Locking Made Easy

Jelena Antić[*] , Georgios Chatzopoulos, Rachid Guerraoui, Vasileios Trigonakis[†]
École Polytechnique Fédérale de Lausanne, Switzerland
jelena.antic@alumni.epfl.ch, {first}.{last}@epfl.ch

ABSTRACT

A priori, locking seems easy: To protect shared data from concurrent accesses, it is sufficient to lock before accessing the data and unlock after. Nevertheless, making locking efficient requires fine-tuning (a) the granularity of locks and (b) the locking strategy for each lock and possibly each workload. As a result, locking can become very complicated to design and debug.

We present GLS, a middleware that makes lock-based programming simple and effective. GLS offers the classic lock-unlock interface of locks. However, in contrast to classic lock libraries, GLS does not require any effort from the programmer for allocating and initializing locks, nor for selecting the appropriate locking strategy. With GLS, all these intricacies of locking are hidden from the programmer. GLS is based on GLK, a generic lock algorithm that dynamically adapts to the contention level on the lock object. GLK is able to deliver the best performance among simple spinlocks, scalable queue-based locks, and blocking locks. Furthermore, GLS offers several debugging options for easily detecting various lock-related issues, such as deadlocks.

We evaluate GLS and GLK on two modern hardware platforms, using several software systems (i.e., HamsterDB, Kyoto Cabinet, Memcached, MySQL, SQLite) and show how GLK improves their performance by 23% on average, compared to their default locking strategies. We illustrate the simplicity of using GLS and its debugging facilities by rewriting the synchronization code for Memcached and detecting two potential correctness issues.

CCS Concepts

•Computing methodologies → Shared memory algorithms; Concurrent algorithms; •Computer systems organization → Multicore architectures;

Keywords

Locking; Adaptive Locking; Locking Middleware; Locking Runtime; Synchronization; Multi-cores; Performance

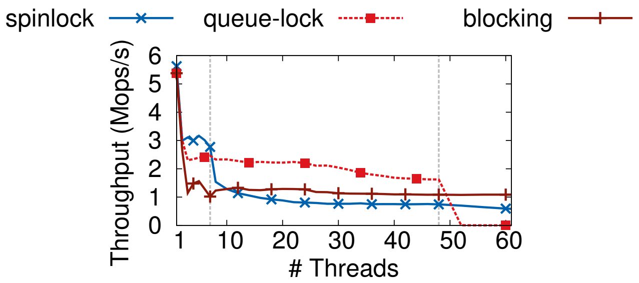

Figure 1: Different lock strategies under varying contention.

1. INTRODUCTION

Locking is arguably the most widely-used synchronization technique in concurrent programming. Essentially, every modern system makes use of locks. Most operating systems (e.g., the Linux kernel), DBMSs (e.g., MySQL), and key-value stores (e.g., Memcached) heavily rely on locks. The wide adoption of locking can be mainly attributed to the need for a simple and fast technique for synchronization. Indeed, locking seems simple at a first glance. However, in practice, there is more to using locks efficiently in systems than meets the eye.

Typically, the programmer must (i) map data to locks, (ii) declare locks, (iii) allocate and initialize locks, (iv) use the locks (i.e., *lock* and *unlock*), and (v) destroy and deallocate the locks. Of course, she has to also select which lock algorithm(s) to use.

These steps are inflexible and error-prone. Every lock object must be explicitly declared, hence, changing the mapping of data to locks can be cumbersome. For example, moving away from the global lock in the Linux kernel required significant effort [14, 43]. Common mistakes during lock-based programming include [19]: (i) accessing uninitialized locks, (ii) trying to acquire the same lock twice, (iii) releasing an already free lock, (iv) releasing a lock that belongs to another thread, and, of course, (v) deadlocks. Theses issues can be difficult to debug.

To make things even more complicated, in order to achieve good performance, one has to fine-tune the locking strategies in use. Indeed, there is no one-size-fits-all lock algorithm [21, 33]. Consider Figure 1 that depicts the performance of different locking strategies under increasing contention. Simple spinlocks are very fast on low contention but do not scale well. Queue-based locks scale well, but are slower than spinlocks on low contention and suffer on multiprogrammed environments (i.e., when the number of threads is larger than the number of hardware contexts). Blocking locks (e.g., `pthread` mutexes) are suitable on that latter case, but are very slow on non-multiprogrammed environments.

[*] Work done while the author was at EPFL. Currently at Google.

[†] Author names appear in alphabetical order.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Middleware'16, December 12 - 16, 2016, Trento, Italy

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4300-8/16/12. . . $15.00

DOI: http://dx.doi.org/10.1145/2988336.2988357

Consequently, selecting the "wrong" algorithm can have detrimental performance results. For example, recent work [21, 30, 32, 33, 46] has shown significant performance improvements in middleware software, such as Memcached and LevelDB, by modifying locking. Accordingly, the designer must select the appropriate locking technique *for each* lock object in order to optimize her system as much as possible. However, the "correct" lock algorithm strongly depends on the lock contention levels which, in turn, depend on the workload. Additionally, workloads tend to change over time, thus the correct lock algorithm is a function of time. Ideally, we would like a single lock algorithm that can dynamically adapt to the workload, delivering the best performance among spinlocks, queue-based locks, and blocking locks, at any point in time.

In this paper, we question whether we can have the cake and eat it too. We present GLS, a *generic locking service* designed to solve all the aforementioned intricacies related to lock-based programming. GLS is a middleware that provides the traditional lock interface, with two main functions to acquire and release a lock. However, in contrast to traditional lock libraries, the locks are fully controlled by GLS, thus the developer does not need to select a lock algorithm, nor to declare, allocate, or initialize any locks. In fact, any arbitrary memory address can be used as a parameter to `gls_lock()`. GLS takes care of mapping the input address to a lock object.

Under the hood, GLS uses a fast concurrent hash table for mapping addresses to objects. As most locks are repeatedly used, this hash table converges to a read-mostly hash table, thus incurring low overhead. Having a central data structure, where all locks are kept, allows us to develop very useful debugging extensions on top of GLS. GLS can detect accesses to uninitialized locks, double locking, etc. Additionally, in §4.2, we show how to build low-overhead deadlock detection on top of GLS, as well as lock profiling tools.

More importantly, the programmer does not need to select the lock algorithm for each individual lock; GLS automatically adapts the lock algorithm to the workload. GLS comes with an adaptive lock called GLK, standing for *generic lock*. GLK keeps track of the contention levels in order to dynamically adapt the algorithm to the needs of the workload. On low contention, GLK behaves as a simple spinlock (i.e., ticket lock [47]). On high contention, GLK turns into a scalable queue-based lock (i.e., MCS lock [47]). On high system load (multiprogramming), GLK transforms to a blocking lock (i.e., mutex lock). The adaptiveness of GLK is per lock, thus a system might contain various locks that operate on different modes depending on their contention levels (e.g., MySQL in §5.2). Naturally, in a system with locking already in place, GLK can be used with or without GLS to minimize the overhead.

Additionally, GLS offers explicit interfaces for many state-of-the-art lock algorithms: test-and-set, test-and-test-and-set, ticket, MCS, CLH [20], and mutex, and allows for easy deployment of more algorithms. These interfaces can be used to manually specify the lock algorithm to be employed for a specific lock object. In §5.1, we show how we use this interface to re-implement locking in Memcached from scratch. The resulting implementation contains 26% less lock-related code than the initial design and delivers 14% higher throughput on our benchmarks.

We evaluate GLS and GLK on various microbenchmarks and software systems. Based on microbenchmark results, we show that GLS adds low overheads compared to directly using locks. Additionally, we show that GLK is always able to capture the needs of the underlying workload, thus adapting to the best algorithm for each workload phase. We plug GLK in various systems: HamsterDB, Kyoto Cabinet, Memcached, MySQL, and SQLite, by overloading the `pthread` mutex library. We improve the performance of these systems by 23% on average, with essentially zero effort. Finally, using the debugging facilities of GLS, we detect two potential correctness issues in Memcached.

The main contributions of this paper are as follows:

- GLS, a middleware locking service that simplifies lock-based programming;
- GLK, a practical lock algorithm that dynamically adapts to the contention of the underlying workload;
- Efficient implementations of GLS and GLK in our locking libraries – available at https://lpd.epfl.ch/site/gls;
- A novel approach for dynamically detecting correctness issues in lock-based systems.

Of course, neither GLS, nor GLK are silver bullets. GLS adds both latency and memory overheads compared to plain locking. Additionally, the adaptiveness of GLK adds a low performance overhead compared to directly using the corresponding lock algorithm. Therefore, a fully-customized system, for a fixed workload configuration, with every lock object set to the correct lock algorithm, will inevitably be slightly faster than with GLK. Finally, one can devise scenarios where GLK will be frequently switching modes. However, we never observe such behavior in practice.

The rest of the paper is organized as follows. In §2, we recall background notions regarding lock-based programming. We introduce GLK in §3 and use it in GLS in §4. We use GLS and GLK to simplify and optimize modern software systems in §5. We discuss related work in §6, and we conclude the paper in §7.

2. LOCK-BASED PROGRAMMING

Locks are objects with two states: *busy* or *free*. A lock can be either free, or have a single *owner*, namely the thread that holds the lock. Locks ensure *mutual exclusion*: Only the owner can proceed with its execution, while any concurrent threads are waiting behind the lock. Locks offer two operations: *lock* and *unlock*. The former is used to *acquire* the lock (i.e., make the current thread the owner of that lock). Unlock *releases* the lock so that it can be subsequently acquired by another thread.

