Lock cohorting: a general technique for designing NUMA locks

Lock Cohorting: A General Technique for Designing NUMA Locks

David Dice, Virendra J. Marathe
Oracle Labs
dave.dice@oracle.com, virendra.marathe@oracle.com

Nir Shavit
MIT and Tel-Aviv University
shanir@csail.mit.edu

Abstract

Multicore machines are quickly shifting to NUMA and CC-NUMA architectures, making scalable NUMA-aware locking algorithms, ones that take into account the machines' non-uniform memory and caching hierarchy, ever more important. This paper presents *lock cohorting*, a general new technique for designing NUMA-aware locks that is as simple as it is powerful.

Lock cohorting allows one to transform any spin-lock algorithm, with minimal non-intrusive changes, into scalable NUMA-aware spin-locks. Our new cohorting technique allows us to easily create NUMA-aware versions of the TATAS-Backoff, CLH, MCS, and ticket locks, to name a few. Moreover, it allows us to derive a CLH-based cohort abortable lock, the first NUMA-aware queue lock to support abortability.

We empirically compared the performance of cohort locks with prior NUMA-aware and classic NUMA-oblivious locks on a synthetic micro-benchmark, a real world key-value store application memcached, as well as the libc memory allocator. Our results demonstrate that cohort locks perform as well or better than known locks when the load is low and significantly out-perform them as the load increases.

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

General Terms Algorithms, Design, Performance

Keywords NUMA, hierarchical locks, spin locks

1. Introduction

In coming years, as multicore machines grow in size, one can expect an accelerated shift towards distributed non-uniform memory-access (NUMA) and cache-coherent NUMA (CC-NUMA) architectures.[1] Such architectures, examples of which include Intel's multi-socket Nehalem-based systems and Oracle's 4-socket 256-way Niagara-based systems, consist of collections of computing cores with fast local memory (e.g. caches shared by cores on a single multicore chip), communicating with each other via a slower inter-chip communication medium. Access by a core to the local memory, and in particular to a shared local cache, can be several times faster than access to the remote memory or cache lines resident on another chip [12].

[1] We use the term NUMA broadly, noting that it includes Non-Uniform Communication Architecture (NUCA) machines as well.

PPoPP'12, February 25–29, 2012, New Orleans, Lousiana, USA.
Copyright © 2012 ACM [to be supplied]...$10.00

Dice [5], and Radović and Hagersten [12] were the first to identify the benefits of designing locks that improve locality of reference on CC-NUMA architectures by developing NUMA-aware locks: general-purpose mutual-exclusion locks that encourage threads with high mutual cache locality to acquire the lock consecutively, thus reducing the overall level of cache coherence misses when executing instructions in the critical section. Specifically, these designs attempt to minimize *lock migration*. We say a lock L *migrates* if two threads running on a different NUMA clusters (nodes) acquire L one after the other.

Radović and Hagersten introduced the *hierarchical backoff lock* (HBO): a *test-and-test-and-set* lock augmented with a new *backoff scheme* to reduce cross-interconnect contention on the lock variable. Their hierarchical backoff mechanism allows the backoff delay to be tuned dynamically, so that when a thread notices that another thread from its own local cluster owns the lock, it can reduce its delay and increase its chances of acquiring the lock next. This algorithm's simplicity makes it quite practical. However, because the locks are test-and-test-and-set locks, they incur invalidation traffic on every modification of the shared global lock variable, which is especially costly on NUMA machines. The issue of fairness that arises because threads backoff with different delays can be addressed, but requires more tuning parameters, which invariably makes the lock's performance highly unreliable.

Luchangco et al. [15] overcame these drawbacks by introducing HCLH, a hierarchical version of the CLH queue-lock [4]. The HCLH algorithm collects requests on each cluster into a local CLH-style queue, and then has the thread at the head of the queue integrate each cluster's queue into a single global queue. This avoids the overhead of spinning on a shared location and eliminates fairness and starvation issues. The algorithm's drawback is that it forms the local queues of waiting threads by having each thread perform an atomic *register-to-memory-swap* (SWAP) operation[2] on the shared head of the local queue, which becomes a contention bottleneck, implying that the thread merging the local queue into the global one must either wait for a long period (10s of microseconds) or globally merge an unacceptably short local queue.

More recently, Dice et al. [7] showed that one could overcome the synchronization overhead of HCLH locks by collecting local queues using the flat-combining technique of Hendler et al. [8], and then splicing them into the global queue. The resulting NUMA-aware FC-MCS lock outperforms previous locks by at least a factor of 2, but uses significantly more memory and is relatively complicated.

In summary, the HBO lock has the benefit of being simple, but is unfair, and requires significant application and platform dependent tuning. Both HCLH and FC-MCS are fair and deliver much better performance, but are rather complex, and it is therefore questionable if they will be of general practical use.

