Be My Guest – MCS Lock Now Welcomes Guests

Tianzheng Wang [*]
University of Toronto
tzwang@cs.toronto.edu

Milind Chabbi
Hewlett Packard Labs
milind.chabbi@hpe.com

Hideaki Kimura
Hewlett Packard Labs
hideaki.kimura@hpe.com

Abstract

The MCS lock is one of the most prevalent queuing locks. It provides fair scheduling and high performance on massively parallel systems. However, the MCS lock mandates a *bring-your-own-context* policy: each lock user must provide an additional context (i.e., a queue node) to interact with the lock. This paper proposes MCSg, a variant of the MCS lock that relaxes this restriction.

Our key observation is that *not all lock users are created equal*. We analyzed how locks are used in massively-parallel modern systems, such as NUMA-aware operating systems and databases. We found that such systems often have a small number of "regular" code paths that enter the lock very frequently. Such code paths are the primary beneficiary of the high scalability of MCS locks.

However, there are also many "guest" code paths that infrequently enter the lock and do not need the same degree of fairness to access the lock (e.g., background tasks that only run periodically with lower priority). These guest users, which are typically spread out in various modules of the software, prefer context-free locks, such as ticket locks.

MCSg provides these guests a context-free interface while regular users still enjoy the benefits provided by MCS. It can also be used as a drop-in replacement of MCS for more advanced locks, such as cohort locking. We also propose MCSg++, an extended version of MCSg, which avoids guest starvation and non-FIFO behaviors that might happen with MCSg.

Our evaluation using microbenchmarks and the TPC-C database benchmark on a 16-socket, 240-core server shows that both MCSg and MCSg++ preserve the benefits of MCS for regular users while providing a context-free interface for guests.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

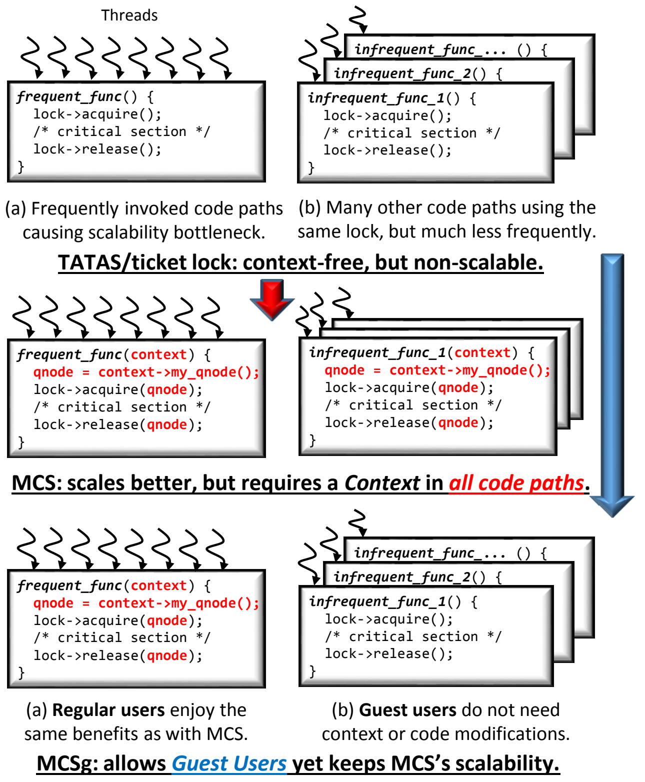
Keywords Spin locks, queued locks, MCS, locking API, fairness, throughput, latency, scalability

1. Introduction

Concurrent threads and processes must coordinate accesses to shared data. A synchronization mechanism for the coordination, typically locks, must scale up to the growing concurrency of today's massively parallel processors.

[*] Work done while with Hewlett Packard Labs.

PPoPP '16 March 12-16, 2016, Barcelona, Spain
Copyright © 2016 ACM 978-1-4503-4092-2/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2851141.2851160


Threads
(a) Frequently invoked code paths causing scalability bottleneck.
(b) Many other code paths using the same lock, but much less frequently.
TATAS/ticket lock: context-free, but non-scalable.
MCS: scales better, but requires a Context in all code paths.
(a) Regular users enjoy the same benefits as with MCS.
(b) Guest users do not need context or code modifications.
MCSg: allows Guest Users yet keeps MCS's scalability.

```
frequent_func() {
  lock->acquire();
  /* critical section */
  lock->release();
}
```

```
infrequent_func_... () {
infrequent_func_2() {
infrequent_func_1() {
  lock->acquire();
  /* critical section */
  lock->release();
}
```

```
frequent_func(context) {
  qnode = context->my_qnode();
  lock->acquire(qnode);
  /* critical section */
  lock->release(qnode);
}
```

```
infrequent_func_1(context) {
  qnode = context->my_qnode();
  lock->acquire(qnode);
  /* critical section */
  lock->release(qnode);
}
```

```
frequent_func(context) {
  qnode = context->my_qnode();
  lock->acquire(qnode);
  /* critical section */
  lock->release(qnode);
}
```

```
infrequent_func_... () {
infrequent_func_2() {
infrequent_func_1() {
  lock->acquire();
  /* critical section */
  lock->release();
}
```

Figure 1. Not all lock users are created equal. Frequent lock users need high scalability while occasional "guests" need not the best performance but context-free locking. MCSg satisfies both users.

Compared to centralized spinlocks, such as test-and-test-and-set (TATAS) and ticket (TKT) locks, software queuing locks such as MCS [15] and CLH [4, 14] locks scale better under high contention. In particular, the MCS lock has two advantages that make it well suited for massively parallel computers. First, the lock-entry (*doorway* [9]) protocol in the MCS lock is *wait-free* and *FIFO* because it uses an atomic swap (`XCHG`) instruction rather than a compare-and-swap (`CAS`) instruction for a thread to enqueue itself. Second, once a thread has enqueued its request, it spins *locally*. Local spinning reduces interconnect traffic among CPU cores.

These advantages led to the successful adoption of MCS locks in real production software. For instance, the Linux kernel has recently begun to replace some of its non-scalable locks with MCS locks. The result is a ∼3–5× performance improvement [2] in major benchmarks. Recent scalable database systems designed for parallel hardware have also adopted MCS locks for synchronization of their internal data structures [7, 8].

1.1 MCS Adoption Challenges and Key Observations

Figure 1 illustrates challenges to adopting MCS locks in complex production systems. The high performance and scalability of MCS locks come with the price of *bring-your-own-context*—each lock user must provide an extra "context", or a *queue node* (`qnode`), in addition to a reference to the lock itself. A `qnode` is typically pre-allocated for each lock user in a NUMA-aware fashion so that the lock user spins locally on its own `qnode`. The `qnode` is often allocated in a region of memory shared between processes so that users in other processes can access it. Whether we allocate `qnodes` on-demand or in advance, the added complexity of bring-your-own-context impedes adoption of MCS locks.

Several developers in Hewlett Packard Enterprise have been contributing to the open-source community for improving the scalability of OSes and databases on many-core servers. One example is the adoption of the MCS lock in the Linux kernel [2]. Throughout the efforts, we have repeatedly observed the performance benefit of the MCS lock as well as the challenge to its adoption. Furthermore, we made two key observations in real codebases that motivated and guided this work.

Key Observation 1: **Complex code paths necessitate a context-free lock interface.** TATAS and TKT locks need no additional contexts. They require only that the user hold a pointer to the lock for acquiring and releasing the lock. Much existing code in real systems assumes such a "context-free" interface. In fact, by far the most widely used locking interface in the Linux kernel consists of the `spin_lock(lock*)` and `spin_unlock(lock*)` macro families, which receive only a pointer to the lock object. Countless invocations of these macros involve infrequent code paths that do not require the same level of scalability or fairness as more frequently executed code paths. Re-writing *all* code paths that employ these macros to appropriately allocate, pass, and de-allocate conventional MCS lock `qnodes` would be a formidable task with meager benefits.

Therefore, the Linux kernel initially limited the adoption of MCS locks only to certain places. The developer who introduced MCS locks to the Linux kernel stated that:

"When trying to convert some of the existing Linux kernel spinlocks to MCS locks, it was especially complicated when the critical section spanned multiple functions. This required some functions to accept additional MCS node parameters, which was not practical." [11]

We observed the same issue in an open-source database [8]. In this case, the database has already adopted MCS locks for higher scalability, pre-allocating `qnodes` for each transaction processing thread. The problem arises when the database needs to add new functions that manipulate shared data protected by the existing MCS locks. The new functions are occasionally invoked from various modules without pre-allocated `qnodes`. Moreover, it is difficult to know *a priori* how many distinct threads will invoke the functions. The issue was more pressing because it impedes *functionality* rather than performance.

Key Observation 2: **In most cases, complex code paths are infrequent while frequent code paths are simple.** A lock protects shared data against various code paths that access the data. While the complexity of each code path's critical section varies, there is a strong correlation between the complexity and frequency of the code paths.