There are numerous lock algorithms. These algorithms mostly differ in the way they handle contention, namely the situation where threads are waiting for a busy lock to become free. Typical designs employ either *busy waiting*, or *blocking* for waiting. With busy waiting, waiting threads remain active, polling the lock until they manage to acquire it. With blocking, waiting threads release their hardware context to the OS. The OS is responsible for unblocking these "sleeping" threads when the owner releases that lock.

There exist various busy-waiting algorithms. Simple spinlocks, such as *test-and-set* (TAS), *test-and-test-and-set* (TTAS), and *ticket lock* [47] (TICKET), use a single memory location on which threads are busy waiting. Spinlocks are fast under low contention due to their simplicity. However, simple spinlocks might generate a lot of coherence traffic on the single memory location (i.e., cache line) of the lock [10]. Queue-based locks (e.g., MCS [47], CLH [20]) generate a queue of waiting nodes so that each thread is spinning on a unique location when busy waiting. Queue-based locks thus remove the single-memory-location bottleneck of simple spinlocks.

The most well-known blocking lock is the *mutex*-lock (MUTEX), part of the `pthread` library. Because the overheads of the OS for blocking and unblocking a thread are high, blocking locks typically employ a busy-waiting period before putting threads to sleep.

At a first glance, programming with locks looks simple. Locking is appealing precisely because of this simplicity. Nevertheless, using locks efficiently (i.e., achieving correct and fast designs) can become cumbersome, mainly because of various correctness and performance problems that are often associated with locks.

Programming with Locks.

In short, programming with locks involves the following steps:

1. Recognizing the various critical sections and the data they protect (i.e., the *granularity* of each lock).
2. Selecting and using concrete lock implementations.
3. Declaring the lock objects. Non-statically allocated locks must not only be declared, but allocated as well.
4. Initializing the locks.
5. Using the locks through their interface (i.e., *lock*, *trylock*, *unlock*) in order to protect the critical sections.
6. Destroying and possibly deallocating the locks.

If the developer implements any of these steps improperly, correctness and performance issues can emerge.

System Correctness with Locks.

The most well-known bugs associated with locks are *deadlocks*. In a deadlock, a thread has acquired a lock l_0 and is waiting for an already acquired lock l_1, the owner of which is waiting for a different acquired lock l_2 and so forth. Finally, there is a thread that has acquired l_n and is waiting for l_0, in which case none of the threads can make progress. Deadlocks are often a result of acquiring locks in the wrong order and are notoriously hard to debug.

Another hard-to-detect issue with locks is using uninitialized locks (i.e., trying to (un)lock a non-initialized lock object). This issue results in executions where the system may or may not work properly, depending on the initial value of the lock object's memory. Other common mistakes with locks are trying to lock the same object twice, unlocking a lock that is already free, or releasing a lock that has been acquired by another thread.[1] As with uninitialized locks, the latter issues can break the system.

These issues are common in practice [19]. In §4.2, we present an easy-to-use debugging extension of GLS for detecting all of these issues. In fact, we use GLS to detect and solve two of these issues in Memcached (§5.1).

System Performance with Locks.

In addition to correctness problems, locks might become a performance bottleneck, mainly due to two different reasons.

First, highly-contended locks can easily become a bottleneck (e.g., the global lock in Memcached v1.4.13 [30] or in the Linux kernel [14, 43]). Removing these bottlenecks might require significant effort. The designer must redesign the critical sections and change the granularity of the locks. This process is of course prone to the correctness issues discussed earlier.

The second and more important reason is that different lock algorithms are suitable for different workloads [21, 33]. As we show in Figure 1, simple spinlocks shine under very-low contention, queue-based locks are by far the best under medium to high contention, and blocking locks are necessary under multiprogrammed workloads. Choosing the "wrong" algorithm under multiprogramming can lead to *livelocks*, situations where although the threads execute, the system throughput is close to zero (see MySQL in §5.2).

Naturally, while designing and implementing a general-purpose system, the designer cannot predict every single deployment or potential runtime fluctuation of the behavior of the workload. Therefore, she must choose the common-denominator lock algorithm (i.e., the one that works even on multiprogramming), namely mutex. Unfortunately, studies have shown that mutex is slow compared to other algorithms in the absence of multiprogramming [21].

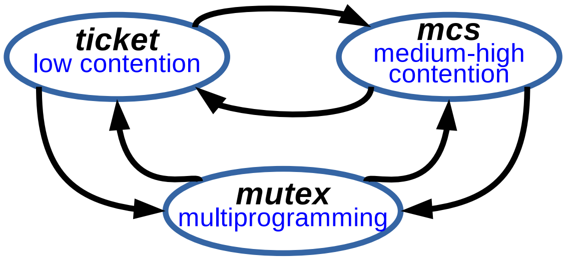

Figure 2: The three modes of GLK.

Locking on Modern Multi-cores.

In order to properly substantiate our claims regarding GLS and GLK, we perform our experiments on two modern multi-cores from Intel. We detail the characteristics of these platforms below:

Name	Model	#Cores	Freq	L1	L2	LLC
Haswell	E5-2680 v3	12	2.5GHz	32KB	256KB	30MB
Ivy	E5-2680 v2	10	2.8GHz	32KB	256KB	25MB

Haswell and *Ivy* are both 2-socket Intel Xeon machines, with 128GB of main memory attached to each socket. Both platforms run Ubuntu 14.04.3 with Linux kernel 3.13.0-63 and `glibc` 2.19, with dynamic voltage and frequency scaling (DVFS) disabled.

3. GLK: A GENERIC LOCK ALGORITHM

As we explain in §1 and §2, different lock algorithms are suitable for different workloads. Simple spinlocks (e.g, ticket locks) are the fastest locks under low contention. Queue-based spinlocks are more suitable under high contention. For instance, MCS locks were recently introduced in the Linux kernel to improve scalability of highly-contended locks [17]. Nevertheless, spinlocks cannot properly handle multiprogrammed environments. Due to busy waiting, threads waiting behind a spinlock might occupy a hardware context instead of a thread that could perform some useful work. This phenomenon is exacerbated in fair locks,[2] where the thread that is next in acquiring the lock might not be scheduled.

Accordingly, we design the *generic lock* algorithm (GLK), based on the premise that there is no one-size-fits-all lock algorithm. GLK adapts to the workload in order to select the most suitable way of waiting behind a busy lock. In brief, GLK collects contention information and, based on this information, periodically adapts the mode it operates in, between *ticket*, *mcs* and *mutex* (Figure 2). We use TICKET instead of other simple spinlocks, as TICKET is fair and more scalable than TAS and TTAS [21]. In what follows, we first describe the design of GLK and then perform a sensitivity analysis for the configuration parameters of GLK on our target platforms.

The GLK Structure.

Figure 3 includes the code for the GLK structure. The structure contains a `lock_type` flag that indicates the current mode of the lock, the three lock objects for *ticket*, *mcs*, and *mutex*, and two counters for gathering statistics (`num_acquired` and `queue_total`). The former counter measures the number of completed critical sections, while the latter contains the amount of queuing behind the lock.

[1] Releasing a lock owned by another thread, or acquiring a lock twice, can be also used as a feature.

[2] Fair locks, such as ticket locks, offer FIFO ordering of lock acquisitions. Queue-based locks are FIFO by design because they are implemented as queues.

```
typedef struct glk {
    glk_type_t lock_type;
    glk_ticket_lock_t ticket_lock;
    glk_mcs_lock_t mcs_lock;
    glk_mutex_lock_t mutex_lock;
    uint32_t num_acquired;
    uint32_t queue_total;
} glk_t;
```

Figure 3: The GLK structure.

Measuring Contention.

GLK contains a configuration parameter on how often to collect contention statistics. On spinlock-mode (i.e., *ticket* or *mcs*), we measure contention as the amount of queuing behind the lock.

Ticket locks provide this queuing information by design [47]. A ticket lock comprises two counters: `ticket` and `owner`. To acquire the lock, the thread grabs a ticket `t` by atomically incrementing (and fetching) the `ticket` field. It then spins until `t` becomes equal to `lock->owner`. To release the lock, the owner simply increments the `owner` field. Consequently, `lock->ticket` − `lock->owner` shows how many threads are waiting behind the lock (plus one for the current owner).

MCS creates a queue of waiting nodes. To measure the amount of queuing, we count the number of nodes while traversing the queue. It is important that this traversal is infrequent, because it breaks the "each node is accessed by a single thread" design goal of MCS.

Finally, the aforementioned queuing information is not sufficient for detecting multiprogramming. Multiprogramming does not relate to the contention of a single lock, but rather to the overall processor load. In other words, multiprogramming might be caused by other applications executing on the machine. Accordingly, to detect multiprogramming, GLK compares the number of running tasks to the number of available hardware contexts. On the first GLK invocation, a background thread is spawned. This background thread is shared across all GLK objects in a system and checks whether there is oversubscription of threads to hardware contexts at the system level, and whether GLK locks must switch to *mutex* mode.

Selecting the GLK Mode.

We perform a sensitivity analysis on when TICKET is faster than MCS (see §3.1). TICKET is consistently faster than MCS when up to three concurrent threads are accessing the lock. We thus configure the transition from *ticket* to *mcs* to happen when the amount of average queuing on a lock is more than three. To avoid frequent, unnecessary transitions, we configure the opposite transition, from *mcs* to *ticket*, to happen when queuing drops below two. Additionally, we keep the exponential moving average of the statistics in order to hide possible short-term workload fluctuations.

