RON: One-Way Circular Shortest Routing to Achieve Efficient and Bounded-waiting Spinlocks

Shiwu Lo, Han-Ting Lin, Yao-Hung Hsieh, and Chao-Ting Lin, *National Chung Cheng University;* Yu-Hsueh Fang, *National Cheng Kung University;* Ching-Shen Lin, *National Chung Cheng University;* Ching-Chun (Jim) Huang, *National Cheng Kung University;* Kam Yiu Lam, *City University of Hong Kong;* Yuan-Hao Chang, *Academia Sinica, Taiwan*

https://www.usenix.org/conference/osdi23/presentation/lo

This paper is included in the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-34-2

RON: One-Way Circular Shortest Routing to Achieve Efficient and Bounded-waiting Spinlocks

Shiwu Lo$^\heartsuit$, Han-Ting Lin$^\heartsuit$, Yao-Hung Hsieh$^\heartsuit$, Chao-Ting Lin$^\heartsuit$, Yu-Hsueh Fang$^\triangle$, Ching-Shen Lin$^\heartsuit$,

Ching-Chun (Jim) Huang$^\triangle$, Kam Yiu Lam$^\diamondsuit$, Yuan-Hao Chang$^\circ$

National Chung Cheng University$^\heartsuit$, National Cheng Kung University$^\triangle$, City University of Hong Kong$^\diamondsuit$, Academia Sinica, Taiwan$^\circ$

Abstract

As the number of processor cores increases, the efficiency of accessing shared variables through the lock-unlock method decreases. A NUMA-aware algorithm, which only considers the transmission delay between processors, may not fully utilize the connection network of a multi-core processor. This limits the scalability of a multi-core processor due to the large amount of low- and variable-cost data sharing between cores. The problem is that the reduction in communication cost cannot compensate for the increase in the time complexity of the spinlocks, and the farthest transmission distance becomes longer with more cores.

We propose a method called Routing on Network-on-chip (RON)[1] to minimize the communication cost between cores by using a routing table and pre-calculating an optimized locking-unlocking order. RON delivers locks and data in a one-way circular manner among cores to (1) minimize global data movement cost and (2) achieve bounded waiting time. Microbenchmarks provide quantitative analysis, while multi-core benchmarks show performance under various workloads.

In terms of user space performance, RON improves the performance of Google LevelDB by 22.1% and 24.2% compared to ShflLock and C-BO-MCS, respectively. In the kernel space, RON is 1.8 times faster than using ShflLock for Google LevelDB. RON-plock solves the problem of oversubscription with constant space complexity and achieves 3.7 times and 18.9 times better performance than ShflLock-B and C-BO-MCS-B, respectively.

1 Introduction

This paper primarily focuses on addressing the lock-unlock problem under high contention. Despite the significant increase in the number of cores in a *central processing unit* (CPU), a fully shared cache memory system can limit the bandwidth of the cache memory, creating a performance bottleneck. To overcome this issue, CPUs can maintain private caches, and processors sharing these private caches are referred to as *Cache Coherent Non-uniform Memory Access*(NUMA) processors (abbreviated as ccNUMA).

Spinlocks and atomic operations are provided to ensure the coherency of shared data in the cache, and programs access shared data in *critical sections* (CS) [5, 6]. However, minimizing data access latency is a crucial issue that can significantly impact CPU performance in accessing shared data in ccNUMA [12, 13]. This depends on the topology of the *Network on Chip* (NoC) and the movement of data between caches, which is triggered by tasks executing in the CPU.

When multiple tasks compete to enter a CS, granting the closest task to the one that just released the lock access can reduce data access latency. However, this can still be costly as core-to-core transmission latencies vary in a CPU [14]. Additionally, allowing the core with the shortest transmission latency to enter the CS may lead to adjacent cores having exclusive access, leading to poor throughput [41].

Inter-core communication limits multicore processor scalability [19, 20]. Transmission latency can be fixed or distance-dependent. While monolithic die processors such as Intel Xeon [2] exhibit similar inter-core communication latency, *Multi-Chip-Module* (MCM) processors like AMD EPYC [2] and Apple M1 Pro [2] use MCM technology to increase the number of cores on a processor affordably and at scale. Next-generation Intel Xeons also use MCM [3], but MCM processors may have varying transmission latency between and within chips.

NUMA-aware spinlocks [25–31] enable cores from the same "NUMA node" to enter the CS in batches. This approach is suitable for multi-core processors, such as AMD EPYC, that have different transmission latencies. We can minimize handover costs by dividing the cores in a multi-core processor into mini-nodes, such as the east and west parts shown in Figure 1, and using a NUMA-aware approach to schedule them. However, transferring locks between cores in a mini-node is not considered in these algorithms. A layered approach (e.g., cohort [25]) can address this, but using too many layers

[1]The source codes of RON can be found at https://github.com/shiwulo/ron-osdi2023.

(e.g., C-TKT-TKT-TKT) can make spinlocks complex and expensive. Modern processors have non-uniform computing power [47], where higher computing power implies a greater ability to acquire locks. Batch-based algorithms often set a "maximal batch size" periodically to prevent starvation and maintain fairness. However, reducing the batch size for fairness can decrease performance. Unfair lock allocation causes unbalanced resource distribution and reduces throughput, as discussed in Section 2.3.

The optimization principle for low-cost communication is similar to that of data routing in computer networks. Although computer networks can use complex algorithms to produce the best route, such methods are often too expensive for small CS in multi-core processors. Therefore, we pre-calculate the shortest circular route including all cores, and the spinlock algorithm generates a path of threads to enter the CS according to the pre-calculated route. The "one-way circular shortest routing" shortens the distance between cores, while the "shortest routing" produces a local optimal solution for handover cost. The "one-way" optimizes handover costs by transferring locks in the direction where more threads are waiting, and the "circular" approach limits the number of times a thread waits to enter the CS. Thus, we schedule tasks on the cores to enter CS in a "one-way circular shortest routing" manner to improve the performance and fairness.

This paper makes three main contributions. Firstly, we propose the simple yet effective concept of "one-way circular shortest routing" to solve the fairness and efficiency issues in spinlocks. Secondly, we identify that long-term fairness alone is insufficient for modern processors, which have cores with varying capabilities to grab locks due to differences in computing frequency. Finally, we provide insights on how single-core spinlocks can work alongside multi-core spinlocks without compromising efficiency and fairness.

In Section 2, we discuss the limitations of NUMA-aware spinlocks in minimizing transmission latency in multi-core processors and the negative impact of unfair spinlocks on throughput. Section 3 presents related work in the field. In Section 4, we propose our fair and efficient spinlock algorithm for ccNUMA. Section 5 addresses performance under oversubscription, while Section 6 compares RON with two well-known algorithms. Section 7 discusses the advantages and disadvantages of RON compared to ShflLock and Linux's qspinlock, and Section 8 concludes the paper.

2 Preliminary and Motivation

2.1 Data Coherence in ccNUMA

Figure 1 shows an example of the multi-core architecture, in which the connection network of each core group is similar to the *CPU CompleX* (CCX) of Advanced Micro Devices, Inc. (AMD). In a multi-core architecture, data stored in the cache memory can be shared among the cores. *Cache coherence non-uniform access* (ccNUMA) uses snoop-based and/or directory-based cache coherence algorithms to maintain consistency of shared data in each cache memory [42]. The snoop method broadcasts messages such as "some shared data has been updated", whereas the directory-based method allows point-to-point communication between nodes. A node can be a core or a group of adjacent cores.

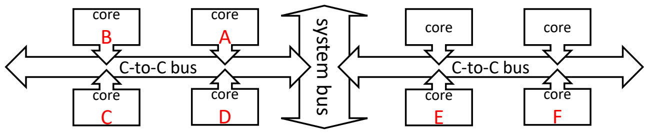


Figure 1: An example NoC architecture of ccNUMA.

2.2 Cost of Spinlocks on Multi-core CPUs

We define the *serializing cost* as the cost of allowing multiple threads to have mutually exclusive access to shared data. Serializing costs are divided into "*contention*" and "*handover*". The contention cost is the cost for determining the next task that can enter the CS. It depends on the data structure and data access method used by a spinlock. For example, the ticket lock [43] is centralized, while MCS spinlock (or called "MCS" for short) [44] is decentralized. In the ticket lock, all threads continuously monitor a variable of the ticket lock, and this can generate a lot of traffics in the NoC. The contention cost also depends on how the threads are granted to enter the CS. The raw spinlock (e.g., GNU's pthread_spin_lock [46], abbreviated as "*Plock*") relies on the NoC to determine when the first thread can enter the CS. The MCS spinlock [44] allows each thread to wait on a different variable. Therefore, MCS spinlocks prevent atomic operations from triggering excessive bus traffic.

The handover cost depends on the speed of transferring shared data between the lock-holding thread and the successive thread. Because spinlock is a shared data structure, a smaller handover cost can also slightly reduce the contention cost. As the example in Figure 1 shows, the processor is divided by two parts, i.e., the west and east parts. The two parts are connected through a system bus. The handover cost of using the C-to-C bus only is 1. The handover cost between the core and the system bus is related to the distance between the core and the system bus. B, C, and F are far away from the system bus, so the handover cost is 3, and the handover cost of A, D, and E is 2.