[2] On some architectures the SWAP operation is emulated using a *compare-and-swap* instruction loop.

This paper presents *lock cohorting*, a new general technique for turning practically any kind of spin-lock or spin-then-block lock into a NUMA-aware lock that allows sequences of threads – local to a given node/cluster – to execute consecutively with little overhead and requiring very little tuning beyond the locks used to create the cohort lock.

Apart from providing a new set of high performance NUMA-aware locks, the important benefit of lock cohorting is that it is a general transformation, not simply another NUMA-aware locking algorithm. This provides an important software engineering advantage: programmers do not have to adopt new and unfamiliar locks into their system. Instead, they can apply the lock cohorting transformation to their existing locks. This will hopefully allow them to enhance the performance of their locks (by improving locality of reference, enabled by the NUMA-awareness property of cohort locks), while preserving many of the original properties of whatever locks their application uses.

1.1 Lock Cohorting in a Nutshell

Say we have a spin-lock of type G that is *thread-oblivious*, that is, allows the acquiring thread to differ from the releasing thread, and another spin-lock of type S that has the *cohort detection property*: a thread releasing the lock can detect if it has a non-empty cohort of threads concurrently attempting to acquire the lock.

We convert a collection of locks S and G into a single NUMA-aware lock by having a single thread-oblivious global lock G and by associating each NUMA cluster i with a distinct local lock S_i that has the cohort detection property. We say a cohort lock is locked if and only if its global lock G is locked. Locks S and G can be of different types. For example, S could be an MCS queue-lock [10] and G a simple *test-and-test-and-set backoff lock* [3] (BO) as depicted in Figure 1. To access the critical section a thread must hold both the local lock S_i of its cluster, and the global lock G. However, the trick is that given the special properties of S and G, once some thread in a cluster acquires G, ownership of the cohort lock can be passed in a deadlock-free manner from one thread in cluster to the next using the local lock S_i, without releasing the global lock. To maintain fairness, the global lock G is at some point released by some thread in the cohort (not necessarily the one that acquired it), allowing a cohort of threads from another cluster S_j to take ownership of the lock.

In more detail, each thread attempting to enter the lock's critical section first acquires its local lock S_i, and based on the state of the local lock, decides if it can immediately enter the critical section or must compete for G. A thread T leaving the critical section first checks if it has a non-empty cohort (some local thread is waiting on S_i). If so, it will release S_i without releasing G, having set the state of S_i to indicate that this is a local release. On the other hand, if its local cohort is empty, T will release G and then release S_i, setting S_i's state to indicate that the global lock has been released. This indicates to the next local thread that acquires S_i that it must re-acquire G before it can enter the critical section. The cohort detection property is therefore necessary in order to prevent a deadlock situation in which a thread leaves the local lock, without releasing the global lock, when there is no subsequent thread in the cohort, so the global lock may never be released.

The cohort lock's overall fairness is easily controlled by deciding when a cluster gives up the global lock. A simple cluster-local policy is to give up the local lock after an allowed number consecutive local accesses. We note that a cohort lock constructed from unfair underlying locks will itself be unfair, but if the underlying locks are fair then the fairness of a cohort lock is determined by the policy that decides when a cohort releases the global lock. If a cohort retains ownership of the global lock for extended periods then throughput may be improved but at a cost in fairness.

The benefit of the lock cohorting approach is that sequences of local threads accessing the lock are formed at a very low cost. Once a thread in a cluster has acquired the global lock, control of the global lock is subsequently passed among contending threads within the cluster – the cohort – with the efficiency of a local lock. In other words, the common path to entering the critical section is the same as a local version of the lock of type S with fairness, as we said, easily controlled by limiting the number of consecutive local lock transfers allowed. This contrasts sharply with the complex coordination mechanisms that create such sequences in the previous top performing HCLH and FC-MCS locks, and the platform-dependent, load-dependent, and application-dependent performance tuning required for the HBO lock.

1.2 Cohort Lock Designs and Performance

It is easy to find efficient locks that are thread-oblivious: the BO or ticket locks have this property, and since the global lock is not expected to be highly contended, they can easily serve as the global locks. With respect to the cohort detection property, there are locks such as the MCS queue lock of Mellor-Crummey and Scott [10] that provide cohort detection by design: each spinning thread's record in the queue has a pointer installed by its successor. There are however locks, for example, BO locks, that require us to introduce an explicit *cohort detection mechanism* to allow releasing threads to determine if other cohort threads are attempting to acquire the lock.

