BPF Hybrid Lock: Using eBPF to communicate with the scheduler

Victor Laforet, Jean-Pierre Lozi, Julia Lawall

BPF Hybrid Lock: Using eBPF to communicate with the scheduler

Victor Laforet
Institut Polytechnique de Paris / Inria
victor.laforet@inria.fr

Jean-Pierre Lozi
Inria
jean-pierre.lozi@inria.fr

Julia Lawall
Inria
julia.lawall@inria.fr

***Abstract*—Multi-core processors have long been available, yet harnessing their full computing potential remains a challenge due to synchronization issues that impact performance. This work addresses the dilemma of choosing between spin locks, known for their responsiveness but prone to collapse in over-subscribed systems, and blocking locks, which are stable but less reactive. We introduce the BPF Hybrid Lock, a novel synchronization technique that dynamically transitions between spin locks and blocking locks based on system subscription. eBPF is used to detect lock holder preemptions.**

Our contributions include algorithms for seamless lock transitions, enhanced condition variable reactivity, and detection of the optimal time to switch from a spin lock to a blocking lock. We provide an implementation of the BPF Hybrid Lock using MCS and Linux Futex locks, showcasing performance improvements in under-subscribed systems, such as LevelDB, where write operation times decreased from 282 to $102\mu s$ and read times from 40 to $25\mu s$.

***Index Terms*—BPF, ebpf, scheduler, locks, hybrid lock, spin lock, blocking lock, condition variables**

I. Introduction

Multi-core processors have been available for decades, but fully harnessing their computing power remains challenging, primarily due to synchronization issues that can hinder multi-core performance [2], [6]. Locks are a common synchronization primitive and come in two types: spin locks, which busy-wait to acquire a lock and are highly responsive, and blocking locks, which sleep when waiting for a lock and are thus less reactive due to the overhead of context switches.

However, spin locks suffer from a major drawback: their performance collapses when more threads try to acquire them than there are available cores on the machine, which is called over-subscription. This is due to the fact that waiting threads, continuously spinning as they await access to the lock, are likely to preempt the critical section, which prevents all progress on the critical path. These lock holder preemptions make spin locks unusable in oversubscribed scenarios. In contrast, blocking locks do not suffer from that issue. Waiting threads sleep and pose no threat of preempting the critical section. Consequently, blocking locks exhibit better performance in over-subscribed environments. However, they perform significantly worse than spin locks in under-subscribed scenarios [5].

In this context, this work introduces a novel locking technique called the BPF Hybrid Lock. This lock is designed to detect lock holder preemptions and seamlessly switch between spin locks and blocking locks. It operates by having lock waiters spin when the system is under-subscribed and block when the system is over-subscribed. Over-subscription is detected using eBPF by monitoring lock holder preemptions.

The goal of the BPF Hybrid Lock is to ensure critical sections make progress without being hindered by other threads. As the approach does not alter thread priorities or any kernel-level thread metadata, as allowed by e.g., Solaris [9], it cannot be exploited by malicious users to avoid preemptions, thereby compromising fairness. The BPF Hybrid Lock relies on eBPF to avoid kernel modifications, and addressing security concerns.

Our contributions are the following: an algorithm for safe transitions between lock types, an algorithm to make condition variables more reactive, a method to detect the optimal time to switch from a spin lock to a blocking lock, and provides an implementation of the BPF Hybrid Lock that uses MCS locks and Linux Futex locks. We improve performance of LevelDB, a key-value store from Google [3] in under-subscribed systems, as we reduce the average time of a write operation from 282 to 102 µs and the average time of a read operation from 40 to 25 µs.

This report is structured as follows. Section 2 introduces the Lock-Switching algorithm and discusses its correctness. Section 3 presents the condition variable switching algorithm. Section 4 proposes an implementation of the BPF Hybrid Lock employing MCS queue locks and Futex locks. Section 5 presents an evaluation of our lock. Section 6 presents related work on the topic of scaling synchronization to multi-core architectures. Finally, Section 7 concludes.

II. Lock-Switching Algorithm

The BPF Hybrid Lock combines the advantages of two different lock algorithms. To seamlessly transition between these two lock algorithms while maintaining correctness, we have developed an algorithm derived from a concept introduced by Antic et al. [1]. We improved it to be able to switch between lock algorithms at any point during the execution. In this section, we first describe the algorithm then discuss correctness as well as deadlock freedom.