In conventional NUMA-aware spinlocks, the order of entering the CS can be arbitrary, for example, A→D→B→C→F→E. Since A, B, C, and D have the same communication cost, they belong to the same group (i.e., mini-node). The same goes for E and F. The handover cost of this order is (A, 1, D, 1, B, 1, C, 3, 3, F, 1, E)=10. This paper proposes to use one-way circular shortest routing to minimize handover costs. By scheduling the order of entering the CS as A→B→C→D→E→F, the handover cost is reduced to (A, 1, B, 1, C, 1, D, 2, 2, E, 1, F)=8. In this example, the one-way circular shortest routing improves the performance by 20% (i.e., $\frac{10-8}{10}$).

2.3 Throughput or Fairness? Take Both !

Various locking techniques have been proposed [39] to improve system throughput in varying levels of contention. Lock algorithms such as Test-And-Set (TAS) [46] and Test-and-Test-and-Set (TTAS) [46] can be used in low-contention systems. Algorithms like *C-BO-MCS* [25] and *Shuffling (ShflLock)* [26] were designed to reduce the handover cost under high contention with hierarchical design or local spinning.

Moreover, the fairness is an issue that needs to be considered to better utilize the protected resources. According to the level of fairness, we define fairness as follows:

1. *Probabilistic fairness*: The chance of each task entering the critical section is the same in probability.

2. *Bounded waiting*: The number of waiting tasks does not exceed a certain multiple of the number of tasks.

Take Test-Test-And-Set (TTAS) [46] as an example. The probability of each thread obtaining a lock on a single core processor system is related to the proportion of the CPU time that the thread can acquire. In such a situation, the TTAS spinlock satisfies probabilistic fairness. Currently, GNU's Pthread library use TTAS spinlock to implement `pthread_spin_lock()`.

A spinlock algorithm is conformed to bounded waiting when it can limit the number of times that other tasks are inserted before a specific task. Ticket lock and MCS [44] are bounded waiting spinlocks. Both of them are based on first-in-first-out (FIFO) mechanism. Although FIFO allows all tasks to enter CS in a fair manner, FIFO also limits the performance of spinlocks on multi-core/NUMA machines. This is because FIFO cannot shorten the data transmission latency.

Most NUMA-aware spinlocks algorithms balance performance and fairness by preventing threads from waiting too long, but some cores may have higher computing power than others due to differences in manufacturing processes [47]. The slight difference in speed will result in the core with the advantage always being able to acquire the lock successfully. Just like in a 100-meter race, the one who gets first place is always Jamaica's Usain Bolt, even though he is only 0.1 seconds faster than the second-place runner. In modern multi-core processors like the AMD 2990WX, some cores have significantly higher lock acquisition capabilities than others. For instance, the lock acquisition capability of cores 0-7 is 20.6 times greater than that of cores 8-31 (refer to Section 6.2.2). As a result, conventional NUMA-aware algorithms may not be able to ensure equal access to the critical section for all threads/cores within a reasonable period.

With joint consideration of both throughput and fairness, we propose a spinlock method that creates one-way circular shortest path and uses this path to minimize the handover cost and ensure bounded waiting time.

3 Related Work

While TTAS spinlock [46] is a simple method to implement POSIX spinlocks in GNU (abbreviated as "Plock") and ensures the consistency of shared data, it is unfair because it tends to provide locks to neighboring cores [21]. Unfairness doubles the execution time of a multi-thread program and causes starvation as shown in [41]. It also increases the variability of latency, making it difficult to guarantee the service quality. The non-scalability of Plock is another serious problem. As shown in [22, 23], although most critical sections are short, increasing the number of cores can cause a system to collapse due to non-scalable locks.

Cohort [25] is a software framework that can combine two NUMA-oblivious locks into a scalable NUMA-aware lock. NUMA nodes compete for the global lock, and unless all threads on the NUMA node leave the CS, the NUMA node will not release the lock. Therefore, threads belonging to the same NUMA node are grouped to enter the critical section, reducing handover costs. Shuffle lock [26] and CNA [27] also use grouping to improve performance. Both are suitable for use with a Linux kernel. However, they cannot effectively reduce the latency of data transmission nor avoid unfairness in a multi-core processor. To obtain good performance under the more complex NUMA architecture, HMCS [28] is based on the concept of Cohort [25] and changes the number of lock levels from 2 to 4. The AHMCS [29] and CLoF [48] algorithms include a mechanism for managing contention and multiple locking methods, allowing different locking methods to be used in different situations. CST-semaphore and CST-mutex locks are applicable to NUMA that support parking [31].

Only dedicated threads or the threads currently holding the lock can execute the code of the CSs of each thread in [32]. In [33], the researchers further proposed turning CSs into an asynchronous execution. Although these methods can optimize data access latency to global data, they take longer to access local data because the code of a CS executes on a specific core.

Programmers can optimize software to better utilize the NoC of ccNUMA when the software uses data-level parallelization and pipe-lining [7, 34, 37, 38]. Stefan Kaestle et al proposed broadcast trees [4] to reduce the communication cost of NUMA machines. However, for multi-threaded programs that use locking mechanisms to protect shared data, these methods may not be suitable.

4 Routing On Network-on-Chip (RON)

In Section 4, we introduced RON, a NUMA-aware algorithm that is specifically designed for highly competitive and multi-core environments. In Section 5, we combined RON with simple spin locks, such as plock or ticket lock, to achieve scalability when the number of threads exceeds the number of cores. As most applications typically have more threads than cores, we utilized the RON-plock combination algorithm in our application-level benchmarks.

4.1 The Idea

In this section, we propose a design called *Routing On Network-on-Chip* (RON) that aims to minimize the handover cost between cores with low contention cost while ensuring fairness in scheduling the threads waiting to enter a CS. Minimizing handover costs can also improve the efficiency of atomic operations, which are based on atomic operations, which rely on cache coherence protocols (such as snoop+dictionary) on multi-core systems. This, in turn, can improve the performance of locks that suffer from contention.

We propose a concept of reducing the total handover cost by scheduling threads waiting to enter the CS in a specific order. This order can be compared to a train passing through all stations. The ownership of the lock is like the train, and each core is like a station. All waiting threads acquire the lock ownership in order, reducing the total handover cost. Engineers optimize train tracks to pass through all stations in the most efficient way possible, even though the route from station A to station B may not be the shortest. Please note that the train track is a *one-way circular route*. Similarly, we define a global schedule for all cores with waiting threads in the system based on the minimum total handover costs, instead of determining the scheduling order using handover costs alone. One-Way Circular Routing can often achieve global optimization. By minimizing total handover costs, we can also improve the efficiency of atomic operations, thereby improving the performance of locks that suffer from contention.

Since the code of spinlocks cannot be too complicated, it is impractical to dynamically calculate the priority of threads waiting to enter the CS. We assume that there is a thread on each core waiting to enter the CS, and then pre-calculate an optimal lock transfer path. The pre-calculate lock transfer path called the *Traveling Salesman Problem Order (TSP ORDER)* of the cores with an efficient TSP algorithm [40]. For the same processor model, the TSP ORDER is the same. RON follows the TSP ORDER to let threads that want to access shared data enter the CS one by one.

To find the TSP ORDER for a multi-core processor, we created a benchmark program to calculate the transmission latency between cores (see Section 6.2.1). Using this information, we built a fully connected weighted graph of cores and solved the TSP problem with a widely-used algorithm [40]. This allowed us to obtain the TSP ORDER that passes through all cores in the graph, which we use for lock ownership transfer to reduce the handover cost with low contention cost.

4.2 The Algorithm

Algorithm 1 presents the RON procedure for one spinlock. We use an array-based method and assume that each core has at most one thread. This method can achieve higher performance under high load compared to using a linked list (similar to MCS [44]). For each spinlock, the array-based RON not only has a "wait flag" for each core, but also places wait flags of adjacent cores, so as to increase the cache efficiency. The data structure of RON is similar to queue spinlock [24] and Linux's qspinlock with constant space complexity. However, queue spinlock [24] cannot handle the situation in which there are more threads than cores (i.e., oversubscription). In the case of oversubscription, Linux's qspinlock does not support all tasks to enter CS in the FIFO order to guarantee bounded waiting. Note that we will introduce how to support oversubscription based on an array-based RON in Section 5. It should be noted that RON is a heuristic algorithm and can provide decent solutions but cannot guarantee optimal solutions. The worst case of RON occurs in low contention scenarios where multiple cores access the same memory locations. To mitigate this issue, cache prefetching can be used to predict and fetch the data, reducing the number of cache misses and improving performance.