More work is needed when the lock algorithms are required to be abortable. In an abortable lock, simply detecting that there is a successor is not enough to allow a thread to release the local lock but not the global lock. One must make sure there is a viable successor, that is, one that will not abort after the thread releases the local lock, as this might leave the global lock deadlocked. As we show, one can convert the BO lock (which is abortable by design) and the abortable CLH lock [14] into abortable (NUMA-aware) cohort locks, which to our knowledge, are the first set of NUMA-aware abortable queue-locks.

We tested our new lock cohorting transformation on an Oracle SPARC Enterprise T5440^{TM} Server, a 256-way 4-socket multicore machine. Our tests show several variations of cohort NUMA-aware locks that outperform all prior algorithms, and in some situations are over 60% more scalable than FC-MCS, the most scalable NUMA-aware lock in the literature. Furthermore, unlike FC-MCS, we found that cohort lock designs are simple to implement and require significantly lower space than FC-MCS. Our novel abortable NUMA-aware lock, the first of its kind, outperforms the HBO lock (abortable by definition) and the abortable CLH lock [14] by about a factor of 6. Our experiments with memcached [2] demonstrate that in some configuration settings cohort locks can improve the application's performance by over 25%, without degrading performance on all other configurations. Finally, our libc allocator experiments demonstrate how cohort locks can directly benefit multithreaded programs and significantly boost their cluster-level reference locality both for accesses by the allocator to allocation metadata and for accesses by the application to allocated memory blocks. In experiments conducted on a memory allocator stress test benchmark [6], cohort locks allow the benchmark to scale up to nearly a factor of 6X, while all other reported locks provided a scalability gain restricted to about 50%.

We describe our construction in detail in Section 2, both our general approach and seven specific example lock transformations. We provide an experimental evaluation in Section 4.

2. The Lock Cohorting Transformation

In this section, we describe our new lock cohorting transformation in detail. Assume that the system is organized into clusters (nodes)

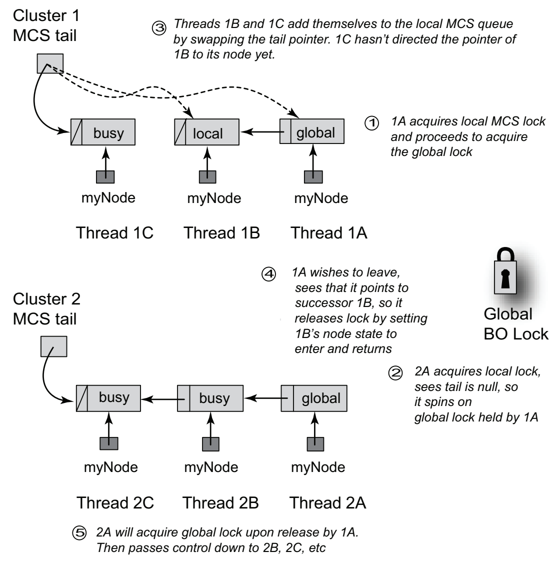
Cluster 1
MCS tail
③ Threads 1B and 1C add themselves to the local MCS queue by swapping the tail pointer. 1C hasn't directed the pointer of 1B to its node yet.
busy
local
global
① 1A acquires local MCS lock and proceeds to acquire the global lock
myNode
myNode
myNode
Thread 1C
Thread 1B
Thread 1A
④ 1A wishes to leave, sees that it points to successor 1B, so it releases lock by setting 1B's node state to enter and returns
Global
BO Lock
Cluster 2
MCS tail
busy
busy
global
② 2A acquires local lock, sees tail is null, so it spins on global lock held by 1A
myNode
myNode
myNode
Thread 2C
Thread 2B
Thread 2A
⑤ 2A will acquire global lock upon release by 1A. Then passes control down to 2B, 2C, etc

Figure 1. A NUMA-aware C-BO-MCS lock for two clusters. A thread spins if its node state is *busy*, and can enter the critical section if the state is *local release*. A thread attempts to take the global lock if it sees the state set to *global release* or if it is added as the first in the queue (setting a null *tail* pointer to its own record).

of computing cores, each of which has a large cache that is shared among the cores local to that cluster, so that inter-cluster communication is significantly more expensive than intra-cluster communication. We use the term *cluster* to capture the collection of cores, and to make clear that they could be cores on a single multicore chip, or cores on a collection of multicore chips (nodes) that have proximity to the same memory or caching structure; it all depends on the size of the NUMA machine at hand. We will also assume that each cluster has a unique *cluster id* known to all threads running on the cluster.

2.1 Designing a Cohort Lock

We describe lock cohorting in the context of spin-locks, although it could be as easily applied to blocking-locks. We assume the standard model of shared memory based on execution histories [9].