A. Algorithm

Let us consider two distinct locks, namely Lock A and Lock B, with the objective of unifying them into a single lock that

can be dynamically switched between Lock A and Lock B at any given moment.

Lock storage Lock algorithms have different storage requirements. Each lock has a type stored within the structure of the BPF Hybrid Lock. Specifically, Lock type A (respectively B) has a type *lockA* (respectively *lockB*) that is stored in the *lock* structure.

Lock state The algorithm must maintain a record of the current lock type. Additionally, to uphold the principle of mutual exclusion, it becomes imperative to preserve the type of the last locked state. Consequently, a lock state variable is integrated as a component of the lock data structure. It is a tuple storing both the current lock type (A or B) as well as the last locked type.

Stable state vs. Transition State We describe two distinctive states: the stable state and the transition state. A stable state arises when the current lock type corresponds to the last locked type. Conversely, a transition state arises when the current lock type is different from the last locked type, signifying a transition from the last locked type to the current type

Lock-Switching Algorithm

```
1  type lock = record
2      lockA : lockA
3      lockB : lockB
4      lock_state: (lockType, lockType)
5
6  procedure lock(L : ^lock)
7      repeat
8          curr, last = L->lock_state
9
10         if not lock_type(L, curr)
11             continue
12
13         if (L->lock_state == (curr, last))
14             if (curr == last)
15                 break
16
17             repeat while
18                 not isfree_type(L, last)
19             L->lock_state = (curr, curr)
20             break
21
22         unlock_type(L, curr)
23
24 procedure unlock(L: ^lock)
25     unlock_type(L, last(L->lock_state))
```

1) Locking: The lock function must correctly attempt to acquire the lock or wait for the lock to become available. It should operate reliably even in scenarios where the lock state undergoes changes.

The lock function initiates by storing the *lock_state* variable (*line 11*). Subsequently, it attempts to acquire the current lock by employing the *lock_type()* function, which takes both the lock and the specified lock type to be acquired. In the event of a lock abortion, this attempt is unsuccessful, leading to the function restarting from the beginning.

During the process of acquiring the lock, it is possible that the lock state may have changed, signifying the necessity to transition to the alternative lock algorithm. Consequently, after obtaining the current lock, we inspect whether there has been a modification to the *lock_state* variable (*line 16*). If such a change is detected, the current lock is released (*line 25*), prompting the function to restart from the initial step.

At this point, in the absence of any switch occurring since the last acquisition, we can proceed with the function. Consequently, if we find ourselves in a stable state, we advance to the critical section, as indicated by the code at *lines 17-18*.

However, if a switch has occurred since the last lock acquisition, it implies that we are presently in a transition state, transitioning from the last locked type to the current lock type. In this transition state, there is a possibility that we have obtained the current lock type while another thread is still in possession of the previous lock type. To prevent simultaneous access to their respective critical sections by two threads, the new thread must await the completion of the critical section of the previous holder. At this stage, we patiently wait for the previous holder to exit its critical section, which is determined by the availability of the last locked type (*lines 20-21*).

Upon confirmation that no other thread is currently within a critical section, we can transition to a stable state by setting the last locked type in the *lock_state* to match the current lock type (*line 22*). Following this adjustment, we proceed to access the critical section (*line 23*).

2) Unlocking: Note that the last locked type remains unaltered while a thread holds that particular lock. Consequently, unlocking is a simpler process than locking, as it involves simply releasing the last locked type from the *lock_state* variable.

3) Switching: The *lock_state* variable only retains a single step back in the history of lock types. Consequently, we have to ensure that two transitions do not occur concurrently, as this could potentially lead to the initiation of a critical section while another thread has yet to release its lock. As a result, switching can exclusively take place in a stable state to avert undesired outcomes. To execute this switch, we employ a Compare-And-Swap instruction, which entails comparing the *lock_state* variable with a stable state and subsequently modifying the current lock type to the desired type.

However, it's worth noting that employing a Compare-And-Swap operation introduces the possibility of failure, necessitating re-execution to achieve the intended effect.

B. Correctness

To ensure the correctness of the algorithm both locks are checked during transition states.

Stable States In stables states, no thread can progress beyond *line 14* on the Lock-Switching Algorithm (II-A) without holding the current lock. Consequently, holding only the current lock is enough to meet the correctness requirements.