The first four lines of Algorithm 1 define the variables:

- `NUM_core`: This variable indicates the total number of cores on the system. It is a *system-scope variable*.
- `TSP_ID_ARRAY[]`: This array stores the mapping of each core ID to its corresponding "TSP ORDER ID" (i.e., TSP_ID), where TSP_ID is the lock transfer order of a core. When a lock is transferred to a core, the thread on this core can be checked to see whether it can enter the CS. This is a *per-process variable*, and each process can have its own routing path (TSP ORDER) because each process owns a different number of cores and can have a different TSP ORDER
- `TSP_ID`: This is the "TSP ORDER ID" of a core, and each thread has its "local version" of `TSP_ID`. Thus, each thread on a different core will get a different value when it accesses the `TSP_ID`. This is a *per-thread variable*. We used "thread_local", a C11 keyword of C language, to declare per-thread variable in Algorithm 1 (Line 2).
- `InUse`: If this is "false", there is no thread in the CS. This is a *per-lock variable*.
- `WaitArray[]`: This array is to indicate which cores' threads are waiting to enter the CS protected by this lock. When a thread wants to enter a CS, its corresponding `WaitArray[TSP_ID]` is set to 1. When the other threads set their corresponding flag in WaitArray[] to

0, the thread can enter the CS. *This is a per-lock array*. In Section 7, we will provide an algorithm for sharing WaitArray[] between different locks.

In Algorithm 1, Lines 3 and 5–8 initialize the variables. Line 5 uses `getcpu()` to get the core ID of the running thread, and uses the core ID to get the TSP ID of the core by looking up the `TSP_ID_Array[]`. The `spin_lock()` in Line 10 informs other threads that the caller thread wants to enter the CS. When no other thread is in the CS, the caller thread can enter the CS (Lines 14-16). Otherwise, it waits for the previous thread in the TSP ORDER to leave the CS (Lines 12-13). Because "checking whether there is no thread in CS (lines 19-21)" and "setting InUse (line 22)" cannot be executed atomically, it is necessary to simultaneously check InUse and waitArray[TSP_ID] in a while loop. Additionally, Line 14's `cmp_xchg()` uses TTAS, a technique commonly used in spinlock implementation to reduce coherence traffic on the cache line.

Algorithm 1 The RON Algorithm

```
1  int TSP_ID_ARRAY[NUM_core]; /*per-process*/
2  thread_local TSP_ID; /*thread-local-storage*/
3  atomic_bool InUse=false; /*per-lock*/
4  atomic_int WaitArray[NUM_core]; /*per-lock*/
5  TSP_ID = TSP_ID_ARRAY[getcpu()]
6  void spin_init()
7      for (each element in WaitArray)
8          element = 0;
9  void spin_lock()
10     WaitArray[TSP_ID]=1;
11     while(1)
12         if (WaitArray[TSP_ID]==0)
13             return;
14         if (cmp_xchg(&InUse, false, true)):
15             WaitArray[TSP_ID] = 0
16             return;
17 void spin_unlock()
18     for (int i=1; i<NUM_core; i++)
19         if (WaitArray[(i+TSP_ID)%NUM_core]==1)
20             WaitArray[(i+TSP_ID)%NUM_core]=0;
21             return;
22     InUse=false;
```

The `spin_unlock()` in Lines 18-21 finds the next thread that wants to enter the CS. Lines 18-20 treat `WaitArray[]` as a circular queue. From the next position of the caller thread (where i is between 1 and `NUM_core`.), it searches for the first thread wanting to enter the CS. Because the thread that wants to enter the CS will set `WaitArray[]` based on its `TSP_ID` (Line 10), the first thread found in the loop of Lines 18-20 is the next thread in the TSP ORDER. In Line 20, `WaitArray[]` of the next thread is set to 0, and the next thread leaves `spin_lock()` (Lines 12-13) to enter the CS. If no thread is waiting, `InUse` is set to false (Line 22).

4.3 Correctness

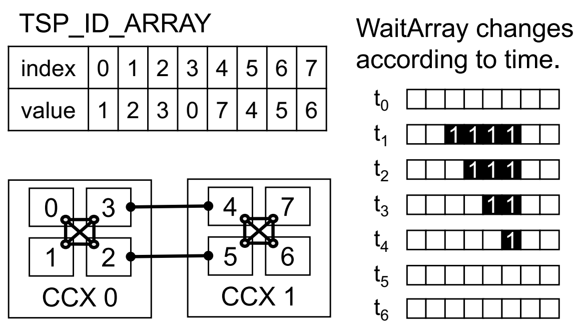
A method must satisfy the following three conditions to ensure the correctness of a CS: (1) mutual exclusion, (2) progress, and (3) bounded waiting. At a minimum, the algorithm used in a software system must satisfy conditions 1 and 2. For instance, GNU's pthread_spin_lock satisfies only conditions 1 and 2, while RON satisfies all three. However, we provide proof of bounded waiting only due to space limitations.

Bounded Waiting: We will prove that the maximum number of waits is the number of threads when each core has at most 1 thread. Each core has a unique `TSP_ID`, and these `TSP_ID`s of cores form a circular queue. RON allows all threads to enter a CS in the order of the TSP ORDER. In the worst case when thread X is ready to enter a CS, all threads on the cores whose TSP ORDERs are before the core of thread X want to enter the CS. Assuming that the total number of threads is "num," thread X needs to wait for (num - 1) threads to leave the CS. In Section 5, RON can support multiple threads on a core. In this case, the maximum number of waits is also the number of threads minus one.

4.4 An Example

RON does not prioritize threads for entering the CS based on arrival order, but instead uses the TSP ORDER of each core. While this approach may not generate the optimal solution in all cases, it provides a heuristic algorithm that works efficiently. Let us use the CPU architecture of AMD as an example to illustrate the mechanism of RON. As Figure 2 shows, two CPU CompleXes (CCXs) are connected by two point-to-point buses. Each CCX contains four cores that are fully connected by a high-speed network. First, we assume that the TSP ORDER of the cores is 3 → 0 → 1 → 2 → 5 → 6 → 7 → 4. The TSP ORDER of a core can be obtained by `TSP_ID_ARRAY[]`. Taking core 3 as an example, we can find that its TSP ORDER is 0 in TSP_ID_ARRAY[3]. We also assume that at time t0, the thread on core 3 is ready to enter the CS. Therefore, `InUse` is set to true (Line 14 in Algorithm 1) and this thread on core 3 enters the CS. Then, all entries of `WaitArray[]` in the graph are null (value 0) at time t0. At time t1, the threads on cores 1, 5, 2, and 6 arrive and are in the Lock Session (LS). Taking the thread on core 1 as an example, its TSP ORDER is `TSP_ID_ARRAY[1] = 2`. Therefore, `WaitArray[2]` is set to 1 (Line 10), and the thread waits for either `WaitArray[2]` (Line 12) or `InUse` (Line 14) to become 0. The `TSP_ID`s of cores 1, 5, 2, and 6 are 2 (`TSP_ID_ARRAY[1]`), 4 (`TSP_ID_ARRAY[5]`), 3 (`TSP_ID_ARRAY[2]`), and 5 (`TSP_ID_ARRAY[6]`), respectively, and their `WaitArray[]` values are set to 1 accordingly.

At time t2, the thread on core 3 leaves the CS. Because the TSP ORDER of core 3 is 0, the search will start from next TSP ORDER (i.e., TSP_ID 1 in this case). Thus, the



index	0	1	2	3	4	5	6	7
value	1	2	3	0	7	4	5	6

Figure 2: An example of RON.

value of `WaitArray[1]` is examined (Lines 18-21), and the first "1" appears in `WaitArray[2]`. Therefore, the thread in core 3 sets `WaitArray[2]` to 0. Since the thread in core 1 has been waiting for `WaitArray[2]` to become 0 (Line 12), it can now enter the CS. Similarly, at times t3 and t4, the threads in cores 2 and 5 enter the CS, respectively. At time t6, the thread in core 6 wants to leave the CS, so it finds all entries of `WaitArray[]` equal to 0. Therefore, it sets InUse to false (Line 22).

In this example, we assume that the handover cost within the same CCX is 1 and that across CCXs is 3. If the CS is entered in the FIFO order (3, 1, 5, 2, 6) as in MCS and Ticket, the total handover cost will be 1 + 3 + 3 + 3 = 10. However, according to RON, it will be entered in the order 3, 1, 2, 5, 6, so the total handover cost is only 1 + 1 + 3 + 1 = 6.

5 More Threads than Cores

In real applications, there may be a situation where the number of running threads is more than the number of cores. We call this oversubscription. RON approach proposed in Section 4.2 cannot handle oversubscription. In this section we propose two methods to solve this problem: RON-ticket and RON-plock. The former provides better fairness (i.e., bounded waiting), while the latter provides better performance and probabilistic fairness. In the following, we first point out that it is not necessary to run all threads with NUMA-aware spinlock algorithms in Section 5.1. By utilizing this observation without violating fairness, we present our solution on supporting oversubscription in Section 5.2.

5.1 Lock Contention Problems on a Core

In oversubscription, multiple threads can run on a single core, which differs from the situation where competing threads are spread across multiple cores. In Figure 3-(a), Thr_1 to Thr_4 correspond to $core_1$ to $core_4$. $core_1$ and $core_2$ belong to NUMA $node_1$, and the other cores belong to $node_2$. If Plock is used and Thr_4 releases the lock, Thr_3 has more probability of entering the CS because Thr_3 and Thr_4 are in the same node. When Thr_3 and Thr_4 continue to request entering the CS, then Thr_1 and Thr_2 may not have the opportunity to enter the CS. In ticket lock, these threads enter the CS in FIFO order.