As described above, we observed many complex critical sections that cannot easily incorporate `qnodes`. It turns out that *all* such complex code paths infrequently enter the lock. Local spinning in these code paths thus does not improve scalability much. We never encountered frequent and complex code paths for an intuitive reason. Such complex critical sections run longer, and thus cannot be repetitively invoked over short durations. In fact, all of the frequent lock users that caused bottlenecks were found to have substantially simple critical sections. For example, the kernel developers found that one of the locking bottlenecks in Linux occurs when a 3-line code block triggers more than $100,000$ lock acquisitions per second [12].

The kernel developer stated that:

"Out of the 300+ places that make use of the dcache lock, 99% of the contention came from only 2 functions. Changing those 2 functions to use the MCS lock was fairly trivial because both of them had straightforward lock/unlock calls within the same function. The call-sites with the complicated locking made up much less than 1% of the bottleneck since they were called less often." [11]

The database developers made a similar statement.

To summarize, we observed that skews and inequality fundamentally abound in locking. Throughout the entire kernel and database code, extremely frequent code paths (e.g., $> 100k$ [/s]) in a highly contended lock are very rare. We can address 99% of the locking bottleneck by imposing the duty of queue-based protocol on a few code paths (i.e., *top 1%* lock users). On the contrary, 99% of the development cost to employ MCS locking is attributed to other, infrequent code paths. Relieving the other 99% lock users from code modification significantly eases the adoption of MCS locks especially because they might have complex critical sections spanning multiple functions and modules. These observations naturally guide us to the dual-interface design of our new MCS lock variant explored in this paper.

1.2 Paper Summary

The key contribution of this paper is a new variant of the MCS lock, the **MCSg** lock, which addresses the aforementioned issues. The MCSg lock provides a different interface for both frequent and infrequent code paths. The interface for frequent code paths, or "regular" users, is the same as that of the MCS lock. It provides the regular users with high scalability and fairness at the cost of bring-your-own-context. Another interface, for infrequent code paths, or "guest" users, is context free, but the code paths receive less scalability and fairness to access the lock.

MCSg facilitates adoption into complex systems in two ways. First, MCSg can replace MCS locks in existing code to allow new guest code paths, such as sporadic background tasks. MCSg is a perfect drop-in replacement for MCS that keeps all the good properties of MCS with minimal code changes. Second, MCSg can replace non-scalable context-free locks in existing code to improve scalability. Unlike with MCS, one can *gradually* adopt MCSg with minimal effort, starting with zero changes (all guest users), identifying a small number of frequent lock users, and then modifying only those specific code paths as regular users.

The rest of this paper is organized as follows. Section 2 details the key properties MCSg is designed to satisfy. Section 3 explains the new MCSg algorithm. Section 4 proposes MCSg++, which extends MCSg to give fair scheduling in the presence of guest users. We expect that many readers are already familiar with MCS locks, hence we discuss the original MCS lock and other related prior work in Section 5. Readers unfamiliar with MCS locks may wish to read Section 5 first. Section 6 empirically evaluates MCSg and MCSg++. Finally, Section 7 concludes.

2. Desiderata

MCS locks have properties that make them applicable to a wide range of settings. MCSg is designed to keep all of them in addition to the new context-free interface. This section details these desiderata to clarify the key principles behind MCSg.

Algorithm 1 MCSg Algorithm. Guests simply spin with CAS. Regular Users differ from original MCS only in Line 7-10.

```
1  def regular_acquire(lock_tail, my_qnode):
     tail_qnode = my_qnode
   retry:
4    pred = atomic_swap(lock_tail, tail_qnode)
     if pred == NULL:
       return
7    elif pred == π:
       # A guest has the lock, put back π and retry
       tail_qnode = atomic_swap(lock_tail, π)
10     goto retry
     else
       # A regular user holds the lock, join the queue
13     my_qnode->flag = WAITING              <mem_release>
       pred->next = my_qnode                 <mem_release>

16     # Spin on my (local) wait flag
       spin_while my_qnode->flag != GRANTED  <mem_acquire>
       return
19
   def regular_release(lock_tail, my_qnode):
     ... Exactly the same as the original MCS lock
22
   def guest_acquire(lock_tail):
     while (!atomic_cas(lock_tail, NULL, π));
25
   def guest_relrease(lock_tail):
     while (!atomic_cas(lock_tail, π, NULL));
```

2.1 High Scalability

As already described in Section 1, MCS provides a wait-free doorway, NUMA-friendly local spinning, and FIFO ordering among lock users. MCSg must maintain the same scalability at least for regular users unless guest users enter the lock extremely often, in which case such guest users should be upgraded to regular users.

2.2 Simplicity, Applicability, and Pluggability

Performance is often deemed as the most important factor of synchronization mechanisms. However, the *simplicity* of the algorithm sometimes weighs even more in practice, especially in huge, complex, and critical codebases, such as OSes and databases. This is where more basic spinlocks (e.g., TATAS) are still preferable. For a locking mechanism, we categorize the concept of *simplicity* into the following aspects.

Single-word Lock State: The MCS lock places just a single word as the shared state, the lock tail. The MCSg lock must keep this property. This has two benefits: reduced space consumption and code complexity. Some locking algorithms require additional words as the lock state. For example, cohort locks [5] have to reserve additional lock memory for each NUMA node.

Even when space consumption is not an issue, the complexity of the code to allocate/deallocate and identify such memory becomes significant. This issue is especially significant when we do not know *a priori* the number of locks or the number of lock users in the system, which is the case for many dynamic storage systems, such as databases and file systems.

For instance, such applications often embed locks into data pages. It is not even known in advance which bytes will be used as a lock. Hence, trivial initialization/destruction by a simple `memzero` is vital.

Context-free Interface: This is the feature the MCS lock does *not* provide. MCSg must provide a context-free interface similar to those of TATAS/TKT locks. It should receive only a pointer to the lock without any thread-specific context or global information.

Inter-Process Uses: The MCS lock is applicable to inter-process mutual exclusion. Many systems run a collection of individual *processes* with shared memory instead of running *threads* in the same process [8, 10, 13, 20]. For instance, virtually all major databases share memory among multiple processes.

Pointers (i.e., virtual addresses) do not work across processes. Hence, a typical MCS lock implementation on shared memory stores an identifier of the process/thread and a memory offset in each `qnode` [8]. This offset approach comes with an added benefit for advanced synchronization methods to be combined with the lock. It allows more bits than raw pointers for additional information, such as delete-flags, ABA counters, etc [6].

Some locking algorithms, however, cannot use offsets because they share a pointer to stack memory or *thread-local-storage* (TLS) that does not allow accesses from another process. MCS does not have this issue; MCSg must also avoid it.

Environment Independence: Some lock algorithms depend on environment-specific features. For example, qspinlock, which has recently been introduced to the Linux kernel [3], requires the kernel-space ability to disable pre-emption. Another example is a locking algorithm that requires efficient access to TLS. Some platforms are equipped with a special register to efficiently support TLS, e.g., the `%fs` segment register in x86 and `tpidr_el` registers in ARM. Without such hardware support, accessing TLS involves far higher costs. Even x86/ARM incurs high overhead for TLS variables in shared libraries or other modules due to the cost of adjusting TLS offset (e.g., `__tls_get_addr`).

MCS does not demand TLS, and MCSg must not.

Composability: The MCS lock can easily be used in combination with other locks, albeit not as easily as TATAS. For instance, an MCS lock can trivially provide the *cohort-detection* [5] property by checking its own `qnode` and provide the *thread-obliviousness* [5] property with a minor change. MCSg should keep the exact same composability as MCS. In other words, MCSg must be a drop-in replacement for MCS, and work everywhere MCS works.

3. Basic MCSg Locks

We now introduce MCSg, a new variant of the MCS lock that satisfies all desiderata in Section 2.

As shown in Algorithm 1, MCSg only slightly modifies the MCS algorithm. MCSg does not require any change to the MCS lock's data structure and adds only additional logic in the lock acquire procedure for regular users. *It behaves exactly the same as the original MCS lock when there are few or no guests.* The basic idea is for guests to treat the MCS lock word (the tail pointer) like a TATAS lock when trying to acquire it; regular users will spin-wait when they notice that a guest has acquired the lock and re-join the queue after the guest has released the lock.

The following subsections describe in more detail how guests and regular users interact with MCSg locks.

3.1 Guests

The lock is treated like a TATAS lock for guests trying to acquire it. Instead of joining the wait queue using an `XCHG` instruction, the guest issues a `CAS` against the lock tail (lines 23–24), trying to change it from `NULL` to a special sentinel value π[1]. The thread

[1] Our implementation uses an exponential backoff strategy in guest lock acquisition to reduce memory traffic. It is an obvious optimization that is orthogonal to the design of MCSg, so we omit it in the rest of the paper.

retries until the CAS succeeds, which indicates that it has acquired the lock.

Lock release is also straightforward for guests although it is more complex than in TATAS. The lock holder issues (and retries if it fails) a CAS against the lock tail (lines 26–27), trying to change it from π back to NULL.

In a nutshell, the guest only needs to issue and retry a CAS in each acquire and release operation. The only requirement for guests is to have a pointer to the lock. They do not have to provide a self-prepared qnode (context). The acquire procedure for regular users provides guarantees for this machinery to work.