Transition States In transition states, one thread holds the current lock type while another thread may hold the previous lock type. The thread in possession of the current lock awaits

the release of the last locked type by the previous holder. Once it is released, no other thread can acquire that lock since it is no longer the current type. The waiting thread then sets the lock to a stable state and proceeds to its critical section.

Assuming both locks are correct, the combination of them with this algorithm is therefore correct throughout the execution.

Deadlock Freedom The algorithm is inherently deadlock-free. This is attributed to the fact that all threads, except one, are in a waiting state for a single lock, while the remaining thread awaits the conclusion of the critical section of the previous holder. Importantly, the execution of the previous holder is unaffected by the *lock_state* change, ensuring the release of the lock after a certain duration, thus eliminating the possibility of deadlock.

III. Condition variable-Switching Algorithm

Another synchronization technique, often employed in conjunction with locks, is the utilization of condition variables. These condition variables enable a thread to await a specific variable value and receive notification when another thread modifies that value. Typically, implementations of condition variables are blocking, which can be less responsive compared to spinning. Our observations have revealed that the use of blocking condition variables significantly reduces the benefits offered by our BPF Hybrid Lock, primarily due to slow wake up time of threads.

To address this issue, we have developed a method that ensures a safe transition between blocking and spinning condition variables. In this section, we first describe the algorithm and then discuss its correctness.

A. Algorithm

This algorithm is similar to the ticket management system. Threads take a ticket when they start waiting then wait for their number to be called when another thread signals that they can wake up. The condition variable contains two tickets: a target and a sequence value.

Condition Variable-Switching Algorithm

```
1  type cond = record
2      target : int // init 0
3      seq : int    // init 0
4
5  procedure wait(L: *lock, C : *cond)
6      target = ++C->target
7      seq = C->seq
8      release(L)
9
10     repeat while target > seq
11         if is_blocking(L->lock_state)
12             futex_wait(&C->seq, seq)
13         seq = C->seq
14
15     acquire(L)
16
17 procedure signal(C : *cond)
18     C->seq++
19     futex_wake(&C->seq, 1)
20
21 procedure broadcast(C : *cond)
22     C->seq = C->target
23     futex_wake(&C->seq, INT_MAX)
```

1) Wait: When a thread needs to wait on a condition variable, it calls the *wait()* function while holding the corresponding lock. The *wait()* function first saves the global target as its own target and increments the global target (*line 6*). At this point the thread has been enqueued and the lock can be released (*line 8*). The thread will then wait in a loop for the sequential value to be greater than or equal to its own target (*line 10*), meaning that the thread is dequeued. The loop will busy-wait when the lock type is spinning. The loop will sleeps with a futex wait on the sequential value when the lock type is blocking (*lines 11-12*). After the loop exits, the lock is acquired again (*line 13*).

2) Signal: The signal function is called by a thread that needs to wake up a single waiting thread. That function increments the sequential value (*line 18*) to dequeue the next thread in the queue then calls *futex_wake* on the sequential variable to wake up that thread (*line 19*).

3) Broadcast: The broadcast function is called by a thread that needs to wake up all waiting threads. That function sets the sequential value to the global target (*line 22*) to dequeue all threads currently in the queue then calls *futex_wake* on the sequential variable to wake up all of the threads (*line 23*).

B. Known issue: `signal()` can sometimes have no effect

We are aware of an issue that could arise from the *futex_wake()* call in the *signal()* function. Two queues are saved in the blocking state of the condition variable. The first queue is based on the ticket (the target is the queue number of the thread). The second queue is maintained by the *futex_wait* system call as threads will be awakened in the order they called this function.

If a thread enqueues itself in the ticket queue then gets preempted before enqueuing itself in the Futex queue, another thread could enqueue itself later in the ticket queue and logically earlier in the Futex queue.

The result would be a call to *signal* not waking up any thread. In our experiments, this case has never happened. However the issue needs to be addressed in future versions of our algorithm.

C. Correctness

To prevent race conditions, in the *wait()* function, the target is retrieved and incremented while the thread is holding the lock. This operation therefore does not have to be atomic.

The rest of the function works the same way as a ticket lock both for spinning and for blocking. Switching from one to another will only change the way the loop waits for the new value. No unwanted behaviour can then happen with threads waiting in different waiting states.

