Received: Added at production | Revised: Added at production | Accepted: Added at production
DOI: xxx/xxxx

RESEARCH ARTICLE

Mutable Locks: Combining the Best of Spin and Sleep Locks

Romolo Marotta[1] | Davide Tiriticco[1] | Pierangelo Di Sanzo[1] | Alessandro Pellegrini[1] | Bruno Ciciani[1] | Francesco Quaglia[2]

[1]DIAG, Sapienza, University of Rome, Italy
[2]DICII, Tor Vergata University, Rome, Italy

Correspondence
Romolo Marotta, DIAG, Sapienza, University of Rome, Italy.
Email: marotta@diag.uniroma1.it

Pierangelo Di Sanzo, DIAG, Sapienza, University of Rome, Italy.
Email: disanzo@diag.uniroma1.it

Francesco Quaglia, DICII, Tor Vergata University, Rome, Italy.
Email: Francesco.Quaglia@uniroma2.it

Summary

In this article we present Mutable Locks, a synchronization construct with the same semantic of traditional locks (such as spin locks or sleep locks), but with a self-tuned optimized trade off between responsiveness and CPU-time usage during threads' wait phases. Mutable locks tackle the need for efficient synchronization supports in the era of multi-core machines, where the run-time performance should be optimized while reducing resource usage. This goal should be achieved with no intervention by the programmers. Our proposal is intended for exploitation in generic concurrent applications, where scarce or no knowledge is available about the underlying software/hardware stack and the workload. This is an adverse scenario for static choices between spinning and sleeping, which is tackled by our mutable locks thanks to their hybrid waiting phase and self-tuning capabilities.

KEYWORDS:
Multi-core platforms, Thread synchronization, Locking supports, Self-tuning, Shared-memory algorithms

1 | INTRODUCTION

Modern multi-core chipsets and the ever-growing adoption of concurrent programming in daily-usage software are posing new synchronization challenges. Non-coherent cache architectures—such as Intel SCC (Single-chip Cloud Computer) [1]—look to be a way for reducing the cost of classical cache-coherency protocols. However, they significantly impact software design, since hardware-level coherency becomes fully demanded from software—which needs to rely on explicit message passing across cores to let updated data values flow in the caching hierarchy. Hardware-Transactional-Memory (HTM) allows atomic and isolated accesses to slices of shared data in multi-core machines. This solution is however not viable in many scenarios because of limitations in the HTM firmware, like the impossibility to successfully finalize (commit) data manipulations in face of hardware events such as interrupts. For this reason, HTM is often used in combination with traditional locking primitives, which enable isolated shared-data accesses otherwise not sustainable via HTM.

The Software Transactional Memory (STM) counterpart avoids HTM-related limitations. However, STM internal mechanisms (e.g. TL2 [2]) still rely on locks to enable atomic and isolated management of the metadata that the STM layer exploits to assess the correctness (e.g. the isolation) of data accesses performed by threads. Furthermore, locking is still exploited as a core mechanism in software-based shared-data management approaches for multi-core machines like Read-Copy-Update (RCU) [3], where readers are allowed to concurrently access shared data with respect to writers, but concurrent writers are anyhow serialized via the explicit usage of locks.

Overall, despite the rising trend towards differentiated synchronization supports, locking still stands as a core synchronization mechanism. Therefore, optimizing the runtime behavior of locking primitives is a core achievement for software operations carried out on nowadays multi-core hardware.

Actually, we can distinguish among two main categories of locks: 1) *spin locks*, which are based on threads actively waiting for the ownership in the access to the target shared resource; 2) *sleep locks*, which make threads not run on any CPU-core until they can acquire the ownership in the access. As well known, the first category can operate in user-space, while the latter requires the support of the underlying Operating System (OS) kernel.

Spin locks are often preferred in HPC applications—where low or none time-sharing interference is expected in the usage of CPU by threads—since they ensure the lowest latency while acquiring the ownership of the lock. However, they do not care about metrics such as CPU usage. In fact, CPU cycles wasted in spinning operations because of conflicting accesses to critical sections may result non-negligible, especially at non-minimal thread counts. Moreover, this might increase the impact of hardware contention on performance, since spin operations typically involve atomic instructions that trigger the cache-coherence firmware. Conversely, sleep locks save CPU cycles and reduce hardware contention, thus representing the obvious alternative to spin locks when resource usage (while synchronizing) is a concern. However, they might increase the latency in the access to a critical section because of delays introduced by the OS while awakening and CPU-dispatching threads.

To tackle the limitations of spin and sleep locks, in this article we present a new synchronization support called mutable lock (also denoted as `mutlock`). Our solution is based on a non-trivial combination of spin and sleep primitives, which gives rise to a state machine driving the evolution of threads in such a way that sleep-to-spin transitions are envisaged as a means to make sleeping threads become ready to quickly enter the critical section when it is released by another thread. Hence, they do not pay the delay caused by the OS awakening phase. On the other hand, the sleep phase is retained as a means for controlling the waste of resources that would otherwise be experienced with pure spin locking.

Our mutable locks ship with the support for the autonomic tuning of the transitions between sleep and spin phases—or the choice of one of the two upon the initial attempt to access the critical section—which is implemented as a control algorithm encapsulated into the locking/unlocking primitives. The main challenge we had to tackle while devising this algorithm has been the one of avoiding anomalies, such as livelocks, potentially caused by inaccurate views of threads on the state of other threads (sleeping or spinning) involved in the access to the mutable lock. We addressed this challenge by embedding an ad-hoc management of metadata (based on smart combinations of atomic machine instructions) into the control algorithm offering the support for the autonomic choices. Furthermore, our solution is fully transparent, and can be exploited by simply redirecting the API of the locking primitive originally used by the programmer to our mutable lock library[†].

The remainder of this article is structured as follows. In Section 2 we discuss related work. Mutable locks are presented in Section 3. Experimental results for a comparison with other lock implementations are reported in Section 4.

2 | RELATED WORK

Spin locks have been originally implemented by only relying on atomic read/write instructions [4,5]. However, this solution had limited applicability to scenarios where the number of threads to synchronize was known at compile/initialization time and could not change at runtime. Such a limitation was overcome thanks to atomic Read-Modify-Write [6] (RMW) instructions, like CAS in modern processors. The main idea behind RMW-based spin locks is the one of repeatedly trying to atomically switch a variable from a value to another one—the so-called test-and-set operation. If a thread succeeds in this operation, it can proceed and execute the critical section, otherwise it has to continuously retry the operation. This is the *spin* phase. Spin locks are greedy in terms of clock-cycles usage, thus leading to non-minimal waste of resources in scenarios with non-negligible likelihood of thread conflict in the access to critical sections. This problem is further exacerbated by the fact that RMW instructions make intensive usage of state transitions in the hardware-level cache coherency protocol. This in turn can impact the cache access latency by other threads, including the one that is currently owning the critical section. Clearly, the more threads spin at the same time, the worse the scenario becomes.

The *test-and*-test-and-set (TTAS) spin lock [7] makes challenging threads continuously check the lock variable until it is released and, only in this case, they try to acquire it via RMW instructions. This allows threads to spin (read the actual value of the lock variable) in cache without disturbing others, thus generating cache/memory traffic only when strictly needed.

Anderson et al. [8] introduce a simple back-off time before attempting to re-acquire the lock. Anyhow, such a strategy requires some variables to be set up, such as the maximum and minimum back off time, which cannot be universal across any hardware architecture and/or workload [9].

Since there is no assurance that a given thread wins the challenge eventually, spin locks might lead to starvation. Mellor-Crummey and Scott [10] introduce the queue spin lock, called MCS, to address this issue. It is a linked list where the first connected node is owned by the thread holding the lock, while others are inserted in FIFO order by threads trying to access the critical section. These threads spin on a boolean variable encapsulated in their individual nodes. This guarantees that each spinning thread repeatedly reads a different memory cell and that a releasing thread updates a cache line owned by a unique CPU-core, which significantly reduces the pressure on the cache management firmware.

In all the above solutions, there is no direct attempt to control the number of threads spinning at each time instant, as instead we do in our mutable locks thanks to the smart combination of spin/sleep phases and sleep-to-spin transitions. Clearly, such a limitation of the literature approaches can lead to catastrophic consequences on performance and resource usage when applications are executed (or simply have phases of execution)

[†] Code available at https://github.com/HPDCS/libmutlock

with more threads than cores. Furthermore, when recurring to FIFO spin locks an anti pattern emerges. The FIFO semantic imposes that, when a thread releases the lock, *one specific thread* has to acquire it—the one standing at the head of the FIFO queue. It follows that delays (for example a CPU de-scheduling) affecting that thread impact all its successors in the queue. This increases considerably the residence time (queue time plus critical section execution time) and, consequently, the overall application performance can be impaired. Spin locks anyhow suffer from the problem of waste of resources in face of conflicting accesses to the critical section. Our mutable locks cope with both these problems, since the smart combination of spin and sleep phases avoids the anti-pattern where a thread running a long critical section is de-scheduled in favor of one simply spinning for the access to the same critical section.

As hinted, sleep locks—based on OS blocking services—represent the opposite solution to synchronization, and are aimed at avoiding the waste of resources (which would take place with spin locks) during wait phases preceding the access to the critical section. OS implementations offer sleep locks since their very beginning, and various improvements in these synchronization constructs have been devised in order to enable flexible synchronization schemes, involving awake conditions resulting as the combination of the state of multiple sleep locks. Examples are the System V semaphores offered by Posix[11] or the wait-for-multiple-object primitive offered by WinAPI[12]. In any case, all the sleep locks based on blocking OS services share the common drawback that, as soon as a critical section is released, there is no guarantee that a thread willing to access the critical section is already CPU-dispatchable (or dispatched). In fact, it might have gone sleeping, thus needing to undergo a wake-up phase bringing it back onto the OS run-queue. Overall, we may experience a delay in the access to the critical section by this thread, which in turn may hamper performance, especially when the critical section is short—a problem exacerbated at higher concurrency.

A lock implementation which copes with the issue of choosing at runtime between spinning and sleeping is the *mutex* offered by the glibc `pthread` library[13]. This lock can work with two different configurations, i.e. default and adaptive. With the default configuration, a thread tries to acquire the lock by performing an atomic test-and-set operation. If this operation fails, the thread goes sleeping. The adaptive configuration is based on the idea of attempting to spin for a while before going to sleep. Such an approach is known as *spin-then-sleep*. In the adaptive mutex, the time after which a thread transits from spinning to sleeping is established by a threshold, which is computed at runtime based on the length of the critical section, and is bounded by the estimated context-switch time.