3.2 Regular Users

In the original MCS lock's acquire procedure, lock users could see either a valid (lines 11–17) or a null pointer (lines 5–6) on swapping the tail pointer. In the MCSg lock's acquire procedure, however, they might see the sentinel value π when the lock is being held by a guest user.

To handle this case, MCSg adds a "swap-and-spin" loop (lines 7–10) for the regular user when it notices that a guest holds the lock. Suppose a guest user has acquired the lock right before a regular user R comes to line 4. R will receive π as the return value on performing XCHG with the lock tail. In this case, R will first XCHG π back to the lock tail. Note that this XCHG can return a value V that might not be the same as a reference to R. This is because, between the time R performed an XCHG with the tail pointer and the time when it performs another XCHG, an arbitrary number of other regular users might have XCHGed the tail pointer and enqueued behind R (lines 12–17). Therefore, when R retries to acquire the lock (lines 4), it XCHGs this latest lock tail V in its list, instead of a reference to its own qnode.

The above procedure causes two atomic operations, yet R does not acquire the lock. One optional optimization for reducing memory traffic, again, is to spin-wait with a backoff until the lock tail becomes non-π. We note that this optimization should be triggered after at least one iteration of XCHG. We empirically observed that, when we read the lock tail for this purpose before the initial XCHG, it causes more traffic on the contended cache line and slows down the most important use case: no or few guests.

Observant readers might have noticed that, when retrying to acquire the lock, a regular user is not guaranteed to maintain its original place in the wait queue; another regular user could conduct the XCHG at line 4 faster and get a return value of NULL, violating FIFO order among *groups*. We discuss and address this issue in Section 4.

3.3 Key Properties of MCSg

Before moving on to the extended version of MCSg, let us analyze the basic MCSg algorithm regarding the desiderata listed in Section 2.

Assuming the frequency of guest users entering the lock is negligibly low, MCSg preserves scalability benefits for regular users brought by the original MCS lock: scalable doorway, local spinning, and FIFO ordering.

For guests, MCSg behaves similarly to a TATAS lock. We rely on a sentinel value π stored in the lock tail to indicate that the lock is held by a guest. Guests' lock acquisitions and releases succeed *iff* the CAS successfully changes the lock tail from NULL to π and from π to NULL respectively. Regular users whose XCHG against the lock tail returns the sentinel π are responsible for swapping π back to the lock tail and then fall back to a spin-retry cycle. As a consequence, guests do not need any context to join and leave the lock, satisfying the **context-freeness**.

MCSg does not change anything on the MCS's lock-state data structure. Therefore, it also maintains the **single-word** lock state

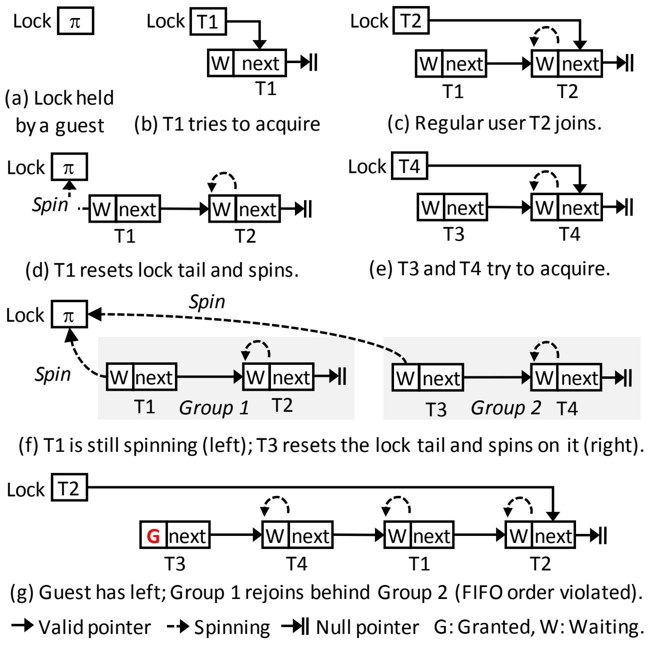


Figure 2. An example of (rare) FIFO order violation that could happen under MCSg. Note that FIFO ordering within each group is preserved.

property, the **inter-process** property, and the **compose-ability**. It does not pose any new requirement on the environment, either. In sum, MCSg is an ideal "drop-in" replacement for MCS locks with minimal changes. In fact, we have replaced MCS locks with MCSg locks in an open source database system [8], replacing its MCS code with just ∼20 LoC changes. Section 6.3 evaluates the performance of the MCSg lock in that database.

4. MCSg++ Extensions

MCSg satisfies the aforementioned scalability and simplicity requirements for regular users. MCSg, however, can potentially starve guest users and violate FIFO order among regular users.

4.1 Issue 1: Guest Starvation

MCSg retries a CAS for guests to acquire and release the lock. Although the steps are straightforward, a guest might starve by repeatedly failing the CAS when competing with regular users and other guests. It is possible that a steady stream of regular users lock out all guests forever. This is due to an inherent limitation of CAS: there is no guarantee that CAS will succeed in a bounded number of steps, violating MCS's wait-freeness of its doorway.

4.2 Issue 2: Non-FIFO Behaviors among Regular Users

Figure 2 pictorially represents how non-FIFO ordering ensues in the presence of guests. In this example, a guest user initially holds the lock (a). Another thread T1 tries to acquire the lock (b). Yet another thread T2 trying to acquire the lock will enqueue itself behind T1 because T2 observed the lock tail was neither NULL nor π (c). T1's XCHG, executed at line 9 of Algorithm 1, will return a lock tail pointing to T2 because T2's XCHG at line 4 happened before T1's at line 9. As shown in Figure 2(d), as an optimization, T1 will start to spin on the lock tail and retry after the guest has released the lock. Meanwhile, T2 has set T1's next field to point to its qnode.

If the guest now released the lock, T1 might notice that the lock tail is pointing to NULL. T1 then executes another XCHG to retry acquiring the lock. In the meantime, another two regular users—`T3` and `T4`—attempt to acquire the lock while `T1` is spinning on the lock tail. `T3` and `T4` will go through the same steps as `T1` and `T2` did, as shown in (e). This might result in the intermediate state shown in (f). After `T3` realizes that a guest is holding the lock, it will also issue an `XCHG` and start spinning on the lock tail. As a result, there could be multiple threads spinning on the lock tail. Each thread spinning on the lock tail leads a *group* of users trying to acquire the lock. We call such threads *group leaders*. Formally, group leaders are the threads that found the lock tail's previous value returned by the `XCHG` to be π. `T1` and `T3` in (f) are two group leaders. `T2` and `T4` are not group leaders but two regular successors that spin on their own `flag` fields.

When the guest releases the lock, the group leaders will notice that the lock tail has changed and retry the `XCHG` (line 4 of Algorithm 1), setting the lock tail to point to the tail of the group (obtained at line 9 of Algorithm 1).

If a later arriving group leader wins in installing its group tail, then FIFO ordering among groups of regular users is not guaranteed. In (g), `T3`'s `XCHG` succeeded earlier than `T1`'s. Group 1 (led by `T1`) then queues up after Group 2 (led by `T3`) by installing `T1` in `T4`'s `next` field.

Assuming that guest users are rare, FIFO ordering is still guaranteed for regular users in most cases. Still, this could potentially become an issue when there is an unexpected burst of guest users in a highly contended lock.

4.3 Guaranteed Guest Lock Acquisition

MCSg++, an enhanced variant of MCSg, addresses the above issues.

The first enhancement addresses the guest starvation issue. The key idea is that a guest user attaches itself to a regular user's `qnode` in a scalable manner. In MCSg++, a guest follows a "declare-and-wait" paradigm to acquire the lock without using a self-provided `qnode`. This protocol is reminiscent of regular users and the CLH lock [4, 14].

In MCSg++, a regular user followed by a guest is responsible for passing the lock to the enqueued guest in its release protocol. To release the lock, the guest atomically swaps the lock tail with `NULL` so that group leaders and other incoming users can resume to compete for the lock.

Handshakes between Guest and Regular Users: MCSg++ relaxes the meaning of π and the use of regular user's `qnode`. In addition to the original meaning of π in MCSg (a guest *has acquired* the lock), in MCSg++ a π value in the lock tail could also mean that a guest is *waiting for* the lock.

MCSg++ introduces sentinel values to the `qnode`'s `next` field for communicating with guests. Therefore, the `next` field has a dual use of (1) holding a pointer to a regular successor and (2) serving as a communication channel between the regular user and its guest successor. MCS and MCSg only use the `next` field for (1).

The sentinel values that could appear in the `next` field are:

- `GW`: The successor is a guest waiting for the lock;
- `GG`: The lock is granted to the guest successor;
- `GA`: The guest successor has acquired the lock;
- `NS`: No successor.

We next describe how guest and regular users interact with the lock.

