Waiting for Locks: How Long Does It Usually Take?*

Christel Baier[1], Marcus Daum[1], Benjamin Engel[2], Hermann Härtig[2], Joachim Klein[1], Sascha Klüppelholz[1], Steffen Märcker[1], Hendrik Tews[2], and Marcus Völp[2]

[1] Institute for Theoretical Computer Science
[2] Operating-Systems Group
Technische Universität Dresden, Germany

Abstract. Reliability of low-level operating-system (OS) code is an indispensable requirement. This includes functional properties from the safety-liveness spectrum, but also quantitative properties stating, e.g., that the average waiting time on locks is sufficiently small or that the energy requirement of a certain system call is below a given threshold with a high probability. This paper reports on our experiences made in a running project where the goal is to apply probabilistic model checking techniques and to align the results of the model checker with measurements to predict quantitative properties of low-level OS code.

1 Introduction

For safety-critical systems such as space, flight, and automotive control systems one wants correctness guarantees for the software not only for the functional behaviour of components but also, e.g., for their timing behaviour. Worst-case execution-time analyses (see e.g. [3,11,22]) are able to provide these guarantees, but only in the form of upper bounds on the execution times of all involved components, which hold even in the most extreme situations. Many computer systems are however either not safety critical or they include fail-safe mechanisms that prevent damage in highly exceptional situations. Quantitative analyses can provide detailed information on the probabilities of certain events or on the average behaviour. First, the requirement that certain requirements hold for all possible execution sequences is a very strong condition. E.g., for uncontended locks the property that a process will find a lock free without waiting might not hold universally, but with some high probability. Second, probabilistic features are crucial for the evaluation of complex architectures such as x86 that are optimised according to their average-performance, for systems that rely on imprecise real-time computing techniques to deal with transient overload [18,6], or for systems that may fail in extreme cases. Third, the probabilistic analysis may guide OS level optimisation justifying, for instance, the use of a simple test-and-test-and-set (TTS) lock implementation over more complex ticket [1] or queue-based locks [15].

* This work was in part funded by the German Research Council (DFG) through the QuaOS project and the collaborative research center 912 Highly-Adaptive Energy-Efficient Computing (HAEC).

M. Stoelinga and R. Pinger (Eds.): FMICS 2012, LNCS 7437, pp. 47–62, 2012.

In this paper, we report on a running project of the operating-system and the formal-methods group at TU Dresden whose aim it is to establish quantitative properties of low-level operating-system (OS) code using probabilistic model checking techniques. By low-level OS code, we mean drivers, the kernel of monolithic operating systems, microkernels or microhypervisors and similar code that directly interacts with hardware devices and that is therefore often optimised to fully exploit the intrinsic behaviour of modern processor architectures. Although the applicability of probabilistic model checking techniques is expected to work in principle, several non-trivial problems have to be addressed.

Modelling. The first challenge is to find a reasonable abstraction level for the formal model based on the probabilistic analysis that will be carried out. E.g., there are several details on the realisation of hardware primitives (such as caches, busses and controllers of the memory subsystem) that have impact on the timing behaviour of low-level OS code. The model must cover all features that dominate the quantitative behaviour, while still being compact enough for the algorithmic analysis. The latter requires to abstract away from details that have negligible impact on the quantitative behaviour or that would render the model unmanageable. The abstraction of many of these details is indispensable because only little information on the hardware realisation is available, and even if these details are known, too fine-grained hardware models make the state-explosion problem unscalable and lead to quantitative results that are too hardware specific. Instead, we incorporate hardware timing effects in the distributions for the execution times and use measurement-based simulation techniques to obtain empirical evidence for the models and the model checking results.

Measurement-based simulation. Generating this evidence from measurement data is the second major challenge because the quantities of interest for many relevant OS-level properties are in a range where measurements significantly disturb the normal system behaviour and where instrumentation-induced noise blurs the results. For instance, the update rate and resolution of CPU-internal energy sensors necessitate a statistical analysis over a multitude of measurements to extract an energy profile for a single system call [5]. To counteract these effects and to obtain empirical evidence for the models and model checking results, we construct microbenchmarks that place the to-be-measured code into a manageable environment and that mimic the formalisation as close as possible.

Quantitative properties. A third major step is the identification of the types of quantitative properties that are relevant for low-level OS code. At a first glance, it seems that constrained reachability conditions such as "what is the probability for threads to find the requested resource locked for longer than 1 microsecond?" can be expressed as probabilistic queries of the form $\mathbb{P}_{=?}(\varphi)$ using comparably simple patterns of path formulas φ in standard temporal logics, such as PCTL. (The notation $\mathbb{P}_{=?}(\varphi)$ refers to the probability for the event specified by φ.) However, the main interest is in deducing probabilities of this kind for the *long run* rather than for fixed initial distributions. Typically, the long-run behaviour of programs (e.g., the time programs hold a certain resource) shows fundamentally different characteristics when compared to the initialisation-phase behaviour (when resources are requested for the first time). These differences are caused by the fact that the system had time to learn and adjust to the program characteristics, e.g., by warming up the disk or processor caches or by adjusting the scheduling parameters of the program to meet its responsiveness and interactivity demands. Queries for questions of the above type must therefore be able to ask for long-run probabilities of path formulas *under the condition* that the system is in a certain set of states. The above question translates into a condition about the states in the long run (e.g., the probability of finding a resource held in the long run) and a temporal formula on the paths starting in these states (e.g., "what is the likelihood that such a held resource will be released and granted to the requesting thread within 1 microsecond?"). We refer to these type of queries as *conditional steady-state queries.* A second class of important queries for low-level OS code asks for the value of a quantity that is not exceeded in the majority of all cases: the *quantile.* Two examples of important quantile-based queries are "how long does a thread wait for a resource in 99.9% of all cases?" and "what is the energy that must remain in the battery to guarantee the complete playback of a certain video with a probability greater than 95%?". To our surprise, neither conditional steady-state queries nor quantile-based queries are supported by state-of-the-art probabilistic model checkers.