IV. Implementation of the BPF Hybrid Lock

The idea behind the BPF Hybrid Lock is to combine two different kinds of locks together: a spin lock and a blocking lock. Different kinds of spin locks can be used. Spin locks like Compare-And-Swap locks, Ticket locks, or Queue locks [7] can be improved by using them in combination with a blocking lock (for example a Futex lock) in a BPF Hybrid Lock.

The implementation of a BPF Hybrid Lock is spin lock specific. Some changes need to be made to the spin lock for the eBPF code to retrieve the thread that is currently holding the lock and to abort spinning when changing from spinning to blocking. In this section, we first describe the way we detect preemptions in eBPF then expose an implementation of the BPF Hybrid Lock for MCS queue locks.

The BPF Hybrid Lock is based on the lock-switching algorithm from Section 2. One of the lock types will be a spin lock (for example MCS). The other lock type will be a blocking lock (for example a Futex lock).

Condition variables are optimized using the condition variable switching algorithm. The state of this algorithm will be based on the current lock type of the lock switching algorithm.

A. Abortable locks

When the BPF Hybrid Lock switches from one lock to another, the switch should be fast, to free up compute resources. If the lock just switched to blocking, threads waiting for the spin lock could be stuck for some time before holding it and releasing it. Waiter threads will then slow down the system until they acquire and release the spin lock.

The solution to that issue is to have abortable locks. The lock-switching algorithm is made to handle abortable locks.

In the example of the MCS lock, the spin loop will periodically check the *lock_state* variable. If it has changed and the current lock type is not MCS anymore, then the loop will exit and the acquire operation will be aborted. The lock-switching algorithm will then restart from the beginning using the new lock type.

Abortable locks are not required by the BPF Hybrid Lock but improve the reactivity of the switch as all waiter threads will exit the spin loop as soon as the *lock_state* variable is changed.

B. Detecting preemptions

Detecting the best time to switch from the spin lock to the blocking lock is done through eBPF. Preemptions of the holder thread trigger a switch to the blocking lock.

The condition variable switching is done at the same time as the lock switching. For now, no detection of preemptions of signaling threads is done.

Detecting preemptions of the thread holding the lock is done through eBPF, using the tp_btf/sched_switch event, which is triggered during each context switch. This event exposes the scheduler variables *prev* and *next*, which reference the previous and the next tasks, respectively, and a boolean *preempt* which represents whether the context switch is a preemption.

Some information is exposed from the user space code to the eBPF code to be used in the event handler. A pointer to a data structure containing information to pinpoint the lock holder that can be retrieved in later events, and a pointer to the *lock_type* variable that can be modified when needed.

Algorithm The eBPF event is processed as follows. First, we only process preemptions by ignoring events with *preempt* as *false*. We retrieve information about which thread is holding the lock. Kernel threads can sometimes preempt other threads while the system is not overloaded or for load-balancing reasons. Therefore, if the next task is a kernel thread, we store the information that this thread has been preempted, in a map and will process it next time it is scheduled again. At this point, if the thread holds the lock and is not preempted by a kernel thread, the holder thread has been preempted and the lock should then switch to blocking. The Compare-And-Swap instruction is then used to change the *lock_type* variable to the blocking lock. If the Compare-And-Swap operation fails, no retry is done as eBPF prevents loops and the operation will be tried again on the next preemption.

This code is executed in the kernel and does not need any polling from the user space code. It is therefore very efficient and its performance impact is negligible.

C. MCS Queue Lock

A MCS Lock is a queue lock whose waiting threads busy-wait for their turn to enter critical section by spinning on a thread-local variable.

Let's consider the following implementation of the MCS lock [7].

MCS Algorithm

```
1  type qnode = record
2      next : ^qnode
3      waiting : Boolean
4  type lock = ^qnode // init to nil
5
6  // parameter I, below, points to a qnode
7  // record allocated (in enclosing scope)
8  // in shared memory locally-accessible
9  // to the invoking processor
10
11 procedure acquire(L : ^lock, I : ^qnode)
12     I->next := nil
13     pred : ^qnode := fetch_store(L, I)
14     if pred != nil // queue non-empty
15         I->waiting := true
16         pred->next := I
17         repeat while I->waiting // spin
18
19
20 procedure release(L : ^lock, I: ^qnode)
21     if I->next = nil // no successor
22         // true iff it stored
23         if compare_and_store(L, I, nil)
24             return
25         repeat while I->next = nil
26     I->next->waiting := false
```