The spin-then-sleep approach has been in-depth studied by Karlin et al.[14] and then by Lim et al.[15]. In the former study, the authors explore various strategies for determining whether and how long to spin before sleeping. Their experimental results show the potential advantages of the spin-then-sleep approach compared to the pure sleep or pure spin approaches, and demonstrate that algorithms which adaptively tune the spinning threshold generally perform better than static ones. The study by Lim et al. claims that, in large scale multiprocessor systems, statically determining the spinning threshold should be preferred, due to the high run-time overhead that may be required to find its optimal value. Thus, the authors propose static methods to choose good values of the threshold depending on the properties of the wait-time distribution, and estimate their competitive factors with respect to the optimal off-line algorithm.

One disadvantage of the glibc `pthread` mutex (in both configurations), and generally of various locks based on the spin-then-sleep approach described above, is that they pay along the critical path the wake-up latency of sleeping threads that access to the critical section. Also, they may suffer from an excessive amount of sleep/wake-up calls.

Some literature studies focus on the selection of the best synchronization algorithm depending on the level of contention. As an example, the proposal by Beng-Hong Lim et al.[16] investigates how to dynamically switch between TTAS and MCS locks. The proposed approach has been later extended by Eastep et al.[17] to specifically address asymmetries in multi-core machines. The authors designed a synchronization framework that dynamically switches between five different spin-lock implementations with the objective of maximizing an application performance metric. GLS (Generic Locking Service)[18] is an adaptive scheme that automatically selects the lock algorithm depending on the workload features. With low contention, GLS uses a simple spin-lock scheme, while with high contention it switches to queue-based locks or, under high system load, to sleeping locks. All these proposals rely on an orthogonal approach compared to our mutable locks, since they use one individual lock implementation at a time, and do not explore the opportunities offered by hybrid approaches. Conversely, mutable locks aim at combining spin locks and sleep locks such that they can coexist at the same time. Nevertheless, we note that mutable locks could be used in the context of the above-mentioned solutions as a candidate lock implementation among the ones that are dynamically selected. This also underlines the fact that mutable locks and the approach taken in these solutions can be considered complementary.

Johnson et. al[19] explore a solution based on the idea of decoupling the scheduling of threads from lock contention management, and show the potential advantages of their approach, which is based on a load control mechanism. Moreover, they discuss some major aspects to tackle, such as obtaining accurate statistics on resource usage by threads and improving OS support for optimizing the communication between OS and application. The main difference between this solution and mutable locks is that the former is essentially built around the concept of load control. It introduces a *daemon* thread that maintains system statistics and decides when to (de)schedule application threads. This thread has to be periodically woken-up, and relies on OS facilities to collect statistics (e.g. the time spent by a thread on CPU and waiting in run-queues). Scheduling decisions are based on the intuition that threads should be removed from CPU in the case of high load, and re-scheduled when the load drops down. Mutable locks do not rely on any load control scheme based on system statistics, and they do not rely on any daemon thread to make scheduling decisions. Mutable

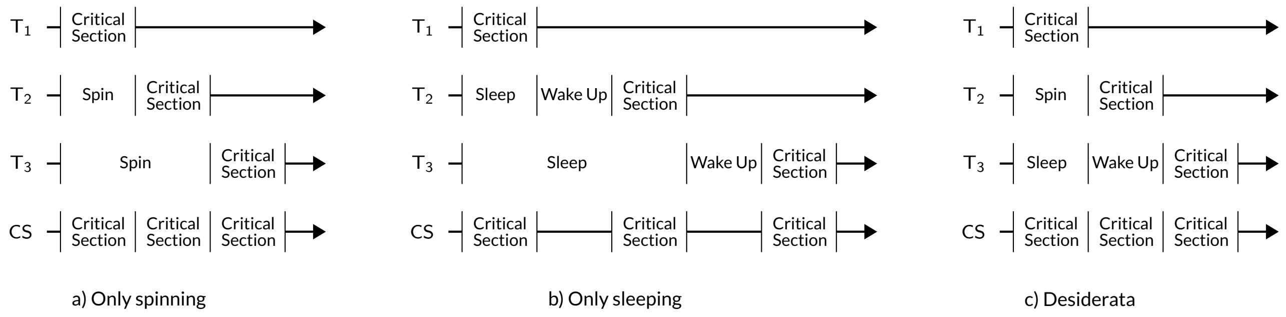


FIGURE 1 Different time-lines related to different lock specifications

locks represent an algorithmic solution completely distributed among the application threads. The adaptive policy of mutable locks exclusively observes what happens when a sleeping thread is woken-up and tries to acquire the lock.

Mutexee[20] is a lock implementation that attempts to reduce the delay in the access to the critical section. It has been designed considering energy efficiency as a primary goal. Mutexee spins using low-energy-footprint instructions, and exploits a bimodal threshold for spin-to-sleep transitions. Also, it introduces a wait phase during the lock release operation in order to increase the probability that an incoming new lock request by another thread can immediately successfully complete, thus saving a pair of sleep/wake-up calls. This favors throughput and energy efficiency against fairness.

Another lock falling in the family of spin-then-sleep locks is called malthusian lock[21]. It is a modified version of the MCS lock where spin phases are replaced with spin-then-sleep waits, and is paired with an additional list of threads which sleep longer than the ones in the main queue of locking requests. Again, this behavior favors throughput against fairness.

One limitation of both mutexee and malthusian locks is that they do not attempt to pro-actively wake-up a sleeping thread that must access the critical section upon the lock release by another thread. In fact, in these approaches, a sleeping thread is woken-up only upon the lock release operation. Hence, like for pure sleep locks, the access to the critical section might be delayed because of the latencies due to OS-level awakening operations. Further, in both of these solutions there is no fine-grain self-tuning mechanism to optimize the mix of spin and sleep phases, with the inclusion of sleep to spin transitions, across different threads. All these limitations are tackled by our mutable locks, which provide a more articulated combination of spin and sleep phases based on a self-tuning mechanism that transparently adapts the lock behavior to the application workload.

Recently, a study that compares a wide range of lock implementations has been presented by Guerraoui et al.[22]. The study clearly shows that no single lock is systematically the best, and that choosing the best one on basis of the hardware and workload characteristics may be not simple. Also, the authors propose a practical selection procedure that can help the developer to choose a good locking algorithm depending on various factors, such as the number of cores/threads, the scheduler load and the lock contention level. As a concluding remark, the authors highlight that the observations from their study call for further research on optimized/dynamic lock algorithms, like the one we propose.

3 | MUTABLE LOCKS

Let us slide towards the description of our mutable locks through the help of an example where three threads running on different CPU-cores compete for accessing the same critical section. Figure 1 shows the same execution scenario with three different lock specifications. For simplicity, but with no loss of generality, we consider the case where all critical-section durations and the times required to wake-up and reschedule a thread are equal, each one requiring one time slot in the representation in Figure 1. The time-line at the bottom (marked as CS) shows the projection of all critical sections executed by threads T1, T2 and T3 on a single time axis, helping the reader to appreciate the overall critical section execution rate. Figure 1a) represents a possible execution resulting by using spin locks (e.g. a TTAS spin lock). As we can see, threads that have lost the challenge of acquiring the lock are always ready to immediately acquire it when it is released by another thread. This makes the critical sections executed at maximum rate (see the CS timeline). The drawback is that the CPU time in three time slots is "wasted" due to spin operations.

Figure 1b) shows the effects with sleep locks, where threads immediately go to sleep when the lock acquisition fails—this is the strategy adopted by the `pthread` mutex in the default configuration. A sleeping thread is woken up only upon the lock release operation by the thread that previously acquired the lock. Consequently, the awakening latency delays the access to the critical section. In this scenario, the critical section execution rate is significantly reduced. In fact, 5 time slots are required to execute 3 critical sections, with respect to 3 time slots required by the spin locks, thus leading to a slow down of 40%. On the other hand, only 2 slots of CPU time instead of 3 are wasted for thread awake and CPU-reschedule operations.

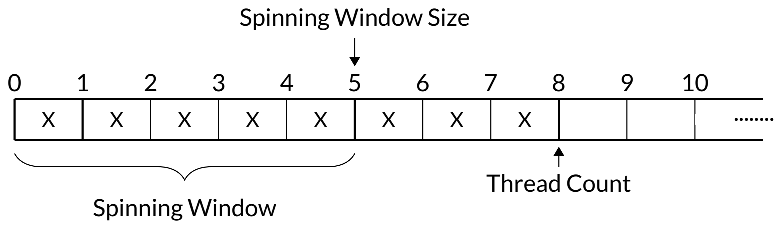


FIGURE 2 Logical representation of a lock with a spinning window

Finally, Figure 1c) shows an ideal scenario where we achieve the best of the two cases: the minimum amount of wasted slots as ensured by the sleep locks (2 slots), and the maximum execution rate of critical sections as provided by spin locks (3 time slot for executing 3 critical sections). This would be possible if we were able to decide which thread has to go to sleep and which one to spin, and to proactively start the wake-up phase of a sleeping thread to let it timely acquire the lock upon its release by another thread. In this scenario, the latency of awakening a thread (T_3) is masked by the critical section execution of the spinning thread (T_2). This behavior is the target of our mutable lock algorithm, which encapsulates the ability to mix spin and sleep phases in an optimized manner, with the inclusion of sleep to spin transitions which are driven by a self-tuning scheme.

3.1 | The Notion of Spinning Window

Mutable locks rely on the notion of *spinning window* (SW). SW identifies the set of threads, among those contending for the lock access, that are allowed to spin. The other contending threads (if any) need to undergo a sleep phase. The maximum cardinality of the set of spinning threads is bounded by the *spinning window size* (SWS). Thus, when a thread tries to acquire the lock, if the number of spinning threads is less than SWS then the thread starts spinning, otherwise it goes to sleep. An example representation is shown in Figure 2, where we have eight contending threads, five of which are spinning and three are sleeping. When a new thread T tries to acquire the lock, it takes the first available slot in the array and, according to its index i, it chooses if it has to spin or sleep. In particular:

$$\begin{cases} \mathrm{i} \in [0, \mathrm{SWS}) & T \text{ goes spinning} \\ \mathrm{i} \in [\mathrm{SWS}, +\infty) & T \text{ goes sleeping} \end{cases}$$

Upon the lock release operation, one random spinning thread accesses the critical section and one sleeping thread wakes up and takes the just freed slot of the SW. The latter is the sleep to spin transition we include in our mutable lock logic. Then, the array cell that is left empty outside the spinning window by the woken up thread will be occupied by shifting the threads associated with larger indexes. This complies with a specification that does not ensure a FIFO policy while serving threads—which can be instead obtained by left-shifting all threads in the array exactly by one position. It follows that this approach allows to control the exact number of spinning threads—including those transiting from the sleep to spin state—by manipulating the value of SWS.

