NUMA-aware lock for Java Lightweight Threads

Anonymous Author(s)

1 Introduction

There exist a lot of different implementations of NUMA-aware locks for standard platform Java threads (“heavy-weight” threads), e.g., CNA [5], HMCS [4], HCLH [6]. All of them use the fact that platform threads are rarely suspended and rarely migrate on another NUMA node: they expect the control over the suspension and that each thread enters and leaves the critical section running on the same NUMA node. If another thread on the current NUMA node tries to acquire the lock, the ownership is passed to it. This way, threads rarely access the remote memory on another NUMA node.

In comparison with the platform threads, *vthreads* or light-weight threads can share address space and resources with other vthreads, reducing context switching time during execution. They work on a pool of platform threads. The platform thread, on which the current vthread is running, is called a *carrier* thread. Two main problems standing behind are that a vthread can be often suspended at an arbitrary time and can often change a carrier thread during its life-time. Moreover, Java’s vthreads scheduler is not aware of the NUMA model and uses a simple work-stealing scheduler. As a result the previously developed locks have a degraded performance due to the suspension and the migration. Thus, our goal is to design an efficient lock for lightweight threads on NUMA architecture.

Our target platform is ARM-based server TaiShan 200 with 128 HiSilicon Kunpeng-920 processor cores in total. It has a complex hierarchical NUMA architecture. Four cores are combined into a group called CCL (CPU Cluster). The change of cache ownership inside one CCL is fast. Then, 8 CCLs are combined into a SCCL (Super CPU Cluster) represented as a NUMA node, where L3 cache is shared. Finally, two NUMA/SCCL nodes are combined on a socket (SoC package in a socket). The scheme of this machine is shown in Figure 3.

2 Implementation details

In this paper, we design a new VNA (Virtual NUMA-aware) lock. At first, we introduce Lock and LockInfo classes, presented in Listing 1. The lock class provides two functions `lock()` and `unlock()`. The lock stores an atomic boolean flag (Line 2) that shows whether some vthread owns the lock and works like the test-and-set lock. Also, it has an array of MCS locks, one for each NUMA node (Line 3). The MCS is cooperative and the vthread is parked when acquiring the lock and unparked by the predecessor.

The helper class LockInfo is returned by the `lock()` function and is used in `unlock()` function. Moreover, there is a field `fastPath`(Line 6) meaning that the lock is acquired via the fast path and should be unlocked in a special way. A field `numaId` (Line 7) is used to point to the corresponding MCS.

Listing 1. Lock and LockInfo structures

```
1 class Lock:
2     AtomicBool flag = false
3     MCS[] mcs = new MCS[numa_cnt]
4
5 class LockInfo:
6     boolean fastPath
7     int numaId
```

A vthread can acquire the lock via a fast or a slow path. The lock function is presented in Listing 2. At first, a vthread tries to acquire a lock using atomic compare-and-swap (CAS) on Line 2. If the operation succeeds then the lock is acquired via the fast path. Otherwise, it goes via the slow path. The vthread gets the identifier of the current NUMA node by executing function `getNumaId()` (Line 4). When the vthread gets its NUMA id, it tries to acquire the corresponding MCS lock (Line 5) and, when acquired, it waits for the successful CAS on the flag (Line 6).

Function `getNumaId()` (Line 4) executes a syscall getcpu [1]. However, if we execute the syscall each time we access a lock, it becomes really expensive. To solve this issue, we cache this identifier in the local memory of a carrier thread. After several accesses, the value is updated. To access the vthread’s carrier we use the thread MethodHandle [2].

The unlock function is presented in Listing 2. First, a vthread sets the flag to `false`. If the lock is acquired via the slow path, the vthread also unlocks the corresponding MCS. It is important, that the vthread should use numaId from lockInfo because it can already be moved to another NUMA node during the execution of the critical section.

Our approach can be seen as a hierarchical lock: inner-socket MCSs and outer-socket spin-lock. We had two ideas in mind: 1) we want to decrease the access to the memory on other NUMA nodes, thus, we use MCS locally, while 2) the lock should be allowed to be taken by several threads, thus, helping with the suspension — given several attempting threads we have more probability that one of them is not suspended; for that, we chose outer spin-lock. We back up the chosen design with the experiments.

3 Experiments

For the benchmarks, we used Java Microbenchmark Harness framework [3]. Our benchmarks start many lightweight threads and each of them runs a work loop. The pseudocode is presented in Listing 3. On each iteration, the vthread: 1) emulates work by multiplying two outer square matrices of the

’18, January 01–03, 2018, New York, NY, USA
2023.

Listing 2. Lock and unlock procedures.

```
1  LockInfo lock():
2    if cas(flag,false,true):
3      return LockInfo(fastPath=true)
4    numaId = getNumaId()
5    mcsNode = mcs[numaId].lock()
6    while !cas(flag,false,true):
7      // spin
8    return LockInfo(numaId=numaId,
9                    fastPath=false)
10 void unlock(LockInfo lockInfo):
11   flag = false
12   if unlockInfo.fastPath:
13     return
14   mcs[unlockInfo.numaId].unlock()
```

same size (Line 3); 2) calls Thread.yield() to emulate the vthread interruption (Line 4); 3) acquires the lock (Line 5); 4) emulates work inside the critical section by multiplying two square matrices (Line 6); 5) calls inner Thread.yield() to emulate the vthread interruption inside the critical section (Line 7); and 6) releases the lock (Line 8).