Guest Lock Acquisition: The acquire protocol for guests under MCSg++ is reminiscent of that in the CLH lock [4, 14]. Instead of retrying `CAS` on a centralized memory location or spinning on its own `qnode`, a guest registers itself in the `next` field of its regular predecessor's `qnode` by storing `GW` there. The guest then spins on the `next` field and waits to be "woken up". A regular user that has a

Algorithm 2 MCSg++ locking protocol for guests.

```
   def guest_acquire(lock_tail):
   retry:
3    pred = atomic_swap(lock_tail, π)
     if pred == NULL:
       return
6    elif pred == π:
       goto retry
     else  # The predecessor is a regular user
9      pred->next = GW                         <mem_release>

       # Wait for the predecessor to pass the lock
12     spin_while pred->next != GG             <mem_acquire>
       # Acknowledge the predecessor
       pred->next = GA                         <mem_release>
15
   def guest_release(lock_tail):
     tail = atomic_swap(lock_tail, NULL)
18   if tail != π:
       tail->next = NS                         <mem_release>
```

guest successor must be responsible to set `GG` in its own `next` field to pass the lock to the guest.

Algorithm 2 gives details on how guests proceed to acquire the lock. Instead of retrying `CAS`, the guest issues an `XCHG` against the lock tail, setting it to π (line 3). A π value in the lock tail makes other incoming users (regular or guest) aware of the existence of a guest. The return value (`pred`) of this `XCHG` identifies the predecessor: no predecessor (`NULL`), a guest (π), or a regular user. If `pred` points to `NULL`, then the lock is not contended and the guest now has successfully acquired the lock (line 5). If `pred` is π, another guest has already acquired the lock or announced its intention to acquire the lock. In this case, the guest simply retries by going back to line 2.[2] When the predecessor is a regular user, the acquiring guest needs to indicate its existence in the predecessor's `qnode` instead of in the lock tail (line 9). The guest then spins on the `next` field in the predecessor's `qnode` (line 12) until the predecessor changes its `next` field to `GG`. The guest then stores `GA` in `next` to inform the predecessor that it has acquired the lock. This acknowledgment is necessary for the regular predecessor to safely reuse the `qnode` for future lock acquisitions.

Guest Lock Release: Releasing the lock as a guest is wait-free in MCSg++: the guest simply atomically swaps the lock tail with `NULL` (line 17 of Algorithm 2). This enables waiting group leaders and guests to compete for the lock again. If the return value of this `XCHG` is not π (i.e., the successor is a regular user), as shown by line 19, the guest "marks" the successor as the tail of the group with a sentinel value `NS`. The regular user will observe `NS` when releasing the lock. The next requester (regular user or guest) whose `XCHG` returns `NULL` will acquire the lock.

As we have discussed above, MCSg++ relies on the `next` field in the `qnode` to track guest status. In particular, it takes advantage of the following invariant:

Invariant 1. *If there is a waiting guest, the lock-holding regular user's `next` field will eventually become non-`NULL`.*

When a regular user fails the `CAS` for lock release in the original MCS lock, the `next` field of the regular user is guaranteed to become non-`NULL` eventually. Invariant 1 still holds in the presence of guests. In Algorithm 2, guests maintain this invariant by storing and reading sentinel values in `next`. We next show regular users' protocols that rely on and also preserve the invariant.

[2] Similarly to the optimization in Section 3.2, one could reduce memory traffic with a backoff when the lock tail is obviously π.

Algorithm 3 MCSg++ locking protocol for regular users.

```
 1 def regular_acquire(lock_tail, my_qnode):
     group_tail = my_qnode
   retry:
     pred = atomic_swap(lock_tail, group_tail)
 5   if pred == NULL:
       return
     elif pred == π:
       while true:
 9       if group_tail == π:
           goto retry   # Retrying with a guest in the tail

         spin_while group_tail->next == NULL  <mem_acquire>
13       if group_tail->next == NS:
           group_tail->next = NULL
           goto retry # No successor, retry as a new group
         elif group_tail->next == GW:
17         # A guest queued up after me; put π in the
           # lock tail for other users to notice
           group_tail = π
           goto retry
21       else
           group_tail = group_tail->next # Follow successors
     else
       my_qnode->flag = WAITING              <mem_release>
25     pred->next = my_qnode                 <mem_release>
       spin_while my_qnode->flag != GRANTED  <mem_acquire>

   def regular_release(lock_tail, my_qnode):
29   if atomic_cas(lock_tail, my_qnode, NULL):
       return

     spin_while my_qnode->next == NULL       <mem_acquire>
33   if my_qnode->next == GW:
       my_qnode->next = GG                   <mem_release>
       # Wait for the guest to pick up the lock
       spin_while my_qnode->next != GA       <mem_acquire>
37   else
       my_qnode->next->flag = GRANTED        <mem_release>
```

Regular User Lock Acquisition: The lock acquisition protocol for regular users in MCSg++ differs from that in MCSg when the acquiring regular user notices that the lock tail contains π. If the lock is not contended or the lock tail is pointing to a regular user, the user can acquire the lock in the same fashion as in MCS and MCSg (lines 5–6 and 23–26 of Algorithm 3).

When a guest is present (either having acquired or still waiting for the lock), the acquiring regular user will form a new group as more regular users come to acquire the lock (lines 7–22). Unlike in MCSg, a regular user in MCSg++ does not immediately put π back to the lock tail. Therefore, unless there are more guests coming to compete for the lock, incoming regular users will queue up after one another. The first regular user whose `pred` points to π will be the group leader. For a group leader, the gist of the algorithm is a loop in which it tries to reach the end of the group (represented by `group_tail`).

Based on Invariant 1, the group leader first waits for its successor field (`next`) to become non-`NULL` (line 12). `group_tail` initially points to `my_qnode`, the requester's own `qnode`, as indicated by line 2. Depending on the status of other concurrent users, the current group tail's `next` field could be `NS`/`GW` or point to another regular user. If it is pointing to a regular user, the group leader simply follows the pointer and enters the next iteration, jumping from line 22 to line 8.

The other two cases are results of interactions with guests. If the current group tail's `next` field is `NS` (line 13), it means a guest has released the lock and "notified" the acquiring group leader that there are no more successors *in the current group* as described in line 16 of Algorithm 2 . As shown by lines 13–15 of Algorithm 3, the group leader can now lead the group to retry as if nothing had happened. The only difference is that the group leader will put the group tail—instead of its own— in the lock tail. This process is similar to how a group leader retries to acquire the lock on behalf of its members in MCSg. The difference is that a group leader in MCSg++ follows the `next` fields to find out the "real" group tail and guest status. MCSg could easily obtain this information from the return value of `XCHG`.

The case illustrated by lines 16–20 happens when another guest—other than the one noticed by the group leader earlier at line 4—tries to acquire the lock. Recall that a guest will register itself to the regular predecessor by storing `GW` in the predecessor's `next` field (line 9 of Algorithm 2). In this case, the acquiring group leader then needs to install π back to the lock tail and retry so that incoming regular users will form new groups (lines 17–20 and 3–4 of Algorithm 3). Concurrent guests in this case will have to retry until a regular user becomes its predecessor or the current guest that is holding the lock exits.

Regular User Lock Release: Similar to the original MCS, releasing an MCSg++ lock as a regular user also starts by attempting a `CAS`, expecting that the lock was not contended (line 29 of Algorithm 3). If this `CAS` fails, it means another guest or regular user has tried to acquire the lock: the releasing regular user is responsible for notifying the successor.

Relying on Invariant 1, the regular user first waits until the `next` field becomes non-`NULL`. If the successor is a regular user, it will register itself in `next`. If the successor is a guest, it stores `GW` and waits for `GG` in the `next` field (lines 9–12 of Algorithm 2). Therefore, Invariant 1 always holds in the presence of guests. To pass the lock to a regular successor, the releasing regular user simply writes to the successor's `flag` field as MCS and MCSg do. To pass the lock to a guest, the releasing regular user puts `GG` in `next`. When the guest detects the transition from `GW` to `GG`, it will set `next` to `GA`, acknowledging the regular user that the guest has acquired the lock. Upon detecting `next` has become `GA`, the regular user can now leave (lines 35–36 of Algorithm 3).

The above state transitions `GW` $\rightarrow$ `GG` $\rightarrow$ `GA` guarantee the integrity of the communication channel between the guest and its regular predecessor. Suppose that a guest was pre-empted after setting its predecessor's `next` field to `GW`. Suppose also that the predecessor has released the lock without waiting for the acknowledgment before the guest starts to spin at line 12 of Algorithm 2. If the regular user started another round of lock acquisition and release using the same `qnode`, the guest might be spinning indefinitely for `next` to become `GG`. Therefore, we need to ensure that the guest has picked up the lock before letting the regular user leave.

4.4 Reducing Non-FIFO Behaviors

In the presence of guests, an older regular user group might queue up after a younger one in MCSg. FIFO order is preserved for all users within the same group, but not always among groups.

MCSg++ does not guarantee FIFO ordering between a regular user and a guest, either. Completely preserving a total order among both regular users and guests generally requires maintaining a list-like structure for them. This violates the simplicity principle for guests because guests need to be associated with some context as well. Because of the relative rareness of guest users, the chance of forming many groups is not high in our targeted use cases. We therefore focus on preserving the FIFO behavior among regular user groups.