Since we are interested in pursuing two goals, maximizing performance and reducing the waste of clock cycles caused by spin operations, SWS should adapt to the workload peculiarities and changes—e.g. to the duration of the critical section and the actual incidence of conflicts in its access. The larger SWS, the lower the access latency is—although if too many threads spin, we get CPU-interference on the one running the critical section—and, at the same time, more computational power is wasted due to spinning cycles. Conversely, a lower value of SWS tends to reduce clock cycles usage, but increases the probability that threads experience late wake ups, stretching the critical section access latency. Overall, a suited value for SWS should ensure that the number of spinning threads is low and the latency of awakening threads is masked by critical sections executed by other threads.

Dynamically adapting the value of SWS at runtime is not a trivial task since the newly chosen value must be correctly reflected onto the actual state of the threads (sleeping or spinning). In more detail, increasing the value of SWS with no other action could make one or more threads to be considered as falling within SW (thus spinning) even if they are currently sleeping (case 1). Consequently, no one will ever try to wake them up and they will sleep unboundedly unless SWS is eventually restored to the original value. On the other hand, reducing the value of SWS, which might lead some spinning thread to fall outside SW (case 2), does not hamper progress. However, it makes a number of threads larger than SWS spin for an unknown period of time, diverging from the desired behavior.

Cases 1 and 2 can occur only under specific conditions. Let Δ be the variation of the value of SWS to be applied at runtime, sws be the value of SWS before the variation is applied, and thc (thread count) be the number of threads waiting to access the lock. Case 1 can occur if and only if $\Delta > 0 \ \wedge \ \mathtt{thc} > \mathrm{sws}$ (C1) while case 2 can occur if and only if $\Delta < 0 \ \wedge \ \mathtt{thc} > \mathrm{sws} + \Delta$ (C2).

Algorithm 1 Mutable Lock Operations

```
procedure ACQUIRE(mutlock m)
    $\Delta \leftarrow 0$
    slept ← false
    $lstate^- \leftarrow$ FAD(m.lstate, +1)                ▷ Increase thread count
    $thc^- \leftarrow lstate^-$.thc
    sws ← $lstate^-$.sws
    if $thc^- \geq$ sws then                               ▷ Check room in the SW
        slept ← true
        m.slp_obj.sleep()
    end if
    spun ← m.spn_obj.lock()                               ▷ Take the lock
    $\Delta \leftarrow$ EvalSWS(spun, slept, m)            ▷ Consult the oracle
    if sws $\neq$ m.lstate.sws then                        ▷ SWS concurrently updated
        return                                            ▷ Ignore oracle and go to CS
    end if
    $\Delta \leftarrow sws + \Delta < 1\ ?\ sws - 1 : \Delta$
    $\Delta \leftarrow sws + \Delta >$ m.max ? m.max$-$sws : $\Delta$
    if $\Delta \neq 0$ then
        tmp ← ($\Delta \ll 32$);
        $lstate^- \leftarrow$ FAD(m.lstate, tmp)           ▷ Apply delta
        thc ← $lstate^-$.thc
        $sws^- \leftarrow lstate^-$.sws
        tmp ← $+\infty$
        sign ← $\Delta/|\Delta|$
        if sign $< 0 \land$ thc $> sws^+$ then              ▷ Condition C2
            tmp ← thc $-$ $sws^+$                          ▷ #spinning threads not in SW
        else if sign $> 0 \land$ thc $> sws^-$ then         ▷ Condition C1
            tmp ← thc $-$ $sws^-$                          ▷ #sleeping threads in SW
        else
            tmp ← 0                                       ▷ No other actions are required
        end if
        tmp ← sign $\cdot \min(|\Delta|, tmp)$
        m.wuc ← m.wuc + tmp                               ▷ Update wake-up counter
    end if
end procedure
```

```
procedure RELEASE(mutlock m)
    if m.wuc $\geq 0$ then                                 ▷ Multiple threads should enter the SW
        $R_{wuc} \leftarrow$ m.wuc
        m.wuc ← 0
    else                                                  ▷ Skip one wake up at a time
        $R_{wuc} \leftarrow -1$
        m.wuc ← m.wuc + 1
    end if
    $lstate^- \leftarrow$ FAD(m.lstate, -1)                ▷ Decrease thread count
    m.spn_obj.unlock()                                    ▷ Release lock
    if $R_{wuc} < 0$ then
        return
    end if
    $thc^- \leftarrow lstate^-$.thc
    sws ← $lstate^-$.sws
    if $thc^- >$ sws then                                  ▷ One thread should enter the SW
        $R_{wuc} \leftarrow R_{wuc} + 1$
    end if
    while $R_{wuc} > 0$ do                                 ▷ Make threads enter the SW
        cnt ← m.slp_obj.wake_up($R_{wuc}$)
        $R_{wuc} \leftarrow R_{wuc}$ $-$ cnt
    end while
end procedure
```

```
procedure EVALSWS(bool spun, bool slept, mutlock m)
    m.cnt ← m.cnt + 1                                     ▷ Count critical sections
    $\Delta \leftarrow 0$
    if slept $\land \neg$spun then                         ▷ Late wake-up event
        $\Delta \leftarrow$ m.sws                          ▷ Suggest to double the SWS
        m.cnt ← 0
    else if m.cnt = K then                                ▷ No late wake ups for K acquisitions
        $\Delta \leftarrow -1$                             ▷ Suggest to decrease the SWS by 1
        m.cnt ← 0
    end if
    return $\Delta$
end procedure
```

The SW specification tackles case 1 by waking up a number of sleeping threads equal to $\min(|\Delta|, \mathrm{thc} - \mathrm{sws})$ instead of 1 as in normal lock release operations. Case 2 is tackled by assigning to threads which are spinning outside the just shrinked SW higher priority in the access to SW with respect to threads that transit from the sleep to the spin state. This can be obtained by simply waking no thread up for a number of release phases equal to $\min(|\Delta|, \mathrm{thc} - (\mathrm{sws} + \Delta))$.

This *mutable lock* algorithm, described in Section 3.2, is de-facto a new thread synchronization algorithm grounded on the notion of locking primitive. At the same time, the change of SWS, in order to adapt its value to the workload, needs to be actuated via some other algorithm, which implements a kind of oracle for optimizing the runtime dynamics under mutable lock-based synchronization. Clearly, different oracles for adapting the SWS value at runtime could be devised, and we provide one of them in Section 3.2. In any case, the mutable lock algorithm is independent of the selected SWS adaptation oracle. This opens to the possibility of studying further variations of thread synchronization dynamics built around the notion of mutable lock.

3.2 | The Mutable Lock Algorithm

A mutable lock is a spin lock, denoted as `spn_obj`, plus other five variables: `sws`, which stores the current SWS; `thc`, which keeps the *thread count*, i.e. the total number of contending threads (including the one holding the lock); `wuc`, which keeps the *wake-up count*, i.e. number of threads to be woken up during a mutable lock release phase; `slp_obj`, which is an object that makes threads sleep until a certain condition becomes true (such as a semaphore or a memory structure managed via the `futex` Linux system call); `max`, which is the maximum SWS set to the number of cores.

In our design, `sws` and `thc` are 32 bits long and are stored in a unique 64-bit word, denoted as `lstate` (lock state), such that $\mathtt{lstate} = \langle \mathtt{sws}, \mathtt{thc} \rangle$. This arrangement allows threads to update one field, get its old value and retrieve the actual value of the other field at once by using an atomic `Fetch&Add` (FAD) machine instruction, commonly supported by off-the-shelf processors.

The operations used to acquire or release the mutable lock are shown in Algorithm 1. Let x be an atomic register (a variable) supporting atomic FAD operations, in our notation x^- and x^+ are the values of x respectively before and after a FAD execution on it. During an acquire phase, a thread

T increases thc via FAD (line A4) and checks whether there is room in SW. If the condition $\texttt{thc}^{-} \geq \texttt{sws}$ holds (line A7)—no room is available in SW—it goes to sleep on slp_obj (line A9). Otherwise it invokes the acquire API of spn_obj (line A11).

As soon as a thread owns the spn_obj, it determines if sws should be updated by invoking EvalSWS. This function, whose parameters will be explained later, implements the SWS adaptation oracle and returns the signed variation Δ to be applied to sws. A thread applies the adaptation suggested by the oracle if and only if the SWS has not been changed while it was sleeping or acquiring the spin lock (lines A13-A15). This prevents concurrent SWS adaptations from overlapping.

The SWS update is performed via FAD on the most 32 significant bits of the lstate field (line A20). Based on the values of Δ, thc, $\texttt{sws}^{-}$ and $\texttt{sws}^{+}$, we know that some countermeasures have to be taken in order to ensure progress of each thread and that the number of spinning threads will be bounded by sws eventually. In particular, if condition C1 occurs, we set the variable wuc to the number of additional threads to be woken up. Conversely, when condition C2 holds, wuc will be set to the number of threads in the spinning state, which are outside the SW, multiplied by -1. Finally, if none of the above conditions holds, no countermeasure is needed at all (line A30). At this point, the computed value will be simply added to wuc (line A33), and the lock acquire phase is completed.

Upon a lock release operation, if $\texttt{wuc} \geq 0$ holds (line R3), its value is copied into a local variable R_{wuc} and then is set to 0, otherwise (line R6) it is incremented by 1 and R_{wuc} is set to -1. Now, thc can be decremented by 1 via FAD and the spn_obj release API allows another thread to get the lock (lines R9 and R10). In order to complete the release operation, we have to ensure that the number of non-sleeping threads is compliant with the current value of sws. Thus, the releasing thread first checks if R_{wuc} is lower than 0. In this case, it can simply return since a previous reduction of sws has made some thread spinning outside SW and, consequently, no additional wake up is required. This is because sws updates and decrements of thc are performed in mutual exclusion (via FAD) and it is ensured that more than sws threads are spinning. If R_{wuc} is greater than or equal to 0, we need to check if an additional thread should be awakened in order to keep the number of spinning threads equal to the current value of sws. In this case (line R16), R_{wuc} is incremented by 1. Finally, the thread can awake R_{wuc} threads by relying on the slp_obj API (line R20). Thanks to these algorithms, the shared variables (thc, sws, wuc) used to keep the state of the lock are updated consistently without resorting to additional locks that could lead to other challenges such as choosing the proper lock implementation.