Outline. In this paper, we report on our experiences with a simple TTS spinlock as a starting point and initial experiment for a more elaborate investigation of the general feasibility of probabilistic model checking techniques and the limitations of existing tool support. Sec. 2 presents the TTS spinlock and a set of relevant quantitative properties. Sec. 3 explains our discrete-time Markov chain model. Sec. 4 presents the results of the quantitative analysis that we have carried out with the model checker PRISM [13]. We explain how we dealt with conditional long-run probabilities and quantile-based properties and report on the results of the measurement-based simulation and the lessons learned. Sec. 5 concludes this paper. Due to space limitations, we present the detailed model checking statistics in an extended version[1] and provide here only a brief summary.

2 A Test-And-Test-And-Set Lock

Fig. 1 shows the C/C++ code of a simple TTS lock. To acquire the lock, the requesting process executes the atomic swap operation in Line 3 to atomically read the value of the shared variable `occupied` and to then set it to `true`. The loop exits if the process was first to perform this swap after another process has released the lock by setting `occupied` to `false`. For as long as in Line 4 `occupied` is `true`, the process only reads this variable to avoid unnecessary contention on the core-to-core interconnect.

```
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  }
```

Fig. 1. Simple TTS spinlock

[1] `http://wwwtcs.inf.tu-dresden.de/ALGI/spinlock-FMICS2012.pdf`

Properties. For our case study, we investigate four questions as representatives for complex conditional long-run and quantile-based properties:

(A1) probability that a process finds a free lock when it seeks to acquire this lock;

(A2) probability of re-acquiring a previously held lock (without spinning) without other processes having acquired the lock in the meantime;

(A3) expected amount of time a process waits for a lock;

(A4) the 95% quantile of the time processes wait for a lock.

Properties such as (A1)–(A4), which characterise the quantitative behaviour of locks, are highly relevant to guide design decisions and optimisation of low-level OS code. For instance, high probabilities in (A1) and (A2) justify the use of less complex lock implementations, respectively of simpler execution-time analyses. An analysis which assumes low fixed costs for acquiring and releasing a lock is justified by a high probability of (A2) because cache eviction of the `occupied` variable is unlikely. The expected waiting time (A3) is important to judge whether a lock implementation is suitable for the common cases of a given scenario. And the quantile in (A4) replaces the worst-case lock acquisition time in imprecise real-time systems [18,6] and in systems with a fail safe override in case of late results. It returns an upper bound t for the lock-acquisition time that will be met with the specified probability (here 95%).

We investigate these measures in a scenario where a fixed number of processes repeatedly acquire a single shared lock, execute a critical section while holding the lock, and then wait for some time after they have released the lock before they attempt to re-acquire it. We call the time between release and the attempt to re-acquire the *interim time* and refer to the code that is executed during this time as the *interim section.* We assume here, that the length of the critical section is more or less constant, while the interim time varies. Further, critical sections are typically very short in comparison with the times when no lock is held. For the distribution of the length of the interim section we draw inspiration from a video decoding example. For video decoding, the different frame types (I and P-frames) lead to clusters of the interim time in certain small intervals. Our approach used for both model checking and the measure-based techniques relies on discretisation of these distributions using finitely many sampling points.

The modelled lock-acquisition pattern allows for the derivation of results about the common-case behaviour of applications when they use a certain operating-system functionality, but gives also rise to extreme-case analyses where one assumes malicious or erroneous applications to attack the operating system. For example, by setting the interim time to the execution time of a system call minus its critical sections it is possible to deduce the contention of locks that protect these sections under the assumption that malicious applications invoke this system call as fast as they can.

3 Markov Chain Model for the Spinlock Protocol

To model the spinlock protocol, we have chosen a *discrete-time Markov chain* (DTMC) model for the following reasons. The clear demand for probabilistic guarantees about the *long-run* behaviour requires a model where steady-state probabilities are mathematically well defined and supported by model checking tools. This, for instance, rules out probabilistic timed automata and other stochastic models with nondeterminism (e.g., Markov decision processes). Continuous-time Markov chains are not adequate given that the distribution specifying the duration of the critical and interim sections are not exponential. Approximations with phase-type distributions lead to large and unmanageable state spaces. From a mathematical point of view, we could use semi-Markovian models with continuous-time uniform distributions, but we are not aware of tools that provide engines for all queries (A1)–(A4).

Preliminaries: Discrete-Time Markov Chains. We briefly explain our notations used for DTMCs and refer to standard textbooks for further details, see e.g. [12,7]. A (probabilistic) *distribution* on a countable set X is a function $\mu : X \to [0,1]$ such that $\sum_{x\in X} \mu(x) = 1$. The support $\mathit{supp}(\mu)$ of μ consists of all elements $x \in X$ with $\mu(x) > 0$. μ is called a Dirac distribution if its support is a singleton. If $C \subseteq X$ then $\mu(C) = \sum_{x\in C} \mu(x)$.

In our approach, a DTMC is a tuple $\mathcal{M} = (S, \mathbf{P}, s_{\mathit{init}}, \mathit{rew})$ where S is a finite state space, $s_{\mathit{init}} \in S$ the initial state, $\mathbf{P} : S\times S \to [0,1]$ the transition probability matrix and $\mathit{rew} : S \to \mathbb{N}$ the reward function. We require $\sum_{u\in S} \mathbf{P}(s,u) = 1$ for all $s \in S$ and refer to $\mathbf{P}(s,u)$ as the probability to move from s to u within one (time) step. A path in $\mathcal{M}$ is a finite or infinite sequence $\pi = s_0\, s_1\, s_2 \ldots$ of states with $\mathbf{P}(s_i, s_{i+1}) > 0$ for all i. Let $\pi(k) = s_k$ be the $(k{+}1)$-st state and $\pi \downarrow k = s_0\, s_1 \ldots s_k$ the prefix consisting of the first $k{+}1$ states. $\mathit{Paths}(s)$ denotes the set of infinite paths starting in state s. The accumulated reward for a finite path $\pi = s_0\, s_1 \ldots s_k$ is $\mathit{Rew}(\pi) = \mathit{rew}(s_0) + \ldots + \mathit{rew}(s_{k-1})$. If π is a finite path then the cylinder set $\mathit{Cyl}(\pi)$ spanned by π consists of all infinite paths π' where π is a prefix of π'. Using well-known concepts of measure theory, given some probabilistic distribution $\mu : S \to [0,1]$, there exists a unique *probability measure* $\Pr^{\mathcal{M}}_{\mu}$ on the σ-algebra generated by the cylinder sets of finite paths such that $\Pr^{\mathcal{M}}_{\mu}\big(\mathit{Cyl}(s_0\, s_1 \ldots s_k)\big) \;=\; \mu(s_0) \cdot \prod_{0\leqslant i<k} \mathbf{P}(s_i, s_{i+1})$. If μ is a Dirac distribution with $\mathit{supp}(\mu) = \{s\}$ we simply write $\Pr_s$ or $\Pr^{\mathcal{M}}_s$ for $\Pr^{\mathcal{M}}_{\mu}$.