Holder Retrieval Detecting the lock holder is a critical part of our approach to trigger the change from spin lock

to blocking lock. Atomically storing the information about which thread currently holds the lock is complex with a queue lock. Our implementation therefore has false positives which means it may switch to a blocking lock when the system is not over-subscribed. The condition used by the eBPF code is `not I->waiting AND ( L == I OR (I->next != NULL AND I->next->waiting))`. We only want to detect threads that are not waiting (*waiting* is *false*). We have two different cases: the thread is the first one in the queue or not. If the thread is the first one in the queue then it holds the lock. If it is not the first one in the queue but the next one is waiting then it holds the lock.

Acquire/Release No changes have to be made in the acquire and release function of the MCS algorithm. These functions are called by the lock-switching algorithm as is when needed.

V. Evaluation

In this evaluation, we aim to show that the BPF Hybrid Lock improves under-subscribed performance compared to blocking locks while maintaining the over-subscribed performance of blocking locks. We conduct experiments on an Intel 2-socket 40-core machine. The first experiment uses a custom microbenchmark capable of replicating various workload scenarios comparing the BPF Hybrid Lock with the MCS lock and the Futex lock. The second experiment evaluates the performance of the BPF Hybrid Lock in LevelDB. We vary the number of threads (ranging from 30 to 50), and the metric assessed is the average time taken by a single operation (read or write).

A. Microbenchmark

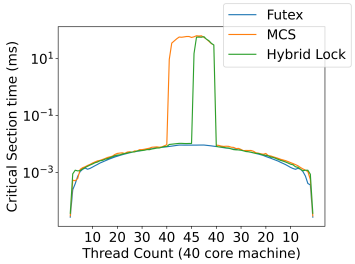
The microbenchmark initializes one lock with one worker thread that acquires and releases the lock in a loop. At regular intervals, new worker threads that access the same lock are spawned. A report on the current number of threads and average time to execute the critical section (acquire the lock, compute the critical section, release the lock) is produced regularly. This microbenchmark can emulate the eBPF code by forcing the lock to switch from one state to another at a specific thread count if needed.

Several parameters can be modified:

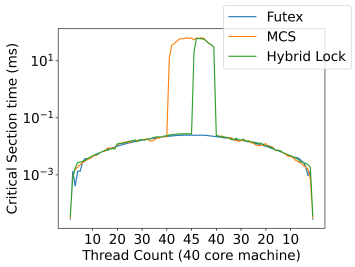
- the number of cache lines accessed
- contention (time between critical sections)
- the type of lock used.

Figure 1 shows microbenchmarks for MCS, Futex and the BPF Hybrid Lock for contention settings of 0, 1000 and 1,000,000 cycles between critical sections and 1 and 5 cache lines accessed. The BPF Hybrid Lock is forced to go back to MCS mode after some time at 45 cores to check its behavior. The relative performance between locks is similar when accessing 1 or 5 cache lines. The same conclusions can therefore be drawn from the graphs with 1 and 5 cache lines.

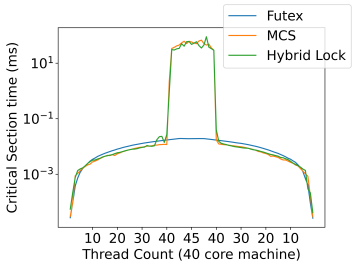
As expected, the spin lock (MCS) experiences a significant degradation of performance when the system is over-subscribed. On the other hand, the Futex lock performs correctly in any situation. Combining the best of both worlds,