We designed a SWS adaptation oracle that targets the objective of keeping SWS close to the minimum value required to mask the wake-up latency of sleeping threads, so that threads can promptly access the critical section and the overall wasted CPU time for spinning is reduced. The oracle uses a strategy similar to the one adopted by the TCP congestion control mechanism[23], which is quite simple but effective in practice. It is implemented by the procedure EvalSWS in Algorithm 1. The oracle design is based on the following observation. When a sleeping thread A is awakened and there are no other threads executing the critical section, it means that SWS may be not large enough to mask the wake-up latency of A. In fact, since A was sleeping, it means that when A tried to acquire the lock there were sws threads spinning (i.e. the condition $\texttt{thc}^{-} \geq \text{sws}$ held true), and that when A was awakened all spinning threads had already terminated the execution of the critical section. When this occurs, the oracle doubles SWS. Conversely, if this condition does not occur for K consecutive critical section executions, the oracle tries to decrease SWS by 1, where K is a configuration parameter. The late wake-up event can be detected very efficiently using two flags, slept and spun, which are updated during the acquisition phase. The first one indicates whether or not the thread has gone to sleep, and the second one indicates whether or not the thread has found the lock owned by another thread. To compute the spun value, the lock procedure (line A11 of Algorithm 1) checks if the lock is not free before entering the spin phase, and returns true in the positive case. We note that this check is required only by our adaptation oracle, not by the basic mutable lock algorithm.

Consider a scenario where, along a give phase of the execution, $V + 1$ is the minimum value of SWS that ensures no late wake up, and when SWS is set to V then late wake ups occur. We can show that our oracle keeps the percentage of late wake-up events bounded by a function of K. We know that the oracle doubles SWS as soon as a late wake up occurs, i.e. when SWS reaches the value V. After this event, SWS is set to 2V, followed by a phase where the oracle starts reducing SWS by 1 every K critical section completions. This phase ends when SWS reaches again the value V. Consequently, SWS repeatedly varies from 2V to V. Since K critical sections complete for each value of SWS in $[V + 1, 2V]$ and 1 critical section completes when SWS is equal to V, we have that $K \cdot \big|[V + 1, 2V]\big| + 1$ critical sections complete, and one late wake up occurs. Hence, the number of late wake-up events over the total number of critical section completions is:

$$\frac{1}{K \cdot \big|[V+1, 2V]\big| + 1} = \frac{1}{KV + 1}.$$

This number reaches the maximum value when $V = 1$. Thus, the upper bound of the number of late wake-up events over the number of critical section completions is $1/(K + 1)$. In conclusion, the value of K can be selected on the basis of the desired maximum percentage of late wake-up events.

4 | EXPERIMENTAL STUDY

In this section, we present the results of an experimental study where we evaluate our mutable locks and compare them with other lock implementations under various workloads. In the first part of this study, we focus on a behavioral analysis, showing experimental results that we achieved with a synthetic benchmark, configured to give rise to different and antipodal contention scenarios. In particular, these results show how the mutable lock takes advantage from its ability to dynamically adapt between sleeping and spinning phases compared to most common lock implementation strategies, i.e. an always-spinning strategy (queue based or not), an always-sleeping strategy, and an adaptive spin-then-sleep strategy. In the second part of the experimental study, we present an extended comparative analysis we carried out with additional benchmarks and a larger set of state-of-the-art lock implementations. All the test-bed applications we used in our experiments are written for Linux using the C/C++ language, and are compiled with GCC 8.2.1. The implementation of our mutable lock [‡] (denoted as `mutlock`) uses a traditional TTAS spin lock as `spn_obj` and a semaphore as sleeping object. To carry out the experimental comparison with the other lock implementations, we used LiTL[22], an open-source POSIX compliant library that allows executing programs that use `pthread` mutex locks also with different lock implementations, and we integrated our mutable lock implementation in LiTL. In our experiments, we set the parameter K of the oracle equal to 10, to keep the expected percentage of late wake-up events below 10%.

4.1 | Mutable Locks Behavioral Analysis

We used a synthetic benchmark, that we name LOCKBENCH[§], which can run an arbitrary number of threads, each one repeatedly executing a critical section protected by a lock, followed by a non-critical section. The length CS of the critical section and the length NCS of the non-critical section are uniformly distributed within the two parametric intervals $[\mathrm{CSL}, \mathrm{CSU})$ and $[\mathrm{NCSL}, \mathrm{NCSU})$, respectively. We show the results achieved with `mutlock` and with four commonly used lock implementations, specifically a TTAS spin lock (that we denote as `ttas`), the `pthread` mutex with the default configuration (denoted as `pt-mutex`), the `pthread` mutex with the adaptive configuration (denoted as `pt-adaptive`), and the MCS lock (denoted as `mcs`). The description of these lock implementations has been given in Section 2. We ran the experiments on a ThinkMate GPX XT10-2260V4-4GPU machine, equipped with 2 Intel Xeon E5-2640 v4, 20 physical cores total, 64GB memory, arranged in 2 NUMA nodes, and CentOS Linux 7.6. As in other comparative experimental studies on locks, we disabled Intel Hyperthreading.

We ran the synthetic benchmark with four different combinations of the intervals $[\mathrm{CSL}, \mathrm{CSU})$ and $[\mathrm{NCSL}, \mathrm{NCSU})$ in order to analyze complementary contention scenarios. We selected the ranges of values of these intervals on the basis of the minimum and maximum execution times, as we observed on the machine we used in our experiment, of typical operations executed on common data structures (such as the one offered by Synchrobench[24]) protected by locks, while varying the data structure size. Then, we selected the interval for the length of the non-critical section in order to mimic different ratios with respect to the length of the critical section. Finally, we varied the number of threads between 1 and 40.

We considered two basic metrics, i.e. the throughput and the CPU time required for synchronization. The former is measured as the number of critical sections executed per second. The latter is the total time spent by the CPU to execute the code required to synchronize the accesses to critical sections. In other words, it represents the cost, in terms of wasted CPU time, spent for thread synchronization.

We show the results in Figure 3. The left column shows the throughput, while the right column shows the CPU time spent for synchronization. Figure 3(a) shows the throughput when both the intervals $[\mathrm{CSL}, \mathrm{CSU})$ and $[\mathrm{NCSL}, \mathrm{NCSU})$ are set to $[0\mu s, 3.7\mu s)$. When the thread count is lower than (or equal to) the number of cores (i.e. 20), `mcs` has the highest throughput. This result is expected because the `mcs` design best fits the NUMA arrangement of memory. In fact, each thread spins on its own cache line and the thread holding the lock touches a single line (the one owned by the next thread in the FIFO queue) for signaling the lock release. `mutlock` has slightly lower throughput than `ttas`, showing up to 8% of overhead for its management. Conversely, `pt-mutex` (`pt-adaptive`) gives rise to 25% (12%) drop in performance and shows its benefits only in case of time-sharing (i.e. when the number of threads is larger than the number of cores), where going to sleep is a smart choice to reduce hardware contention. However, for the time-sharing scenario, `mutlock` is superior to all the other solutions, especially when observing the average behavior across different thread counts. Also, thanks to its self-tuning strategy—that provides at any time a suited combination between the number of spinning and sleeping threads—it definitely contrasts the excessive waste of CPU time required for synchronization with `mcs`, which also shows a drop in performance with time-sharing due to its FIFO semantic (see Figure 3(b)).

In the second set of experiments we varied the upper bound of the critical-section length, setting it to $366\mu s$. Essentially, we incremented the size of the critical section by increasing CSU by a factor of 100, thus moving to a scenario with high contention. The throughput results, presented in Figure 3(c), show that spinning (potentially for a very long time) is convenient only with low thread counts (up to 4). Conversely, `pt-mutex` and `pt-adaptive` show their advantages having a maximum and stable throughput with thread count larger than 4. `mutlock` maintains a stable

