eBPF Hybrid Lock: Scalable Spin-based User-Space Locking (Poster)

Victor Laforet, Jean-Pierre Lozi, Julia Lawall

▶ **To cite this version:**

Victor Laforet, Jean-Pierre Lozi, Julia Lawall. eBPF Hybrid Lock: Scalable Spin-based User-Space Locking (Poster). SOSP 2023 - The 29th ACM Symposium on Operating Systems Principles, Oct 2023, Koblenz, Germany. hal-04262326

HAL Id: hal-04262326
https://inria.hal.science/hal-04262326v1

Submitted on 27 Oct 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

eBPF Hybrid Locks: Scalable Spin-based User-Space Locking

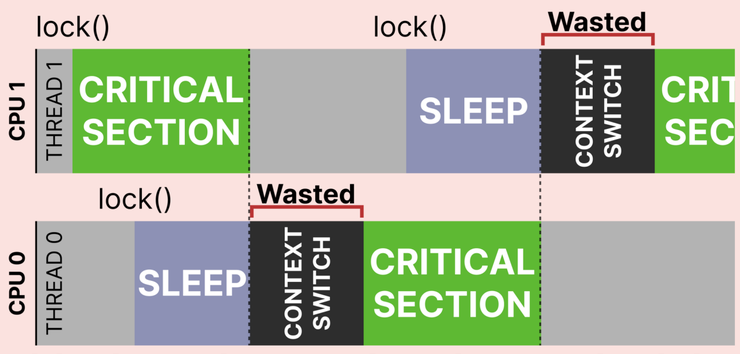
Victor Laforet, Jean-Pierre Lozi, Julia Lawall
Inria

Problem

Blocking locks are **not affected** by **system subscription** but are always **slower to react than spin locks.**

Blocking Locks

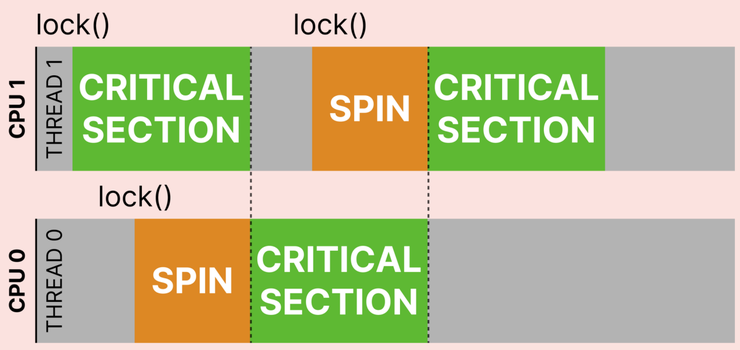
- **Waiting is cheap and allows other threads to progress**
- **Slow to wake up**



Critical Sections wait to be rescheduled

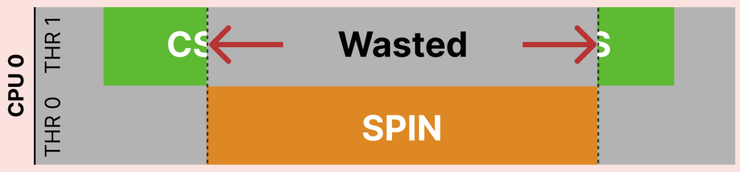
Spin Locks

- **Quick to react**



Critical Sections start immediately

- **Performs poorly in over-subscribed scenarios** (more threads than cores)



Thread 1 is preempted by Thread 0*

Approach

➔ **Spin** to **improve reactivity.**
➔ **React to preemptions** of the **lock holder** and **speed up re-scheduling**:

- Detect preemptions using **eBPF**
- Switch to a **blocking lock** to **release CPU resources** for the **lock holder**.