For specifying measurable path events (i.e., sets of paths that belong to the σ-algebra generated by the cylinder sets of finite paths), we use the standard LTL-like notations with the symbols $\bigcirc$ (next), $\mathcal{U}$ (until) and $\Diamond$ (eventually) and time-bounded variants thereof. For $X, Y \subseteq S$, $\bigcirc X$ stands for the set of infinite paths π with $\pi(1) \in X$. $X\, \mathcal{U}^{=k}\, Y$ denotes the set of infinite paths π such that $\pi(n) \in X \setminus Y$ for $0 \leqslant n < k$ and $\pi(k) \in Y$. We write $X\, \mathcal{U}^{\leqslant K}\, Y$ for the union of the sets $X\, \mathcal{U}^{=k}\, Y$ where k ranges over the elements in $\{0, 1, \ldots, K\}$ and $X\, \mathcal{U}\, Y$ for the union of the sets $X\, \mathcal{U}^{=k}\, Y$ where $k \in \mathbb{N}$. $\Diamond\, Y$, $\Diamond^{=k}\, Y$ and $\Diamond^{\leqslant K}\, Y$ are short forms of $S\, \mathcal{U}\, Y$, $S\, \mathcal{U}^{=k}\, Y$ and $S\, \mathcal{U}^{\leqslant K}\, Y$, respectively. To deal with query (A2), we will also use LTL-like formulas with cascades of until-operators and suppose here the standard LTL-semantics for paths. For further details see [20,4,21].

For $m \in \mathbb{N}$, $\theta_m : S \to [0,1]$ denotes the state distribution for $\mathcal{M}$ after m steps. Formally, θ_0 is the Dirac distribution with $\mathit{supp}(\theta_0) = \big\{s_{\mathit{init}}\big\}$ and $\theta_{m+1} = \mathbf{P} \cdot \theta_m$ for $m \geqslant 0$. The function $\theta : S \to [0,1]$,

$$\theta(s) \;=\; \lim_{k\to\infty} \tfrac{1}{k+1} \cdot \sum_{m=0}^{k} \theta_m(s),$$

is called the *steady-state distribution* for $\mathcal{M}$. Then, $\theta(s) > 0$ iff s belongs to a bottom strongly connected component (BSCC) that is accessible from s_{init}. If C is a set of states with $\theta(C) > 0$ and Π a measurable set of paths then the *conditional long-run probability* for Π (under condition C) is defined by:

$$\mathbb{P}^{\mathcal{M}}\big(\,\Pi \,\big|\, C\,\big) \;\overset{\text{def}}{=}\; \sum_{s\in C} \theta(s)/\theta(C) \cdot \Pr^{\mathcal{M}}_s(\Pi)$$

Here, $\theta(s)/\theta(C)$ is the conditional steady-state probability for state s, again under condition C. The intuitive meaning of $\theta(s)/\theta(C)$ is the portion of time spent in state s on long runs relative to the total time spent in states of C. With the factor $\Pr^{\mathcal{M}}_s(\Pi)$, the above weighted sum represents the long-run probability for the event specified by Π under condition C. Analogously, conditional steady-state average values of random variables can be defined as weighted sums. (A3) will be formalised as an instance of *conditional long-run accumulated reward* for reaching a goal set Y defined by:

$$\mathbb{R}^{\mathcal{M}}\big(\,\Diamond Y \,\big|\, C\,\big) \;\overset{\text{def}}{=}\; \sum_{s\in C} \theta(s)/\theta(C) \cdot \mathrm{ExpAccRew}^{\mathcal{M}}_s(\Diamond Y)$$

where $\mathrm{ExpAccRew}^{\mathcal{M}}_s(\Diamond Y)$ denotes the expected accumulated reward for reaching Y from state s. It is defined by:

$$\sum_{r=0}^{\infty} r \cdot \Pr^{\mathcal{M}}_s\big\{\pi \in \mathit{Paths} : \exists k \in \mathbb{N} \text{ s.t. } \pi \in \Diamond^{=k}\, Y \wedge \mathit{Rew}(\pi{\downarrow}k) = r\big\}$$

Markov Chain Model for the Spinlock Protocol. To analyse the quantitative behaviour of the spinlock protocol for n processes $P_1, \ldots, P_n$, we use a DTMC that results as the synchronous parallel composition of one module representing the spinlock in Fig. 1 (see Fig. 3) and one module for each of the processes (see Fig. 2). The t_i's are integer variables that serve as timers for the critical and noncritical section. Distribution ν models the *interim time*, i.e., the time required for the (noncritical) activities of the processes between two critical sections, including the request to acquire the lock, but without the spinning

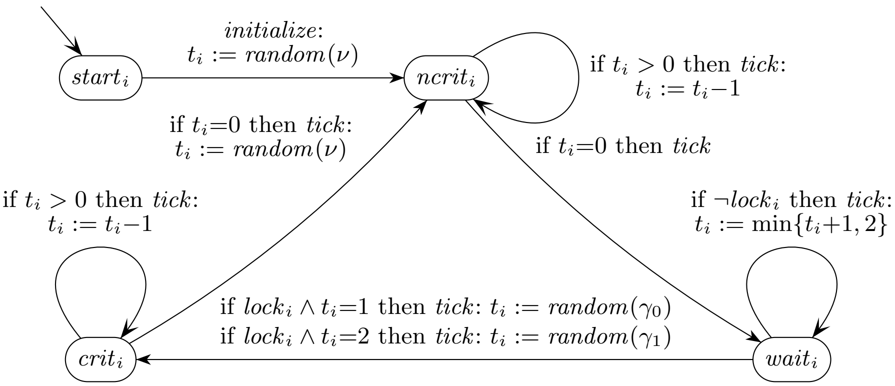

initialize:
$t_i := random(\nu)$
$start_i$
$ncrit_i$
if $t_i > 0$ then $tick$:
$t_i := t_i{-}1$
if $t_i{=}0$ then $tick$:
$t_i := random(\nu)$
if $t_i{=}0$ then $tick$
if $t_i > 0$ then $tick$:
$t_i := t_i{-}1$
if $\neg lock_i$ then $tick$:
$t_i := \min\{t_i{+}1, 2\}$
if $lock_i \wedge t_i{=}1$ then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