A *lock* is an object providing mutual exclusion with *lock* and *unlock* methods, implemented in shared memory, and having the usual safety and liveness properties (see [9]). At a minimum we will require that the locks considered here will provide mutual exclusion and be deadlock-free. In addition, we define the following properties:

Definition A lock *x* is *thread-oblivious*, if in a given execution history, for a *lock* method call of x by a given thread, it allows the matching *unlock* method call (the next *unlock* of x that follows in the execution history) to be executed by a different thread.

Definition A lock x provides *cohort detection* if one can add a new predicate method *alone?* to x so that in any execution history, if there is no other thread concurrently executing a lock method on x, *alone?* will return `true`.

Note that we follow the custom of not using linearizability as a correctness condition when defining our lock implementations. In particular, our definition of *alone?* refers to the behavior of concurrent lock method calls and says *if* rather than *iff* so as to allow false-positives: there might be a thread executing the lock operation and *alone?* (not noticing it) could still return true. False-positives are a performance concern but do not affect correctness. False-negatives, however, could result in loss of progress. This weaker definition is intended to allow for very relaxed and efficient implementations of *alone?*.

We construct a NUMA-aware cohort lock by having each cluster i on the NUMA machine have a local instance S_i of a lock that has the cohort detection property, and have an additional shared thread-oblivious global lock G. Locks S_i, $i \in \{1 \ldots n\}$ (where n is the number of clusters in the NUMA system), and G can be of different types, for example, the S_i could be slight modifications of MCS queue-locks [10] and G a simple *test-and-test-and-set backoff lock* [3] (BO) as depicted in Figure 1.

The *lock* method of a thread in cluster i in a cohort lock operates as follows. The state of the lock S_i is modified so that it has a different detectable state indicating if it has a *local release* or a *global release*.

1. Call *lock* on S_i. If upon acquiring the lock the lock method detects that the state is:
    - A *local release*: proceed to enter the critical section.
    - A *global release*: proceed to call the *lock* method of the global lock G. Once G is acquired, enter the critical section.

We define a special *may-pass-local* predicate on the local lock S_i and the global lock G. The *may-pass-local* predicate indicates if the lock state is such that the global lock should be released. This predicate could, for example, be based on how long the global lock has been continuously held on one cluster or on a count of the number of times the local lock was acquired in succession in a *local release* state. It defines a tradeoff between fairness and performance, as typically the shorter successive access time *may-pass-local* grants to a given cohort, the more it loses the benefit of locality of reference in accessing the critical section.

Given this added *may-pass-local* predicate, the *unlock* method of a thread in cluster i in a cohort lock operates as follows.

1. Call the *alone?* method and *may-pass-local* on S_i.
    - If both return `false`: call the *unlock* method of S_i, setting the release state to *local release*. The next owner of S_i can directly enter the critical section.
    - Otherwise: call the *unlock* method of the global lock G. Once G is released, call the *unlock* method of S_i, setting the release state to *global release*.

As can be seen, the state of the lock upon release indicates to the next local thread that acquires S_i if it must acquire G or not, and allows a chain of local lock acquisitions without the need to access the global lock. The immediate benefit is that sequences of local threads accessing the lock are formed at a very low cost: once a thread in a cluster has acquired the global lock, ownership is passed among the cluster's threads with the efficiency of a local lock. This reduces overall cross-cluster communication and increases intra-cluster locality of reference when accessing data within the critical section.

3. Cohort Lock Designs

Though most locks can be used in the cohort locking transformation, we briefly explain six specific constructions here: The first four are non-abortable (do not support timeouts [13]) locks and the last two are abortable (timeout capable) locks. Of the non-abortable locks, we first present a simple *test-and-test-and-set* *backoff lock* [3] (which we will refer to as the BO lock) based cohort lock that employs a BO lock globally and local BO locks per NUMA cluster. We refer to this lock as the C-BO-BO lock. The second lock is a similar combination of ticket locks [10], which we call the C-TKT-TKT lock. The third is a combination of a global BO lock, and local MCS locks [10] per NUMA cluster. The last non-abortable lock contains MCS locks both globally and locally. For the abortable locks, we first present an abortable variant of the C-BO-BO lock, which we call the A-C-BO-BO lock, and then we present an abortable cohort lock comprising of an abortable global BO lock and abortable local CLH locks [14], which we call the A-C-BO-CLH lock.

3.1 The C-BO-BO Lock

In the C-BO-BO lock, the local and global locks are both simple BO locks. The BO lock is trivially thread-oblivious. However, we need to augment the local BO lock to enable cohort detection by exposing the *alone?* method. Specifically, to implement the *alone?* method we need to add an indicator to the local BO lock that a successor exists.

To that end we add to the lock a new *successor-exists* boolean field. This field is initially `false`, and is set to `true` by a thread immediately before it attempts to CAS the test-and-test-and-set lock state. Once a thread succeeds in the CAS and acquires the local lock, it writes `false` to the *successor-exists* field, effectively resetting it. The *alone?* method will check the *successor-exists* field, and if it is `true`, a successor must exist since it was set after the reset by the local lock winner. *Alone?* returns the logical complement of *successor-exists*.