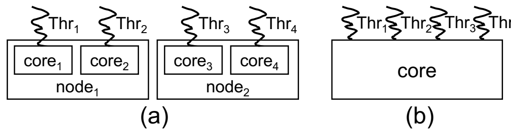


Figure 3: Threads on NUMA nodes vs. threads on a core.

Figure 3-(b) is the same as Figure 3-(a), but all threads belong to the same core. Taking Linux as an example, the execution order of threads on the same core depends on the scheduler. If Plock is used, when Thr_4 (abbreviation for thread 4) releases the lock, the next task that enters the CS is the task executed after Thr_4. Therefore, the chance of Thr_1 to Thr_3 entering the CS is proportional to their chance of getting CPU time. Because RON guarantees that each core has an equal chance of obtaining the lock, the fairness of threads obtaining the lock on different cores depends on whether the scheduler is fair. The fairness of the ticket lock is the same as in the example shown in Figure 3-(a).

5.2 RON with Oversubscription Support

In RON, the element in WaitArray indicates whether a thread on that core is waiting to enter the CS. In this section, each element of WaitArray indicates how many threads are waiting for the lock on that core (for RON-ticket and RON-plock) and the order in which they enter the CS (for RON-ticket).

The RON-ticket is given in Algorithm 2. Each lock has an array consisting of the elements corresponding to each core and the elements consist of two variables: grant and ticket.

Each core has its own nWait variable, which behaves more like thread-local storage. When a thread is waiting to enter the CS from the LS, it uses the atomic_fetch_add(nWait, 1) operation to check whether there is a thread in the CS or not. This operation is performed on the nWait variable of the core that the thread is running on. If no thread is in the CS, then the waiting thread can enter. To enter the CS, the thread uses the atomic_fetch_add(ticket, 1) operation to set the l_ticket variable (Line 6). The thread then waits on the while loop (Lines 7-10) until it is its turn to enter the CS. If the thread is not the next thread that should enter the CS of the core (that is, $grant - l\_ticket \neq 1$), the thread releases the CPU (Lines 8-9) and tries again later. When a thread leaves the CS, it first checks to see if there is any waiting thread (Line 13). If there is a waiting thread, it searches for a core with a waiting thread (Lines 14-19). Once a core with a waiting thread is found, it increases the grant of that core by 1 (Line 17), allowing the waiting thread to enter the CS.

The RON-plock is shown in Algorithm 3. Each lock has an array consisting of the elements corresponding to each core and the elements consist of two variables: numWait and lock.

Algorithm 2 The RON-ticket Algorithm

```
1  struct TicketLock {grant=0, ticket=1;}
2  atomic_int nWait=0; //per-lock variable
3  TicketLock WaitArray [NUM_CORE]; //per-lock
       variable
4  TSP_ID = TSP_ID_ARRAY[getcpu()]
5  void spin_lock()
6   if(atomic_fetch_add(&nWait, 1) == 0) return;
7   l_ticket = atomic_fetch_add(&WaitArray[TSP_ID
        ].ticket, 1);
8   while(1)
9    if(WaitArray[TSP_ID].grant-l_ticket≠1)
10       sched_yield();
11   if(l_ticket==WaitArray[TSP_ID].grant) return;
12 void spin_unlock()
13  if(atomic_fetch_sub(&nWait, 1) == 1) return;
14  next = (TSP_ID+1)%NUM_core;
15  while(1)
16     if(WaitArray[next].grant - WaitArray[next].
           ticket ≤ -2)
17         atomic_inc(&WaitArray[next].grant, 1);
18         return;
19     next = (next+1)%NUM_core;
```

Algorithm 3 The RON-plock Algorithm

```
1  struct PLock {numWait=0, lock=MUST_WAIT;}
2  atomic_bool InUse=false;//per-lock variable
3  PLock WaitArray[NUM_core];//per-lock variable
4  void lock()
5   atomic_inc(&WaitArray[TSP_ID].numWait);
6   while(1)
7    if (cmpxchg(&WaitArray[TSP_ID].lock,HAS_LOCK,
         MUST_WAIT))
8     return;
9    if (cmpxchg(&InUse, false, true))
10    return;
11 void unlock()
12  atomic_dec(&WaitArray[TSP_ID].numWait);
13   for(int i = 1;i < NUM_core+1;i++)
14    if(WaitArray[(TSP_ID+i)%NUM_core].numWait>0)
15     WaitArray[(TSP_ID+i)%NUM_core]=HAS_LOCK;
16     return;
17  InUse=false;
```

Each thread that wants to enter the CS must use atomic_inc() to set the numWait to which it belongs. When the lock of a core is HAS_LOCK, the thread currently executing on the core can enter the CS (Lines 7-8). To increase cooperation between the lock-unlock algorithm and the scheduler, `yield()` can be used when multiple threads are executing on a single core. Although `yield()` is a system call and can have overhead equivalent to `futex()`, for user-mode threads, it can be a user-mode function that transfers control to other threads on the same core. When the thread leaves the CS, it searches for the next core whose numWait is not equal to 0 and sets the lock of that core to HAS_LOCK. If necessary, `yield()` can be used again to allow other waiting threads on the same core to proceed. The proof of correctness is shown in the supplementary material.

6 Performance Evaluations

6.1 Evaluation Platform and Settings

In the performance evaluation experiments, we used a AMD Threadripper 2990WX with 64 cores (/32 physical cores) with a GNU/Linux operating system. The kernel version was 5.4. The compiler used gcc-9.3 with the optimization parameter -march=znver1 -O3, which enabled gcc-9.3 to perform the optimization for the Threadripper microarchitecture. All experiments were conducted 100 times, and their results were averaged. The source codes of RON in this section can be found at https://github.com/shiwulo/ron-osdi2023.

For a more complete comparison with other methods, we used the LiTL framework [39]. We compiled RON as a shared library. We wrote Algorithm 3 into a program that is compiled with LiTL. By using `LD_PRELOAD`, RON can be compared with other methods on different benchmarks. AMD Threadripper is a chip-NUMA. There are four dies in the chip. Each die has two CCXs, each of which has four cores. Moreover, the Linux `numastat` command shows that 2990WX has 4 NUMA nodes.

The cache coherence protocol operates at the cache line granularity, which means that low latency also implies high bandwidth. Therefore, the transmission latency obtained from the experiments shown in Figure 4 not only informs the design of inter-core locking algorithms but also provides insights into the underlying hardware's performance characteristics. By profiling the inter-core latency, an operating system can optimize the lock-unlock algorithms accordingly. Furthermore, detailed microarchitecture information about the NoC from CPU vendors can lead to even better performance. In calculating the transmission latency from core X to core Y, we make core X read 100 integers (2 cache lines in this case) from DRAM, and then we calculate the time for core Y to read the 100 integers from core X's cache. As 2990WX is a ccNUMA architecture, Y will read 100 integers from X's cache. It should be noted that not all dies on a 2990WX are the same due to differences in the manufacturing process. AMD puts the best cores on die 0, which means that the transmission latency of die 0 is lower. AMD and Intel support "AMD Turbo Core" and "Turbo Boost Max 3.0", respectively. The operating system can learn how to make better use of the CPU by being aware of the best die. However, traditional NUMA-aware spinlocks cannot achieve the fairness they claim in such processors, which will be discussed in Section 6.2.2.

After obtaining the handover time (i.e., transmission speed) for each pair of cores, we used Google OR-Tools [40] (A solver for NP-complete problems, providing a usable solution.) to determine the TSP ORDER for the cores as shown in Figure 4. We see that the TSP-ORDER first visits all the cores in the same CCX and then the CCXs on the same die. Finally, the TSP-ORDER visits each die in the clockwise direction. Then, we generate TSP_ID_ARRAY[] according to the TSP ORDER for Algorithm 1.

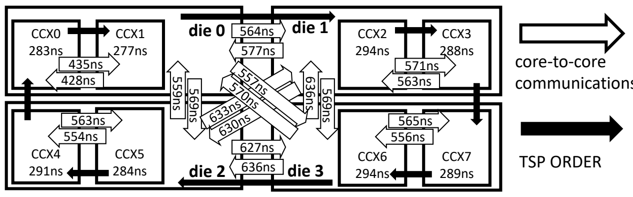


Figure 4: The core-to-core communication latencies and TSP ORDER of AMD 2990WX

We used microbenchmarks in Section 6.2 to analyze the characteristics of the algorithm, and used general benchmarks in Section 6.3 to understand the performance under various usage scenarios. We compared RON with the following algorithms. Please note that what we describe below are the performance characteristics of each algorithm, not the implementation details.

1. Plock: The GNU Pthread's spinlock. A thread that intends to enter a CS will test the lock until its value equals to 0. When a thread leaves a CS, it sets the lock to 0. The first core that observes that the lock is 0 can enter the CS. The closer to the core the lock is released, the more likely it is for the core to enter the CS.

2. Ticket: This method allows each task waiting to enter the CS to have a "ticket" number. The thread waits until the "grant" is equal to its ticket number. The wait loops of all waiting threads use atomic instructions to continually query the value of the "grant", which consumes limited NoC bandwidth.