Fig. 2. Control flow graph of process P_i

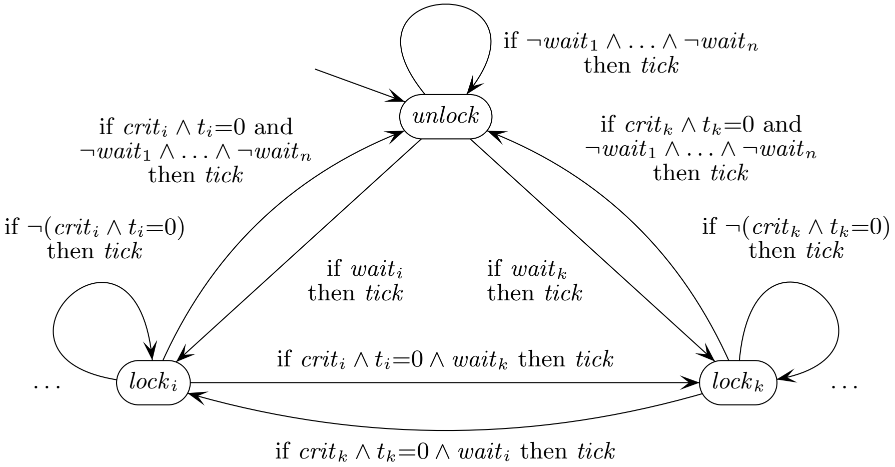

unlock
if $\neg wait_1 \wedge \ldots \wedge \neg wait_n$ then $tick$
if $crit_i \wedge t_i{=}0$ and $\neg wait_1 \wedge \ldots \wedge \neg wait_n$ then $tick$
if $crit_k \wedge t_k{=}0$ and $\neg wait_1 \wedge \ldots \wedge \neg wait_n$ then $tick$
if $\neg(crit_i \wedge t_i{=}0)$ then $tick$
if $\neg(crit_k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

Fig. 3. Control flow graph of the spinlock

time. Distributions γ_0 and γ_1 serve to model the total length of the critical section (including lock acquisition and release). In order to account for cache effects that lead to different time behaviour for the lock acquisition, depending on whether the lock is taken with and without spinning, we use two distributions for the critical section length. Distribution γ_0 is used when the lock was obtained without spinning, while distribution γ_1 is used if some spinning was necessary. To obtain a DTMC model, ν and γ_0, γ_1 are discrete distributions with finitely many sampling points in the relevant time intervals. E.g., the distribution $\nu(8) = \nu(10) = \nu(12) = \frac{1}{3}$ indicates that with equal probability the duration of the interim time is 8, 10 or 12 time units. The assignment $t_i := random(\nu)$ means that a sample is drawn according to distribution ν and assigned to t_i.

For each process P_i, we distinguish four locations. Location $start_i$ serves to set the timers for the first noncritical phase. The other three locations have the obvious meaning. Location $wait_i$ signals that P_i is trying to acquire the lock. If the lock is granted to P_i, the lock process switches to location $lock_i$, which in turn enables the transition from $wait_i$ to $crit_i$. In location $wait_i$, process P_i waits until the lock has been made accessible for it. In location $wait_i$, variable t_i does not serve as a timer. Instead, t_i indicates whether process P_i has just entered the waiting location ($t_i \in \{0, 1\}$) or P_i is spinning as some other process is holding the lock ($t_i{=}2$). The control flow graph of the lock process contains for each P_i one location $lock_i$ (indicating that P_i may take or holds the lock) and one location *unlock* (the lock is free). For the synchronisation, we followed the approach of PRISM's input language with synchronisation over shared actions. Action *initialize* has to be synchronised by all processes, while *tick* indicates one time step and must be executed synchronously by all processes and the lock.

The states of the DTMC $\mathcal{M}$ for the composite model have the form $s = \langle \ell_1, \ldots, \ell_n, m, t_1{=}b_1, \ldots, t_n{=}b_n \rangle$ where ℓ_i is the current location of process P_i, m the current location of the lock and b_i the current value of variable t_i. Then,

P_i is spinning in state s iff $s.m{\neq}lock_i$, $s.\ell_i{=}wait_i$ and $s.b_i{=}2$, where we use the dot notation to refer to parts of a state s. If process P_i performs its last critical action and moves from location $crit_i$ to $ncrit_i$ then either the lock returns to its initial location *unlock* (if no process P_k is spinning) or there is a uniform probabilistic choice for the lock to move to one of the locations $lock_k$ where process P_k is spinning. To compute the long-run average spinning time (query (A3)), we deal with the reward function $rew\_spin_i(s) = 1$ for each state s where process P_i is spinning. For all other states s, we have $rew\_spin_i(s) = 0$.