The periodic background thread that detects multiprogramming wakes up approximately every 100 us. If the thread detects multiprogramming, it sets a library-wide flag to inform locks that they must switch to *mutex* when they next try to adapt. However, locks switching to *mutex* mode might cause a system-load decrease, as threads block on *mutex*. Thus, the locks might continuously fluctuate between *mutex* and the other modes. To avoid this effect, we detect and avoid consecutive transitions from *mutex* to spinlocks, by exponentially increasing the number of consecutive rounds with no oversubscription required to switch away from *mutex*.

Additionally, locks that face close-to-zero contention do not cause a problem on multiprogramming. In fact, these locks should be simple spinlocks in order to complete these critical sections as fast as possible. Therefore, GLK objects that operate with minimal queuing do not switch to *mutex*, but remain in *ticket* mode.

```
1  void glk_lock(glk_t* lock) {
2      do {
3          glk_type_t curr_type = lock->lock_type;
4          switch(curr_type) {
5          case TICKET_LOCK:
6              glk_ticket_lock(lock);
7              break;
8          case MCS_LOCK:
9              glk_mcs_lock(lock);
10             break;
11         case MUTEX_LOCK:
12             glk_mutex_lock(lock);
13         }

15         if (lock->lock_type == curr_type &&
               !glk_try_adap(lock, curr_type))
16             break;
17         else glk_unlock_type(lock, curr_type);
18     } while (1);
19 }
```

Figure 4: The GLK lock function.

We implement a version of MUTEX for GLK that incorporates the collection of statistics required for adaptation. We also modify the lock to support deadlock detection. Our MUTEX implementation is more lightweight than the one in the `pthread` library, as it does not include the various sanity checks of the latter. These sanity checks (and more) are provided by GLS in debug mode (see §4.2).

Adapting the GLK Mode.

GLK contains a configuration parameter on how often adaptation must be attempted. When adaptation is triggered, the thread currently holding the lock checks the lock statistics and decides which GLK mode to use.

To acquire the lock, a thread (i) accesses the `lock_type` flag of the lock, (ii) acquires the corresponding "low-level" lock object, (iii) checks that `lock_type` has not been modified (if it has been modified, the thread releases the low-level lock object and restarts), and (iv) performs the adaptation (if needed). This approach has the benefit that the lock and unlock functions of TICKET, MCS, and MUTEX can be used almost unmodified. The only modifications in their lock functions are in order to update the queuing and the number of completed critical sections statistics.

The GLK lock function is included in Figure 4. The `glk_try_adap` function first checks whether it should try to adapt the GLK's mode and then checks the statistics to decide on which mode the lock will execute on. If a mode-transition happens, `glk_try_adap` returns true, hence the lock operation is restarted (line 15).

Correctness.

The correctness of GLK is not obvious at a first glance. To understand GLK, we need to consider all possible interleavings of executions. First, we can differentiate between executions with and without concurrent adaptation(s). In the latter case, without any GLK adaptation, GLK is trivially correct because it acts exactly as the low-level lock indicated by `lock->lock_type`.

Then, we will show that only a single thread t can succeed the if statement in line 15 of Figure 4 at a time. If t succeeds `lock->lock_type == curr_type`, then no other thread can succeed this statement before t either adapts the lock, or releases the low-level lock. Any other concurrent thread that uses the same lock type as t will block at the low-level lock. All other concurrent threads will fail the statement. Accordingly, just t (i.e., one thread at a time) has the chance to trigger adaptation and enter the

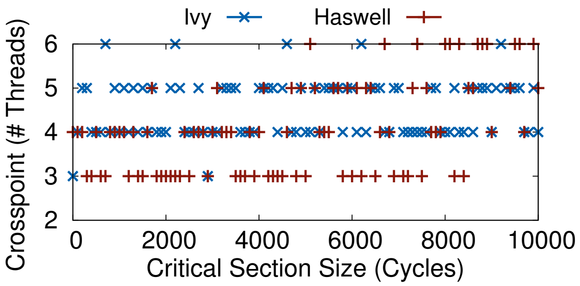

Figure 5: Performance crosspoint: The number of threads that should be concurrently accessing a lock object so that MCS outperforms TICKET.

critical section. If no adaptation happens, `glk_try_adap` returns false and t enters the critical section. Otherwise, `glk_try_adap` returns true, thus t releases the low-level lock and re-runs the while loop. In that case, any thread can take the position of t in succeeding the first clause of line 15, without breaking the correctness of our algorithm.

Including Additional Lock Algorithms.

In GLK, we chose to use a minimum number of algorithms to cover the needs of most workloads. As such, we included a TICKET spinlock algorithm for low contention cases, an MCS queue lock algorithm for contended locks and a MUTEX lock algorithm in order to cope with oversubscription of threads to hardware contexts. In our experience, and as we show in our evaluation of GLK, these locks are sufficient for a wide range of workloads. However, additional lock algorithms can be included in GLK: By modifying the lock selection algorithm previously described and by introducing new selection criteria, users can add more specialized lock algorithms to address cases where such algorithms could yield better performance (e.g., cohort locks [24]).

3.1 GLK Sensitivity Analysis

We perform a sensitivity analysis of the three most important parameters of GLK: (i) the contention thresholds that control the switch from/to *ticket* mode, (ii) adaptation period, and (iii) queue sampling period. Based on our analysis, we set the default GLK settings, which we believe are suitable for most workloads and x86 hardware platforms (in our evaluation we did not need to change any of the settings). However, our scripts for this sensitivity analysis can be used to fine-tune GLK for specific hardware platforms and workloads, if necessary.

ticket vs. mcs Mode.

Figure 5 shows how many threads must contend for a single lock in order for an MCS lock to outperform a TICKET lock. On both of our platforms, TICKET is typically faster than MCS on up to three threads. Accordingly, we configure GLK in our experiments so that it switches from *ticket* to *mcs* mode when the average queuing behind the lock is more than three.

Adaptation and Queue Sampling Periods.

Figure 6 contains the relative throughput of GLK compared to GLK with adaptation disabled, under the extreme scenario of empty critical sections. For the 2-thread execution we fix the non-adaptive GLK to *ticket* mode, while for the 8-thread execution to *mcs* mode. For both experiments the results show that – as expected – short adaptation periods incur a high performance overhead. In both cases, the results stabilize as we increase the adaptation period. Accordingly, in our experiments, we set the adaptation period to 4096 critical sections and the sampling period to 128 critical sections. With these settings, we obtain a sufficient number (4096/128 = 32) of queuing samples per adaptation.

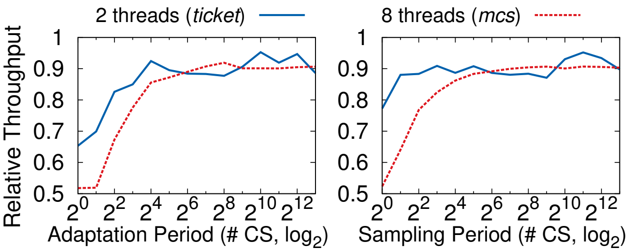

Figure 6: Relative throughput of GLK in *ticket* and *mcs* modes compared to GLK with adaptation disabled, depending on the adaptation period (left) and the queue sampling period (right).

3.2 GLK Evaluation

We start by evaluating the overhead of GLK due to its adaptiveness. We then compare the performance of GLK with TICKET, MCS, and MUTEX on a set of microbenchmarks. In §5, we plug in and compare the performance of these algorithms in software systems. Before we proceed with the evaluation, we describe the experimental settings in our microbenchmarks.

Experimental Methodology.

Threads execute in a loop, performing lock and unlock operations on lock object(s). On every run, we configure (i) the number of threads, (ii) the number of lock objects, and (iii) the duration of the critical section (in CPU cycles). Furthermore, after every loop iteration, threads wait for a short duration to avoid long runs [48]. On every loop iteration, each thread selects a lock object at random. Our results use the median value of 11 repetitions of 10 seconds each. We do not pin threads to cores, but let the OS do the scheduling. Additionally, for fairness and for avoiding false cache-line sharing, we pad all locks to 64 bytes (one cache line). Due to space limitations, we only show the results on our *Haswell* machine (see §2). We get very similar results on our *Ivy* machine.

Overhead.

We evaluate the overhead of GLK due to its adaptiveness. Compared to the vanilla TICKET, MCS, and MUTEX algorithms, GLK additionally executes the following steps: (i) it accesses the `lock_type` flag, (ii) it increments the `num_acquired` counter on almost every critical section,[3] (iii) it updates the queuing statistics of the lock every 128 critical sections, (iv) it tries to adapt the lock type every 4096 critical sections, and (v) it checks the `lock_type` flag for a second time. Additionally, the conditional statement on whether adaptation should be attempted is executed on every lock acquisition.

Figure 7 shows the throughput of GLK on three distinct configurations, each suitable for TICKET, MCS, and MUTEX, respectively. On the single-thread execution, all acquires and releases are local and uncontested (we use empty critical sections). GLK operates on *ticket* mode but is 22% slower than TICKET, mainly because of the switch statement on the `lock_type` when locking and unlocking. Even if we turn adaptation off (i.e., overhead steps (ii)-(iv)), GLK is still 19% slower than TICKET. This comparison reveals an inherent

[3] In TICKET mode, we take advantage of the `current` counter of the lock to avoid incrementing the `num_acquired` counter.

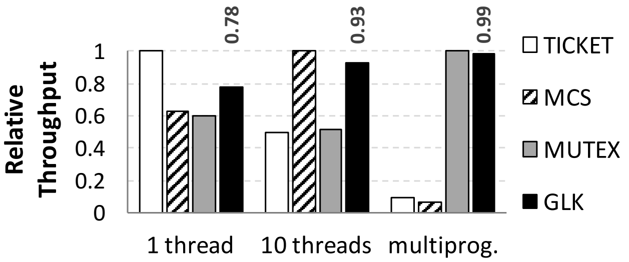

Figure 7: Relative throughput of GLK compared to the best per-configuration lock on various configurations.