[‡] The source code is publicly available at https://github.com/HPDCS/libmutlock

[§] The source code is publicly available at https://github.com/HPDCS/lockbench

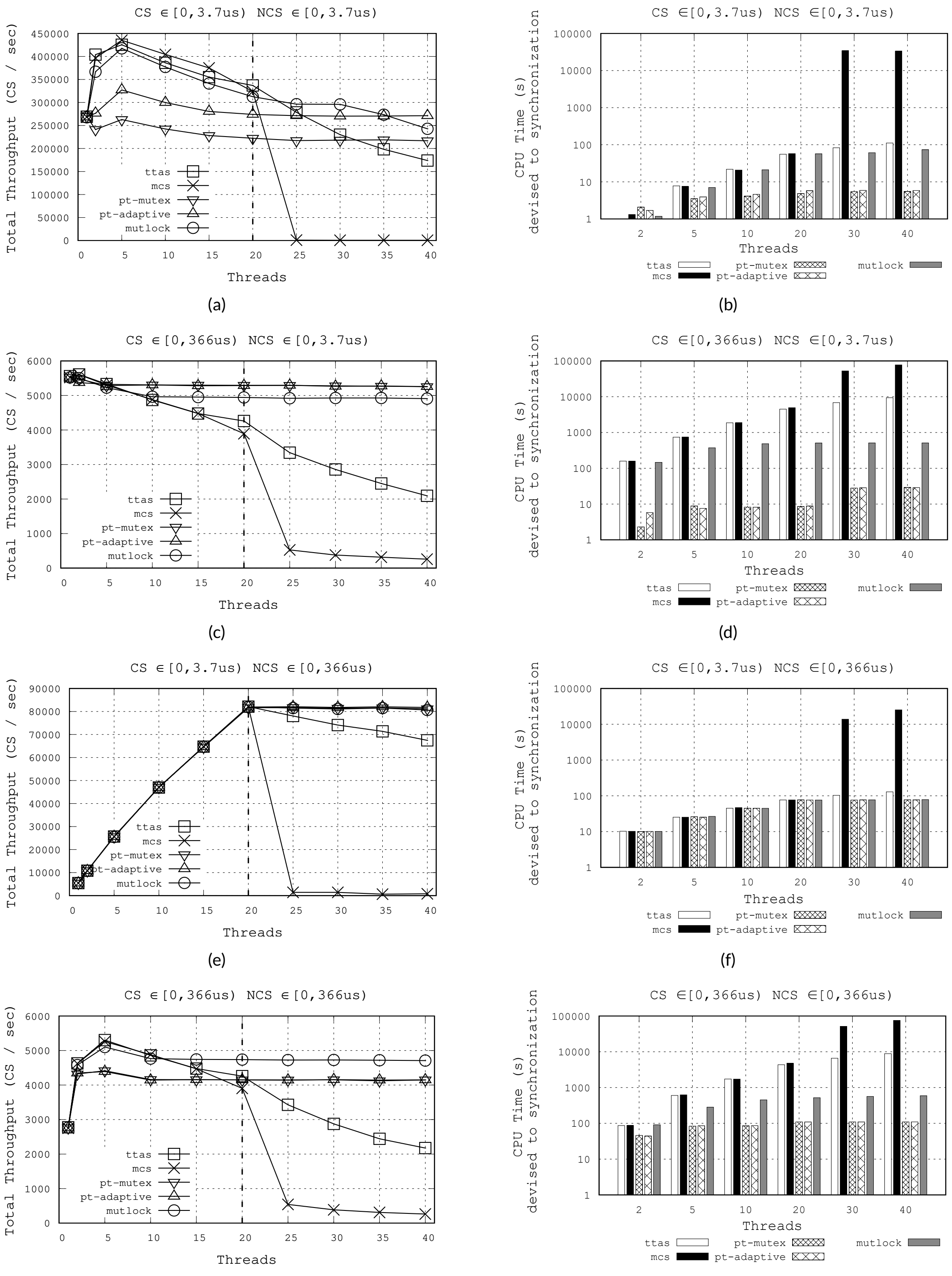


FIGURE 3 Tests consist in repeatedly executing critical and non-critical sections, whose lengths are uniformly distributed in the interval [CSL, CSU) and [NCSL, NCSU) respectively. Left charts show throughput (the higher the better), while right charts show CPU time spent in synchronization (the lower the better)—`pt-exp` refers to the mean of `ttas` and `pt-mutex` values.

throughput, a bit lower than pt-mutex and pt-adaptive, since it continues to keep a few threads spinning to mask the wake up latency. However, since critical sections are very long, such latencies are negligible and going to sleep allows to reduce hardware contention. Finally, mutlock reduces the CPU time spent for synchronization of an order of magnitude w.r.t. spin locks for large thread counts (beyond 10) as shown in Figure 3(d).

In the third set of experiments, we consider a short critical section and a long non-critical section. Specifically, $[\mathrm{CSL}, \mathrm{CSU})$ is set to $[0\mu\mathrm{s}, 3.7\mu\mathrm{s})$ and $[\mathrm{NCSL}, \mathrm{NCSU})$ is set to $[0\mu\mathrm{s}, 366\mu\mathrm{s})$, thus leading to very low lock contention. In such a scenario, all lock implementations have generally similar performance (see the throughput curves in Figure 3(e)) and CPU times (see Figure 3(f)), except for the time-shared case with ttas, where the throughput drops down, and the configuration with 40 threads and mcs, where the CPU time notably increases.

The last case we investigated, namely the one with both critical section and non-critical section uniformly distributed in $[0\mu\mathrm{s}, 366\mu\mathrm{s})$, represents a scenario where critical sections are very long, but moderately accessed. This reduces the differences between the pure spinning strategy (see ttas) and mutlock, as shown by the throughput results in Figure 3(g). On the other hand, it exacerbates the benefits of making a controlled number of threads spinning in order to save CPU time, reducing hardware contention and not paying wake up latencies. Clearly, the higher throughput of mutlock over mutexes comes at a price in terms of CPU time spent for synchronization (see Figure 3(h)). However, mutlock still has two orders of magnitude lower waste of CPU time compared to mcs, and one order of magnitude lower waste of CPU time compared to ttas.

To carry out an overall evaluation of the mutable lock behavior, we analyzed the observed results also from another perspective. Let us assume to have an ideal lock implementation (that we denote as best_lock) that for each thread count is able to automatically select the best lock implementation among all those of our tests. We calculated, for each lock implementation, its average throughput over a given range of thread counts, and then we calculated the ratio between it and the average throughput of best_lock over the same range. Such a metric gives a measure of how much the behavior of a lock implementation is close to best_lock. We calculated this metric for the four different configurations of LOCKBENCH and for three different ranges of thread counts: a) *no time-sharing* (namely with thread count between 1 and 20), b) *time-sharing* (with thread count between 21 and 40), and c) *overall* (with thread count between 1 and 40). Figure 4 reports the results. The value 1 represents the best performance achievable by a lock implementation, i.e. its throughput is the same as best_lock. By observing the overall behavior, in 6 out of 12 cases the performance of mutlock is not less than 99% compared to best_lock, and in the other 6 cases it is between 93% and 97% compared to best_lock. Accordingly, in the worst case (the case of no time-sharing in Figure 4(a)) mutlock achieves 93% of the performance of best_lock. None of the other lock implementations shows better overall results. In fact, among them, the best one is pt-adaptive. Compared to best_lock, its performance is no less than 99% in 6 out of 12 cases, below 90% in 4 cases, and below 80% in the other two cases.

In Figure 4 we also add a bar, denoted as pt-exp, which is the mean between the values of ttas and pt-mutex. When the pt-exp bar appears lower than mutlock, it means that if we have no any a-priori knowledge about the performance achievable with a given workload using ttas or pt-mutex (so we might have chosen one of the two with equal probability), mutlock represents a better choice. Compared to pt-exp, mutlock shows better performance in 11 out of 12 tested scenarios, with a maximum gain of about 23%. This makes mutlock a good candidate when operating with uncertain conditions because of an unpredictable (or difficult to be reasoned) workload and/or possible interference by virtualized hardware on, e.g., the length of critical sections.

4.2 | Extended Comparison with Alternative Solutions

In this section, we present an extended experimental study of our mutable locks. The objective of this study is to carry out an overall evaluation of our solution in terms of achievable performance compared to a wider range of lock implementations taken from the literature, and with respect to different workloads. To this aim, in addition to the lock implementations that we used in the behavioral analysis in Section 4.1, we include recent state-of-the-art lock proposals that, similarly to mutable locks, have been designed to take advantage of the combination of spinning and sleeping. Specifically, in this study we include: 1) MCS with the *spin-then-sleep* policy (denoted as mcs-sts)[22], i.e. an MCS lock where spinning threads go to sleep after a given time, 2) Malthusian locks (denoted as malthusian) and 3) Mutexee (denoted as mutexee). The details about these lock proposals can be found in Section 2. We used the implementations offered by LiTL. As suggested by the authors of LiTL, we set the variable SPINNING_THRESHOLD (that establishes the time after which a spinning thread transits to the sleeping phase) based on the duration of a round-trip context switch[14], which we measured using LMBench[25]. We ran all the experiments of this study on an AWS EC2 m5.metal machine, equipped with 2 Intel Xeon Platinum 8260 Processor, 48 physical cores total, 384GB memory, arranged in 2 NUMA nodes, and Ubuntu Server 18.04 LTS.

In this study, we extended the set of benchmarks including, in addition to our synthetic benchmark LOCKBENCH, two largely used micro-benchmarks taken from Synchrobench[24], and a well know high-performance key/value database that makes an extensive use of locks for thread synchronization, i.e. Upscaledb[26]. Synchrobench is a micro-benchmark suite designed to evaluate synchronization techniques on common data structures. Among the data structure implementations offered by Synchrobench, we selected the ones designed for the C language which use locks as synchronization construct, i.e. a hashtable (that we denote as HASHTABLE) and a linked list (LAZY LIST). As input parameters, we used the default

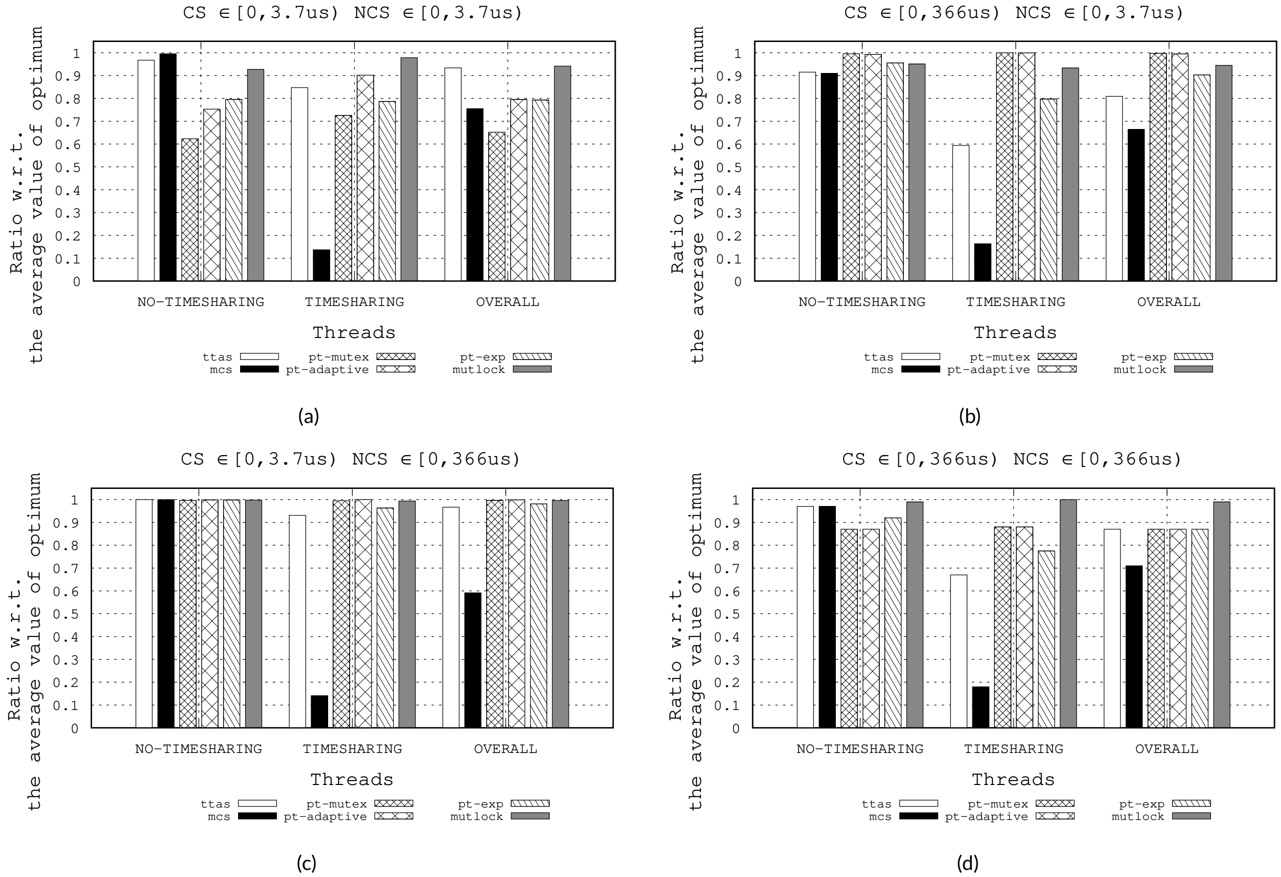


FIGURE 4 Bars represent the ratio between the average throughput of a given lock implementation and the average throughput of the lock implementation offering the best performance for each thread count (the higher the better)— `pt-exp` refers to the mean of `ttas` and `pt-mutex` values.