The lock releaser uses the *alone?* method to determine if it can correctly release the local lock in *local release* state. If it does so, the following lock owner of the local lock implicitly inherits ownership of the global BO lock. Otherwise, the local lock is in the *global release* state, in which case, the new local lock owner must acquire the global lock as well. Notice that it is possible that another successor thread executing *lock* exists even if the field is `false`, simply because the post-acquisition reset of *successor-exists* by the local lock winner could have overwritten the successor's setting of the *successor-exists* field. This type of incorrect-false result observed in *successor-exists* is allowed – it will at worst cause an unnecessary release of the global lock, but not affect correctness of the algorithm.

However, incorrect-false conditions can result in greater contention at the global lock, which we would like to avoid. To that end, a thread that spins on the local lock also checks the *successor-exists* flag, and sets it back to `true` if it observes that the flag has been reset (by the current lock owner). This is likely to lead to extra contention on the cache line containing the flag, but most of this contention does not lie in the critical path of the lock acquisition operation. Furthermore, intra-cluster write-sharing typically enjoys low latency, mitigating any ill-effects of contention on cache lines that might be modified by threads on the same cluster. These observations are confirmed in our empirical evaluation.

3.2 The C-TKT-TKT Lock

The C-TKT-TKT lock has the ticket lock [10] as both the local lock, as well as the global lock. A traditional ticket lock consists of two counters: *request* and *grant*. A thread intending to acquire the lock first atomically increments the *request* counter and then spins, waiting for the *grant* counter to contain the incremented *request* value. The lock releaser subsequently releases the lock by incrementing the *grant* counter.

The ticket lock is trivially thread-oblivious; a thread can increment the *request* and another thread can correspondingly increment the *grant* counter. Cohort detection is also easy in the ticket lock; all the thread needs to do is determine if the *request* and *grant* counters match, and if not, it means that there are more requesters waiting to acquire the lock.

In C-TKT-TKT, a thread first acquires the local ticket lock, and then the global ticket lock. To release the C-TKT-TKT lock, the owner first determines if it has any cohorts that may be waiting to acquire the lock. The *alone?* method is a simple check to see if the *request* and *grant* counters are the same. If not, it means that there are additional requests posted by waiting cohort threads. In that case, the owner informs the next cohort in line that it has inherited the global lock by setting a special *top-granted* field that residees in the local ticket lock. [3] It then releases the local ticket lock by incrementing the *grant* counter. If the *request* and *grant* counters are the same, the owner releases the global ticket lock and then the local ticket lock (without setting the *top-granted* field).

3.3 The C-BO-MCS Lock

The design of the C-BO-MCS lock, depicted in Figure 1, is also straightforward. The BO lock is a simple test-and-test-and-set lock with backoff, and is therefore thread-oblivious by definition: any thread can release a lock taken by another.

We remind the reader that an MCS lock consists of a list of records, one per thread, ordered by their arrival at the lock's *tail* variable. Each thread adds its record to the lock by performing a swap on a shared *tail*. It then installs a *successor* pointer from the record of its predecessor to its record in the lock. The predecessor, upon releasing the lock, will follow the successor pointer and notify the thread of the lock release by writing to a special *state* field in the successor's record.

The MCS lock can be easily adapted to be the local cohort detecting lock as follows. We implement the *alone?* method by simply checking if a thread's record has a non-null successor pointer. The release state is augmented so that instead of simple *busy* and *released* states, the *state* field encodes *busy*, *release local* or *release global*. Each thread will initialize its record state to busy unless it encounters a null tail pointer, indicating it has no predecessor, in which case it is in the *release global* state and will access the global lock.

With these modifications, the global BO lock and local modified MCS locks can be plugged into the cohort lock protocol to deliver a NUMA-aware lock.

3.4 The C-MCS-MCS Lock

The C-MCS-MCS lock comprises a global MCS lock and local MCS locks. The cohort detection mechanism of the local MCS locks is the same as in C-BO-MCS. So the implementation of the local MCS lock remains the same. However, the thread-obliviousness aspect is somewhat more interesting.