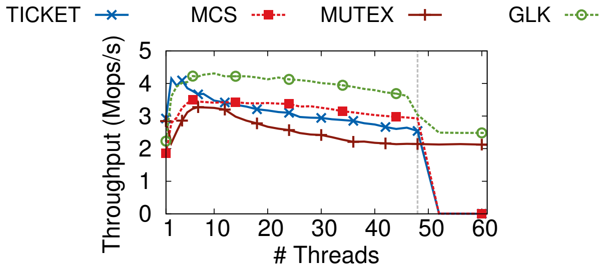

Figure 9: Eight locks on varying contention.

overhead of GLK: An adaptive lock must access a flag to find the lock type to use. Still, GLK delivers 24% and 30% more throughput than MCS and MUTEX, respectively.

The second workload includes 10 threads and empty critical sections, a configuration suitable for MCS locks. In this case, GLK is 7% slower than MCS. Again, even if we remove adaptation from GLK, GLK is still 6% slower than MCS. The overhead in *mcs* mode is lower than in *ticket*, because there is actual contention behind the lock, thus the overhead is partly hidden by waiting. Again, regardless of the overhead of GLK, GLK is still significantly faster than TICKET and MUTEX, respectively.

Finally, the third configuration of Figure 7 involves 10 threads and multiprogramming (we initialize 48 additional threads that just spin locally). In this configuration, GLK is only 1% slower than MUTEX, but much faster than TICKET and MCS. As we have pointed out, fair locks suffer on multiprogrammed workloads.[4]

Single Lock Behavior.

We evaluate the behavior of a single lock as the number of concurrent threads increases. Critical section are 1024 cycles long. Figure 8 includes the results of this experiment. As expected, on a low thread count (i.e., up to 3 threads), TICKET is the fastest lock. On these executions, GLK operates in *ticket* mode and follows the performance of TICKET closely. As we increase the contention further, GLK switches to *mcs* mode, thus behaving similarly to MCS. Finally, when using more than 48 threads, where there is oversubscription of threads to hardware contexts, GLK executes in *mutex* mode, in order to handle multiprogramming.

Multiple Locks Behavior.

We experiment with eight locks as the number of concurrent threads increases, using critical sections of 1024 cycles. On every iteration, each thread selects one of the locks at random, using a zifpian skewed distribution with alpha set to 0.9. In other words, some locks are more frequently utilized than the rest. The two most busy locks serve 34% and 18% of the requests, respectively. Intuitively, in a software system, some locks might be more contended than others: This is one of the cases that we aim at capturing with GLK. The developer must not have the difficult duty of identifying contended locks and customizing their algorithm accordingly.

Figure 9 includes the results of this test. For up to three threads, all eight locks face low contention. Thus, TICKET and GLK (in *ticket* mode) are the fastest. For more threads, the contention on one to two locks increases, thus MCS is the most suitable choice. GLK is able to adapt to *mcs* mode only these highly-contended locks, while keeping the rest in *ticket* mode. This behavior results in GLK being approximately 20% faster than MCS on the non-multiprogrammed configurations. Under multiprogramming, GLK uses *mutex* mode for the two contended threads, while the rest remain in *ticket* mode.

Varying Workload.

We evaluate the behavior of a single lock when the contention varies over time. More precisely, the execution is broken into phases of 0.5-1s.[5] In each phase, the number of threads that execute is selected at random from 1-24. Additionally, 30 background threads run on the processor. These background threads represent other applications that could be executing on the same machine.

Figure 10 shows the throughput of different lock algorithms as a parameter of time and per-phase configuration. On average, GLK delivers 15% higher throughput than the second fastest lock, namely MCS. GLK achieves this by dynamically adapting its configuration during every phase, depending on the needs of the workload. For instance, in phase 3, where contention is very low, GLK switches to *ticket* mode, thus delivering the same performance as TICKET. In contrast, in phases 0-1, where contention is very high, GLK switches to *mcs* to better scale with the number of contending threads. Finally, in multiprogrammed phases, such as 2 and 10, GLK switches to *mutex* in order to avoid potential performance degradation caused by busy waiting.

Conclusions.

GLK can successfully adapt to the needs of the underlying workload at runtime, in order to deliver performance that is close to the best lock algorithm at any point in time. Of course, due to the adaptation overhead, GLK is usually slightly slower than the best per-configuration lock algorithm for fixed workloads. Based on these results, we claim that GLK delivers close-to-optimal performance for any workload and configuration combination. Additionally, as we show in §5, in systems with many locks and complex interactions, GLK can outperform any lock algorithm precisely because of its adaptiveness.

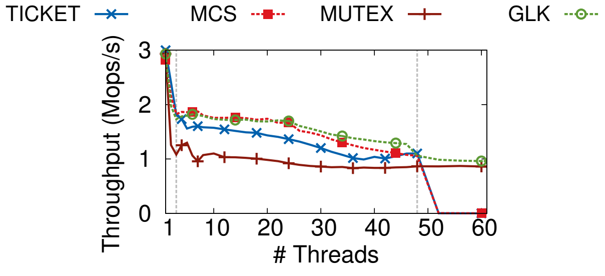

Figure 8: A single lock on varying contention.

[4] There do exist techniques, such as time-published queue-based locks [34], for alleviating this problem.

[5] Note that GLK is able to adapt with much more fine-grain granularity. Assuming 1M acquires/s, adaptation happens approximately every 4 ms, hence the phase length can be as low as roughly 10 ms.

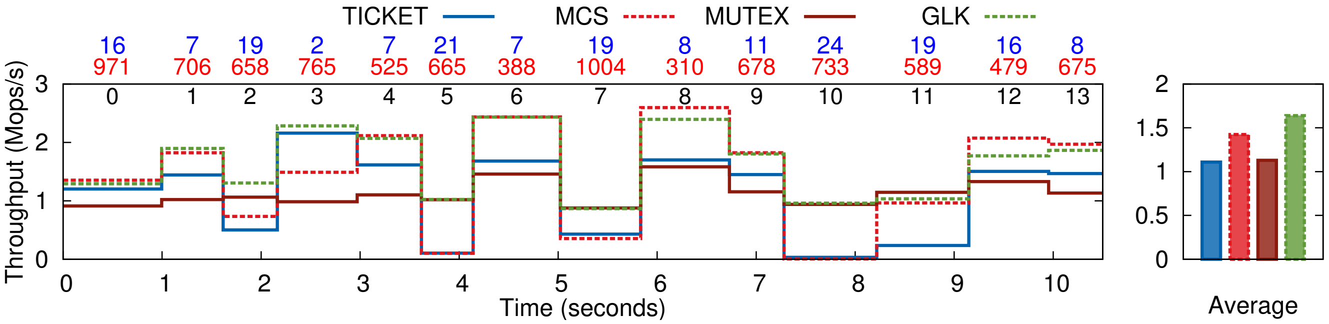

Figure 10: One lock under varying contention levels over time. The numbers on top of the graph represent the number of threads on each phase (blue, top), the critical section duration in cycles (red, middle), and the phase id (black, bottom).

4. GLS: A GENERIC LOCKING SERVICE

GLS is a *generic locking service* that simplifies lock-based programming by handling many of the complexities that developers must typically cope with when using locks. GLS provides the classic lock interface (e.g., functions for locking and unlocking). However, unlike traditional lock libraries, GLS accepts any arbitrary memory address (or even value) as an input parameter to `gls_lock`. With GLS, `gls_lock(17)` is a valid lock invocation. GLS maps the input address to a lock object behind the scenes. Accordingly, the user of GLS does not need to worry about declaring and initializing locks.

Additionally, GLS offers two very useful extensions: (i) a debugging extension that can detect the most common bugs in lock-based programming, and (ii) a profiler that reports the amount of contention on each lock. We first describe the default design and implementation of GLS and then we detail the debugging and profiler extensions of GLS.

4.1 Programming with GLS

Interface.

Table 1 presents the interface of GLS. Apart from the initialization functions, GLS includes various calls for locking and unlocking using different algorithms. These functions accept any arbitrary value as an input, except for `NULL`. The default interface of GLS (`gls_lock`) utilizes the GLK algorithm. In addition, GLS offers an explicit interface to six other algorithms.

Implementation.

GLS is essentially a cache for locating the lock object that corresponds to an address. We implement GLS on top of a modified version of the lock-based CLHT hash table [22]. CLHT has several properties that are necessary in GLS: (i) it uses cache-line-sized buckets, hence operations typically complete with at most one cache-line transfer, (ii) searching for a key is a read-only, wait-free operation, (iii) failing to insert a key is also read-only and wait-free, and (iv) it is resizable. Consequently, when the lock objects used in a system are stable (i.e., there are not many allocations and deallocations of locks), the CLHT hash table in GLS becomes a read-mostly hash table, thus incurring low overhead. The workflow of `gls_lock` is as follows:

```
1 int gls_lock(void* addr) {
2     glk_t* lock = (glk_t*) gls_clht_put(addr);
3     return glk_lock(lock);
4 }
```

In line 2, GLS is searching in the hash table for the lock object that corresponds to the given address. We modify `clht_put` to create and initialize a new lock object for `addr` if `addr` is not found. If `addr` already exists in the hash table, then the corresponding lock object is returned. The `gls_unlock` function uses `gls_clht_get` to fetch the lock which maps to the given address. (As we show in §4.2, if `gls_clht_get` returns `NULL`, GLS detects that an uninitialized lock is used in unlock.) The `gls_A_lock` functions perform the same workflow as `gls_lock`, but initialize and use the lock function of the corresponding algorithm `A`.