3. MCS: Because all tasks waiting to enter the CS are queued in a linked list, when a thread leaves the CS, it only needs to set the "wait flag" of the next task to false. Setting the wait flag of next thread is more efficient than multicasting when the CPU supports a directory cache coherence algorithm. MCS does not optimize the interconnect latency in multi-core architectures.

4. C-BO-MCS: The thread should first acquire the MCS lock of the NUMA node to which the thread belongs. Then, it must compete with threads on other NUMA nodes to obtain a back-off lock. If a core neighbors to the core that obtains the C-BO-MCS lock, it has a higher priority to enter the CS. With this method, threads belonging to the same node can be grouped together to reduce handover costs.

5. ShflLock (also known as Shuffle Lock): This also uses grouping to improve performance. Shuffle can specify that a thread in the queue is responsible for shuffling. However, when the task that is allowed to enter the CS is shuffling the queue, the thread cannot enter the CS immediately and system performance may decrease.

6.2 Microbenchmarks for Quantitative Analysis

6.2.1 Evaluation Platform and Settings

Here, we analyzed each spinlock method in a quantitative manner through a controllable microbenchmark. In each set of experiments in this section, each thread is bound (i.e., `sched_setaffinity()`) to a hardware thread and executes Algorithm 4. Because we have SMT (Simultaneous multithreading) enabled, there are 2 hardware threads per core. The total number of software threads is 64. In the while loop (Lines 2–9), a thread in the *lock section* (LS) (Line 3) requests entry into the *critical section* (CS) (Lines 4-5). After the thread enters the CS, each entry in SharedData is read and written, and the lock is released into the *unlock section* (US) (Line 6) when the thread leaves the CS. The `clock_gettime()`, defined in the POSIX.1-2001 standard, is called in the *non-critical section* (nCS) (Lines 7-9) until the elapsed time of the nCS exceeds the value of nCS_size±15% in Line 9. We first evaluate the throughput (Figure 5) and fairness (Figure 6) of each algorithm, and then analyze their efficiency in terms of handover (Figure 7) and contention (Figure 8). Please note that in these 4 experiments, except for adding the code for measuring time (i.e., `clock_gettime()`) and the code for statistics, the experimental parameters are the same.

Algorithm 4 Testing Program and Measurements

```
1  void thread():
2      while(1):
3          spin_lock();       //LS
4          for (each element in SharedData): //CS
5              element = element + 1;        //CS
6          spin_unlock();    //US
7          t = clock_gettime(); //nCS
8          //syscall overhead, rdtscp implement in
               userspace
9          while(clock_gettime()-t > nCS_size*rand
                (0.85~1.15));
```

6.2.2 Results of Microbenchmarks

As shown in Algorithm 4, a shorter nCS implies a heavier workload because the lock request rate is higher. The upper and lower parts of Figure 5 are the performance when the contention is low and high, respectively. RON can provide the best locking efficiency in both cases. Under low load conditions (nCS = 400K∼120K), the performance of Plock is second only to RON. When nCS is lower than 40K, the performance of Plock drops rapidly when the load is heavy. The performance curves of MCS and ShflLock are similar, which may be because they queue tasks waiting for CS in a linked list.

ShflLock and C-BO-MCS perform worse than MCS in some cases (i.e., nCS > 10K). This is because these two algorithms implicitly treat the handover costs between cores belonging to the same die as equal. Therefore, they cannot optimize the handover costs of different cores in different dies (Please see the example in Section 2.2). Further, since the difference in communication cost between 2990WX cores is only 3.13 times at most, an algorithm with too-high time complexity reduces the benefits that can be obtained. Because RON uses a pre-calculated TSP ORDER that is optimized for all cores, it can achieve higher performance at a lower cost. According to our simulation results (in the supplementary material), ShflLock and C-BO-MCS performs better when the transmission latency between cores on the same chip/die is almost the same. In other situations, RON performed best.

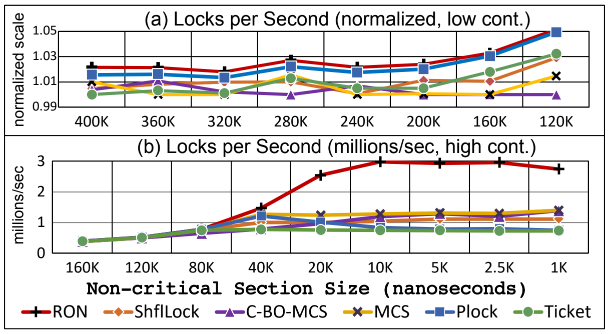


Figure 5: Locks per second under different loads

Figure 6 shows the number of locks acquired by each core in the case of short-term (1 second) and long-term (10 seconds). The lower the coefficient of variation (CV), the better the fairness. In Figure 6, we see that RON, MCS, and Ticket perform equally well, in terms of CV. That is almost equal to 1%. In long-term fairness, when non-critical section (nCS) < 80K ns, the CV of Plock starts to rise. When nCS < 20K ns, the CV of ShflLock and C-BO-MCS both starts to rise. In order to better understand the performance of spinlock algorithms in long-term fairness, we let the ShflLock, C-BO-MCS, and Plock execute for 1,000 seconds with nCS = 10K and their CVs are 35%, 71% and 96%, respectively. *Fairness factor* is described by Dice et al. [25]. It is the most common metric to measure fairness. The value of fairness factor is between 0.5 and 1. A complete fair spinlock's factor is 0.5 and a complete unfair spinlock's factor is 1. The fairness factor of the ShflLock, C-BO-MCS andPlock are 0.68, 0.85 and 0.8, respectively.

In terms of software design, each thread in Plock competes fairly for locks. C-BO-MCS is based on two fair spinlocks, namely backoff [44] and MCS. ShflLock allows threads on the same node (i.e., die) of the lock holder to elevate their positions in the queue for a limited number of times. Note that it is difficult to analyze in detail why these algorithms do not meet long-term fairness perfectly, so only Plocks is analyzed to provide insights into the interaction between multicore processors and spinlocks.

In the past, the multi-core processor had to execute at a frequency that all cores could run correctly. The worst core determines the maximum clock frequency that a multi-core processor can run. Now each core can run on its highest frequency [47]. According to the experimental results of the Plock with nCS=10k, the ability of cores 0-7 and 32-39 to obtain locks is 20.6 times that of cores 8-31 and 40-63. Therefore, we roughly conclude that when the load becomes heavier, algorithms that meet long-term fairness may not achieve the expected fairness on modern multi-core processors. [47].

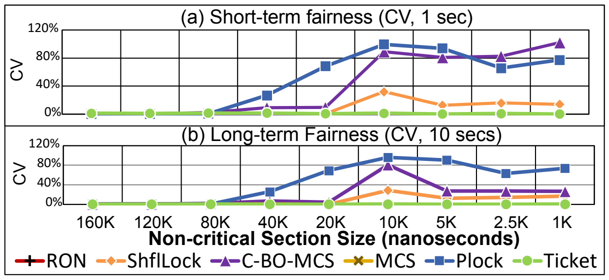


Figure 6: Long-/short- fairness of different algorithms.

The experimental parameters of Figure 7 are the same as Figures 5 and 6, but we changed the X coordinate from nCS (Line 8 in Algo. 4) to the number of threads waiting to enter CS (i.e., the number of threads in LS, Line 3 in Algo. 4). The more the number of threads waiting, the better the performance of a algorithm that can optimize the handover cost. In Figure 7, the Y axis is the time required to access the shared data. For example, in the case of RON under load nCS = 10K, the number of threads in LS is 45, and the handover time is 100 ns. Under the same load, the number of threads of C-BO-MCS in LS is 52, and handover time is 190 ns. RON is almost the best spinlock in terms of handover time. With more tasks in the LS, the path formed by each task selected by RON is closer to TSP ORDER, because each core has a higher probability to have a thread waiting for entering the CS. When the number of tasks in LS increases from 0 to 15, the efficiency of accessing shared data doubles (from 210ns to 100ns). When the LS changes from 60 to 62, the efficiency is reduced by 7%. This is the reason for the reduced efficiency when the nCS is 1K in Figure 5.

Plock is slightly better (0.07%) than RON when the lock contention is very low (nCS=120K). Although Plock's handover cost is low, its performance is not good. Since Plock uses `cmp_xchg` (`compare_exchange`) to solve the lock contention problem, The hardware may need to execute `cmp_xchg` continuously until the return value of only one task is equal to true. This wastes the limited bandwidth of

NoC and is time consuming. The handover time of C-BO-MCS and ShflLock is better than MCS. However, these two methods are too complicated, resulting in the performance lower than expected. Both Ticket and MCS arrange tasks to enter the CS in the order of their arrival. Since Ticket does not give each thread a wait flag, all threads will constantly monitor the wait flag, thus consuming a lot of NoC bandwidth. Ticket has the worst handover cost.