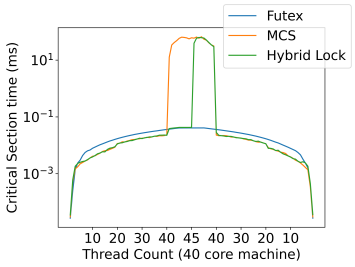
(a) 0 cycles, 1 cache line



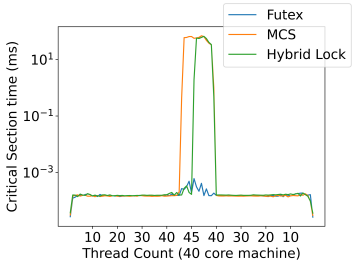
(b) 0 cycles, 5 cache lines



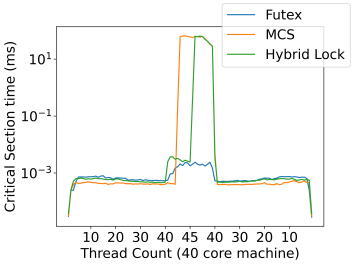
(c) 1000 cycles, 1 cache line



(d) 1000 cycles, 5 cache line



(e) 1,000,000 cycles, 1 cache lines



(f) 1,000,000 cycles, 5 cache lines

Fig. 1: Microbenchmarks on a 40-core machine

the BPF Hybrid Lock maintains the best performance in both scenarios. When under-subscribed, the BPF Hybrid Lock performs as well as the MCS lock. Once the system experiences over-subscription it switches and bahaves as the blocking lock (Futex).

To know more about the correlation between contention and lock performance, we modified the microbenchmark to compute average critical section time as a function of contention for three cases: under-, equal- and over-subscription (30, 40 and 50 threads) as shown in Figure 2.

At low contention (high number of cycles between critical sections), the microbenchmark does not use the lock enough to leave room for improvement. Very high contention tends to make MCS and Futex locks perform the same in under- and equal-subscription which reduces the potential improvement of the BPF Hybrid Lock. Contention between 10^2 and 10^6 cycles therefore seems to be the best target for the BPF Hybrid Lock.

Thanks to our microbenchmark, we can see that our solution has the best performance of both locks in the case of MCS and Futex with varying thread count and contention. We also identified the range of contention where the BPF Hybrid Lock improves performance the most. Implementing the BPF Hybrid Lock in a real-world application also improves performance, as we'll show in the next section.

B. Database Benchmark

We have implemented the BPF Hybrid Lock in LiTL [4], a library for interposing another lock and another condition variable algorithm in place of the *pthread* algorithms. LiTL uses `LD_PRELOAD` to link our functions with any code using the *pthread* library.

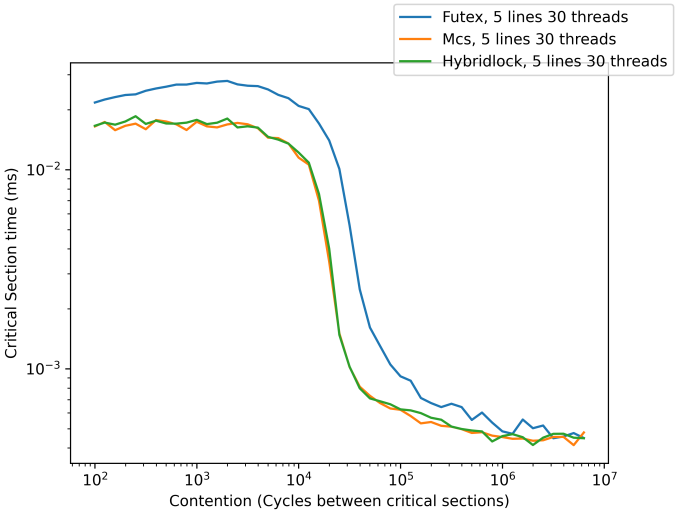
LevelDB is a key-value store from Google. It uses a couple locks in combination with condition variables. The impact of the BPF Hybrid Lock without condition variables is negligible on LevelDB.

We have done two benchmarks with LevelDB. One writes randomly to the database. The other reads randomly from the database. The benchmarks are run with different thread counts (from 30 to 50 on a 40-core machine) and the metric is the average time taken by a single operation (read or write). Figure 3 shows that the BPF Hybrid Lock performs better overall on both benchmarks