Function	Description
`gls_init()`	Initialize GLS
`gls_destroy()`	Stop GLS and cleanup

`gls_lock(void* m)`
`gls_trylock(void* m)`
`gls_unlock(void* m)`
	Lock, trylock, or unlock `m` using GLK algorithm

`gls_A_lock(void* m)`
`gls_A_trylock(void* m)`
`gls_A_unlock(void* m)`
	Lock, trylock, or unlock `m` using algorithm `A`. `A` can be `tas`, `ttas`, `ticket`, `mcs`, `clh`, or `mutex`

`gls_free(void* m)`	Remove `m` from GLS

Table 1: GLS interface.

Lock-cache Optimization.

In locking, the most common pattern involves acquiring and later releasing the same lock, without accessing any other locks in the meantime (the opposite case is called *lock nesting*). Additionally, there is temporal locality of accesses: A lock that is used by a thread will be reused in the near future by the same thread with high probability. To optimize for these patterns, we introduce a single object cache in GLS. This cache keeps track of the address and the lock object of the latest lock that has been accessed. If a lock/unlock operation finds the target address in the cache, there is no need to access the GLS hash table. On a cache miss during locking, the cache is updated with the target address-lock object pair.

Memory Overhead.

GLS adds memory overhead over traditional locking, mainly due to the hash table. CLHT keeps up to three key-value pairs per cache line (64 bytes), hence the minimum overhead introduced by GLS is one-third of a cache line per lock. In our experience, the CLHT used in GLS typically achieves 60-70% occupancy, thus we estimate the overhead per lock to be approximately 32 bytes, or 50% of a cache-line-sized lock object.

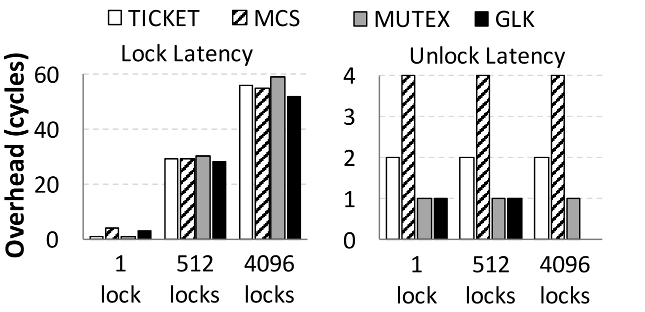

Figure 11: Latency overhead of GLS over directly using locks on a single thread.

Performance Overhead.

We evaluate the performance overhead of GLS compared to using locks directly.

Single Thread. Figure 11 depicts the latency overhead of GLS on a single thread, depending on how many locks are accessed. On each iteration, the thread picks a lock at random. Evidently, with a single lock, the overhead of GLS is just a few cycles due to the lock cache that is always able to locate the lock without accessing the GLS hash table. The same applies for the unlock latencies in this experiment, regardless of the number of locks: Unlock operations always hit on the lock cache (no lock nesting). Without the cache, the unlock latency overhead is close to the GLS overhead in lock functions. With 512 locks, GLS adds approximately 30 cycles overhead, which corresponds to roughly 100% increase in lock latency. As we increase the number of locks, the size of the hash table does not fit in the L1 cache, thus the overhead of GLS increases.

Multiple Threads. Figure 12 compares the throughput of different lock algorithms when used in GLS to direcly using the lock algorithm, with 10 threads competing for 1, 512, or 1024 locks (high, medium and low contention respectively). Each thread randomly chooses among the locks and spends 1024 cycles in the critical section. When locks are not contested (4096 locks to choose from), the overhead of GLS is proportional to the duration of the critical section. In the presence of contention, however, the overhead of GLS can be masked by waiting.

4.2 Debugging with GLS

GLS can be configured to detect several lock-related issues. In order to detect potential bugs, including deadlocks, GLS in debug mode keeps track of the owner of each lock object. In what follows, we describe the issues that GLS can detect. Note that some of these issues could be seen as features depending on the semantics of the specific lock algorithm in use.

Uninitialized Locks.

GLS handles the mapping of addresses to lock objects. Accordingly, GLS can detect when a thread is trying to access an uninitialized lock. Upon releasing a lock, uninitialized locks are detected when the target address does not exist in the hash table, which means that the lock was not acquired before. In order to detect uninitialized locks while trying to acquire a lock, GLS adds special values in the overloaded MUTEX locks for static initialization (instead of the default ones used by the `pthread` mutex). When trying to acquire a lock, if the call to `gls_clht_put` does not find an address-lock mapping, there are two possible cases: either the lock has been statically initialized, or the lock has not been initialized at all. To discern between the two, GLS checks whether the MUTEX object contains the special values used for static initialization. If that is the case, the lock was statically initialized, otherwise an error is detected. Due to the static declaration of these locks, it is certain that if not initialized, they will contain a zero value [1].

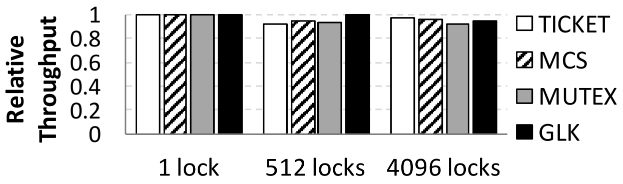

Figure 12: Relative throughput of GLS over directly using locks on 10 threads.

Double Locking.

GLS can check whether the current owner of a lock tries to acquire that same lock again. GLS implements this functionality by comparing the id of the thread that is performing the operation with the id of the lock owner. Of course, double locking is a subset of deadlock detection.

Releasing a Free Lock.

Releasing an already free lock can either break some lock algorithms (e.g., TICKET), or result in race conditions where a thread is trying to acquire the lock while another is falsely releasing it at the same time. GLS checks whether an unlock function call operates on a lock that is already free.

Releasing a Lock with Wrong Owner.

GLS checks that the owner id of the lock to be released is the same as the id of the thread performing the unlock operation.

Deadlocks.

Deadlocks can be very hard to detect and debug [28]. GLS implements a technique for detecting and reporting deadlocks at runtime. GLS implements deadlock detection in the following steps:

1. GLS augments the hash table with a *waiting* array that indicates which lock each thread is waiting on (if any).
2. Before calling the lock/trylock function in the `gls_clht_put` invocation, GLS records that the corresponding thread is waiting on the target address.
3. When the lock is acquired (or the trylock function completes), GLS updates the waiting structure to indicate that this thread is not waiting behind that lock anymore.
4. Additionally, the owner of that lock is set to the thread id. For trylock, the owner id is set iff the lock was successfully acquired.
5. On lock release, the owner id for that lock is cleared.

Based on the aforementioned steps, if the thread is waiting behind a lock for a long time (i.e., more than a second), GLS triggers the following deadlock-detection procedure:

```
// the invoking thread is waiting on wait_lock
wait_on  = wait_lock;
do {
     /* find the owner of the lock that the
        previous thread is waiting on */
    owner_id = gls_debug_get_onwer(wait_on);
    // if the invoking thread re-appears
    if (owner_id == gls_get_id())
         gls_debug_deadlock_print();
    // find the lock that owner_id is wating on
    wait_on = gls_debug_get_wait_on(owner_id);
 } while (wait_on_lock != NULL);
```

In short, the invoking thread $owner_0$ tries to find a cycle with $owner_0 \rightarrow waiting_0 \rightarrow owner_1 \rightarrow waiting_1 \rightarrow ... \rightarrow owner_0$ relationships. If such a cycle exists (a series of relationships that starts and ends with the same id) a deadlock is detected. In that case, GLS prints the details of the cycle as well as the backtrace of the call that caused the deadlock:

```
[GLS]WARNING> DEADLOCK 0x1ad0010 - cycle detected
[2   waits for 0x1ad0010] ->
[9   waits for 0x1acfff4] ->
[8   waits for 0x1acfff8] ->
[2   waits for 0x1ad0010]
[BACKTRACE] Execution path:
[BACKTRACE]#0 ./stress_error_gls(glk_lock+0x4b)
gls/src/glk.c:392
[BACKTRACE]#1 ./stress_error_gls(gls_lock+0x54)
gls/src/gls.c:196
[BACKTRACE]#2 ./stress_error_gls(test+0x248)
gls/bmarks/stress_error_gls.c:160
```

Note that this output is a simplified version of the actual output. GLS can provide more details for debugging the deadlock easier (e.g., automatically setting GDB breakpoints and printing the actual `pthread` ids of threads).

Removing GLS Deadlock-detection Overhead.

The overhead of GLS's detection technique is high, because every lock and unlock operation has to update some metadata in the GLS hash table. In our microbenchmarks, GLS in debugging mode performs up to 4 times slower than without debugging. However, this metadata (regarding waiting-for relations and lock ownership) is only checked when the deadlock-detection procedure is triggered. Accordingly, we can avoid updating this metadata in normal operation and only have the threads update it when they are stuck behind a lock for a significant amount of time. With this approach, deadlock detection happens after a couple of invocations to the detection procedure, but we almost completely remove the overheads while threads operate normally.