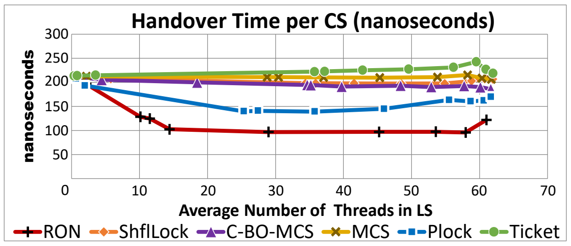


Figure 7: Handover cost and the number of thread in LS.

In Figure 8, we set the size of the shared data accessed in the critical section to 0 (Line 4 in Algorithm 4). The X axis is the number of tasks waiting to enter the CS, and the Y axis is the time of each thread executing one round (including LS, US, CS and nCS, i.e., Lines 2-8). The size of the non-critical section (Lines 7 and 8) ranges from 160K to 1K. Therefore, the main factor in performance is the locking and unlocking efficiency of an algorithm. RON is almost the best algorithm. Its performance is slightly worse than that of Plock (0.2%) when the loading is extremely light. RON has a better performance for two reasons. First, TSP ORDER is pre-calculated. Second, the lower handover cost makes atomic operations more efficient. We use experiments to analyze the efficiency of atomic operations of RON. When we schedule threads to perform atomic operations through TSP ORDER, the efficiency of atomic operations is 1.6 times that of random order.

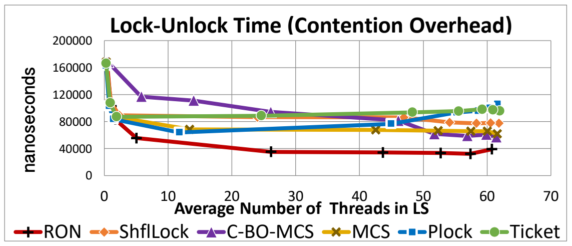


Figure 8: Contention cost.

6.2.3 Oversubscribe

In each set of experiments of this section, each thread binds (i.e., `sched_setaffinity()`) to a core and executes Algorithm 4. Each core has at most $\lceil num\_thread \div num\_core \rceil$ threads. Because RON-ticket shares a key property with RON, that is, bounded waiting, RON-ticket was used for performance evaluation in the previous section. In this section, microbenchmarks are used to evaluate the performance of RON-ticket and RON-plock. In the case of oversubscription, two factors affect performance. The first one is whether the thread holding the lock is scheduled out. Second, if the algorithm specifies the next thread entering the CS, and whether it is scheduled out.

In Figure 9, RON-plock and RON-ticket perform better in the case of overbooking, where the y-axis denotes "millions locks" per second. Although C-BO-MCS(-B) and ShflLock(-B) also support oversubscription, the number of lock-unlock operations per second is dropped quickly.

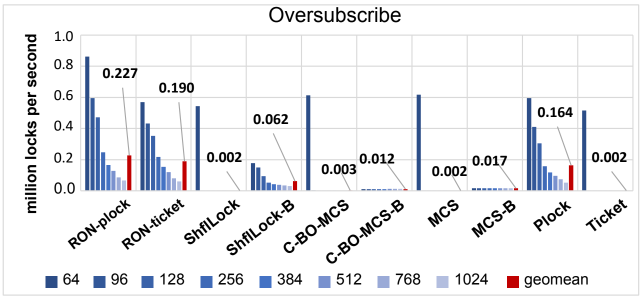


Figure 9: Performance of algorithms under oversubscribe.

RON-plock and Plock use intuitive methods (e.g., test-test-and-set) to solve the problem of oversubscribe. As long as the lock-holder is not scheduled out, Plock will allow a thread to enter CS (it is because that all threads wait on the same variable.). RON-plock is similar to Plock, except that all threads on the next core are scheduled out. Because RON-plock is based on RON, the performance of RON-plock is better than Plock. RON-ticket, ShflLock-B, and C-BO-MCS-B use system calls (i.e., futex() and yield()) to prevent the thread from spinning meaninglessly. ShflLock-B's unlock() directly wakes up the next thread. However, C-BO-MCS-B's unlock() may wake up all threads that can enter CS. RON-ticket makes the next thread that can enter the CS busy waiting, and other threads on that core enter the sleep state. For the same reason as RON-plock, RON-ticket has better performance.

6.2.4 Scalability

In this section, we investigate how algorithm performance changes with an increase in the number of threads used. Our experiment was conducted on the 2990WX, which has SMT technology. During the experiment, each thread accesses 100 integers in the critical section, while the non-critical section takes 10,000 $\pm 15\%$ nanoseconds.

As shown in Figure 10, it indicate that the performance of the RON-plock improves with an increase in the number of threads when the thread count is less than 64. This experiment yields results similar to those in Figure 5 because "executing more threads simultaneously" and "having shorter non-critical sections" both lead to greater competition for entering the critical section. However, when the number of threads exceeds 64, the performance of these algorithms is determined by their ability to handle the oversubscribe problem.

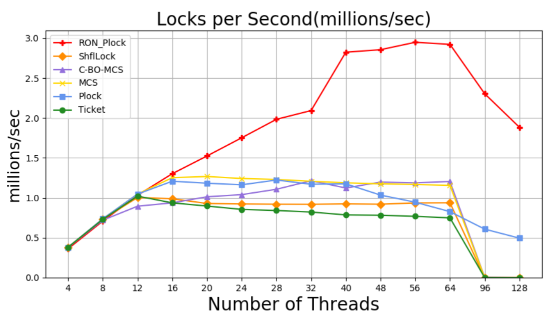


Figure 10: Locks per second under different number of threads

6.3 Application-level Benchmarks

We pick five different application-level benchmarks representing different performance bottlenecks. For the consistency of the experiment, the RON implementation here uses RON-ticket, which has identical features to RON (bounded waiting).

6.3.1 LevelDB (Key-value Database)

Here, we used Google's LevelDB to test the performance of the spinlock. The Horizontal axis of Figure 11 is the algorithm tested, and the vertical axis is the time cost for every operation reported by LevelDB. Because of the difference data scales, "fillsync" is normalized to MCS and others normalized to Ticket.

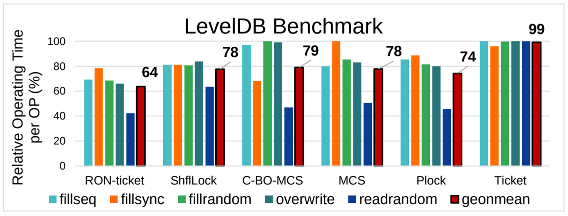


Figure 11: Google's LevelDB.

We use the db_bench in LevelDB to evaluate performance with 1 million entries and 64 threads. For each spinlock, fillseq, fillsync, fillrandom, overwrite, and readrandom have been tested. The last one is the geometric average of LevelDB's 5 tests. RON-ticket, ShflLock and C-BO-MCS are spin locks optimized for ccNUMA or NUMA. Please note that RON-ticket is RON with oversubscribe support and it also satisfies bounded waiting. Compared with ShflLock and C-BO-MCS, the performance of RON is better by 22.1% and 24.2%, respectively.

MCS is slightly better than ShflLock and C-BO-MCS for LevelDB, although the latter two are optimized for NUMA. This may be because these two algorithms are designed to overcome the huge transmission overhead between two CPUs. However, there is not much difference in the communication cost between the cores on AMD 2990WX. When the load is high, ShflLock and C-BO-MCS may only perform local optimization.

Consider the situation with four NUMA nodes, where ShflLock and C-BO-MCS serve node X, and the load on node X always has a thread waiting to enter the CS. At this time, although there are many threads waiting for the CS on other NUMA nodes to enter, ShflLock and C-BO-MCS tend to let tasks on node X enter the CS. Since RON uses TSP ORDER to arrange the cores to enter the CS, RON does not suffer from the problem of local optimization.

6.3.2 Benchmarks in Different Contention Levels

We applied an additional four different application benchmarks to evaluate the performance of different algorithms. These algorithms are selected from LiTL [39] and cover both high and low contention scenarios. Volrend and Raytrace are from the SPLASH2x benchmark set representing extreme and high levels of contention, respectively. For the extreme level, more than 40 threads are waiting to acquire the same lock instance. For the high level, there are about 10 to 40 threads waiting to acquire a lock. Dedup and Ferret are from the PARSEC3.0 benchmark set and respectively represent pressure on the memory and relatively low levels of contention [39]. In Figure 12, the vertical axis is the elapsed time of the benchmark task (including geomean of LevelDB Figure 11). Because of the different data scales, the numbers are the percentage to where the algorithm performs the worst for each task.

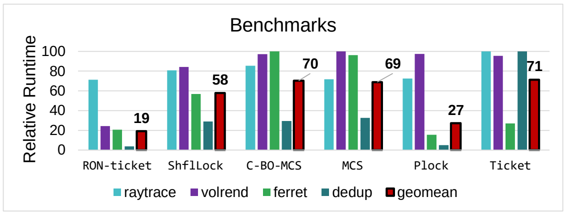


Figure 12: Applications with different contention level.

The bottleneck of Raytrace is a lock contention, protecting a single counter with about a million acquisitions every second. RON-ticket, MCS, and Plock accomplished the task with around 70% of elapsed time. MCS is optimized for multi-core systems with dedicated caches for each core to reduce the overhead of lock contention and well fitted in high level of contention. The code of Plock is not optimized for multi-core. However, the core adjacent to the core that released the lock is more likely to successfully perform the atomic operation compare_exchange() to acquire the lock. Thus, Plock is implicitly optimized for multi-core platforms.