values as defined in their source code release [¶]. We used four different values for the parameter *initial size*, i.e. 4096, 8192, 16386 and 32768, respectively. This parameter establishes the initial size of the data structure, thus affecting (in inverse proportion) the lock contention level. We set the parameter *range* equal to the value 65536. As for Upscaledb [26] (that we denote as UPSCALEDB), we ran Ups_Bench [27], using four different configurations of the mix of *insert/find/remove* operations, i.e. 40%/20%/40%, 30%/40%/30%, 20%/60%/20% and 10%/80%/10%, respectively. Such a set of configurations generate very different workload profiles. As for the other input parameters, we used the default values of the benchmark, and set the *inmemorydb* option active.

For each benchmark, we ran the experiments varying the thread count between 1 and 96 (we remark that we used a machine with a total of 48 physical cores). Again, we consider three different ranges of thread counts: a) *no time-sharing* (thread count between 1 and 48), b) *time-sharing* (thread count between 48 and 96), and c) *overall* (thread count between 1 and 96). The results are shown in Figure 5 - 8. Each bar represents the ratio between the average throughput of a given lock implementation and the average throughput of `mutlock` over the specific thread count range.

The results with LOCKBENCH (Figure 5) in the case of *no time-sharing* show that there are configurations where some lock implementations are subject to a non-minimal performance loss compared to `mutlock`. In particular, this happens for the first configuration (left-most one in the chart) with `mcs-sts`, `malthusian`, `pt-adaptive` and `mutexee`. As for the other lock implementations, there is no relevant difference compared to `mutlock`. In any case, it is possible to notice that, for one configuration `mutlock` is the best performing one, and for the other three configurations its performance is very close to the best one. In the worst case, the performance of `mutlock` is exceeded only by `mcs` of no more than 7,6%. In the case of *time-sharing*, a noticeable performance loss with respect to `mutlock` is experienced by various lock implementations, which is more evident for some configurations with `mcs`, `mcs-sts` and `malthusian`. `mutlock` appears not to be affected by time-sharing

[¶] Available at https://github.com/gramoli/synchrobench

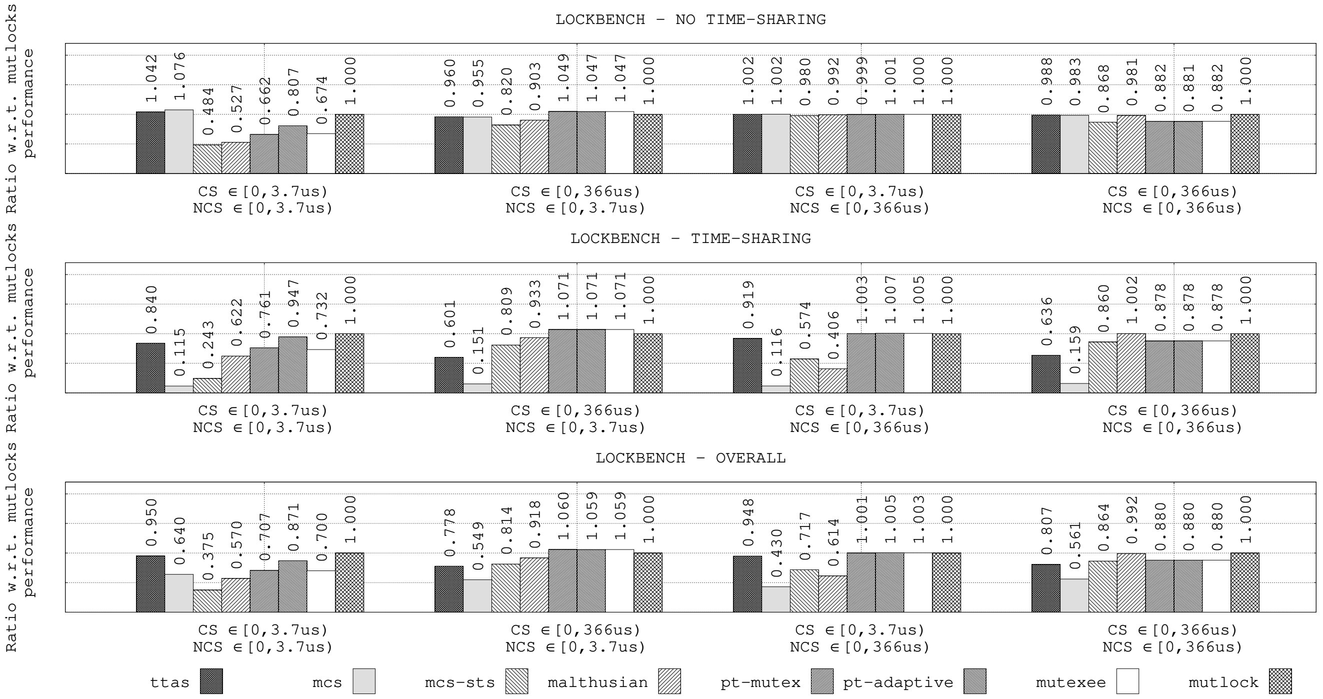


FIGURE 5 Performance comparison of `mutlock` and other lock implementations with LOCKBENCH. Bars represent the ratio between the average throughput of a given lock implementation and the average throughput of mutable locks (the higher the better).

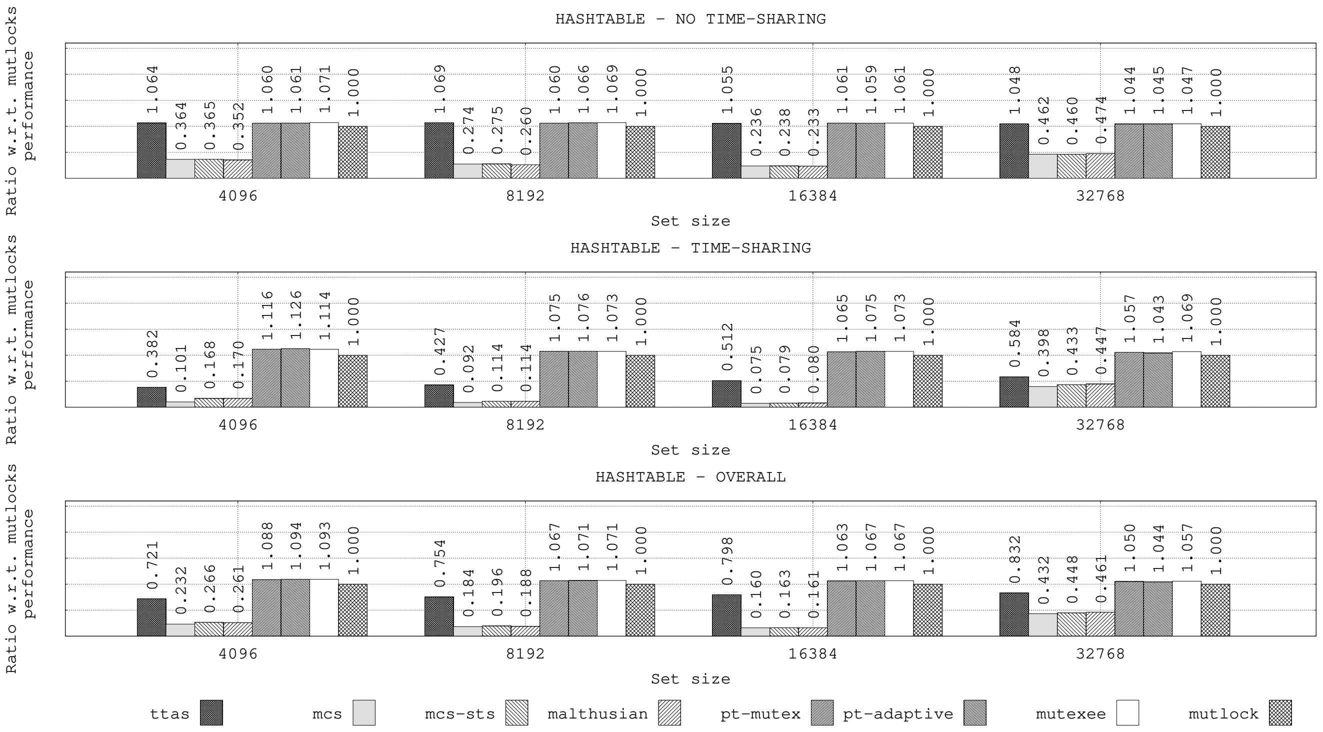


FIGURE 6 Performance comparison of `mutlock` and other lock implementations with HASHTABLE (Synchrobench). Bars represent the ratio between the average throughput of a given lock implementation and the average throughput of mutable locks (the higher the better).