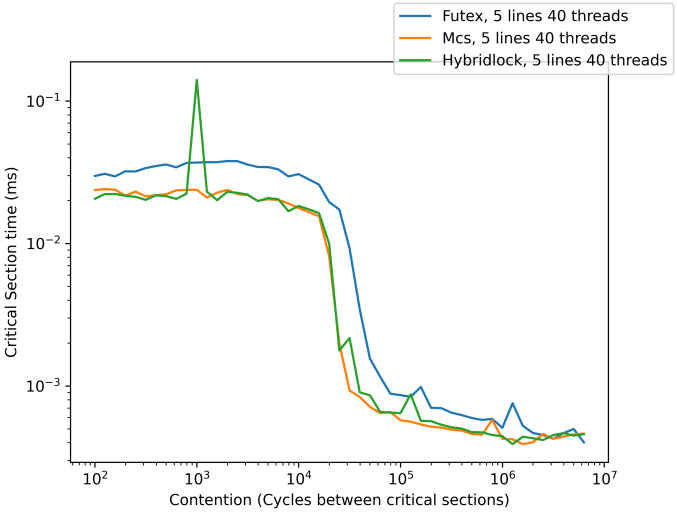
The BPF Hybrid Lock performs about 3 times and 37% faster respectively for writes and for reads on under-subscribed systems. The BPF Hybrid Lock performs similarly to the original LevelDB version on the benchmark with random writes starting from 40 threads. The BPF Hybrid Lock has performance comparable to the original version on the random reads benchmark for 40 threads and about 8% faster for 45 and 50 threads.

Considering no modifications to LevelDB code were required to produce these results, it should be possible to achieve speed ups on other applications directly by using the LiTL library.

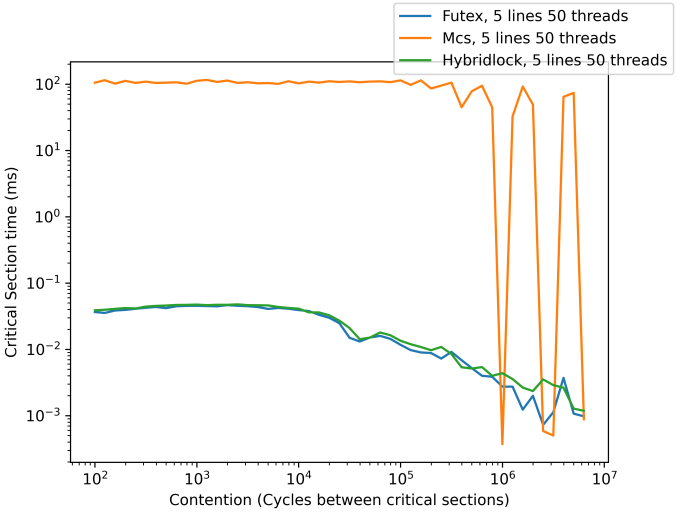
The performance advantage of the BPF Hybrid Lock is useful for applications that often have low thread counts, as



(a) 30 threads - under-subscription

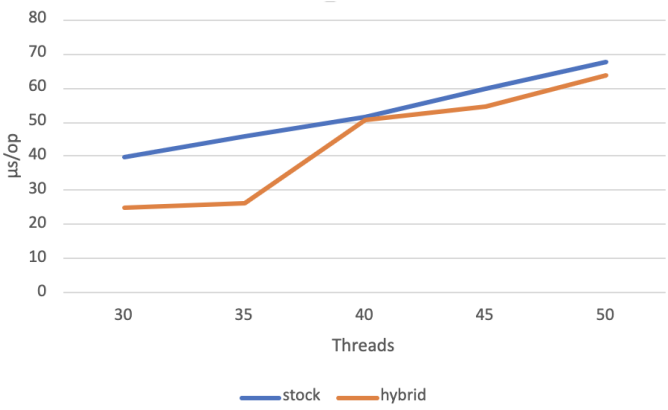


(b) 40 threads - equal-subscription

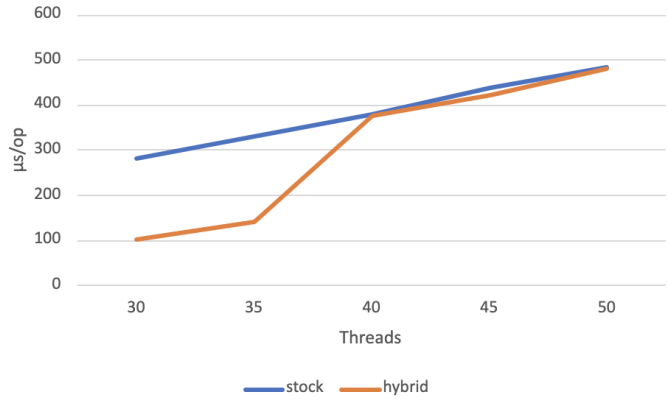


(c) 50 threads - over-subscription

Fig. 2: Contention microbenchmarks on a 40-core machine and 5 accessed cache lines



(a) Random reads



(b) Random writes

Fig. 3: LevelDB benchmarks on a 40-core machine

the use of spin locks means the application is more reactive. The hybrid lock is useful in case the system becomes over-subscribed, as gracefully switching to a blocking lock prevents the performance collapse that using a spin lock in such scenarios would result in.