A key property of MCS is what is called *local spinning* [10], where a thread spin-waits on its MCS queue node, and is informed by its predecessor thread that is has become the lock owner. Thereafter, the thread can enter the critical section, and release the lock by transferring lock ownership to its queue node's successor. The thread can subsequently do whatever it wants with its MCS queue node; it usually deallocates it. In order to make the global MCS lock thread-oblivious, the thread that enqueues its MCS queue node in the global MCS lock's queue cannot always get its node back immediately after it releases the C-MCS-MCS lock – the node has to be preserved in the MCS queue so as to let another cohort thread release the lock. We enable this feature by using thread-local pools of MCS queue nodes. A thread that posts a request node in the global MCS lock must get a free node from its local pool. On releasing

[3] The *top-granted* flag is reset by the thread that observed it set and took possession of the the local ticket lock.

the global lock, the lock releaser can return the node to the original thread's pool. This circulation of MCS queue nodes can be done very efficiently and does not impact performance of the lock [13].

With this extra modification we achieve a thread-oblivious MCS lock, which can be combined with the local MCS locks that are enabled with cohort detection to deliver the NUMA-aware C-MCS-MCS lock.

3.5 The C-TKT-MCS Lock

The C-TKT-MCS lock combines local MCS queue locks with a global ticket lock. We believe this lock combines the best of C-TKT-TKT and C-MCS-MCS: First, because the global lock is a ticket lock, it does not contain the complexity of circulating queue nodes between threads as in C-MCS-MCS. Second, since the local locks are MCS locks instead of ticket locks, the C-TKT-MCS lock retains their local-spinning property. As we shall see in Section 4, having local MCS locks indeed helps C-TKT-MCS to scale better than C-TKT-TKT.

3.6 Abortable Cohort Locks

The property of *abortability* [13] in a mutual exclusion lock enables threads to abandon their attempt of acquiring the lock while they are waiting to acquire the lock. Abortability poses an interesting difficulty in cohort lock construction. Even if the *alone?* method, which indicates that a cohort thread is waiting to acquire the lock, returns false (which means that there exists a cohort thread waiting to acquire the lock), all the waiting cohort threads may subsequently abort their attempts to acquire the lock. This case, if not handled correctly, can easily lead to a deadlock, where the global lock is in the acquired state, and the local lock has been handed off to a cohort thread that no longer exists, and may not appear in the future either.

Thus, we must strengthen the requirements of the lock cohorting transformation with respect to the *cohort detection* property: if *alone?* returns false, then some thread concurrently executing the local *lock* method will not abort before completing the local *lock* method call. Notice that a thread that completed acquiring the local lock with the *release local* lock state cannot be aborted since by definition it is in the critical section.

3.6.1 The A-C-BO-BO Lock

The A-C-BO-BO lock is very similar to the C-BO-BO lock that we described earlier, with the difference that aborting threads also reset the *successor-exists* field in the local lock to inform the local lock releaser that a waiting thread has aborted. Each spinning thread reads this field while spinning, and sets it in case it was recently reset by an aborting thread. Like the C-BO-BO lock, in A-C-BO-BO, the local lock releaser checks to see if the *successor-exists* flag was set (which indicates that there exist threads in the local cluster that are spinning to acquire the lock). If the *successor-exists* flag was set, the releaser can release the local BO lock by writing *release local* into the BO lock. [4]

However, at this point the releaser must double-check the *successor-exists* field to determine if it was cleared during the time the releaser released the local BO lock. If so, the releaser conservatively assumes that there may be no other waiting cohort, and atomically changes the local BO lock's state to *global release*, and then releases the global BO lock.

[4] Note that the BO lock can also be in 3 states: *release global* (which is the default state, indicating that the lock is free to be acquired, but the acquirer must thereafter acquire the global BO lock to execute the critical section), *busy* (indicating that the lock is acquired by some thread), and *release local* (indicating that the next acquirer of the lock implicitly inherits ownership of the global BO lock).

3.6.2 The A-C-BO-CLH Lock

The A-C-BO-CLH lock has a BO lock as its global lock (which is trivially abortable), and an abortable variant of the CLH lock [14] (A-CLH) as its local lock. Like the MCS lock, the A-CLH lock also consists of a list of records, one per thread, ordered by the arrival of the threads at the lock's tail. To acquire the A-C-BO-CLH lock, a thread first must acquire its local A-CLH lock, and then explicitly or implicitly acquire the global BO lock.

Because we build on the A-CLH lock, we will first briefly review it as presented by Scott [14]. The A-CLH lock leverages the property of "implicit" CLH queue predecessors, where a thread that enqueues its node in the CLH queue spins on its predecessor node to determine if it has become the lock owner. An aborting thread marks its CLH queue node as aborted by simply making its predecessor explicit in the node (i.e. by writing the address of the predecessor node to the prev field of the thread's CLH queue node). The successor thread that is spinning on the aborted thread's node immediately notices the change and starts spinning on the new predecessor found in the aborted node's prev field. The successor also returns the aborted CLH node to the corresponding thread's local pool.