Formalisation of Queries (A1)–(A4). In the sequel, we use propositional formulas over the locations and conditions on the values of $t_1, \ldots, t_n$ to characterise sets of states. For instance, $crit_i$ is identified with the set $\{s \in S : s.\ell_i = crit_i\}$, where S denotes the state space of the DTMC $\mathcal{M}$ for the composite system. Condition $request_i = wait_i \wedge t_i{=}0$ characterises the set of states s in $\mathcal{M}$ where process P_i has just requested the lock. The set of states where process P_i spins is specified by $spin_i = wait_i \wedge t_i{=}2 \wedge \neg lock_i$. Finally, $release_i = crit_i \wedge t_i{=}0$ characterises the states s where process P_i is just performing its last critical actions and the lock is to be released next. The relevant quantitative measures (A1)–(A4) of Section 2 now correspond to the following values.

$$\begin{array}{lll} \text{(A1)} & \mathbb{P}^{\mathcal{M}}\big(\varphi_1 \,\big|\, request_i\big) \quad \text{where} & \varphi_1 = \bigcirc\, lock_i \\ \text{(A2)} & \mathbb{P}^{\mathcal{M}}\big(\varphi_2 \,\big|\, release_i\big) \quad \text{where} & \varphi_2 = \bigcirc\,(\, unlock\, \mathcal{U}\ lock_i\,) \\ \text{(A3)} & \mathbb{R}^{\mathcal{M}}\big(\Diamond\, lock_i \,\big|\, request_i\big) & \\ \text{(A4)} & \min\big\{\, t \in \mathbb{N} : \mathbb{P}^{\mathcal{M}}\big(\Diamond^{\leqslant t+1} lock_i \,\big|\, request_i\big) \geqslant 0.95 \,\big\} & \end{array}$$

(A1), (A3) and (A4) refer to the conditional steady-state distribution for the condition that process P_i has just performed its first request operation. (A1) corresponds to the long-run probability under the condition $request_i$ for the path event $\varphi_1 = \bigcirc\, lock_i$ stating that in the next time step process P_i will win the race between the waiting processes, i.e., it will enter its critical section without spinning. (A3) stands for the long-run average spinning time from states where P_i has just requested its critical section. Replacing the condition $request_i$ in (A3) with $wait_i \wedge t_1{=}1 \wedge \neg lock_i$, we obtain the long-run average spinning time, provided that the first attempt to acquire the lock was not successful. The quantile in (A4) corresponds to the minimal number t of time steps minus one such that the long-run probability from the $request_i$-states for the path event $\Diamond^{\leqslant t+1} lock_i$ stating that the lock will be granted for process P_i in $t{+}1$ or fewer steps is at least 0.95. For the long-run probability of acquiring the lock, again without interference by other processes (see (A2)), there are several reasonable formalisations. For the constraint that no process other than P_i requested the lock in the mean time, we can deal with the path event $\varphi_2 = \bigcirc(\, unlock\, \mathcal{U}\ lock_i)$ under the condition that process P_i will release the lock in the next step (condition $release_i$). The variant of (A2) where other processes may have acquired and released the lock in between the critical sections of P_i and where P_i has not to spin and hence experiences a low overhead lock can be formalised as $\mathbb{P}^{\mathcal{M}}\big(\varphi_2' \,\big|\, request_i\big)$ for the event specified by the LTL-formula $\varphi_2' = \bigcirc\big(lock_i\ \mathcal{U}\ (ncrit_i \wedge (\neg spin_i\ \mathcal{U}\ crit_i))\big)$. The treatment of (A2) using φ_2 or φ_2' is rather complex since the standard treatment of LTL-queries relies on a probabilistic reachability analysis of a product construction of a deterministic ω-automaton and the DTMC (see e.g. [2]). To avoid this automaton-based approach, one can replace φ_2 and φ_2' with a simpler reachability condition when refining the control flow graph of P_i by duplicating the control loop $\mathit{ncrit}_i\ \mathit{wait}_i\ \mathit{crit}_i$. This can be realised by introducing a Boolean variable b that flips its value after leaving the critical section, see the extended version of this paper, and justified using an appropriate notion of bisimulation. The control flow graphs for the lock and the other processes remain unchanged. Then, instead of φ_2 and φ_2' we can then deal with $\psi_2 = (\mathit{lock}_i \vee \mathit{unlock})\ \mathcal{U}\ (\mathit{crit}_i \wedge b)$ and $\psi_2' = \neg \mathit{spin}_i\ \mathcal{U}\ (\mathit{crit}_i \wedge b)$ under the condition $\mathit{release}_i \wedge \neg b$ and $\mathit{request}_i \wedge \neg b$, respecively. Indeed, our experiments show that the analysis of $\mathcal{M}'$ with the modified queries is more efficient than the analysis of $\mathcal{M}$.

4 Quantitative Analysis of the TTS Spinlock

Our general approach to the quantitative analysis of low-level operating-system code proceeds in four steps: (1) The targeted operating-system code is formalised at a suitable level of abstraction in the input language of a probabilistic model checker. For our TTS lock case study, we used the prominent model checker PRISM [13]. (2) The parameters of the model are determined with the help of measurements in the targeted setting. If this is not possible due to unduly high interference with the instrumentation code, we extract the relevant code and measure it in the form of a microbenchmark in a controlled environment. (3) The queries of interest are evaluated both by the model checker and with a microbenchmark that again executes the code in a manageable environment. This step is necessary to determine how well the model corresponds to the targeted setting. (4) The queries are evaluated on the parameters obtained for the real workload and if possible cross-checked against measurements in this setting. There are two situations in which such a comparison is infeasible, namely when the interference between the instrumentation code and the targeted operating-system code fundamentally changes the behaviour of the latter or when the analysis is performed with parameters of a system that does not yet exist.

The measurements done in the microbenchmarks do as well interfere with the to-be-analysed code. However, these measurements only extract the required model parameters (possibly only one at a time). The challenge is to construct a setting where this interference is limited only to those parameters that are not currently extracted. We now report on our experiences in adjusting the model with the help of a microbenchmark and measurement-based simulation.

Measurement-Based Simulation. For the extraction of the two distributions γ_0 and γ_1 for the critical section and for the distribution ν for the interim section, we can resort to random sampling and similar techniques [14] to deduce the characteristics of the workload we seek to investigate. Very short critical sections and the acquire and release costs of the lock must however be measured in an environment where it is possible to control side effects on the quantity of interest. For the TTS lock, we construct such an environment by mimicking the behaviour of the formal model in the critical and interim section. When a process enters a critical section, it selects pseudorandomly a sampling point of the distribution γ_0 if the lock was free and of γ_1 if it had to spin. A counter in the spinning loop (in Line 4 in Fig. 1) reveals whether or not the process was able to obtain the lock without spinning. We pass this counter in a processor register not to disturb the timing of the TTS code. The actual instrumentation consists of three reads of the per core time stamp counter, before and after acquiring the invocation of the `lock()` function in Line 2 and after the `unlock()` function returns. Both the critical and interim section consist of a loop of an integer instruction: `rep; dec %eax`. We have confirmed that the execution time of this loop is very regular. The loop executes for a time that is proportional to the sampling point selected for the respective critical or interim section.