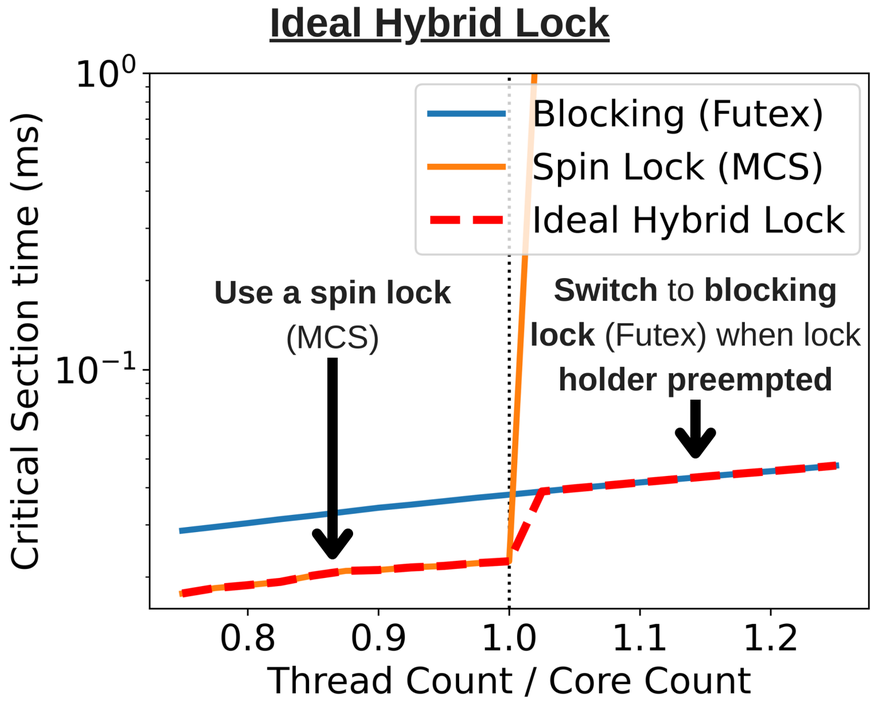
Previous work

- **Disabling preemptions** (Solaris [1])
  ➔ Can **compromise fairness/not portable**
- **Spin-then-wait** policies [2]
  ➔ **Overhead** with **many threads**
- **Decoupling contention management from scheduling** [3]
  ➔ Needs **user daemon thread** with **timer** to **track global overload metric**

[1] Solaris man page: schedctl start(3c)
[2] John K. Ousterhout. Scheduling techniques for concurrent systems. IEEE Distributed Computer Systems, 1982
[3] F. Ryan Johnson, Radu Stoica, Anastasia Ailamaki, and Todd C. Mowry. Decoupling contention management from scheduling. ASPLOS XV, 2010

Principle

Ideal Hybrid Lock



Context Switch Detection

- **eBPF `tp_btf\sched_switch`** event
- Kernel threads ignored

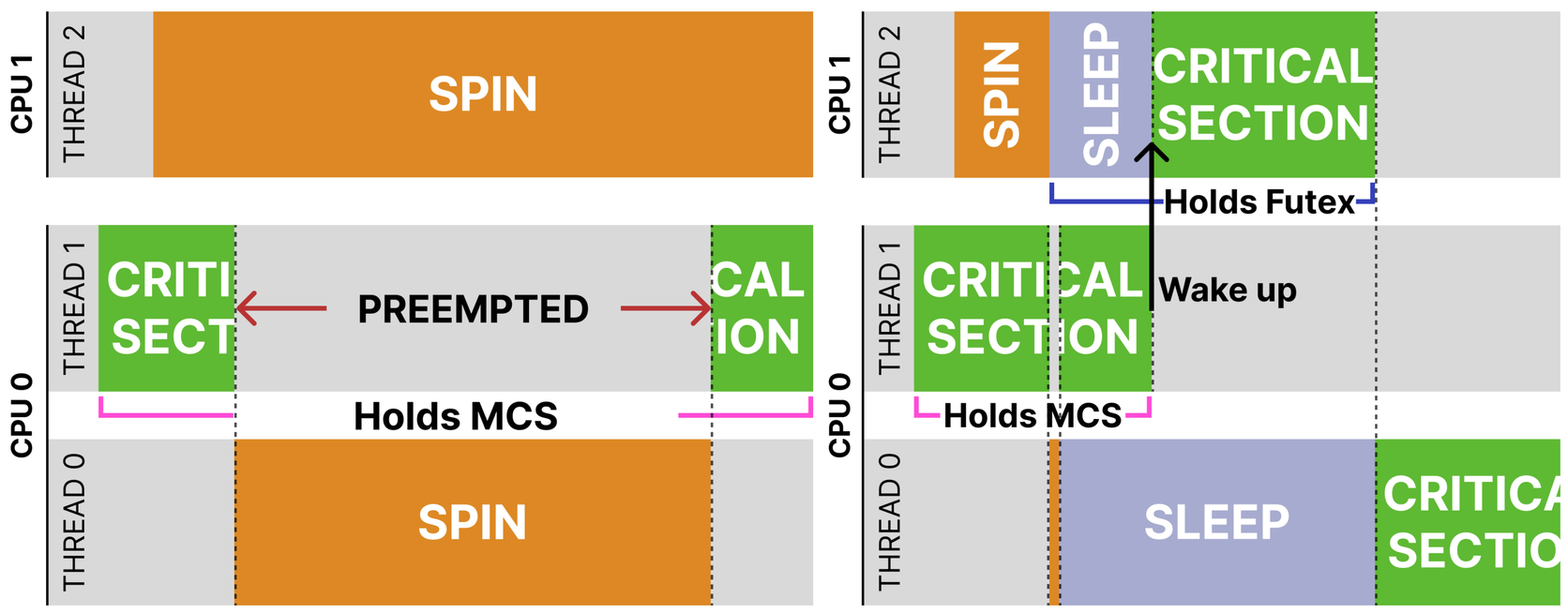
Abortable MCS

The **MCS component** of the hybrid lock is **abortable** to **instantly switch** to blocking and **improve reactivity.**