VI. Related work

Many studies have been conducted to try to improve locking algorithms [6], [2], [5], [8]. Some improve the performance of highly-contended locks with a spin lock that can improve data locality and remove the need for synchronization [6]. Others, like the BPF Hybrid Lock, try to combine the best of both worlds, spin locks and blocking locks [5], [8].

RCL [6] dedicates an entire core to the execution of critical sections. Threads on other cores perform a remote procedure call to their critical sections and spin while waiting for the code to be executed. This avoids the need for synchronization between critical sections on the dedicated core and improves data locality as shared data protected by the RCL lock can stay in the cache of the server core. However, this approach has high overhead as a core has to be dedicated to the lock. Waiting threads also spin while waiting which wastes resources in loaded systems. The BPF Hybrid Lock eliminates spinning in over-subscribed systems.

Ousterhout [8] proposed a two-phase lock algorithm. In a first phase, the lock spins and waits for the lock to become available. After a set amount of time, the lock stops spinning and blocks, triggering a context switch. This approach combines great reactivity during the first phase and low resource usage in the second phase. However, heavily-loaded systems still suffer from the resource usage of the thread in the first phase. The BPF Hybrid Lock adapts to the current system load and improves performance.

Johnson et al. [5] present a solution that share similarities with the BPF Hybrid Lock. Their lock spins by default and signals a random subset of waiting threads to use a blocking lock when the system load requires it. This approach ensures great reactivity of the lock by keeping some spinning threads and improves resource usage by lowering the number of spinning threads. However, their approach requires extensive modifications of the operating system and is not portable. eBPF on the other hand is available on all recent Linux distributions and makes the BPF Hybrid Lock usable out-of-the-box on most systems.

VII. Conclusion

The BPF Hybrid Lock is a modern locking technique that combines a spin lock and a blocking lock and tries to use the best performing one according to system load. The key idea is to detect preemptions (unwanted context switches) of the thread that holds the lock using eBPF. When such a preemption is detected, the lock switches from spinning to blocking, thus reducing the amount of threads busy-waiting and releasing resources for the critical section. Our performance evaluation shows that the BPF Hybrid Lock improves performance when an application runs on an under-subscribed system and maintains performance for over-subscribed systems. The implementation of the BPF Hybrid Lock in the LiTL library allows for it to be used in a wide range of applications.

In future work, we will consider the detection of decreasing system load to switch back to a spin lock and thus increase reactivity, which will improve the usability of the BPF Hybrid Lock. Furthermore, other applications will be evaluated.

Availability The implementation of the BPF Hybrid Lock is not currently available online but will be shared on request.

Acknowledgments We would like to thank Inria Paris and the WHISPER team, as well as the Grid5000 project for the compute resources. Victor Laforet also thanks Jean-Pierre Lozi and Julia Lawall for the comments, suggestions and help they have provided all along the project.

REFERENCES

[1] Jelena Antic, Georgios Chatzopoulos, Rachid Guerraoui, and Vasileios Trigonakis. Locking made easy. *Middleware*, 2016.

[2] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. *SOSP*, 2013.

[3] Google. LevelDB is a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values. https://github.com/google/leveldb.

[4] Hugo Guiroux. LiTL: Library for Transparent Lock interposition. https://github.com/multicore-locks/litlhttps://github.com/multicore-locks/litl.

[5] F. Ryan Johnson, Radu Stoica, Anastasia Ailamaki, and Todd C. Mawry. Decoupling contention management from scheduling. *ASPLOS XV*, 2010.

[6] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. *Usenix ATC*, 2012.

[7] John M. Mellor-Crummey and Michaël L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM TOCS*, 1991.

[8] John K. Ousterhout. Scheduling techniques for concurrent systems. *IEEE Distributed Computer Systems*, 1982.

[9] Solaris. Solaris man page: schedctl_start(3c). https://docs.oracle.com/cd/E86824_01/html/E54766/schedctl-start-3c.html.