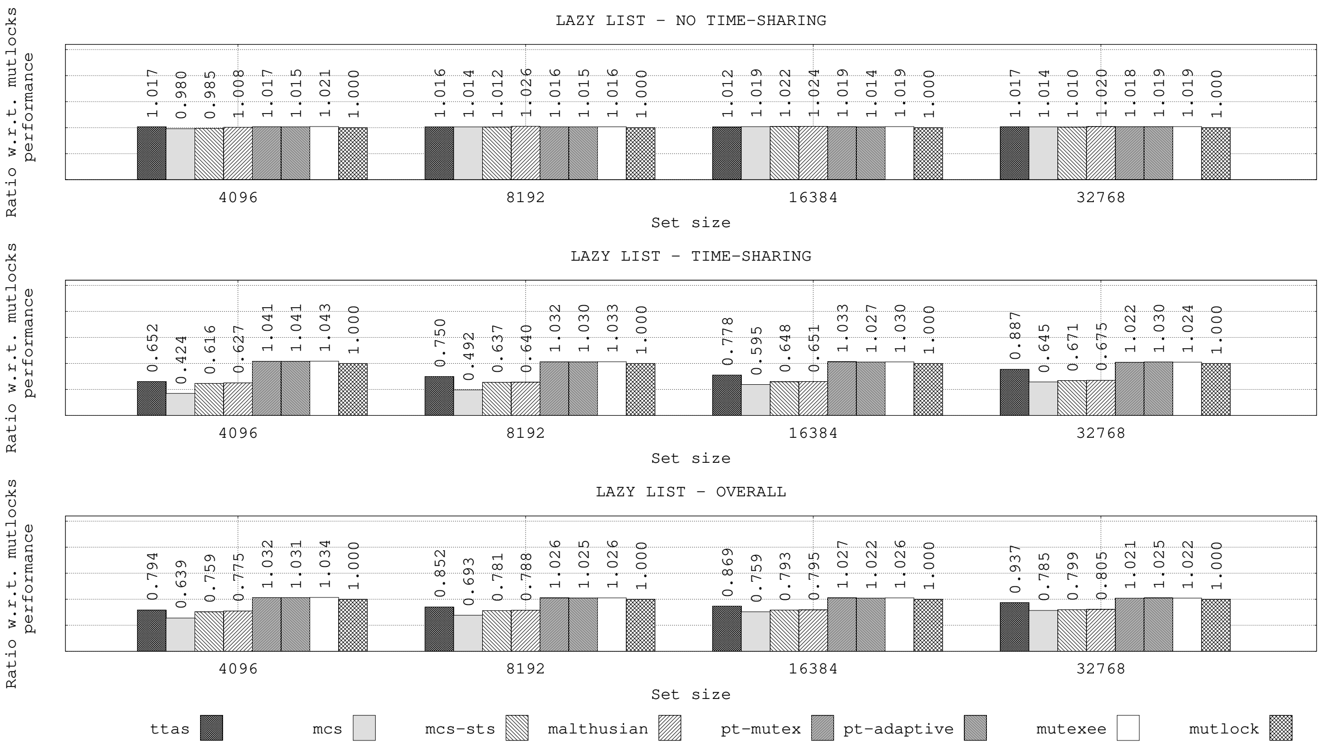


FIGURE 7 Performance comparison of `mutlock` and other lock implementations with LAZY-LIST (Synchrobench). Bars represent the ratio between the average throughput of a given lock implementation and the average throughput of mutable locks (the higher the better).

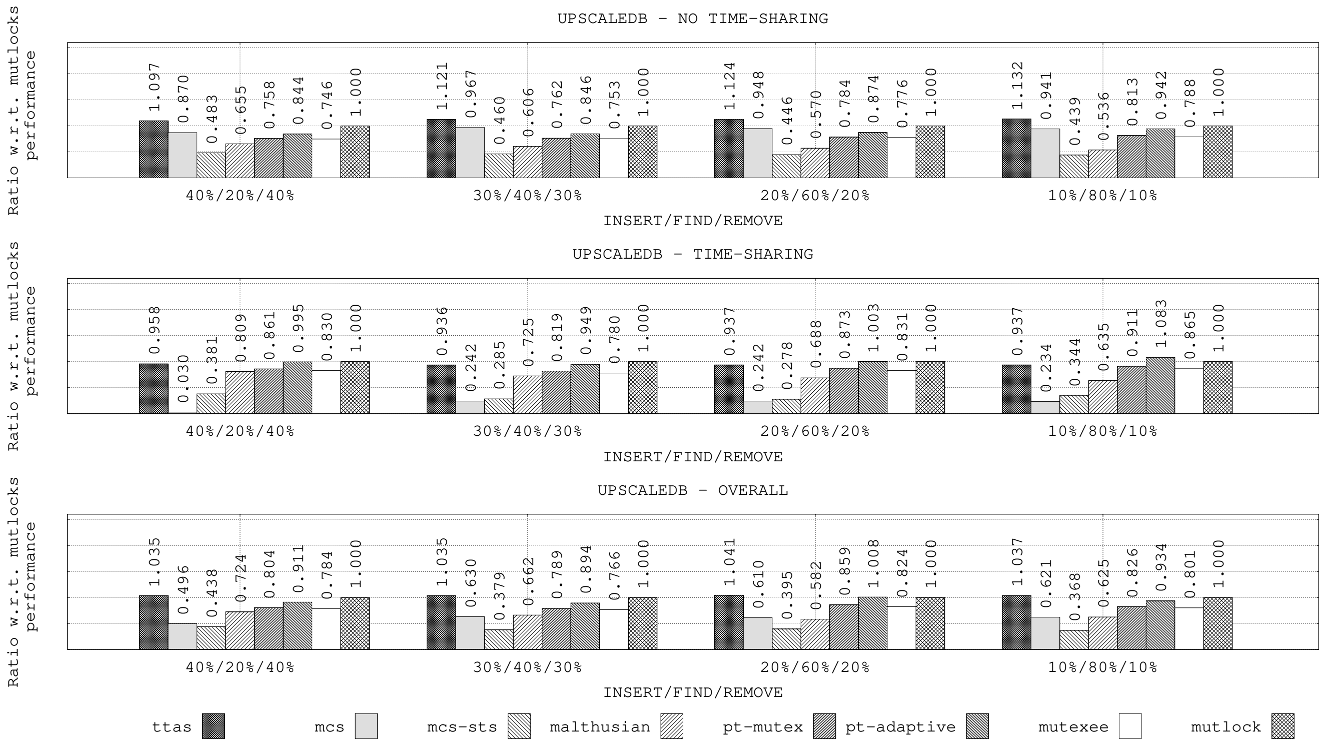


FIGURE 8 Performance comparison of `mutlock` and other lock implementations with UPSCALEDB. Bars represent the ratio between the average throughput of a given lock implementation and the average throughput of mutable locks (the higher the better).

TABLE 1 Average values of the ratio between the average throughput of a given lock implementation and the average throughput of mutable locks. The average is calculated over the four configurations of each benchmark. The different colors help the reader to immediately identify the favorable and unfavorable cases for `mutlock`, as indicated in the legend.

	NO TIME-SHARING				TIME-SHARING				OVERALL			
	Lock bench	Hash table	Lazy List	Upscale DB	Lock bench	Hash table	Lazy List	Upscale DB	Lock bench	Hash table	Lazy List	Upscale DB
TTAS	1.00	1.06	1.02	1.12	0.75	0.48	0.77	0.94	0.87	0.78	0.86	1.04
MCS	1.00	0.33	1.01	0.93	0.14	0.17	0.54	0.19	0.54	0.53	0.44	0.59
MCS-STS	0.79	0.68	0.66	0.46	0.62	0.46	0.38	0.32	0.69	0.55	0.50	0.39
MALTHUSIAN	0.85	0.69	0.66	0.59	0.74	0.46	0.39	0.71	0.77	0.56	0.50	0.65
PT-MUTEX	0.90	1.04	1.04	0.78	0.93	1.04	1.07	0.87	0.91	1.04	1.05	0.82
PT-ADAPTIVE	0.93	1.03	1.04	0.88	0.98	1.04	1.07	1.01	0.95	1.04	1.06	0.94
MUTEXEE	0.90	1.04	1.04	0.77	0.92	1.05	1.07	0.83	0.91	1.04	1.06	0.79

Legend	$(0, 0.90]$	$(0.90, 0.95]$	$(0.95, 1.05)$	$[1.05, 1.10)$	$[1.10, +\infty)$

in any configuration, unlike most of the other lock implementations. The scenario *overall* demonstrates that `mutlock` is the best performing one for 2 out of 4 configurations (the first one and the last one in the graph). For the configuration with minimal contention level (the third one), the difference between `mutlock` and the best performing one (i.e. `pt-adaptive`) is negligible (0.5%). For the second configuration, the performance of the best lock implementation (i.e. `pt-mutex`) oversteps the performance of `mutlock` by only 6%.

The results with HASHTABLE (Figure 6) show that, independently of whether time-sharing is used or not, it is possible to clearly distinguish a group of lock implementations that generally show good performance for all the benchmark configurations, showing values that are always quite close to 1. This group includes `mutlock`, `pt-mutex`, `pt-adaptive` and `mutexee`. Conversely, the other lock implementations suffer, at least in one case, from a significant performance degradation. The case *overall* confirms that `mutlock` falls in the group of the best performing solutions. In the worst case, `mutlock` is surpassed by a performance surplus of 9.4% achieved by `pt-adaptive`.

The results with LAZY LIST (Figure 7) and *no time-sharing* show that there are no relevant performance differences between the various lock implementations. This is mainly caused by the reduced contention level of this benchmark due to the optimistic lock acquisition, which makes threads compete only for modifying the data structure and not for traversing it. On the other hand, in the *time-sharing* case, the performance of half of them is remarkably affected. The set of lock implementations that appear more robust against the thread scheduling/descheduling dynamics caused by time-sharing, and the consequent lock contention dynamics, includes again `mutlock`, `pt-mutex`, `pt-adaptive` and `mutexee`. The case *overall* shows that `mutlock`, in the worst case, is surpassed only by `mutexee` by no more than 3.4%.

Finally, the results with UPSCALEDB (Figure 8) show that the various lock implementations behave quite differently. With *no time-sharing*, only `ttas` offers better results than `mutlock`, with a performance improvement of about 10-12%. However, `mutlock` clearly provides higher performance than all the other lock implementations for all the configurations. With *time-sharing*, the performance of `ttas` falls below the performance of `mutlock`, and the performance of `mcs` noticeable drops down. For 2 out of 4 configurations, `mutlock` achieves the best performance, and for the other two configurations it is surpassed only by `pt-adaptive`. However, the difference between `pt-adaptive` and `mutlock` is non-minimal only with one configuration, while being still bounded by 8.3%. The case *overall* shows that `mutlock` is surpassed only by `ttas`, with a maximum performance difference limited to 4.1%, and by `pt-adaptive` only for one configuration with a difference of about 0.8%. None of the other lock implementations shows better results than `mutlock`.