When the **lock holder is preempted**:

- Waiter threads all **switch to Futex**
- *Thread 2* acquires the Futex lock and **waits for *Thread 1* to end its critical section.**
- **Thread 1** will **wake up Thread 2**

eBPF Hybrid Lock (MCS/Futex)

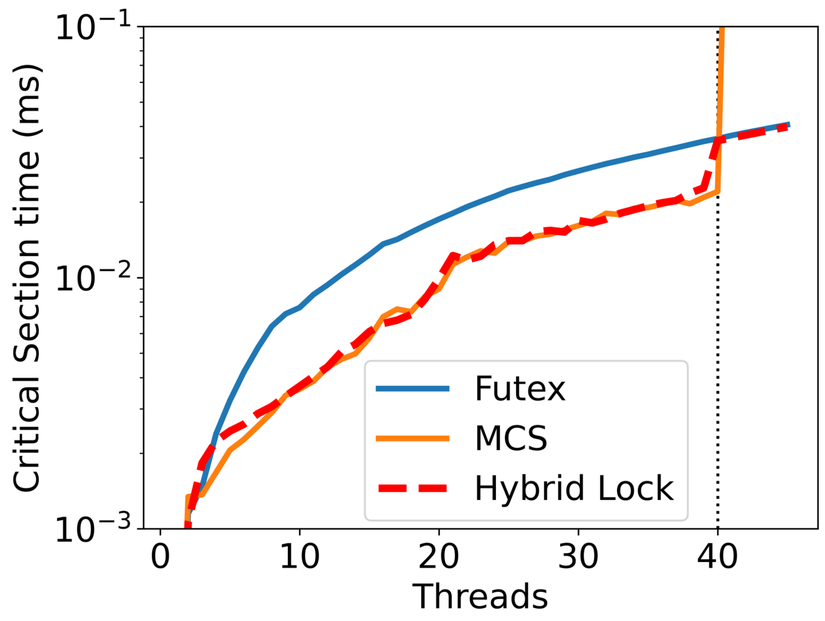


Without eBPF Hybrid Lock* ➔ **With eBPF Hybrid Lock***

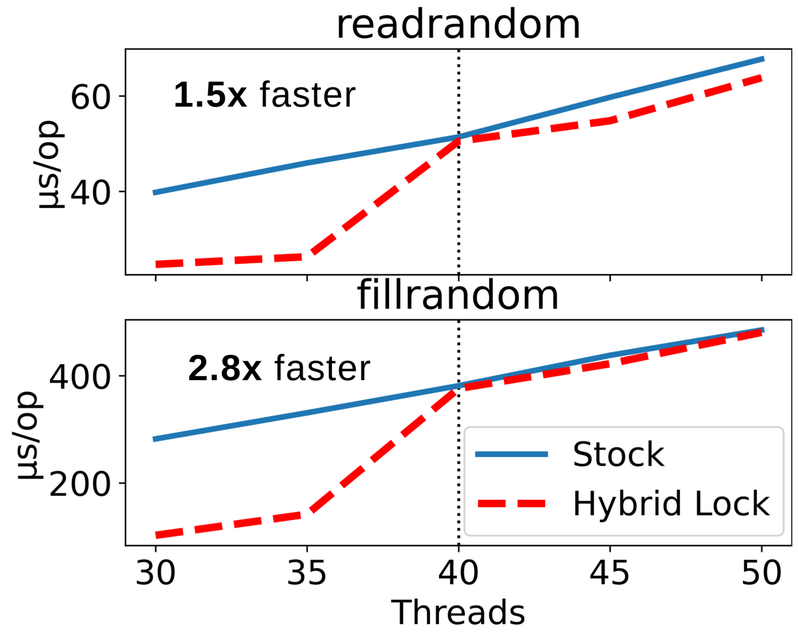
Results: Microbenchmarks and LevelDB

40-thread Intel machine: 2 sockets (10 cores/20 threads each)

Microbenchmark (lower is better)



LevelDB benchmarks (lower is better)



Discussion

Detection:

- **Restartable sequences** (André Almeida)
- Atomic MCS enqueue (Double-CAS) for **accurate lock holder detection**
- Use **preemption address** for **accurate lock holder detection** with **unmodified MCS**

Hybrid Lock Algorithm:

- **When** do we switch **back to spinning?** E.g., hybrid **lock** is **free**
- Only **switch current spinners** to blocking
- **Increase priority** of **holder** on preemption

***Context switches not shown**

eBPF Hybrid Lock: Scalable Spin-based User-Space Locking

Victor Laforet*
Jean-Pierre Lozi
Julia Lawall
first.last@inria.fr
Inria
France

1 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 [1, 4]. 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 [3].