Our solution is inspired by ticket locks. A ticket lock has two counters: `ticket_owner` and `next_ticket`. Threads atomically read and increment `next_ticket` to enter the lock. The thread whose ticket equals to `ticket_owner` can enter the critical section. Other threads will spin for `ticket_owner` to match their tickets.

Algorithm 4 MCSg++'s lock acquire protocol that reduces FIFO order violations among regular user groups. Additional logics are added on top of Algorithm 3, most of which are omitted for clarity.

```
   def regular_acquire(lock_tail, my_qnode):
2    group_tail = my_qnode
     my_ticket = INVALID_TKT
   retry:
     if my_ticket $\neq$ INVALID_TKT:
6      spin_while(ticket_owner != my_ticket) <mem_acquire>
     pred = atomic_swap(lock_tail, group_tail)
     if pred == NULL:
       if my_ticket $\neq$ INVALID_TKT:
10       atomic_increment(ticket_owner)
       return
     elif pred == $\pi$:
       if my_ticket == INVALID_TKT:
14       my_ticket = atomic_increment(next_ticket)
       ... lines 8 -- 22 of Algorithm 3 ...
     else
       if my_ticket $\neq$ INVALID_TKT:
18       atomic_increment(ticket_owner)
       ... lines 24 -- 26 of Algorithm 3 ...
```

Similarly, MCSg++ maintains these two counters in addition to the lock tail. Whenever a regular user realizes it is a group leader, it obtains a ticket and waits for its turn before retrying. Algorithm 4 shows the revised lock acquire protocol for regular users. For clarity we omit most of the code that overlaps with Algorithm 3. At line 3 a user enters the lock without obtaining a ticket. It then conducts the XCHG operation at line 7. If the regular user finds that the return value is π (line 13), it is a group leader and will obtain a ticket by atomically reading and incrementing next_ticket. This is usually done through an atomic-fetch-and-add instruction. The group leader continues as illustrated by lines 8–22 of Algorithm 3. Before the group leader retries at line 7, it first spins on the ticket_owner to wait for its turn (lines 5–6). Ticketing gives ordering to all group leaders competing for the lock in the presence of guests.

The ticket_owner field is incremented under two circumstances: (1) the group leader acquired the lock (lines 9–10) and (2) the group leader has to queue after another group (lines 17–18). In case (1), the group leader waited for its turn and the return value of the XCHG is NULL (line 8). Case 2 is the "unlucky" scenario where another regular user's XCHG succeeded earlier while the group leader is waiting for its turn. This could happen if a regular user got the lock right after a guest released the lock (i.e., the regular user's XCHG returned NULL) while the group leader was spinning on next_ticket. Therefore, ticketing only *reduces* FIFO order violations among regular user groups. We quantify the impact of case 2 in Section 6. In most cases, ticketing can reduce 50–70% of FIFO order violations.

4.5 Discussion

Under MCSg, guests might starve in the presence of a steady stream of regular users. MCSg++ uses XCHG to allow guest users to attach after regular users, which upon lock release will pass the lock to the awaiting guest. MCSg++ makes it easier for guests to acquire the lock. However, we also note that MCSg++ does not always guarantee a guest will be able to acquire the lock, especially when guests are the majority. Specifically, if there is a single guest and an arbitrary number of regular users, then the guest will enter its doorway in bounded time. If most users are guests, however, or if there is a steady stream of guests intermixed with regular users, then an individual guest can starve.

5. Prior Work

Our work in this paper stands upon the shoulders of much prior work on queuing locks. In this section, we briefly discuss the MCS and CLH queuing locks, followed by other related work.

5.1 MCS-Lock

Mellor-Crummey and Scott [15] invented the MCS lock. In an MCS lock, the lock word represents a tail pointer to a linked list of lock requesters. Each lock requester arrives with its own record and swaps the tail pointer to its own record using the XCHG atomic primitive. Thus, the tail pointer always points to the last requester, or NULL if none. The record has two cache-line aligned fields: (1) a status flag and (2) a pointer (next) to a successor record. Swapping the tail pointer informs each requester of its predecessor. If there is none, then such requester immediately enters the critical section. If there is a predecessor, then such requester sets the flag field in its record to WAITING, installs a reference to its own record in its predecessor's next field, and spins on its flag field until the flag is toggled to GRANTED. The release protocol involves setting a successor's flag field to GRANTED, if present. If there is no successor, then such releaser CASes the tail pointer to NULL. If the CAS fails due to some successor XCHGing the tail pointer, then the releaser waits until the successor installs the next pointer and then toggle's the successor's flag field. In the MCS lock, the record node is both brought by the requester during acquisition and reclaimed by the requester after releasing the lock.

5.2 CLH-Lock

The CLH lock [4, 14] is a variant of the MCS lock where each lock requester, instead of spinning on its own flag, spins on its predecessor's. Each queue node in a CLH lock maintains a pointer to its predecessor, whereas in the MCS lock it maintains a pointer to its successor. The head of the queue is a dummy node. A key difference between CLH and MCS is that, in CLH, a requester leaves behind its record for its successor and reclaims its predecessor's record during its release protocol. As a result of reclaiming the predecessor's record, the CLH lock needs to additionally manage its memory. Scott [17] proposes a technique to avoid the overhead by thread-local memory allocations.

5.3 Advances in MCS and CLH

There have been many prior efforts to enhance the MCS and CLH locks to accomplish other objectives. Mellor-Crummey and Scott [16] relaxed the MCS lock for *reader-writer synchronization* accommodating multiple readers in a critical section. They explored three variants—fair-reader-writer, reader-preference, and writer-preference locks. For example, the fair-reader-writer lock enables a requestor to safely access fields in its predecessor's record. A reader who is enqueued immediately after another reader can notice the status of its predecessor (waiting or holding the lock) and enter its critical section without waiting for the predecessor to finish. If a successor reader finishes before a predecessor reader, the last finishing reader takes the additional responsibility of passing the lock to the first waiting writer.

Scott and Scherer [18] enhanced both MCS and CLH locks with the *timeout capability* allowing an enqueued process to abort after a period of waiting. The enhancement to the CLH lock adds additional states—transient, leaving, and recycled—to the flags field. These states are used to establish a handshake among the aborting thread, its predecessor, and its successor. When a thread wanting to abort is at the tail of the queue, it follows a more complex protocol to ensure consistency with an intervening successor that might abort as well. The modifications to the MCS lock are even more complex, especially when the aborting thread is at the tail of the queue. The MCS queue is transformed into a doubled-linked list from its original singly-linked list. The modified MCS lock uses a few special values to make sure an aborting thread leaves without causing dangling pointers in the linked list.

5.4 Queueing Locks that support Guests

The idea of using "special" value(s) to indicate deviation(s) from normal behavior is not uncommon in the synchronization literature. For example, the aforementioned MCS and CLH locks with timeout also used a handful of "special" values when aborting. However, we are not aware of any prior work that uses such a special value in order to treat one type of user differently from another. Put in another way, to the best of our knowledge, this paper is the first one to explore the problem of occasional guest users that cannot provide a context (i.e., a queue node).

K42: Notable prior art in terms of context-less locking is the variant of MCS lock implemented in K42 [1]. It is a queue lock, but a lock user does not need to provide a context. However, it loses the wait-freeness of MCS's doorway protocol because it uses `CAS` to enter the lock acquisition. Experiments in Section 6 observe a degraded scalability due to this. Moreover, K42 uses the stack memory of the thread as the queue node. This rules out its use in inter-process locking and in cohort-locking described next.

In Section 6, we evaluate a variant of the K42 lock that addresses these shortcomings with TLS as [17] proposes. However, we observe that this approach demands efficient TLS support in the platform and also does not scale as well as MCS/MCSg due to its lack of NUMA-awareness.

5.5 Lock Cohorting

In the context of NUMA locks, Dice et al. [5] devised lock cohorting, which composes two different synchronization protocols. Cohort locks are two-level locks—a global lock and a local lock. The global lock can be of any kind, whereas the local lock should be able to express the fact that there are waiters—e.g., MCS, CLH, ticket etc. The cohort locks dedicate a local lock per socket on a node and there is one global lock. Each thread wanting to enter the critical section competes for its local lock. The first thread to acquire the local lock proceeds to compete for the global lock; other threads wait for the local lock. Once a thread acquires the global lock and finishes its critical section, it releases its local lock if it notices local waiters. A waiting thread immediately enters the critical section without competing for the global lock after it is granted a local lock, effectively inheriting the global lock. A thread can pass the global lock within its NUMA domain for a "threshold" number of times to take advantage of locality. On reaching the threshold, the global lock is relinquished to another NUMA domain. A global back-off lock (BO) with local MCS locks makes a cohort C-BO-MCS lock. Similarly, one can devise C-BO-CLH, C-MCS-MCS, C-CLH-CLH, C-BO-TKT, C-TKT-MCS locks, among others.