To carry out an overall assessment of the results of our extended experimental study, we report in Table 1 the average values calculated over the four configurations of each benchmark. The cells of the table are marked with different colors that allow to easily identify the favorable and unfavorable cases for `mutlock`. Green cells identify the favorable cases for `mutlock`, i.e. cases where the average ratio between the throughput of a given lock implementation and the throughput of `mutlock` is below 0.9 (dark green cells) or in the interval $(0.9, 0.95]$ (light green cells). White cells identify the cases where the performance difference is less relevant, i.e. cases where the average ratio is in the interval $(0.95, 1.05)$. Finally, red cells identify the unfavorable cases for `mutlock`, i.e. cases where the average ratio is in the interval $[1.05, 1.1)$ (light red cells) or larger than 1.1 (dark red cells).

By observing the cell colors in Table 1, it is possible to easily notice that the majority of cells is marked as dark green. This holds true for the case of *no time-sharing* and also *time-sharing*, and, consequently, *overall*. Furthermore, various values of these cells show that `mutlock` achieves an average throughput which is 2 times (up to 7 times in the best case) higher than the other lock implementations. In a minority of cases the cells are marked as white. A smaller number of cells are marked as light green or light red. More important, only one cell is marked as dark red, i.e. with `ttas` and UPSCALEDB in the *no time-sharing* case, which is the worst case for `mutlock`. However, we note that the performance improvement of `ttas` is limited to 12% in this case, and it is reduced to 4% when considering the case *overall* of the same benchmark.

Overall, the results of our experimental study show that mutable locks can offer noticeable performance advantages with various workloads compared to several state-of-the-art lock implementations. At the same time, they show that, with workloads for which some other lock implementations offer better performance, mutable locks are generally more resilient to performance degradation compared to the other lock implementations, and achieve results which are close to the best one. We believe that these advantages come from the novel way introduced by mutable locks to combine sleeping and spinning phases with respect to the other solutions, and from the possibility offered by mutable locks to adapt the size of the spinning window depending on the workload. We remark that the spinning window adaptation strategy is decoupled from the mutable lock implementation, and is freely customizable. This paves the way for future work to improve mutable locks through the exploration of alternative adaptation strategies.

5 | SUMMARY

In this article, we presented a synchronization construct called mutable lock, a locking mechanism that uses a new approach to combine spin and sleep locks. It is based on the concept of spinning window, that allows to adaptively control the number of threads enabled to spin before accessing a critical section to guarantee responsiveness. We showed the potential advantages of this new approach through experimental data achieved with workloads which give rise to different lock contention scenarios. By the results, mutable locks can ensure better performance than other state-of-the-art lock implementations with different workloads and a reduced performance loss in the most adverse cases. Also, they allow a significant reduction of the waste of CPU usage compared to spin-lock based solutions. All these benefits are achieved with no intervention by the application programmer, also thanks to the possibility offered by mutable locks to adapt the spinning window to the workload. Finally, the possibility to customize the adaptation of the spinning window offers further possibilities for additional investigations.

References

1. Held J. "Single-chip Cloud Computer", an IA Tera-scale Research Processor. In: Guarracino MR, Vivien F, Träff JL, et al., eds. *Euro-Par 2010 Parallel Processing Workshops*Springer Berlin Heidelberg; 2011; Berlin, Heidelberg: 85–85.
2. Dice D, Shalev O, Shavit N. Transactional Locking II. In: DISC'06. Springer-Verlag; 2006; Berlin, Heidelberg: 194–208
3. McKenney PE, Slingwine JD. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In: ; 1998; Las Vegas, NV: 509-518.
4. Dijkstra EW. Solution of a Problem in Concurrent Programming Control. *Commun. ACM* 1965; 8(9): 569–. doi: 10.1145/365559.365617
5. Lamport L. A New Solution of Dijkstra's Concurrent Programming Problem. *Commun. ACM* 1974; 17(8): 453–455. doi: 10.1145/361082.361093
6. Kruskal CP, Rudolph L, Snir M. Efficient Synchronization of Multiprocessors with Shared Memory. *ACM Trans. Program. Lang. Syst.* 1988; 10(4): 579–601. doi: 10.1145/48022.48024
7. Rudolph L, Segall Z. Dynamic Decentralized Cache Schemes for Mimd Parallel Processors. In: ISCA '84. Association for Computing Machinery; 1984; New York, NY, USA: 340-347
8. Anderson TE. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Trans. Parallel Distrib. Syst.* 1990; 1(1): 6–16. doi: 10.1109/71.80120
9. Scott ML. Shared-Memory Synchronization. *Synthesis Lectures on Computer Architecture* 2013; 8(2): 1–221. doi: 10.2200/S00499ED1V01Y201304CAC023
10. Mellor-Crummey JM, Scott ML. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.* 1991; 9(1): 21–65. doi: 10.1145/103727.103729
11. IEEE , The Open Group . POSIX.1-2017. https://pubs.opengroup.org/onlinepubs/9699919799/; 2018.
12. Microsoft . Windows API Index. https://docs.microsoft.com/en-us/windows/win32/apiindex; 2019.

13. Free Software Foundation . GNU C Library. https://www.gnu.org/software/libc/; 2019.

14. Karlin AR, Li K, Manasse MS, Owicki S. Empirical Studies of Competitve Spinning for a Shared-memory Multiprocessor. *SIGOPS Oper. Syst. Rev.* 1991; 25(5): 41–55. doi: 10.1145/121133.286599

15. Lim BH, Agarwal A. Waiting Algorithms for Synchronization in Large-scale Multiprocessors. *ACM Trans. Comput. Syst.* 1993; 11(3): 253–294. doi: 10.1145/152864.152869

16. Lim BH, Agarwal A. Reactive Synchronization Algorithms for Multiprocessors. *SIGPLAN Not.* 1994; 29(11): 25–35. doi: 10.1145/195470.195490

17. Eastep J, Wingate D, Santambrogio MD, Agarwal A. Smartlocks: Lock Acquisition Scheduling for Self-aware Synchronization. In: ICAC '10. ACM; 2010; New York, NY, USA: 215–224

18. Antić J, Chatzopoulos G, Guerraoui R, Trigonakis V. Locking Made Easy. In: Middleware '16. ACM; 2016; New York, NY, USA: 20:1–20:14

19. Johnson FR, Stoica R, Ailamaki A, Mowry TC. Decoupling Contention Management from Scheduling. *SIGARCH Comput. Archit. News* 2010; 38(1): 117–128. doi: 10.1145/1735970.1736035

20. Falsafi B, Guerraoui R, Picorel J, Trigonakis V. Unlocking Energy. In: USENIX Association; 2016; Denver, CO: 393–406.

21. Dice D. Malthusian Locks. In: EuroSys '17. ACM; 2017; New York, NY, USA: 314–327

22. Guerraoui R, Guiroux H, Lachaize R, Quéma V, Trigonakis V. Lock-Unlock: Is That All? A Pragmatic Analysis of Locking in Software Systems. *ACM Trans. Comput. Syst.* 2019; 36(1). doi: 10.1145/3301501

23. Jacobson V. Congestion Avoidance and Control. In: SIGCOMM '88. ACM; 1988; New York, NY, USA: 314–329

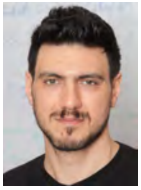
24. Gramoli V. More Than You Ever Wanted to Know About Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms. In: PPoPP 2015. ACM; 2015; New York, NY, USA: 1–10

25. McVoy L, Staelin C. Lmbench: Portable Tools for Performance Analysis. In: ATEC '96. USENIX Association; 1996; Berkeley, CA, USA: 23–23.

26. Rupp C. Upscaledb. https://upscaledb.com; 2019. Referenced November 13, 2019.

27. Rupp C. Ups Benchmark. https://upscaledb.com/benchmarking.html; 2019. Referenced November 13, 2019.

AUTHOR BIOGRAPHY



Romolo Marotta received the Bachelor's degree, the Master's degree and a PhD in Computer Engineering at Sapienza, University of Rome in Italy. He is currently a member of the High Performance and Dependable Computing Systems research group at the same institution. His research activities mainly focus on parallel and concurrent programming.

Davide Tiriticco received the BSc degree in Engineering in Computer Science at Sapienza, University of Rome in 2014 and he is currently an MSc student in Engineering in Computer Science. He joined Advanced Computer Systems ACS S.r.l from 2013 to 2018 and since 2019 he joined the Aerospace & Defense department at Exprivia S.p.A. His main areas of research interests are High Performance Computing applications and Parallel and Concurrent Computing.

Pierangelo Di Sanzo received a M.S. degree and a Ph.D. degree in Computer Engineering from Sapienza University of Rome. He was an associate researcher at the Italian National Interuniversity Consortium for Informatics, and currently he is a postdoctoral researcher at Sapienza University of Rome. His research interests lie in the area of concurrent programming and transactional systems, with special interest on performance analysis, modeling, optimization and energy efficiency.

Alessandro Pellegrini received a B.S. degree and a M.S. degree and a PhD in Computer Engineering at Sapienza, University of Rome. His main research area is on parallel and distributed architectures and applications, where he has published more than 50 technical articles. In 2015 he won the Sapienza prize for the best PhD thesis of the year. He has worked as a researcher at some national and international research centers (CINI, CINFAI and IRIANC).

Bruno Ciciani is Full Professor of Computer Engineering at Sapienza University of Rome. His research interests include fault tolerance, computer architectures, distributed systems, manufacturing yield prediction, performance and dependability evaluation. In these fields he has published more than 130 papers. He spent more than two years as visiting researcher at the IBM Thomas J. Watson Research Center (N.Y.).

Francesco Quaglia received his MS in Electronic Engineering in 1995 and his PhD in Computer Engineering in 1999, both from Sapienza University of Rome, where he has worked as Assistant Professor and then Associate Professor from September 2000 till June 2017. Since then, he works as Full Professor at the University of Rome Tor Vergata. His research interests include parallel and distributed computing systems and applications, operating systems, high performance computing, cyber-security, and fault tolerance.