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 [5], 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 [2] 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.

2 Principle

The foundation of the BPF Hybrid Lock lies in its lock-switching algorithm, which allows for smooth transitions between the two lock types, i.e., a spin lock and a blocking lock at any point during the execution. For the most part, this algorithm operates similarly to a standard lock. However, in scenarios where the lock type has changed since the last lock acquisition, the algorithm waits for the previous holder to exit its critical section before transitioning back to a stable state. This process guarantees mutual exclusion and avoids conflicts.

The algorithm is switched to blocking when a lock holder preemption is detected using the eBPF *tp_btf/sched_switch* event, which occurs during each context switch. When processing this event, only forced context switches, i.e., preemptions are considered. If the preempted thread is holding the lock, the eBPF code switches to blocking by modifying the *lock_state* variable directly. For now, detection of the need to switch back to the spin lock is not implemented. The eBPF code executes in the kernel without the need for user space polling, making it highly efficient and minimally impacting system performance.

We can improve the efficiency of our algorithm using two methods. First, we want to allow the algorithm to instantly switch from one lock to the other and improve its responsiveness by choosing abortable lock algorithms. This capability is crucial when transitioning from spinning to blocking, as it makes it possible to switch instantly from the spin lock to the blocking lock, thereby quickly releasing cores for the lock holder. Second, to enhance the responsiveness of condition-variable-dependent applications, we developed an

*Student, Presenter

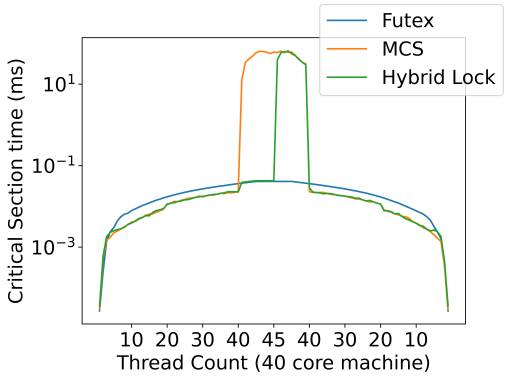


Figure 1. Microbenchmark comparing the BPF Hybrid, MCS and Futex locks performance with an increasing then decreasung thread count on a 40 core machine, accessing 5 cache lines and waiting for 1000 cycles between iterations. The BPF Hybrid Lock is switched back to MCS in the middle of the graph.

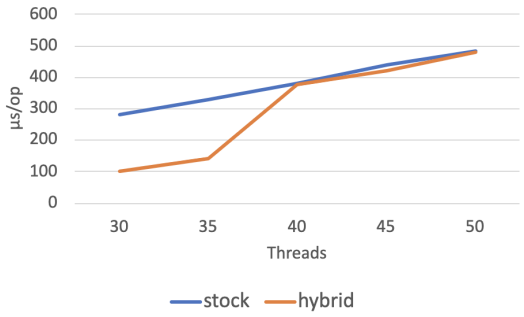


Figure 2. LevelDB random writes benchmark on a 40-core machine comparing the performance of the original version and the BPF Hybrid Lock version.

algorithm that complements the lock-switching technique. This algorithm allows for safe transitions between blocking and spinning condition variables. It operates based on the principles of the ticket queue management system and synchronizes its switching with the lock.

We provide an implementation of the BPF Hybrid Lock we described and evaluate its performance in the next section. This implementation utilizes MCS as the spin lock and employs a lock based on Linux Futex as the blocking lock.

3 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).

As shown in Figure 1, the spin lock (MCS) experiences a significant degradation of performance when the system is over-subscribed. On the other hand, the performance of the Futex lock does not degrade. Combining the best of both worlds, 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 behaves as the blocking lock (Futex).

To evaluate our algorithm in a real-world application, we use LevelDB, which utilizes a single global lock along with a condition variable. The BPF Hybrid Lock version consistently outperforms the original version with both write and read operations when the system is under-subscribed. It demonstrates a performance boost, with approximately 2.8 times better efficiency for writes (as seen in Figure 2) and 1.5 times better efficiency for reads (not shown) compared to the original version. In over-subscribed scenarios, the BPF Hybrid Lock version performs on par with the original LevelDB version for both reads and writes.

In summary, the BPF Hybrid Lock demonstrates its utility in improving multi-core system performance through both the microbenchmark and the LevelDB database benchmark. It offers improved performance in under-subscribed systems while maintaining performance in over-subscribed settings.

4 Future Work

Future work on the BPF Hybrid Lock will involve the design of lock APIs to accommodate different lock algorithms, a separate detection mechanism for condition variables, the detection of situations warranting a switch back to a spin lock and an extended evaluation with more real-world applications.

References

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

[2] 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.

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

[4] 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.

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