In the case of extreme levels of contention, the performance of Plock and MCS starts to drop while the ShflLock and RON-ticket can handle the stress. Under extreme level of contention (Volrend), RON-ticket achieved its best performance, taking only 24.3% of the elapsed time of the ticket to accomplish task. The bottleneck of Volrend is the lock contention for protecting different task queues with around 40 threads waiting. This benchmark verified that the RON algorithm generally performs best under higher contention. With more cores possessed by threads spinning for the lock instance, the routing path can massively reduce the handover cost.

However, under low levels of contention, RON-ticket only performs second best in all six algorithms. Ferret is a parallelized software with about 2000 times of acquisition for every second. While RON-ticket uses around 20% of the elapsed time, Plock takes only around 15.6% of the elapsed time of C-BO-MCS, outperforming RON in this specific benchmark. The lower contention of the lock leads to the sparseness of the `WaitArray`, which results in leaping on the routing path and lowers the benefit. Ticket guarantees fairness as threads keeping querying the global variable to know whether they can enter the CS. Ticket fits the task with low level of contention. However, under higher pressure, the bandwidth consumed by lock contention limits the bandwidth that can be used by handover.

Moreover, according to the results of Dedup, RON-ticket and Plock gave low memory pressure. Dedup allocates numerous locks (266k) [39], which puts pressure on the memory if the components of the lock are not reusable. The reusability of components like WaitArray and `Get_TSP_ID` gives RON-ticket the ability to handle numerous lock allocations.

In summary, RON algorithms can handle different levels of contention, especially higher levels. With higher contention RON algorithms achieve better performance relatively, but Plock remains a better algorithm for low levels of contention. With reusable components, Both Plock and RON put low pressure on the memory while allocating numerous lock instance.

7 RON in GNU/Linux Kernel

7.1 Implementation

As shown in the performance evaluation section, RON is more suitable for multi-core computers than methods that support NUMA in user space applications. In this section, we shows whether RON is suitable for Linux kernel. In our implementation, the line of code (LoC) is 47.

In the Linux kernel, the lock-acquire and lock-release are implemented by `queued_spin_lock(struct qspinlock *lock)` and `queued_spin_unlock(struct qspinlock *lock)`, respectively. Both functions have only one parameter, `lock`. By rewriting these two functions, we implement RON in the Linux kernel. We use `*lock` as `InUse` in the RON algorithm (Line 3 in Algorithm 1).

In order to achieve the same space efficiency as qspinlock, only one `WaitArray` (Line 4 in Algorithm 1) is in kernel. In user space, a task sets `WaitArray[TSP_ID]` (Line 10 in Algorithm 1) to wait for entering the CS. In the Linux kernel, the task writes the address of `lock` (that is the parameter of `queued_spin_lock`) to `WaitArray[TSP_ID]` for entering the CS. When the thread leaves CS, the thread will check one by one whether there is an element with a value equal to `lock` in `WaitArray` . Therefore, a busy-waiting task is only awakened by the task holding the same `lock`.

If the space of the `WaitArray` has been used up, the other tasks wait on `InUse` (that is `*lock` in kernel space). Tasks waiting for `InUse` do not enter CS in TSP ORDER. This design method is the same as Linux's qspinlock, though it is not perfect but good enough (compromise to O(1) space complexity). In terms of memory usage, RON requires 4 bytes for each lock (that is the size of `struct qspinlcok`) and 28 bytes for each core ($28 \times 64 = 1792$ bytes for AMD 2990WX).

7.2 Evaluations

In this section, the Linux kernel version is 5.12.1. We apply the patch of ShflLock into the `qspinlock.c` of Linux. Therefore, in this section, we will compare the performance of the Linux kernel using qspinlock, ShuflLock and RON. We use a microbenchmark and `db_bench` of Google LevelDB to measure the performance of RON in Linux kernel. In the experiment, we do not use `LD_PRELOAD` to change the behavior of LevelDB. The purpose of microbenchmark is to measure performance under high load conditions. The microbenchmark is implemented by forking 64 child processes, and every child process creates 2048 threads to execute 64 `mmap()` and `munmap()` function calls. We use `strace` to evaluate the time taken for each system call.

As shown in Figure 13, ShflLock doesn't perform well on both microbenchmark and LevelDB. ShflLock is suitable for multi-socket NUMA machines but ours is a single-socket machine. RON performs better than qspinlock under high load conditions. In terms of geometric average of microbenchmark, the performance of Linux kernel with RON is 1.35 times that of Linux kernel with qspinlock. In terms of LevelDB, RON and qspinlock are about the same in terms of geometric average. In these five experiments, the performance of these three algorithms on readseq are almost the same. RON performs better than qspinlock in fillsync bacause the contention is high. RON and qspinlock both have their own strengths.

Intuitively we can combine qspinlock and RON to achieve better performance. qspinlocks can encode the first two tasks that want to enter CS into the `lock` variable (the parameter of `queued_spin_lock()`). In this way, the performance of qspinlock is very good under low contention. The purpose of this experiment is to explore the effectiveness of RON, so we did not use Linux optimization techniques to improve performance under low contention conditions.

	microbenchmark (microseconds)				
system call	mmap	clone	mprotect	munmap	geomean
qspinlock	923	183	252	43	207
ShflLock	2029	206	456	39	294
RON	592	185	264	19	153

(a)

	LevelDB					
	fillseq	fillsync	fillrandom	overwrite	readseq	geomean
	MB/sec	op/sec	MB/sec	MB/sec	MB/sec	
qspinlock	20.2	511.4	17.8	17.2	487.3	68.8
ShflLock	7.9	390.5	7.5	7.1	485.3	38.1
RON	18.3	687.1	16.3	15.7	482.7	68.9

(b)

Figure 13: Performance comparisons on Linux kernel.

8 Conclusion

We propose a RON spinlock algorithm that delivers locks and data in a one-way circular manner among cores with the awareness of the performance differences of cores, so as to minimize the system-level handover cost and achieve bounded waiting for threads among cores. In particular, "one-way" is for minimized system-level handover cost and "circular" is for bounded waiting of threads to enter CS. In addition, the proposed RON algorithm can also resolve the oversubscription issue without losing its scalability. A series of experiments were conducted to evaluate the efficacy of the proposed algorithm. Compared with ShflLock and C-BO-MCS, the performance of RON in google leveldb has increased by 22.1% and 24.2% respectively. In terms of kernel space performance, compared with using ShflLock, RON can improve the performance of Google LevelDB by 1.8 times.

9 Future work

This paper addresses the issue of unfairness caused by different execution frequencies on multi-core processors, as well as the efficiency of inter-core data transfer. The proposed method is particularly suitable for highly competitive scenarios. Although high competition can be a bottleneck for performance, low competition is a more common scenario where simple algorithms often have good performance. Therefore, in future research, we will investigate how to dynamically switch algorithms (such as plock and RON) at runtime. We will also evaluate the performance of RON by implementing it using linked-list methods to offload the runtime of unlocking to the locking process.

Acknowledgement

We thank our shepherd, Aurojit Panda, and the anonymous reviewers for their valuable feedback and suggestions. This work was supported in part by Ministry of Science and Technology (MOST) of Taiwan under grant nos. 111-2221-E-194 -017 -MY3, 111-2223-E-001-001, 111-2923-E-002-014-MY3, 111-2221-E-001-013-MY3, and 112-2927-I-001-508.

References

[1] L. T. Su, S. Naffziger and M. Papermaster, "Multi-chip technologies to unleash computing performance gains over the next decade," 2017 IEEE International Electron Devices Meeting (IEDM), 2017, pp. 1.1.1-1.1.8, doi: 10.1109/IEDM.2017.8268306.

[2] Nicolas Viennot (Sep 19, 2022). Measuring CPU core-to-core latency. https://github.com/nviennot/core-to-core-latency

[3] Sally Ward-Foxton (08.19.2021). Intel Brings Chiplets to Data Center CPUs. EETimes. https://www.eetimes.com/intel-brings-chiplets-to-data-center-cpus/

[4] Stefan Kaestle, Reto Achermann, Roni Haecki, Moritz Hoffmann, Sabela Ramos, Timothy Roscoe: Machine-Aware Atomic Broadcast Trees for Multicores. OSDI 2016: 33-48

[5] https://www.freebsd.org/cgi/man.cgi?query=atomic&sektion=9&format=html

[6] Daniel Sorin; Mark Hill; David Wood, "A Primer on Memory Consistency and Cache Coherence," Morgan & Claypool, 2011.

[7] Pradip Kumar Sahu and Santanu Chattopadhyay. 2013. A survey on application mapping strategies for Network-on-Chip design. J. Syst. Archit. 59, 1 (January, 2013), 60–76. DOI:https://doi.org/10.1016/j.sysarc.2012.10.004

[8] Rajesh Chopra, Yang-Trung, LinSailesh Kumar, "Generating physically aware network-on-chip design from a physical system-on-chip specification, " US Patents US10218580B2, Application granted in 2019.