6. Evaluation

This section empirically evaluates MCSg/MCSg++ and compares them with other candidates described in Section 5 to confirm the following claims:

- MCSg maintains the same ***scalability*** of MCS when guests are rare (§ 6.2);
- MCSg admits guests yet preserves all the good properties of MCS as a ***drop-in replacement*** (§ 6.3);
- MCSg++ gives more fairness to guests and can reduce FIFO order violations (§ 6.4).

6.1 Setup

We conducted experiments on a server equipped with 16 Xeon E7-4890 processors clocked at 2.8 GHz, each of which has 15 cores. The server has 240 physical cores and 12 TB of DDR3 DRAM clocked at 1333 MHz. The processor has 256 KB of L2 cache per core and 38 MB of L3 cache per socket. We use a microbenchmark and a database workload for our experiments.

All threads are pinned to physical cores in a *compact* manner, meaning we assign threads to a minimal number of sockets. We always leave an unused core per socket so that `watchdog` and other kernel tasks do not hinder our threads. We do not use hyperthreaded hardware contexts to maximize the performance under high contention. We thus use at most 224 cores over 16 sockets.

Lock algorithms: We have implemented MCSg and MCSg++ in addition to MCS, TATAS, CLH, and a few more variants of MCS for comparison, detailed below. To measure the overhead and effectiveness of MCSg++'s ticketing machinery for preserving FIFO ordering among regular users, we have also implemented **MCSg+**, a stripped-down version of MCSg++ without the ticketing machinery. To show that MCSg can be used as a drop-in replacement of MCS everywhere, we implemented a cohort lock, **C-MCSg-MCS**, in which we use an MCSg lock in place of the global MCS lock. The C-MCSg-MCS lock is described in detail in Section 6.3.

The `qnodes` used by MCS, CLH, and C-MCSg-MCS are pre-allocated. Regular users in MCSg and MCSg++ also use pre-allocated `qnodes`. These pre-allocated `qnodes` are organized as a global array, with an entry for each thread. For **CLH**, we use the algorithm described in [17]. Because a successor in CLH spins on its predecessor's `qnode`, a `qnode` cannot be reused until the successor notices the `next` field has been changed by its predecessor. In this implementation, each thread inherits its predecessor's `qnode` to solve this problem.

Our implementation of **K42**'s MCS variant follows the algorithm described in [17], which allocates `qnodes` on the stack. We also compare with our own extension of K42 to analyze its performance without the restriction on inter-process use described in Section 2. Instead of using a stack-allocated `qnodes`, our **K42-TLS** maintains a global `qnode` pool from which a thread can atomically borrow and return a `qnode`. To avoid the cost of borrowing and returning a `qnode` each time, K42-TLS uses a thread-local (TLS) variable to hold the borrowed `qnode` for each thread. This introduces another requirement on efficient TLS support in the platform, but this experiment is run on a CPU that satisfies the requirement (`x86_64`, on which Linux can use `%fs` register for TLS).

Likewise, our CLH implementation uses a global array of pre-allocated `qnodes` for inter-process use without stack variables. To enable guest access, we implement **CLH-TLS**, a variant CLH with standard interface based on the proposal in Section 4.3.2 of [17]. We use a TLS variable to store `thread_qnode_ptrs[self]` as [17] recommends.

Finally, a **TATAS** lock is implemented as a baseline. Each thread in the critical section reads four cache lines and then releases the lock. We run each experiment for ten seconds and report the averages of five runs.

Database Workload: MCS is used in various complex systems for its performance and simplicity. For example, modern database systems designed for massively parallel hardware, such as Shore-MT [7] and FOEDUS [8]. In such systems, there often are infrequent yet various background tasks in addition to regular worker threads that run database transactions.

We have implemented MCSg in FOEDUS for its superior performance on many-core systems. The guests are background threads that need to hold a page lock when installing snapshots.

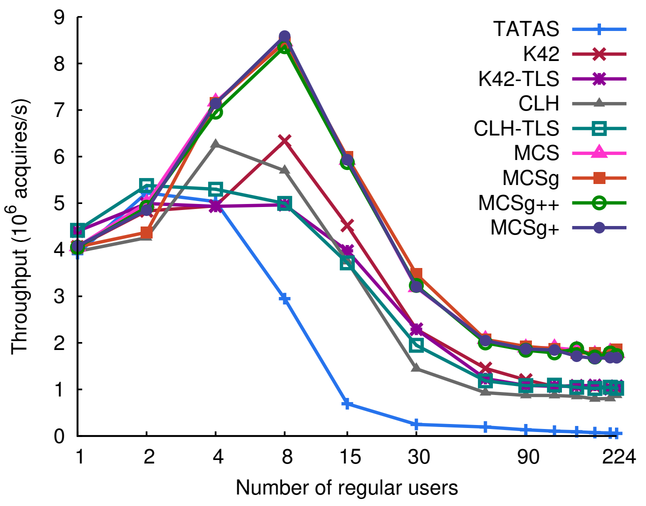


Figure 3. Lock throughput without guests. MCSg and MCSg++ keep the scalability of MCS. K42 sacrifices some scalability due to its use of `CAS`; CLH's node inheritance design puts more pressure on the interconnect, making it less scalable than MCS.

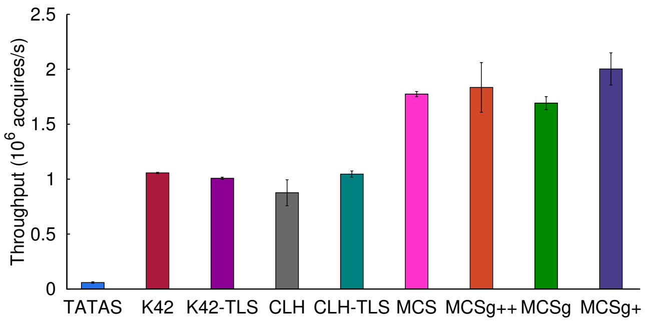


Figure 4. Lock throughput with 223 regular users and one guest. The number for MCS with 224 regular users are used as a baseline. A small number of guests do not affect the throughput of MCSg.

Table 1. Throughput of TPC-C's Payment transaction under high contention (one warehouse, 192 threads). MCSg achieves the same performance as MCS yet provides a standard interface for guests.

Lock	Throughput (MTPS)	Standard deviation
TATAS	0.33	±0.095
MCS	0.46	±0.011
MCSg	0.45	±0.004

MCSg is applied to FOEDUS's existing codebase.[3] We run TPC-C [19], a standard database benchmark that models an online transaction processing workload of a wholesale supplier when processing orders. We use the `Payment` transaction, which is a write-heavy transaction that updates the customer's balance and generates relevant statistics about the warehouse. We run `Payment` with 192 worker threads. The database size is set to one warehouse to generate enough contention. We compare its throughput (million transactions per second, or MTPS) among MCS, MCSg, and TATAS (supposing FOEDUS had to use a centralized lock to allow guests).

6.2 Maintaining MCS's Scalability

In this section, we focus on two basic settings: no guests and one guest. MCSg and MCSg++ should preserve the scalability of the original MCS when there are few guests. In both settings, we evaluate the locks under high contention: threads repeat the *acquire–access–release* cycle without any delays.

No Guests: Although the sole purpose behind MCSg and MCSg++ is to allow guests, it is crucial to maintain the performance in existing code paths for MCSg and MCSg++ to be a superior drop-in replacement of the original MCS.

Figure 3 shows the throughput of each lock implementation with a varying number of regular users. Both MCSg and MCSg++ match MCS's throughput.

K42 and CLH exhibit less scalability than MCS/MCSg. Although K42 can handle guests, its use of `CAS` rather than `XCHG` causes many retries under high contention, making it less scalable than MCS/MCSg.

K42-TLS suffers from the same issue. Furthermore, the additional overhead to access TLS variables makes it slightly slower than K42, about 10%-20% around the peak performance under low contention. Under higher contention, the `CAS` effect becomes more important so that the overhead of TLS access is less visible.

CLH and CLH-TLS do not scale as well as the MCS family because of the node-passing design. CLH-TLS largely follows the performance of CLH although not as close as K42-TLS and K42 because the algorithm in [17] has a few performance improvements over the original CLH. Still, we observed that the lack of NUMA-awareness in CLH fundamentally limits the scalability in many-socket servers. In fact, we observed that the performance of CLH/K42 and their TLS versions is much closer to MCS/MCSg's on a smaller number of NUMA sockets, such as 2-sockets rather than 16-sockets in this experiment.

MCSg, MCSg+ and MCSg++ keep the scalability of MCS. They achieve much better performance under high contention thanks to the local spinning protocol.

With One Guest: We now repeat the same experiment with one guest and 223 regular users. Locks that cannot support guests (MCS and CLH) are excluded from this experiment, but we still include the throughput of MCS and CLH with 224 regular users for reference. Figure 4 shows the throughput of regular users in the presence of one guest user. The throughput of MCSg is statistically equivalent to that of MCS. MCSg++ has a slight slowdown due to additional ticketing mechanism, but still matches the original MCS lock. In line with the results shown in Figure 3, K42 and CLH-TLS do not scale as well as other MCS variants. K42-TLS again closely follows the performance of K42. TATAS, not surprisingly, does not scale at all. We describe how the locks perform with more guests in Section 6.4.