4.3 Profiling with GLS

GLS can be configured to operate in a low-overhead profiler mode. In this mode, GLS reports per-lock statistics regarding the average queuing behind the lock, the lock acquisition latency, and the critical-section duration. For instance, in SQLite the output is similar to the following:[6]

```
[GLS] queue: 0.03 | l-lat: 96    | cs-lat: 194
 @ (0x7fe6318eb660:sqlite3.c:pthreadMutexEnter)
[GLS] queue: 4.50 | l-lat: 13963 | cs-lat: 2848
 @ (0x7fe6318eb4e0:sqlite3.c:pthreadMutexEnter)
```

We use this profiler mode to easily detect highly-contended locks that are likely to become a scalability bottleneck as we scale a system. For example, in §5, we use the profiler to better redesign locking in Memcached and to better understand the performance results on various systems.

Additionally, GLK can be configured to print the mode transitions that it performs, as well as the reason behind each transition. This output can be used to better understand potential variations in the workload behavior. It can also be used to decide on a predetermined lock algorithm that is the most suitable for a given lock object in a system (in case the designer selects to use a per-lock custom algorithm).

5. GLS / GLK IN LOCK-BASED SYSTEMS

We modify locking in various concurrent systems in order to evaluate the effectiveness of GLS and GLK. In most systems, modifying locks is as simple as overloading the `pthread` mutex functions with our own lock implementations. We first show how we can easily employ GLS in debugging and optimizing Memcached. We then plug GLK in various modern software systems in order to improve their performance.

5.1 Re-engineering Memcached with GLS

Debugging Memcached.

We notice that when we overload `pthread` mutexes with certain lock algorithms (i.e., TICKET, MCS and GLK), Memcached (v. 1.4.20) hangs. We identify this behavior as the perfect opportunity to show the debugging capabilities of GLS in action. Indeed, GLS reports the following output:

```
[GLS]WARNING> LOCK 0x6344e0 - Uninitialized lock
   [BACKTRACE] #0 memcached/thread.c:662
   [BACKTRACE] #1 memcached/assoc.c:72
[GLS]WARNING> UNLOCK 0x62a494 - Already free
   [BACKTRACE] #0 memcached/slabs.c:836
   [BACKTRACE] #1 memcached/assoc.c:249
```

The first warning is about locking the uninitialized `stats_lock` in `assoc.c`. The second one involves unlocking the `slabs_rebalance_lock` before it is ever acquired. Based on the output of GLS, we easily fix these two issues. Notice that these issues do not manifest with MUTEX, because (i) the `stats_lock` is always initialized to zero due to its static declaration [1], and (ii) unlocking a free lock with MUTEX leaves the lock in the same state. However, the first issue (locking the uninitialized `stats_lock`) is indeed a programming error, as the behavior of MUTEX in such cases is undefined [9]. The second issue (unlocking the `slabs_rebalance_lock` without acquiring it) can either be an error, or not, depending on the configuration of `pthread` mutex.

Optimizing Memcached.

The main goal of GLS is to make programming with locks easier. To showcase using GLS in practice, we re-implement synchronization in the Memcached key-value store from scratch, using the GLS API directly. We remove the code for lock declaration and initialization and replace the calls to `pthread` locks with calls to GLS. We use the `gls_lock` and `gls_unlock` functions and let GLK choose the most suitable locking technique for our workloads. We remove 26% of the synchronization code of the application and modify 186 lines of code in total. This implementation is the way we expect that most GLS users will be using the service, allowing for fast and easy development of lock-based applications.

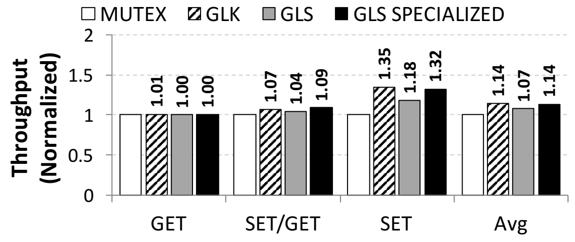

Figure 13: Normalized (to MUTEX) throughput of our Memcached implementations on *Ivy*.

[6] SQLite wraps mutex calls in a function. We could get the actual location of this invocation by printing the backtrace of the call.

HamsterDB [2] Version: 2.1.7 # Threads: 2	An embedded key-value store. We run three tests with random reads/writes, varying the read-to-write ratio among 10% (WT), 50% (WT/RD), and 90% (RD).
Kyoto [3] Version: 1.2.76 # Threads: 4	An embedded NoSQL store. We stress Kyoto with a mix of operations for three database versions (CACHE, HT DB, B+-TREE).
Memcached [5] Version: 1.4.22 # Threads: 8	An in-memory cache. We evaluate Memcached using a Twitter-like workload [42]. We vary the get ratio from 10% (SET), 50% (SET/GET), to 90% (GET). We dedicate one socket on each machine to the Memcached server and the other to the clients.
MySQL [7] Version: 5.6.27	A relational DBMS. We use Facebook's LinkBench and tune the MySQL server following Facebook's guidelines [4] for in-memory (MEM) and SSD-drive (SSD) configurations.
SQLite [8] Version: 3.8.5	A relational DB engine. We use TPC-C with 100 warehouses varying the number of concurrent connections (i.e., 8, 16, 32, and 64).

Table 2: Software systems and configurations.

Figure 13 compares the performance of Memcached when using MUTEX, GLK, as well as GLS using GLK on our *Ivy* platform. The GLS version is 7% slower than directly using GLK. This is due to the overheads of GLS. Still, the GLS version is 7% faster than the default Memcached implementation with MUTEX.

We then set out to better understand the performance of locking in Memcached. We first observe the output of GLK and notice that most locks operate in TICKET mode, which hints at these locks facing little contention. We then use the GLS profiler mode (see §4.3) to understand the different requirements and behavior of each lock. What we discover is that all the locks used in Memcached exhibit low contention, with the exception of specific global locks (e.g. the `stats_lock`).

Accordingly, we devise a second version of Memcached, again using GLS. This time we use the explicit interface of GLS (see Table 1), choosing the lock algorithm that best suits each lock. We use MCS locks for the contended global locks and TICKET locks for the internal hash table and all other locks. GLS allows us to use any lock algorithm by simply modifying the lock and unlock function calls. This enables us to avoid the adaptation overheads and tailor our synchronization code to the optimal strategy. Figure 13 shows that the performance gains from this implementation are significant. Specifically, GLS SPECIALIZED achieves 14% higher throughput than the default MUTEX version, the same as GLK. Programmers with experience in lock-based programming can use this interface to achieve higher performance while still benefiting from the simplicity of GLS.

5.2 Optimizing Systems with GLK

We modify locking in five software systems. We select the set of systems so that they employ locking in diverse ways, including concepts such as global locking, fine-grained locking, reader-writer locks, and conditional variables. Table 2 includes a short description of the systems that we use, as well as the workloads that we use to evaluate them. All of our experiments use a dataset size of 10 GB, except for the MySQL SSD configuration. For this experiment, we use a dataset of 100 GB. We tune each system to achieve maximum throughput and configure the number of threads used based on the maximum-throughput configuration with their default MUTEX locks. Figures 14 and 15 show the throughput of the target systems when employing different lock algorithms on our *Ivy* and *Haswell* platforms respectively.

Overall.

For both platforms, we see that in 14 out of 15 configurations for *Ivy* (and 12 out of 15 for *Haswell*), GLK improves the performance over the default MUTEX lock on the target systems. The performance gains range from 1% to 80% on our *Ivy* machine, and from 3% to 53% on our *Haswell* machine. On average, GLK delivers 25% higher throughput than MUTEX on *Ivy* and 21% higher on *Haswell* by selecting the most appropriate locking technique per lock, per phase, and per configuration. Naturally, there do exist configurations where spinlocks are sufficient. In these cases, the performance of both TICKET and MCS is similar. Additionally, there are few configurations where MUTEX delivers the highest throughput. In these configurations, where GLK does not deliver the highest throughput, GLK is slower only up to 8% than the best performing lock. In contrast, in the configurations where GLK delivers the best performance, we see the power of adaptiveness, as no static algorithm can capture the characteristics of the workload. The overall trends in the results are intuitive. In low-contention configurations, TICKET (i.e., simple spinlocks) delivers the best performance due to its simplicity. On higher contention levels, MCS is the fastest. Finally, in multiprogrammed configurations, blocking locks, such as MUTEX, are necessary.

HamsterDB.

The HamsterDB embedded key-value store [2] relies on a global lock. Of course, the contention on that lock is very high. We measure with the GLS profiler that with N worker threads, the average queuing behind the lock is always close to $N-1$. Consequently, we use just two threads as the application cannot scale further. TICKET delivers the best performance. GLK operates in *ticket* mode, delivering throughput very close to TICKET. MUTEX lags behind in performance because it employs, unnecessary for this workload, block and unblock invocations.

Kyoto Cabinet.

The Kyoto Cabinet NoSQL store [3] comes in two flavors: a hash table, and a B+-tree-based implementation. The hash-table version has two extensions, a store (HT DB) and a cache (CACHE). All three versions protect the main data structure with a highly-contended global reader-writer lock.[7]

Additionally, the hash-table versions use 16 mutexes, each protecting a group of buckets. These locks typically face very low contention: We use GLS in profiler mode to measure the contention and discover that the average queuing behind locks is less than 0.1 and 0.05 for CACHE and HT DB, respectively. However, the throughput of CACHE is approximately 10-times higher than that of HT DB, which means that there is significantly higher traffic on the locks of the former. Additionally, CACHE utilizes up to 10 levels of lock nesting. Nesting with MCS locks is expensive because, for each lock, threads must find and use a separate queue node. These behaviors are reflected in the results of CACHE, where TICKET is significantly faster than MCS. For CACHE, GLK operates in *ticket* mode and thus delivers throughput very close to TICKET. On HT DB, locks are accessed less frequently, hence the performance gains obtained by changing the locks used are smaller than those on CACHE.