The local lock in our A-C-BO-CLH builds on the A-CLH lock. For local lock handoffs, much like the A-CLH lock, the A-C-BO-CLH leverages the A-CLH queue structure in its cohort detection scheme. A thread can identify the existence of cohorts by checking the A-CLH lock's tail pointer. If the pointer does not point to the thread's node, it means that a subsequent request to acquire the lock was posted by another thread. However, now that threads can abort their lock acquisition attempts, this simple check is not sufficient to identify any "active" cohorts, because the ones that enqueued their nodes may have aborted, or will abort.

In order to address this problem, we introduce a new *successor-aborted* flag in the A-CLH queue node. We colocate the *successor-aborted* flag with the prev field of each node so as to ensure that both are read and modified atomically. Each thread sets this flag to false, and its node's prev field to *busy*, before enqueuing the node in the CLH queue. An aborting thread atomically (with a CAS) sets its node's predecessor's *successor-aborted* flag to true to inform its predecessor that it has aborted (the thread subsequently updates its node's prev field to make the predecessor explicitly visible to the successor).

While releasing the lock, a thread first checks its node's *successor-aborted* flag to determine if the successor may have aborted. If not, the thread can release the local lock by atomically (using a CAS instruction) setting its node's prev field to the *release local* state (just like the release in C-BO-MCS). This use of a CAS coupled with the colocation of prev and *successor-aborted* fields ensures that the successor thread cannot abort at the same time. The successor can then determine that it has become the lock owner. If the successor did abort (indicated by the *successor-aborted* flag), the thread releases the global BO lock, and then sets its node's state to *release global*.

Our use of a CAS instruction to do local lock handoffs seems quite heavy-handed. And we conjecture that indeed it would be counter-productive if the CAS induced cache coherence traffic between NUMA clusters. However, since the CAS targets memory that is likely to already be resident in cache of the local cluster in writable state, the cost of local transactions is quite low – equivalent to a store instruction hitting the L2 cache on the system we used for our empirical evaluation.

3.7 Bounding Local Lock Handoff Rates

All the locks described above are deeply unfair, and with even modest amounts of contention can easily lead to thread starvation. To address this problem, we add a *may-pass-local* method that increments a simple counter of the number of times threads in a cohort have consecutively acquired the lock in a *release local* state. If the counter crosses a threshold (64) in our experiments, the lock releaser releases the global lock, and then releases the local lock, transitioning it to the *release global* state. This simple solution appears to work very effectively for all our algorithms.

4. Empirical Evaluation

We evaluated cohort locks, comparing them with the traditional, as well as the more recent NUMA-aware locks, on multiple levels: First we conducted several experiments on microbenchmarks that stress test these locks in several ways. This gives us a good insight into the performance characteristics of the locks. Second, we integrated these locks in memcached, a popular key-value data store application, to study their impact on real world workload settings. Third, we modified the libc memory allocator to study the effects of cohort locks on allocation intensive multi-threaded applications; we present results of experiments on a microbenchmark [6].

Our microbenchmark evaluation clearly demonstrates that cohort locks outperform all prior locks by at least 60%. Additionally, the abortable cohort locks scale vastly better (by a factor of 6) than the state-of-the-art abortable locks. Furthermore, cohort locks improved the performance of memcached by about 20% for write-heavy workloads. Finally, our libc allocator experiments demonstrate that simply replacing the lock used by the default Solaris allocator with a cohort lock can significantly boost cluster-level reference locality for accesses by the allocator to allocation metatdata and for accesses by the application to allocated blocks, resulting in improved performance for multi-threaded application that make heavy use of memory allocation services.

In our evaluation we compare the performance of our non-abortable and abortable cohort locks with existing state-of-the-art locks in the respective categories. Specifically, for our microbenchmark study, we present throughput results for our C-BO-BO, C-TKT-TKT, C-BO-MCS, C-TKT-MCS and C-MCS-MCS cohort locks. We compare these with MCS [10] (as a base line NUMA-oblivious lock), and other NUMA-aware locks, namely, HBO [12], HCLH [15], and FC-MCS [7]. We also evaluated our abortable cohort locks (namely, A-C-BO-BO and A-C-BO-CLH) by comparing them with an abortable version of HBO, and the abortable CLH lock [14].

Memcached uses pthread locks for synchronization. To test our locks with memcached, we decided to adhere to the policy of not changing the memcached sources or its binary. This choice is facilitated by the fact that the pthread library is dynamically linked to the application. So we can easily use a Solaris LD_PRELOAD *interpose* library that installs any kind of lock we want under the pthread library API. The scalability results for memcached are reported in Section 4.2. For the libc allocator experiments, we used the same interpose library to inject our locks into the allocator.