6.3 Being MCS's Drop-in Replacement

MCS is simple and pluggable. It can be used in conjunction with many other techniques and in various complex systems. This section demonstrates that MCSg can be used a drop-in replacement of MCS in the context of databases and cohort-locks.

MCSg in a Database System: As Section 6.1 described, we have implemented MCSg in FOEDUS and then run TPC-C. Table 1 shows the throughput of TPC-C's `Payment` transaction under high contention with one warehouse and 192 worker threads. We observe that MCS and MCSg improve end-to-end performance by up to 50% compared to TATAS. MCSg has performance equivalent to MCS yet allows guest users. Although the 50% difference is not as dramatic as the orders of magnitude differences seen in the microbenchmarks, it is a surprisingly significant improvement

[3] Available at `https://github.com/hkimura/foedus_code`.

considering that database transactions are substantially more complex than locking itself. The complexity is also the source of much higher variance in this experiment. Especially with TATAS, the progress of transactions is highly random, sometimes causes milliseconds of latency to a single transaction that usually finishes in sub-microseconds.

Because MCSg does not change the data structure and interface of the original MCS, the change required to adopt MCSg in FOEDUS was minimal—as few as 20 LoC—showing MCSg's pluggability. Keeping the same data structure and interface is vital. Like many other complex systems, FOEDUS needs to embed many lock objects in each fix-sized data page (usually of 4 KB). Combining MCS with other locks (e.g., to compose a cohort lock) or adopting locks with a different data structure/interface incurs much more space overhead and code complexity compared to MCS and MCSg, which occupy only a single word.

Although K42 provides a standard lock interface, it requires the ability to point to each thread's stack memory. Its use is therefore limited to *intra-process* synchronization. Like many other databases, FOEDUS uses shared-memory for most data objects to be referenced *across processes*. K42 cannot be implemented in such systems because the pointer must be valid across in all processes. One can extend K42 for multi-process use with TLS, such as our K42-TLS. However, as we observed in previous experiments, it incurs another requirement on efficient TLS support, an overhead to access TLS variables, and complexity of a global `qnode` pool.

MCSg satisfies all the requirements and works as a pure drop-in replacement for MCS without any special hardware or additional complexity.

MCSg in a Cohort Lock: MCS is an important building block for more advanced/complex locks. Cohort locking [5] is such a composite lock implementation. For example, the C-MCS-MCS lock uses an MCS lock as the global lock and another for each NUMA node to get better scalability. In order to show that MCSg can be used as a drop-in replacement anywhere MCS is useful, we used MCSg to compose a cohort lock, C-MCS*g*-MCS, which uses MCSg as the global lock, and the original MCS as local locks. Guests contend directly on the global lock. Regular users first try to acquire the local MCS lock. The first winning regular user will then compete for the global lock and pass it to its successors for a "threshold" number of times. In our experiments, we set the threshold to 64, which is the recommended value in [5].

Table 2 shows the throughput in the microbenchmark with 224 regular users. C-MCSg-MCS outperforms MCSg and MCSg++ by 3×, which coincides with the findings of [5]. Plugging MCSg in place of MCS is trivial. Composing C-MCSg-MCS thus did not require any more effort than C-MCS-MCS. In large NUMA systems with the need to support guests, C-MCSg-MCS is a preferable lock implementation. MCSg can also provide equivalent enhancement to other flavors of cohort locks and MCS-based hierarchical locks.

6.4 Getting more Fairness with MCSg++

We have discussed the performance of regular users in previous sections. This section focuses on guests and their interactions with regular users.

Guest Starvation: Recall that MCSg might starve guests because guests rely on `CAS` to acquire and release the lock. This will become a serious problem when the lock is highly contended. To solve this problem, MCSg++ uses `XCHG` for guests to acquire and release the lock, with extra handshake protocols between regular users and guests. Figure 5 compares the average latency using locks that support guests for a single guest to acquire the lock with 223 regular users and one guest. The y-axis is in log scale. MCS is missing from the figure because it does not support guests.

Table 2. Microbenchmark comparison of MCS, MCSg, MCSg++ and C-MCSg-MCS with 240 regular users (no guests).

Lock	Throughput	Average latency
MCS	1.77M/s	126.21μs
MCSg	1.86M/s	120.34μs
MCSg++	1.72M/s	129.58μs
C-MCSg-MCS	5.22M/s	42.79μs

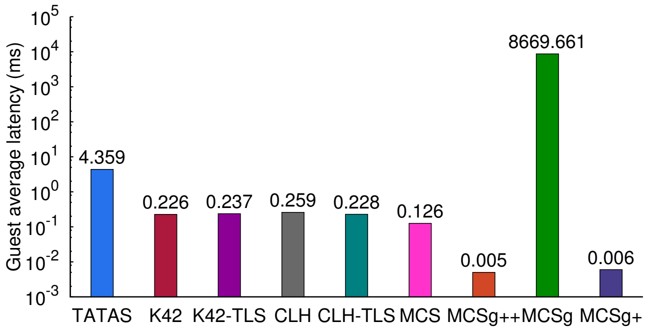


Figure 5. Average latency for a guest to acquire the lock when running at 223 regular users and one guest. The number for MCS with 224 regular users are used as a baseline. MCSg starves guests, while MCSg+ and MCSg++ siginificantly improves guests' fairness.

As shown in the figure, when the lock is heavily contended, guests starve under MCSg because `CAS` does not have any bounded guarantee on when a guest can get the lock. In almost all runs, the guest under MCSg acquires the lock only for a few times. K42 supports guests by giving them an implicit `qnode` living on the stack, which gives guests equal opportunity as regular users. However, K42 could issue many `CAS`es during lock acquire and release, resulting in high latency for both guests and regular users. We observed the exact same behavior in K42-TLS. MCSg++ and MCSg+ achieve orders of magnitude lower average guest latency because they issue a single `XCHG` instead of `CAS` in most operations. Moreover, the only guest is the absolute minority among all users in this experiment. Therefore, under MCSg+ and MCSg++ the guest will be able to easily find a regular predecessor to attach to and grab the lock thereafter with low latency.

Figure 6 shows the total throughput with a varying number of guests. The total number of threads is fixed at 224. Among the locks we tested, K42/K42-TLS and CLH/CLH-TLS give fair scheduling for both guests and regular users. Therefore, they all maintained steady performance across the x-axis in Figure 6. MCSg behaves similarly because it favors regular users much more and starves guests. Correspondingly, Figure 7 shows that MCSg does not perform well for guests. MCSg+ and MCSg++ both give much more fairness to guests, but at the cost of lower throughput for regular users when there are many guests, as shown by Figure 8. This is because of the complexity to handle handshakes between guest and regular users, while MCSg does not have.

FIFO Ordering: In order to evaluate the effectiveness of MCSg++'s ticketing mechanism, we compare the number of FIFO order violations among regular users that happen during 10-second runs with a varying number of guests. Since MCSg often starves guests under high contention, we focus on comparing MCSg+ and MCSg++ in this experiment. We use a TLS counter to record the number of FIFO order violations at runtime, and sum up all the counters after each experiment. As shown in Algorithm 4, a group

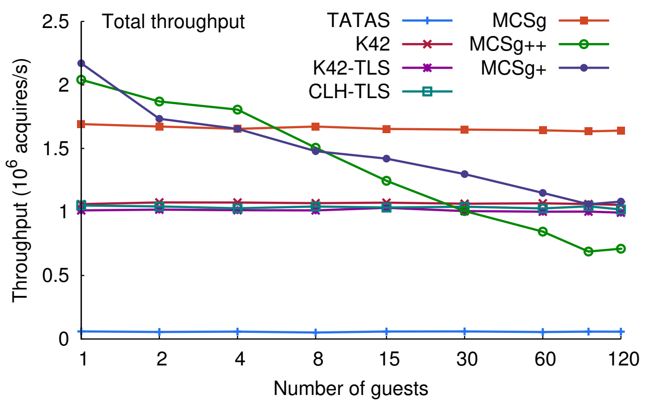


Figure 6. Total throughput when running at a varying number of guests (224 users in total). MCSg favors regular users and starves guests, thus providing stable performance. K42 and K42-TLS also show stable performance because they treat all users equally. MCSg+ and MCSg++ sacrifice total throughput to give guests fairness.

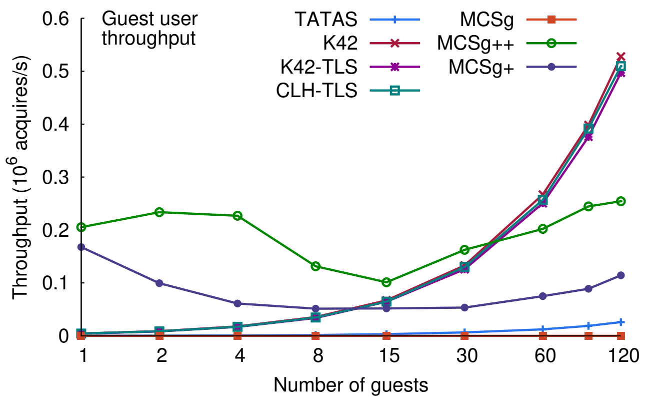


Figure 7. Guest throughput when running at a varying number of guests with a total number of users of 224. MCSg++ performs consistently better than MCSg+, because ticketing causes more bias toward guests; regular users are "throttled" by waiting for turns.