The tree-version uses reader-writer locks for the nodes of the tree and mutexes for a custom cache of the tree nodes. These mutexes are highly contended: Both MCS and GLK significantly outperform TICKET and MUTEX.

[7] For TICKET, MCS, and GLK, we overload the `pthread` reader-writer locks with our custom TTAS-based implementation.

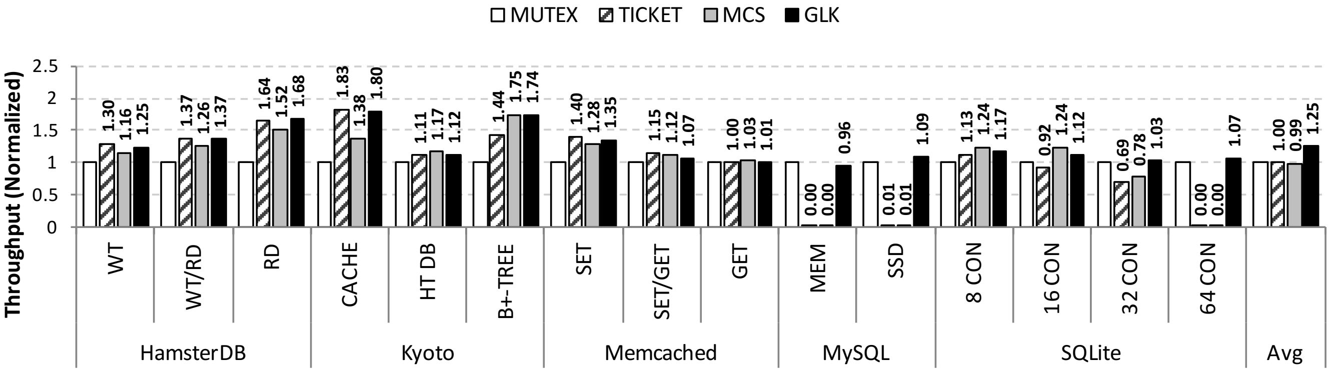

Figure 14: Normalized (to MUTEX) throughput of various systems with different locks on our Ivy machine.

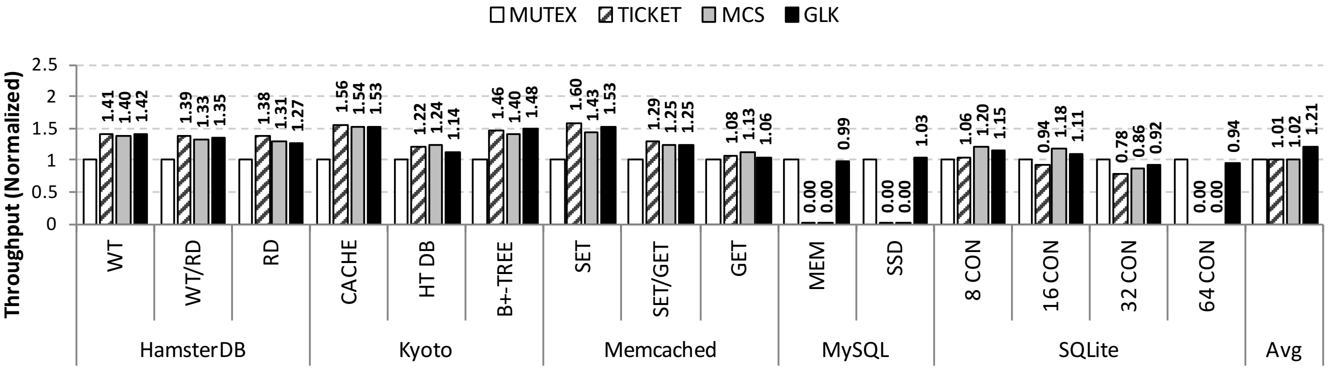

Figure 15: Normalized (to MUTEX) throughput of various systems with different locks on our Haswell machine.

Memcached.

As we describe earlier in this paper (Section 5.1), the locks in our Memcached experiments typically face low contention. Thus, TICKET delivers the highest performance for Memcached. Consequently, with GLK, most locks operate in *ticket* mode, achieving performance that is very close to that of TICKET for both platforms.

MySQL.

MySQL is a complex system that handles most low-level synchronization with custom locks (semaphores). Clearly, neither simple, nor queue-based spinlocks are sufficient for MySQL: In both workloads, MySQL oversubscribes threads to hardware contexts. The result is a livelock for both MCS and TICKET that deliver less than 100 operations per second. Notice that the fairness of these two locks exacerbates the problem. In comparison, a TTAS lock – not shown in the graph – gives 90% and 50% lower throughput than MUTEX on the MEM and SSD workloads, respectively.

On MySQL, we see an inherent limitation, but also the true power of adaptiveness. On the in-memory workload, GLK is 4% slower than MUTEX on *Ivy* and 1% slower on *Haswell*. This difference in throughput is due to the adaptiveness overhead. Directly using the mutex implementation of GLK results in the exact same throughput as MUTEX. However, on the SSD workload we see the power of adaptiveness: Many locks in MySQL are lightly contended, thus using *ticket* mode instead of *mutex* results in 9% higher throughput on *Ivy* and 3% higher throughput on *Haswell*. In complex systems, such as MySQL, it is nearly impossible to manually customize every single lock with the "correct" algorithm.

SQLite.

SQLite is based on a B-tree data structure. SQLite implements concurrency with both coarse and fine-grained locks. SQLite uses a MUTEX for each database (e.g., each new connection), another for memory allocation, and a last one for protecting the database cache. However, the nodes of the B-tree are protected by custom reader-writer locks. The mutexes of SQLite become contended as we increase the number of connections.

With 8 and 16 connections, MCS gives the best performance, with GLK following closely. However, on 32 connections, the workload has some phases with multiprogramming, thus the performance of MCS and TICKET drops. As expected, on 64 connections, using fair spinlocks results in livelocks. In comparison, TTAS – not shown in the graph – delivers 30% lower throughput than MUTEX. On both 32 and 64 connections, GLK achieves slightly better throughput than MUTEX on *Ivy* and follows closely MUTEX on *Haswell*, with lightly contended locks remaining in *ticket* mode.

Conclusions.

GLK is able to adapt and capture the needs of all the 15 workloads on the five systems that we evaluate. Doing so, GLK improves the performance of these systems by 25% and 21% on average on our two platforms, with practically zero effort on the developer's side. Even on the configurations that GLK does not deliver the best performance, due to the overheads of adaptation, GLK is only up to 8% slower than the best performing lock. Consequently, our experimental results validate our claim that GLK can deliver close-to-optimal performance regardless of the configuration.

6. RELATED WORK

Lock Algorithms.

Apart from the traditional spinlock algorithms (e.g., test-and-set, ticket locks [47]), there are several efforts towards designing more scalable locks. Mellor-Crummey et al. [47] and Anderson [10] introduce several alternatives, such as queue-based locks. Luchangco et al. [45] design a hierarchical CLH lock [20] for NUMA architectures. Dice et al. [24] generalize the design of NUMA-aware hierarchical locks by introducing a technique for converting any lock algorithm to be hierarchical. Chabbi et al. [18] and Zhang et al. [60] design locks for NUMA systems, delivering performance in various contention scenarios. David et al. [21] analyze various lock algorithms on different platforms and point out that "every lock has its fifteen minutes of fame." Our GLS middleware and GLK algorithm directly build on top of the observations and the results of such prior work. Concurrently with our work, Guiroux et al. [33] study the performance of lock algorithms in applications and confirm the previous findings. Their LiTL library uses a similar approach to GLS, but with the aim of easily switching between lock algorithms.

Adaptiveness in Locks.

Karlin et al. [39] analyze the cost of busy waiting and blocking and show that adaptive techniques (i.e., adapting the amount of spinning before threads block) deliver the best performance. Similarly, Falsafi et al. [29] design a mutex algorithm that dynamically adapts the amount of spinning when locking/unlocking, to achieve higher energy efficiency. Many modern lock designs, such as the classic mutex lock, both on Linux [6] and on Solaris [56], include this type of adaptiveness. Similarly, the Hotspot 7 JVM modifies its semaphore algorithm at runtime to save memory, based on the idea of thin locks [12]. If there are no threads waiting behind the lock, the JVM represents the lock with just a few bits, otherwise it converts the lock to keep track of the queue of waiting threads. Lock elision [31, 52, 53] aims at reducing the overhead of locking when critical sections do not actually conflict. A thread can optimistically execute its critical section without acquiring a lock. If a data conflict appears, then the thread rolls back and executes the critical section normally. Diniz and Rinard [25] introduce dynamic feedback, where the compiler generates various synchronization policies that can then be selected at runtime based on sampling.

Lim and Agarwal [41] propose reactive locks, an adaptive synchronization scheme that switches between different protocols and waiting strategies. In this sense, GLK can be viewed as an instance of reactive locks. However, GLK is a fully practical adaptive lock algorithm, tailored to modern multicores. GLK includes three concrete lock algorithms which cover the needs of modern software systems, tuned approaches for collecting contention statistics for the locks, as well as a low-overhead method for detecting multiprogramming and switching to a blocking lock when necessary.

Johnson et al. [38] present a locking technique (LC) that decouples waiting from thread scheduling. A global monitor controls how many threads must block, while the remaining threads use time-published MCS locks [34]. The background thread of GLK is similar to this global monitor. However, in GLK, every lock object is a "normal" lock with just a hint on whether it should employ the *mutex* mode.

Debugging Locks.