Calling Thread.yield() inside and outside the critical section is necessary because otherwise a vthread may never be interrupted and other vthreads will not run at all.

By changing the sizes of the matrices, we control the contention level on the lock. Before running the benchmark, we pin platform threads to the cores.

Listing 3. vthread's procedure in benchmark

```
1 void run():
2   for i in 0..10000:
3     matrixA * matrixB
4     Thread.yield()
5     lock.lock()
6     matrixC * matrixD
7     Thread.yield()
8     lock.unlock()
```

We run experiments on a system with 128 HiSilicon Kunpeng-920 cores (4 NUMA nodes) running CentOS Stream 9 and OpenJDK Java 19. On the plots, OX axis represents the number of vthreads and OY axis shows the total number of iterations per millisecond (higher is better). Each point was aggregated on 5 independent runs.

We compare our VNA lock (with a queue per NUMA) and VNA_2_Q (a queue per socket) against fair and unfair ReentrantLock, CNA, HMCS (two-tiered, as three-tiered one works worse), and HSPIN (hierarchical spin) locks. HSPIN calls Thread.yield() after some spins on the flag, because, otherwise, the system might be blocked when all carrier threads are actively spinning.

At first, we consider a low contention case: the size of outer matrices is 100 while the size of inner matrices is 15 (Figure 1). VNA with four queues is up to 30% more efficient than other locks when the number of vthreads exceeds 64, i.e., it uses more than two NUMA nodes.

The results of the benchmark with absent outer work and inner matrices with size 50 (Figure 2) is the high contention case. VNA with four queues achieves the best throughput when the number of vthreads is more than 32.

The experiments support our basic ideas: 1) we need an outer spin-lock to help with the suspension — HMCS and VNA with just two MCSs works worse; 2) locally, it is better to use MCS as HSPIN works worse.

Figure 1. Throughput. 10000 iterations. Outer matrices are 100×100, inner matrices are 15×15.

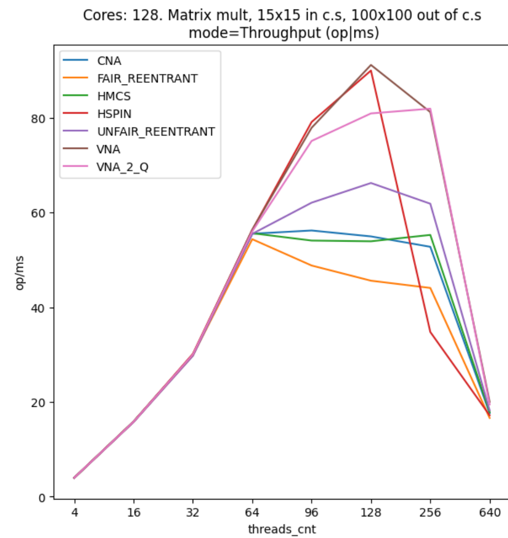
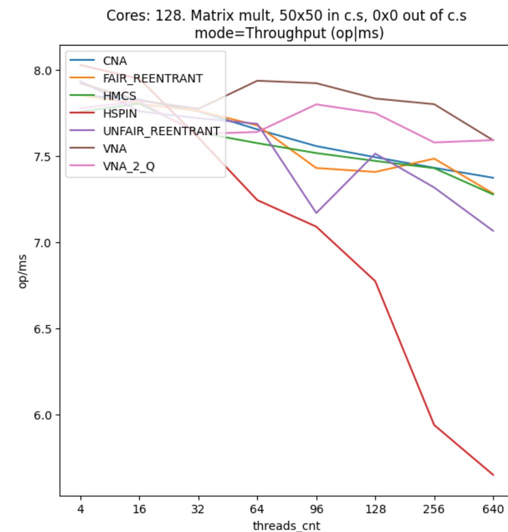


Figure 2. Throughput. 10000 iterations. Absent outer matrices, inner matrices are 50×50.



4 Conclusion

We designed a NUMA-aware lock for lightweight threads in Java. Our lock has a higher throughput for lightweight threads than standard or previously developed NUMA-aware locks on benchmarks with different levels of contention. As an additional advantage, our lock is easy to understand.

References

[1] [n. d.]. *getcpu(2) — Linux manual page*. https://man7.org/linux/man-pages/man2/getcpu.2.html

[2] [n. d.]. *Java Reflection, but much faster*. https://www.optaplanner.org/blog/2018/01/09/JavaReflectionButMuchFaster.html

[3] [n. d.]. *OpenJDK: JMH*. https://openjdk.org/projects/code-tools/jmh/

[4] Milind Chabbi, Michael Fagan, and John Mellor-Crummey. 2015. High Performance Locks for Multi-Level NUMA Systems. *SIGPLAN Not.* 50, 8 (jan 2015), 215–226. https://doi.org/10.1145/2858788.2688503

[5] Dave Dice and Alex Kogan. 2019. Compact NUMA-Aware Locks. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, Article 12, 15 pages. https://doi.org/10.1145/3302424.3303984

[6] Victor Luchangco, Dan Nussbaum, and Nir Shavit. 2006. A hierarchical CLH queue lock, Vol. 4128. 801–810. https://doi.org/10.1007/11823285_84

A Machine Layout

Figure 3. NUMA hierarchy of Kunpeng-920 with 128 cores