Quantitative Analysis Using PRISM. For the quantitative analysis of the DTMC of the TTS spinlock we used the probabilistic model checker PRISM [13]. We mainly concentrated on (A1)–(A4), but also considered functional and a few more quantitative queries. To obtain empirical evidence in the model and in the model checking results we compare the model checking results with measurements of the model mimicking a microbenchmark. Unfortunately, PRISM has no direct support for computing conditional long-run probabilities or quantile-based queries. We extended the PRISM code by operators that compute conditional long-run probability $\mathbb{P}^{\mathcal{M}}\big(\,\varphi\,\big|\,C\,\big)$ and conditional long-run accumulated rewards $\mathbb{R}^{\mathcal{M}}\big(\,\Diamond\, Y\,\big|\,C\,\big)$ where φ is a PCTL path formula and Y, C sets of states. Although there is also no direct support for (A4) in PRISM, quantiles that refer to the amount of time until some event occurs can be calculated with the same iterative bottom-up computation scheme as for bounded reachability properties. In (A4), we are interested in the minimal $t \in \mathbb{N}$ such that the conditional long-run probability is greater than some fixed probability value. For finding the minimal value $t \in \mathbb{N}$ with the above property efficiently, we modified the implementation of the bounded until operator in PRISM to store the intermediate probabilities for all $0 \leqslant j \leqslant t$. Instead of a more direct evaluation at the MTBDD level of PRISM, this storing of intermediate results allows for an external script to check the gradually increasing values of the bounded until formula without restarting the model checking for each such check.

Lessons Learned. During the analysis of the queries (A1)–(A4) and in the course of performing and evaluating the measurements for the simulation, we encountered several difficulties that we would like to share.

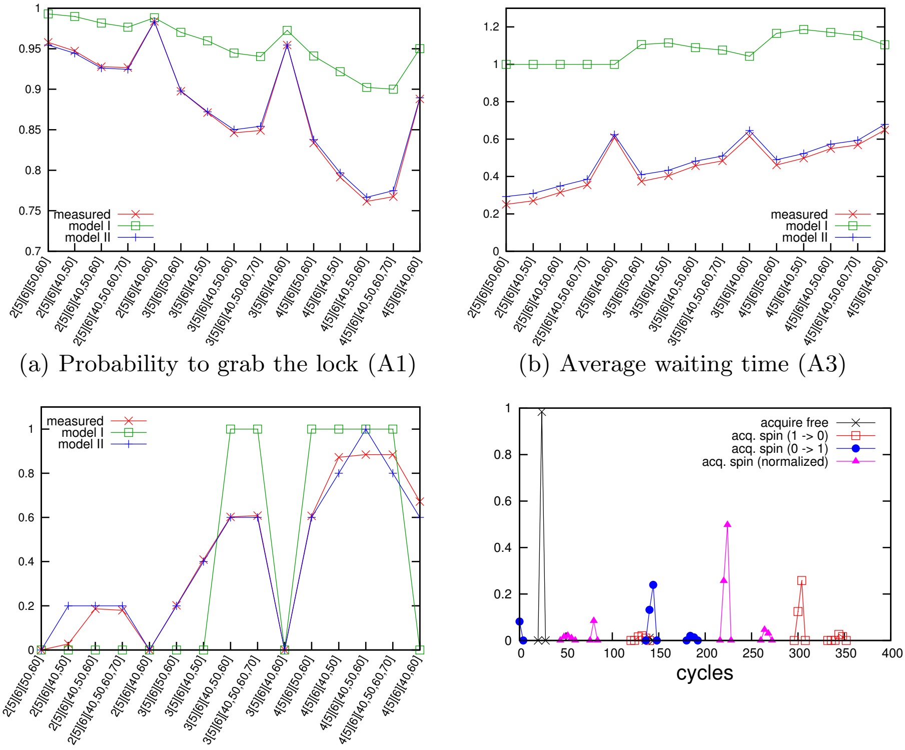
Cascade effects. We first evaluated (A1)–(A4) using a model where the length of the critical section and the interim time are deterministic, i.e., where $\gamma_0 = \gamma_1$ and ν are Dirac distributions with values T_{crit} and T_{int} and where $T_{crit} \ll T_{int}$. A simulation run of the model revealed a probabilistic choice of the order in which processes acquire the lock for the first time. For all subsequent turns, the processes received the lock in this same order and without having to spin. Small variations in the lock acquisition times and in the points in time when the processes start prevented the measurement-based simulation from entering a similar cascade. These cascade effects, however, do not appear for more realistic models where at least ν is a distribution with $|supp(\nu)| \geqslant 2$. Assuming that the values for ν are much larger than those for γ_0 and γ_1, such DTMC models only have one bottom strongly component, which justifies to apply the measurements for just a few simulation runs.

Short critical sections. One of the first workloads we evaluated, was a system call of the Nova microhypervisor [19] in which very short TTS-lock protected critical sections alternate with relatively long sequences of interim activities. The measure-based approach encountered situations where the lock-acquisition times exceeded the critical section length. To avoid a too fine granular (discrete) time domain for the DTMC model that would rule out the feasibility of model checking techniques, we used time domains of different granularity for the measurements and the DTMC model and relate them via a scaling factor *sf*. E.g., for the scaling factor $sf = 1000$, one time unit in the DTMC model corresponds to 1000 cycles (approx. 362 ns) on the target system.

Varying acquisition times. In an early version of the microbenchmark, we adjusted the `rep; dec %eax` loops, which together with the pseudo random choice of a sampling point mimic the distributions of the critical and interim sections, to the same constant value for all processes. More precisely, for $\gamma(x) = 1$, we adjusted the loop in critical to consume as close as possible to 1000 cycles minus the time required to execute the instrumentation code. However, the average costs for acquiring and releasing a lock increase with the number of processes that require this lock. One explanation for this behaviour could be that the costs for invalidating a cache line when acquiring or releasing the lock vary with varying number of cores. This is because the on-chip networks in modern multicore processors connect cores in a point-to-point fashion while maintaining information about the locations of copies of cache lines.