Debugging locks is notoriously hard. The main techniques available today can be categorized into *static* and *dynamic*. Static techniques identify deadlocks by analyzing the lock and flow graphs of an application and by employing heuristics for identifying the prerequisites for a deadlock [28, 50, 58]. Dynamic deadlock-detection techniques employ ways of monitoring the acquired locks for each thread and discovering cycles in the locks that are being locked [40, 51, 54, 57]. More specifically, Koskinen et al. [40] implement a deadlock detection algorithm that allows lock operations to expire. Pyla et al. [51] implement a runtime for deadlock detection for applications that use `pthreads`. Samak et al. [54] use dynamic analysis and execution traces to identify possible deadlocks. GLS includes a low-overhead, dynamic debugging mode, which can identify issues in lock-based applications.

Alternatives to Traditional Locks.

Transactional memory, in software (STM) [27, 49, 55] or in hardware (HTM) [23, 36], replaces locks with transactions as a concurrency-control mechanism. On the one hand, STMs are typically slower than locks, due to their instrumentation overhead. On the other hand, HTMs are not mature enough and cannot yet fully replace locking [59]. Flat combining [35] is a technique for optimizing coarse-grained locks (e.g., the global lock of a queue). With flat combining, a critical section translates to a message to a dedicated server thread that executes the request on behalf of the invoking thread. Similarly to flat combining, RCL [44] overloads the lock function for highly-contended locks with remote procedure calls on a dedicated server thread. Unlike flat combining and RCL, GLK optimizes both lightly- and highly-contended locks.

Locks in Systems.

There are various efforts in operating systems to minimize sharing [15], to remove lock-related bottlenecks [16, 17], or to completely avoid locks [11, 13] in order to resolve scalability bottlenecks. Similarly, in data stores and DBMSs, recent projects [26, 30, 32, 37, 44, 46] show significant performance improvements in systems such as Memcached and RocksDB, mainly by removing contended locks. GLS is designed to make development of such systems easier, while achieving high performance.

7. CONCLUSIONS

We introduced GLS, a *generic locking service* and the accompanying GLK lock algorithm. GLS is a middleware that makes lock-based system development significantly easier (i) by removing the need for manually handling lock declaration and initialization, (ii) by detecting several common lock-related correctness issues, and (iii) by profiling and reporting per-lock statistics, such as the contention behind a lock and its acquisition latency. GLK simplifies things further, by monitoring the contention levels of a lock in order to dynamically adapt the locking algorithm to the needs of the underlying workload, delivering the best performance among spinlocks, queue-based locks, and blocking locks.

We showed that GLS can simplify concurrent programming, adding low overheads compared to directly using a locking algorithm. We also showed that GLK is always able to capture the needs of the workload, adapting to the best-performing algorithm for each workload phase. We used GLS to re-implement synchronization in Memcached, resulting in 26% less lock-related code and achieving 14% higher throughput. We also used the debugging facilities of GLS to detect two locking issues in the initial Memcached implementation. Finally, we used GLK in five software systems: HamsterDB, Kyoto Cabinet, Memcached, MySQL, and SQLite, comparing against their default locking algorithm, as well as the different modes that GLK can operate in. We improved the performance of these five systems by 23% on average, on two different platforms, using GLK, with essentially zero effort.

Acknowledgements

We wish to thank the anonymous reviewers for their helpful comments on improving the paper. Part of this work was supported by the European Research Council (ERC) Grant 339539 (AOC).

References

[1] C99 Standard. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

[2] HamsterDB. http://hamsterdb.com.

[3] Kyoto Cabinet. http://fallabs.com/kyotocabinet.

[4] Facebook LinkBench Benchmark. https://github.com/facebook/linkbench.

[5] Memcached. http://memcached.org.

[6] Pthread Mutex Lock. https://sourceware.org/git/?p=glibc.git;a=blob_plain;f=nptl/pthread_mutex_lock.c;hb=HEAD. Accessed: 2016-09-07.

[7] MySQL. http://www.mysql.com.

[8] SQLite. http://sqlite.org.

[9] Information technology Portable Operating System Interface (POSIX) Base Specifications, Issue 7. *IEEE/ISO/IEC 9945*, 2009.

[10] T. E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *TPDS '90*.

[11] C. Arad, J. Dowling, and S. Haridi. Message-passing Concurrency for Scalable, Stateful, Reconfigurable Middleware. Middleware '12.

[12] D. F. Bacon, R. B. Konuru, C. Murthy, and M. J. Serrano. Thin Locks: Featherweight Synchronization for Java. PLDI '98.

[13] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. SOSP '09.

[14] A. Bergmann. BKL: That's All, Folks. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=4ba8216cd90560bc402f52076f64d8546e8aefcb. Accessed: 2016-09-07.

[15] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An Operating System for Many Cores. OSDI '08.

[16] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. OSDI '10.

[17] D. Bueso and S. Norton. An Overview of Kernel Lock Improvements. LinuxCon '14.

[18] M. Chabbi and J. Mellor-Crummey. Contention-conscious, Locality-preserving Locks. PPoPP '16.

[19] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating Systems Errors. SOSP '01.

[20] T. S. Craig. Building FIFO and Priority-queuing Spin Locks from Atomic Swap. Technical report, University of Washington, 1993.

[21] T. David, R. Guerraoui, and V. Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. SOSP '13.

[22] T. David, R. Guerraoui, and V. Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. ASPLOS '15.

[23] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco, W. Mesard, M. Moir, K. Moore, and D. Nussbaum. Applications of the Adaptive Transactional Memory Test Platform. TRANSACT '08.

[24] D. Dice, V. J. Marathe, and N. Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. PPoPP '12.

[25] P. C. Diniz and M. C. Rinard. Eliminating Synchronization Overhead in Automatically Parallelized Programs Using Dynamic Feedback. *TOCS '99*.

[26] S. Dong. Reducing Lock Contention in RocksDB. http://rocksdb.org/blog/521/lock. Accessed: 2016-09-07.

[27] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui. Why STM Can Be More Than a Research Toy. *CACM '11*.

[28] D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. SOSP '03.

[29] B. Falsafi, R. Guerraoui, J. Picorel, and V. Trigonakis. Unlocking Energy. ATC '16.

[30] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. NSDI '13.

[31] P. Felber, S. Issa, A. Matveev, and P. Romano. Hardware Read-write Lock Elision. EuroSys '16.

[32] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar. Scaling Concurrent Log-structured Data Stores. EuroSys '15.

[33] H. Guiroux, R. Lachaize, and V. Quéma. Multicore Locks: The Case Is Not Closed Yet. ATC '16.

[34] B. He, W. N. Scherer, and M. L. Scott. Preemption Adaptivity in Time-Published Queue-Based Spin Locks. HiPC '05.

[35] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat Combining and the Synchronization-parallelism Tradeoff. SPAA '10.

[36] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. ISCA '93.

[37] L. Jin. Avoid Expensive Locks in Get(). http://rocksdb.org/blog/677. Accessed: 2016-09-07.

[38] F. Johnson, R. Stoica, A. Ailamaki, and T. Mowry. Decoupling Contention Management from Scheduling. ASPLOS '10.

[39] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical Studies of Competitive Spinning for a Shared-memory Multiprocessor. SOSP '91.

[40] E. Koskinen and M. Herlihy. Dreadlocks: Efficient Deadlock Detection. SPAA '08.

[41] B.-H. Lim and A. Agarwal. Reactive Synchronization Algorithms for Multiprocessors. ASPLOS '94.

[42] K. Lim, D. Meisner, A. Saidi, P. Ranganathan, and T. Wenisch. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. ISCA '13.

[43] R. Lindsley and D. Hansen. Bkl: One Lock to Bind Them All. Ottawa Linux Symposium, 2002.

[44] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote Core Locking: Migrating Critical-section Execution to Improve the Performance of Multithreaded Applications. ATC '12.

[45] V. Luchangco, D. Nussbaum, and N. Shavit. A Hierarchical CLH Queue Lock. Euro-Par '06.

[46] A. Matveev, N. Shavit, P. Felber, and P. Marlier. Read-log-update: A Lightweight Synchronization Mechanism for Concurrent Programming. SOSP '15.

[47] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *TOCS '91*.

[48] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. PODC '96.

[49] M. Mohamedin, R. Palmieri, S. Peluso, and B. Ravindran. On Designing NUMA-aware Concurrency Control for Scalable Transactional Memory. PPoPP '16.

[50] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective Static Deadlock Detection. ICSE '09.

[51] H. K. Pyla and S. Varadarajan. Avoiding Deadlock Avoidance. PACT '10.

[52] R. Rajwar and J. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. *MICRO '01*.

[53] A. Roy, S. Hand, and T. Harris. A Runtime System for Software Lock Elision. EuroSys '09.

[54] M. Samak and M. K. Ramanathan. Trace Driven Dynamic Deadlock Detection and Reproduction. PPoPP '14.

[55] N. Shavit and D. Touitou. Software Transactional Memory. PODC '97.

[56] Sun Microsystems. Multithreading in the Solaris Operating Environment, 2002.

[57] P. Triantafillou. An Approach to Deadlock Detection in Multidatabases. *Information Systems '97*.

[58] A. Williams, W. Thies, and M. Ernst. Static Deadlock Detection for Java Libraries. ECOOP 05.

[59] L. Xiang and M. L. Scott. Software Partitioning of Hardware Transactions. PPoPP '15.

[60] M. Zhang, F. C. M. Lau, C.-L. Wang, L. Cheng, and H. Chen. Scalable Adaptive NUMA-aware Lock: Combining Local Locking and Remote Locking for Efficient Concurrency. PPoPP '16.