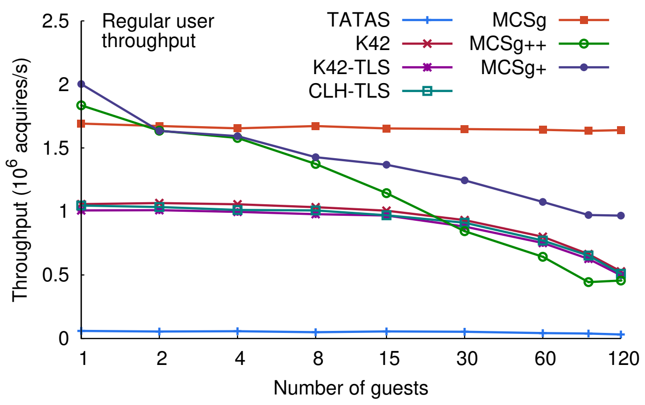


Figure 8. Regular user throughput when running at a varying number of guests. The number of total users is fixed to 224. The numbers for MCSg+ and MCSg++ drops faster than K42 and CLH families due to their bias toward guests. Compared to MCSg+, MCSg++'s is more biased toward guests because with ticketing, regular users are "throttled" by waiting for turns.

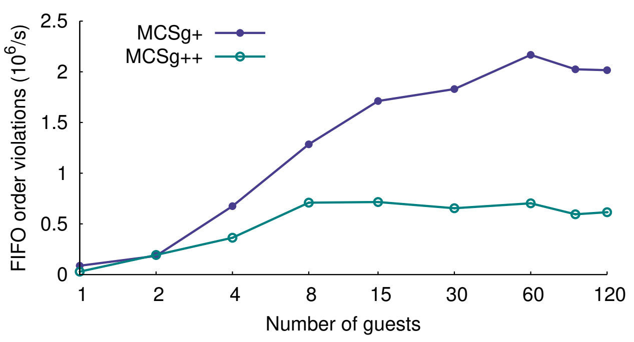


Figure 9. The number of priority inversions with 224 users in total. Ticketing in MCSg++ can reduce up to 70% of priority inversions among regular user groups.

leader can exit the retry loop only in two cases: (1) its `XCHG` returned `NULL` (lines 5–6), or (2) it successfully attached after another regular user (lines 24–27). Therefore, combined with Algorithm 3, we increment the TLS counter whenever a group leader queues after another regular user with a valid ticket (lines 24–27 of Algorithm 3). We added the ticketing machinery in MCSg+ for counting FIFO order violations only. Regular users acquire tickets like they do with MCSg++, but do not use them. In other experiments, MCSg+ does not implement the ticketing machinery at all.

Figure 9 shows the result. Compared to MCSg+, MCSg++ can reduce FIFO order violations for up to 70%. Because a regular group leader in MCSg++ waits for its turn via the ticketing machinery, guests have a higher chance to acquire the lock while regular group leaders are waiting for their turns. Figure 7 verifies that ticketing favors guests. MCSg++ consistently achieves much higher guest throughput than MCSg+.

Finally, we note that ticketing trades off regular user throughput for guest performance. Figure 8 shows that, as the number of guests increases, the regular user throughput of MCSg++ drops faster than that of MCSg+. With more guests, regular users have a higher chance to be "chopped" and form more groups to acquire and wait on their tickets, causing a higher contention.

7. Conclusions

We have described a new variant of MCS locks, which allows lock acquisition and release without any bring-your-own-context and without degrading the high scalability of MCS locks. The key observation behind this work is that complex multi-thread/multi-process software often has two kinds of lock users: *regular* users and *guest* users.

MCSg behaves as an MCS lock to regular users and as a centralized lock to guest users, providing benefits of both. We recommend using MCSg as a drop-in replacement for existing locks in three scenarios.

The first scenario is to replace an existing MCS lock that needs to allow guest users. As we have observed in Section 6.3, it requires only a small change to transform an existing MCS lock to MCSg.

The second scenario is to replace an existing centralized spinlock (e.g., TATAS) that is a scalability bottleneck. It is trivial for the developer to replace the lock with an MCSg lock where all existing lock users (i.e., functions) are guests. Then, the developer can gradually identify the few most frequent lock users and modify them to be regular users with MCS qnodes. While the lock will enjoy high scalability as an MCS lock, the majority of lock users can still remain as guests without any code change.

The third scenario is to use MCSg as a building block for combined locks, such as cohort locking. MCSg keeps the simplicity and pluggability of MCS locks, hence it can be used wherever MCS locks could be used. For example, C-MCSg-MCS instead of C-MCS-MCS and C-BO-MCSg++ instead of C-BO-MCS can provide the guest user functionality in addition to the high scalability of the original cohort locks.

Finally, we have also proposed an extended version of MCSg, MCSg++. MCSg++ provides guest users with a guaranteed lock acquisition on highly contended locks. MCSg++ also alleviates priority inversion between groups of regular users at the cost of less simplicity (e.g., no longer a single word). We recommend developers to start with MCSg because of its simplicity and perfect compatibility to the original MCS lock. As frequent code paths are upgraded to regular users, MCSg rarely poses any issue. When it is somehow difficult to upgrade a frequent code path to be a regular user (e.g., an *untouchable* code path in a complex legacy codebase), we recommend MCSg++ to ameliorate the issues.

Acknowledgement

We would like to express our sincere appreciation to several kernel hackers at Hewlett Packard: Davidlohr Bueso, Wai Man Long, Scott Norton, and especially Jason Low. They gave us helpful discussions on this paper and made significant contributions to improve the scalability of the Linux kernel. Their contribution in Linux kernel 3.10 and onwards is one of the most influential achievements that applies the ideas in the locking community to the real world. We acknowledge editorial help from our colleagues at Hewlett Packard Labs and constructive suggestions from the anonymous reviewers as well as the passionate shepherd of this paper, Michael Scott.

References

[1] M. Auslander, D. Edelsohn, O. Krieger, B. Rosenburg, and R. Wisniewski. Enhancement to the MCS lock for increased functionality and improved programmability. *U.S. patent application number 20030200457 (abandoned)*, 2003.

[2] D. Bueso and S. Norton. An overview of kernel lock improvements. *LinuxCon North America*, 2014. http://events.linuxfoundation.org/sites/events/files/slides/linuxcon-2014-locking-final.pdf.

[3] J. Corbet. MCS locks and qspinlocks. *LWN*, 2014. https://lwn.net/Articles/590243/.

[4] T. Craig. Building FIFO and priority-queuing spin locks from atomic swap. *Technical Report TR 93-02-02, Department of Computer Science and Engineering, University of Washington*, 1993. ftp://ftp.cs.washington.edu/tr/1993/02/UW-CSE-93-02-02.pdf.

[5] D. Dice, V. J. Marathe, and N. Shavit. Lock cohorting: a general technique for designing NUMA locks. *PPoPP*, pages 247–256, 2012.

[6] D. Gifford and A. Spector. Case Study: IBM's System/360-370 Architecture. *CACM*, 30(4):291–307, Apr. 1987.

[7] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: A Scalable Storage Manager for the Multicore Era. In *EDBT*, pages 24–35, 2009.

[8] H. Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. *SIGMOD*, pages 691–706, 2015.

[9] L. Lamport. A new solution of dijkstra's concurrent programming problem. *CACM*, 17(8):453–455, Aug. 1974.

[10] S. Li, T. Hoefler, and M. Snir. NUMA-aware shared-memory collective communication for MPI. *HPDC*, pages 85–96, 2013.

[11] J. Low. Personal communication, 2015.

[12] J. Low et al. [patch] timer: Improve itimers scalability. *LKML*, 2015. https://lkml.org/lkml/2015/10/14/822.

[13] M. Luo, D. K. Panda, K. Z. Ibrahim, and C. Iancu. Congestion avoidance on manycore high performance computing systems. *ICS*, pages 121–132, 2012.

[14] P. S. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. *International Symposium on Parallel Processing*, pages 165–171, 1994.

[15] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM TOCS*, 9 (1):21–65, Feb. 1991.

[16] J. M. Mellor-Crummey and M. L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. *PPoPP*, pages 106–113, 1991.

[17] M. L. Scott. Shared-memory synchronization. *Synthesis Lectures on Computer Architecture*, 8(2):1–221, 2013.

[18] M. L. Scott and W. N. Scherer. Scalable queue-based spin locks with timeout. *PPoPP*, pages 44–52, 2001.

[19] Transaction Processing Performance Council. TPC benchmark C standard specification, revision 5.11. 2010.

[20] Y. Yan, S. Chatterjee, Z. Budimlic, and V. Sarkar. Integrating mpi with asynchronous task parallelism. In *EuroMPI*, pages 333–336, 2011.