Controlled measurement environment. We realised early variants of our microbenchmark as a Linux user-level application, disabled all obvious sources of interference and raised the priority of this application into the otherwise empty real-time priority band. From the results, we classified spikes as interference and ignored these points in our comparison with the formal model. Still, we experienced high fluctuations of the measurement results, which could not easily be explained. In a repeated measurement on top of a small kernel binary, which we just used to bootstrap our microbenchmark and to communicate the results after the measurement part completed, these variations did not reappear. We therefore take this effect as an indicator and as a warning that the interference of large operating-system kernels on short executing microbenchmarks should not be underestimated and best be avoided whenever this is possible.

Need for two critical distributions. After having performed the above adjustments, we observed a discrepancy between the model checking results and the measurements of approx. 20% (see the model I results in Fig. 4(a)–4(c)). Following an in-depth search for possible causes both in the model and in the



(a) Probability to grab the lock (A1)

(b) Average waiting time (A3)

(c) 95% quantile (A4)

(d) Frequency of acquire free and spin

Fig. 4. (a)-(c) results for model I and II and (d) the histogram for acquire time

microbenchmark, we identified small variations between the costs of acquiring a lock with and without spinning. In the course of this search, we encountered several other possible causes such as the quantisation due to scaling, which showed similar small variations in the measurements. To confirm these factors, we adjusted a copy of the measurement data to mimic the effect we suspected to determine the magnitude of the impact this effect could have. If this impact was in a range where it could have explained a significant part of this discrepancy, we adjusted the formal model accordingly. For the small variations between lock acquisition times, we changed the model from a single Dirac distribution for the critical section length to the two Dirac distributions γ_0 and γ_1 in Fig. 2.

Unfortunately, the value of the singleton sampling point of γ_1 could not be directly measured because it would require the inclusion of `rdtsc` in the spinning loop (Line 4 in Fig. 1) to read the core cycle counter, which would significantly change the timing behaviour of the lock. We therefore calculate the average costs for acquiring a lock under the condition that a process had to spin by comparing the time stamps of the releasing cores with the time stamps of the lock acquiring cores. However, although the time stamp counter for cores on the same die are derived from the same clock, there is an offset caused by the barrier on which these cores synchronise to start the measurement at approx. the same point in time. The two dashed lines in Fig. 4(d) show the acquire spin costs per core. The figure also shows the acquire spin costs after normalising the offset between the two clocks. We choose the spike as the sampling point for γ_1. The scaling factor *sf* = 1000 that we used to deal with the single-distribution model (where $\gamma(1) = 1$) was no longer adequate for integrating the resulting distributions ($\gamma_0(5)=\gamma_1(6)=1$) into the DTMC model. We therefore decreased the scaling factor to *sf*=200.

Evaluation of the Results. Fig. 4(a)–4(c) show the results of our quantitative analysis of the queries (A1)–(A4). We omit the plot for (A2) because, for selected distributions, the chance to re-acquire the lock without interference by other processes is close to zero. The plots compare our measurements with the results from two models. Model I is our earlier, simpler model which uses only one distribution for the critical section ($\gamma_0=\gamma_1$). Model II is more precise because it uses different distributions γ_0 and γ_1 as explained before. The x-axis displays the number of processes and the distributions used for model II and, with scaling factor *sf*=200, for the measurement. The label $3[5][6][40,50,60]$ stands for n=3 processes and the distributions $\gamma_0(5)=\gamma_1(6)=1$ and $\nu(40)=\nu(50)=\nu(60)=\frac{1}{3}$. The distribution for model I, which uses scaling factor *sf*=1000, is obtained by dividing all values by 5 and using the value of γ_0 for γ_1 too. Thus, for model I, the same label stands for the model with 3 processes, $\gamma_0(1)=\gamma_1(1)=1$ and $\nu(8)=\nu(10)=\nu(12)=\frac{1}{3}$. We performed the measurements on an Intel i7 920 quadcore machine at 2.67 GHz. The benchmark and the sampling area for storing the measurement results fitted completely in the on-die caches. Fig. 4(a) shows that the introduction of γ_1 reduced the error between the measured and analysed results from 20% for model I to below 1% for model II. The discrepancy between model II and the measurements in Fig. 4(c) is due to the quantisation of the model. That is, the model considers changes of the waiting time only in steps of *sf*=200 cycles, which corresponds to one fifth (0.2) of the critical section length. The differences between the results obtained for the models with parameters $n[5][6][40,50,60]$ and $n[5][6][40,60]$ (where $n \in \{2,3,4\}$) illustrate that not only the mean value, but also the variance of distribution ν for the interim time has non-negligible impact on (A1)-(A4).

For model I (*sf*=1000 and $\gamma_0=\gamma_1$), we used PRISM to compute (A1)–(A4) for the DTMC with up to six processes. For model II (*sf*=200, $\gamma_0 \neq \gamma_1$) the analysis has been carried out with up to four processes. Because of the reduced scaling factor, model II is far more complex. E.g., for n=4 processes and the distribution $\nu(8) = \nu(14) = \frac{1}{2}$ the DTMC for model I has about 10^4 states, while for model II with the corresponding distribution $\nu(40) = \nu(70) = \frac{1}{2}$ the DTMC has about $2.5 \cdot 10^6$ states. In a nutshell, for the DTMC with n=6 and *sf*=1000 PRISM needs a few minutes for all queries, while for n=4 and *sf*=200 the computation can take a few hours. In most cases, the computation of the steady-state probabilities is the most time consuming part. For more information on the PRISM statistics (MTBDD sizes, time for the model construction and the quantitative analysis) we refer to the extended version.

5 Conclusions

The paper presents a first step towards the application of probabilistic model checking techniques for the quantitative analysis of low-level operating-system code. We reported on the difficulties we encountered when analysing a simple test-and-test-and-set (TTS) spinlock with the model checker PRISM and on our solutions to address them. A major challenge was to find an appropriate level of abstraction that allows to capture all relevant behaviour and allows to abstract from the precise timing behaviour of the cache and other CPU parts. We performed extensive measurement-based simulations of real spinlocks to demonstrate that our abstract model does indeed reproduce important aspects of the timing behaviour. We considered a representative list of properties that are of high interest to the system designer when he has to choose the right lock implementation. These properties involve conditional steady-state probabilities and quantiles. To overcome the lack of direct tool support for both types of queries, we added the relevant features in the PRISM code.