[9] C. Wu et al., "A Multi-Objective Model Oriented Mapping Approach for NoC-based Computing Systems," in IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 3, pp. 662-676, 1 March 2017.

[10] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta and J. Hennessy, "The directory-based cache coherence protocol for the DASH multiprocessor," In Proceedings of the 17th Annual International Symposium on Computer Architecture (ISoA), Seattle, WA, USA, pp. 148-159, 1990.

[11] Chinya Ravishankar, James Goodman, "Cache Implementation for Multiple Microprocessors,"in Proceedings of IEEE Computer Conference, pp. 346–350, Feb 1983.

[12] Ruibo Wang, Kai Lu, and Xicheng Lu. 2009. Investigating transactional memory performance on ccNUMA machines. In Proceedings of the 18th ACM international symposium on High performance distributed computing (HPDC '09). Association for Computing Machinery, New York, NY, USA, 67–68. DOI:https://doi.org/10.1145/1551609.1551625

[13] R. R. Iyer and L. N. Bhuyan, "Design and evaluation of a switch cache architecture for CC-NUMA multiprocessors," in IEEE Transactions on Computers, vol. 49, no. 8, pp. 779-797, Aug. 2000, doi: 10.1109/12.868025.

[14] K. A. Bowman, A. R. Alameldeen, S. T. Srinivasan and C. B. Wilkerson, "Impact of die-to-die and within-die parameter variations on the throughput distribution of multi-core processors," Proceedings of the 2007 international symposium on Low power electronics and design (ISLPED '07), 2007, pp. 50-55, doi: 10.1145/1283780.1283792.

[15] "Ampere® Altra® offers up to 80 cores at up to 3.0 GHz", 80 cores, https://amperecomputing.com/altra/

[16] " AMD EPYC™ 7003 Series Processors scale from 8 to 64 cores", 64 cores, https://www.amd.com/en/processors/epyc-7003-series

[17] "Intel® Xeon® Platinum 8380 Processor (60M Cache, 2.30 GHz)", 40 cores,https://www.intel.com/content/www/us/en/products/details/processors/xeon/scalable/platinum.html

[18] "Arm-based AWS Graviton2 processors", 64 vCPU, https://aws.amazon.com/tw/ec2/instance-types/x2/

[19] Abdul Naeem, Xiaowen Chen, Zhonghai Lu, Axel Jantsch. "Scalability of relaxed consistency models in NoC based multicore architectures". ACM SIGARCH Computer Architecture News. April 2010.

[20] B. K. Daya et al., "SCORPIO: A 36-core research chip demonstrating snoopy coherence on a scalable mesh NoC with in-network ordering," 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), 2014, pp. 25-36, doi: 10.1109/ISCA.2014.6853232.

[21] scientiaesthete. 2012 "pthreads: thread starvation causedby quick re-locking", Retrieved June 20, 2019 fromhttps://stackoverflow.com/questions/12685112/pthreads-thread-starvation-caused-by-quick-re-locking

[22] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris and Nickolai Zeldovich, "Non-scalable locks are dangerous", in Proceedings of the Linux Symposium (OLS2012), Ottawa, Canada, July 2012.

[23] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An Analysis of Linux Scalability to Many Cores. In Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI). USENIX Association, Vancouver, Canada, 1–16.

[24] J. M. Mellor-Crummey and M. L. Scott. "Algorithms for scalable synchronization on shared-memory multi-processors," ACM Transactions on Computer Systems,9(1):21–65, 1991.

[25] David Dice, Virendra J. Marathe, and Nir Shavit. 2015. Lock Cohorting: A General Technique for Designing NUMA Locks. ACM Trans. Parallel Comput. 1, 2, Article 13 (January 2015), 42 pages. DOI:https://doi.org/10.1145/2686884

[26] Sanidhya Kashyap, Irina Calciu, Xiaohe Cheng, Changwoo Min, and Taesoo Kim. 2019. Scalable and practical locking with shuffling. In Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19). Association for Computing Machinery, New York, NY, USA, 586–599. DOI:https://doi.org/10.1145/3341301.3359629

[27] Dave Dice and Alex Kogan. 2019. Compact NUMA-aware Locks. In Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19). Association for Computing Machinery, New York, NY, USA, Article 12, 1–15. DOI:https://doi.org/10.1145/3302424.3303984

[28] Milind Chabbi, Michael Fagan, and John Mellor-Crummey. 2015. High performance locks for multi-level NUMA systems. SIGPLAN Not. 50, 8 (August 2015), 215–226. DOI:https://doi.org/10.1145/2858788.2688503

[29] Milind Chabbi and John Mellor-Crummey. 2016. Contention-conscious, locality-preserving locks. SIGPLAN Not. 51, 8, Article 22 (August 2016), 14 pages. DOI:https://doi.org/10.1145/3016078.2851166

[30] Dave Dice, Virendra J. Marathe, and Nir Shavit. 2011. Flat-combining NUMA locks. In Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures (SPAA '11). Association for Computing Machinery, New York, NY, USA, 65–74. DOI:https://doi.org/10.1145/1989493.1989502

[31] S Kashyap, C Min, T Kim, Scalable numa-aware blocking synchronization primitives, USENIX Annual Technical Conference, 2017

[32] LOZI, J., DAVID, F., THOMAS, G., LAWALL, J., and MULLER, G. "Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multi-

threaded Applications." USENIX Annual Technical Conference '12.

[33] David Klaftengger, Konstantinos Sagonas and Kjell Winblad, "Queue Delegation Locking", IEEE Transaction Parallel and Distributed Systems, vol. 29, no. 3, pp.687-704, March 2018.

[34] B. K. Joardar, R. G. Kim, J. R. Doppa, P. P. Pande, D. Marculescu and R. Marculescu, "Learning-Based Application-Agnostic 3D NoC Design for Heterogeneous Manycore Systems," in IEEE Transactions on Computers, vol. 68, no. 6, pp. 852-866, 1 June 2019, doi: 10.1109/TC.2018.2889053.

[35] W. Amin et al., "Performance Evaluation of Application Mapping Approaches for Network-on-Chip Designs," in IEEE Access, vol. 8, pp. 63607-63631, 2020, doi: 10.1109/ACCESS.2020.2982675.

[36] S. Das, J. R. Doppa, P. P. Pande and K. Chakrabarty, "Monolithic 3D-Enabled High Performance and Energy Efficient Network-on-Chip," 2017 IEEE International Conference on Computer Design (ICCD), 2017, pp. 233-240, doi: 10.1109/ICCD.2017.43.

[37] C. Wu et al., "A Multi-Objective Model Oriented Mapping Approach for NoC-based Computing Systems," in IEEE Transactions on Parallel and Distributed Systems,vol. 28, no. 3, pp. 662-676, 1 March 2017.

[38] Aryan Deshwal, Nitthilan Kanappan Jayakodi, Biresh Kumar Joardar, Janardhan Rao Doppa, and Partha Pratim Pande. 2019. MOOS: A Multi-Objective Design Space Exploration and Optimization Framework for NoC Enabled Manycore Systems. ACM Trans. Embed. Comput. Syst. 18, 5s, Article 77 (October 2019), 23 pages. DOI:https://doi.org/10.1145/3358206

[39] Rachid Guerraoui, Hugo Guiroux, Renaud Lachaize,Vivien Quéma, and Vasileios Trigonakis. "Lock–Unlock:Is That All? A Pragmatic Analysis of Locking in Soft-ware Systems," ACM Transactions on Computer Systems,Volume 36 Issue 1, March 2019.

[40] "OR-Tools | Google Developers". Retrieved June, 28, 2019 from https://developers.google.com/optimization/

[41] Jonathan Corbet, "Ticket spinlocks," Retrieved from https://lwn.net/Articles/267968/

[42] J. Hennessey and D. Patterson, Computer Architecture: A Quantitative Approach. Morgan Kaufmann, 2017.

[43] David P. Reed and Rajendra K. Kanodia. 1979. Synchronization with event counts and sequences. Communications of the ACM 22, 2 (1979), 115–123. DOI:https://doi.org/10.1145/359060.359076

[44] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems 9, 1 (1991), 21–65. DOI:https://doi.org/10.1145/103727.103729

[45] D. M. Tullsen, S. J. Eggers and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," Proceedings 22nd Annual International Symposium on Computer Architecture, 1995, pp. 392-403.

[46] Thomas E. Anderson. 1990. The performance of spin lock alternatives for shared-memory multiprocessors. IEEE Transactions on Parallel and Distributed Systems 1, 1 (1990), 6–16. DOI:https://doi.org/10.1109/71.80120

[47] Btarunr. Windows 10 2H19 Update to Have "Favored Core" Awareness, Increase Single-threaded Performance. online https://www.techpowerup.com/259688/windows-10-2h19-update-to-have-favored-core-awareness-increase-single-threaded-performance

[48] Rafael Lourenco de Lima Chehab, Antonio Paolillo, Diogo Behrens, Ming Fu, Hermann Härtig, Haibo Chen. CLoF: A Compositional Lock Framework for Multi-level NUMA Systems. Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, October 2021, Pages 851–865. DOI: https://doi.org/10.1145/3477132.3483557