Related work. Many researchers performed case studies with probabilistic model checkers for mutual exclusion protocols and other coordination algorithms for distributed systems (see e.g. [8,16,17] or the PRISM web pages [13]). While some of these case studies address the analysis of randomised protocols, we deal with non-randomised operating-system primitives (the TTS spinlock). Our models rely on stochastic assumptions on the execution times of the critical section and interim sections of competing processes). Unlike the wide range of case studies with continuous-time models, where rates of exponential distributions specify, e.g., the frequency of the arrival of requests or the average duration of events (see e.g. [7]), we deal with a DTMC model and discretisations of non-exponential distributions. Of course, there have been plenty of case studies with the DTMC engine of PRISM, but we are not aware of experiments that have been carried out where the modelling and evaluation process was accompanied by measure-based techniques. To the best of our knowledge, none of these case studies considers conditional steady-state probabilities or quantile-based queries. The majority of work on model checking low-level operating-system code concentrates on properties in the safety-liveness domain. E.g., [23] used model checking to find serious file system bugs. Formal quantitative analyses often consider only worst-case execution times as measure. For the special case of probabilistic worst-case execution times (i.e. queries similar to Query (A3)), [3] presents a timing-schema based on independent or only pairwise dependent (i.e., joint) execution profiles.

Future work. It would be very interesting to scale the analysis of basic and more advanced spinlocks for CPUs with more than 100 cores. Such results could justify the use of more simple locks with less overhead for certain tasks. Given the exponential growth of the system model with the number of cores, this is an extremely challenging task and requires clever encoding, abstraction and reduction techniques such as symmetry reduction. Other promising candidates are bisimulation quotienting techniques as supported by the model checker MRMC [9,10] and the use of sophisticated (MT)BDD-based techniques to increase the efficiency of PRISM's symbolic engine.

References

1. Anderson, T.E.: The performance of spin lock alternatives for shared-memory multiprocessors. IEEE Trans. Parallel Distrib. Syst. 1(1), 6–16 (1990)
2. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press (2008)
3. Bernat, G., Colin, A., Petters, S.: WCET analysis of probabilistic hard real-time systems. In: RTSS 2002, pp. 279–288. IEEE (2002)
4. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. Journal of the ACM 42(4), 857–907 (1995)
5. Hähnel, M.: Energy-utility functions. Diploma thesis, TU Dresden, Germany (2012)
6. Hamann, C.-J., Löser, J., Reuther, L., Schönberg, S., Wolter, J., Härtig, H.: Quality-assuring scheduling - using stochastic behavior to improve resource utilization. In: RTSS 2001, pp. 119–128. IEEE (2001)
7. Haverkort, B.: Performance of Computer Communication Systems: A Model-Based Approach. Wiley (1998)
8. Irani, S., Singh, G., Shukla, S.K., Gupta, R.: An overview of the competitive and adversarial approaches to designing dynamic power management strategies. IEEE Trans. VLSI Syst. 13(12), 1349–1361 (2005)
9. Katoen, J.-P., Kemna, T., Zapreev, I., Jansen, D.: Bisimulation Minimisation Mostly Speeds Up Probabilistic Model Checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 87–101. Springer, Heidelberg (2007)
10. Katoen, J.-P., Zapreev, I., Hahn, E., Hermanns, H., Jansen, D.: The ins and outs of the probabilistic model checker MRMC. Perform. Eval. 68(2), 90–104 (2011)
11. Knapp, S., Paul, W.: Realistic Worst-Case Execution Time Analysis in the Context of Pervasive System Verification. In: Reps, T., Sagiv, M., Bauer, J. (eds.) Wilhelm Festschrift. LNCS, vol. 4444, pp. 53–81. Springer, Heidelberg (2007)
12. Kulkarni, V.: Modeling and Analysis of Stochastic Systems. Chapman & Hall (1995)
13. Kwiatkowska, M., Norman, G., Parker, D.: Probabilistic symbolic model checking with PRISM: A hybrid approach. STTT 6(2), 128–142 (2004)
14. Liedtke, J., Islam, N., Jaeger, T., Panteleenko, V., Park, Y.: Irreproducible benchmarks might be sometimes helpful. In: ACM SIGOPS European Workshop, pp. 242–246. ACM (1998)
15. Mellor-Crummey, J., Scott, M.: Scalable reader-writer synchronization for shared-memory multiprocessors. In: PPOPP 1991, pp. 106–113. ACM (April 1991)
16. Norman, G.: Analysing Randomized Distributed Algorithms. In: Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.-P., Siegle, M. (eds.) Validation of Stochastic Systems. LNCS, vol. 2925, pp. 384–418. Springer, Heidelberg (2004)
17. Norman, G., Parker, D., Kwiatkowska, M., Shukla, S., Gupta, R.: Using probabilistic model checking for dynamic power management. Formal Aspects of Computing 17(2), 160–176 (2005)
18. Shih, W.K., Liu, J.W.-S., Chung, J.-Y.: Algorithms for scheduling imprecise computations with timing constraints. SIAM J. Comput. 20(3), 537–552 (1991)

19. Steinberg, U., Kauer, B.: NOVA: a microhypervisor-based secure virtualization architecture. In: EuroSys 2010, pp. 209–222. ACM (2010)
20. Vardi, M.: Automatic verification of probabilistic concurrent finite-state programs. In: FOCS 1985, pp. 327–338. IEEE (1985)
21. Vardi, M.: Probabilistic Linear-Time Model Checking: An Overview of the Automata-Theoretic Approach. In: Katoen, J.-P. (ed.) ARTS 1999. LNCS, vol. 1601, pp. 265–276. Springer, Heidelberg (1999)
22. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem - overview of methods and survey of tools. Trans. Embedded Comput. Syst. 7(3), 1–53 (2008)
23. Yang, J., Twohey, P., Engler, D., Musuvathi, M.: Using model checking to find serious file system errors. ACM Trans. Comput. Syst. 24(4), 393–423 (2006)