We implemented all of the above algorithms in C and compiled them with the GCC 4.4.1 at optimization level -O3 in 32-bit mode. The experiments were conducted on an Oracle T5440 series machine which consists of 4 Niagara T2+ SPARC chips, each chip containing 8 cores, and each core containing 2 pipelines with 4 hardware thread contexts per pipeline, for a total of 256 hardware thread contexts, running at a 1.4 GHz clock frequency. Each chip has a 4MB L2 cache, and each core has a shared 8KB L1 data cache. For all the NUMA-aware locks, a Niagara T2+ chip is the NUMA clustering unit, so in all we had 4 NUMA clusters.

Memcached was evaluated using a standard client application called *memaslap*, which is a part of a larger suite of memcached applications called *libmemcached*. Results reported were averaged over 3 test runs.

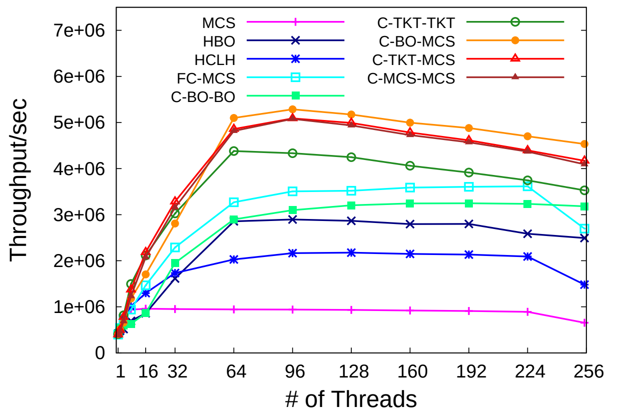

Figure 2. The graph shows the average throughput in terms of number of critical and non-critical section pairs executed per second. The critical section accesses two distinct cache blocks (increments 4 integer counters on each block), and the non-critical section is an idle spin loop of up to 4 microseconds.

4.1 Microbenchmark Evaluation

4.1.1 Scalability

We constructed what we consider to be a reasonable representative microbenchmark, LBench, to measure the scalability of various lock algorithms. LBench launches a specified number of identical threads. Each thread loops as follows: acquire a central shared lock, access shared variables in the critical section, release the lock, and then execute a non-critical work phase of about 4 microseconds. The critical section reads and writes shared variables residing on two distinct cache lines. At the end of a 60 second measurement period the program reports the aggregate number of iterations completed by all threads as well as statistics related to the distribution of iterations completed by individual threads, which reflects gross long-term fairness. Finally, the benchmark can be configured to tally and report lock migrations.

Figure 2 depicts the performance of the non-abortable locks on LBench. (We conducted other experiments varying the critical section length, non-critical section length, and number of cache lines accessed within the critical section, but observed similar results to those reported). As a baseline to compare against, we measured the throughput of the MCS lock, which is a classic scalable queue lock. This lock performed the worst because it does not leverage reference locality, which is critical for good performance on NUMA architectures. The HCLH, HBO and FC-MCS locks perform as expected – with FC-MCS generally performing the best among the three. HBO's performance, which is better than HCLH in this workload, is highly sensitive to the underlying workload, and is generally very unstable (as we will see in Section 4.2).

Our C-BO-BO lock scales very well, approaching the performance of FC-MCS. Because it is based on the BO lock, C-BO-BO is sensitive to backoff parameters – different workloads might require different backoff parameters for the best possible performance. However, this sensitivity is related only to the parameters associated with local backoff locks, unlike HBO, where the backoff parameters need to be tuned for both the local and remote backoffs. Under C-BO-BO we expect that the global lock will remain lightly contended; and in fact, in our implementation, threads contending at the global BO lock continuously spin on it and never backoff, much like the "bare bones" test-and-test-and-set lock. Our C-TKT-TKT lock scales even better (generally 30-40% better than the prior state-of-the-art NUMA-aware lock, FC-MCS). C-BO-MCS scales

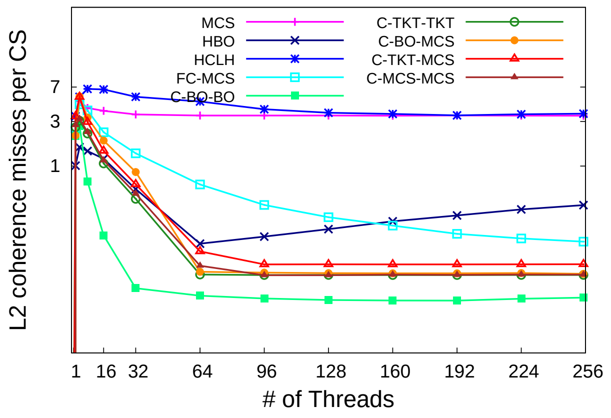

Figure 3. The graph shows the average number of L2 cache coherence misses per critical section for the experiment in Figure 2 (lower is better). The Y-axis is in log scale.

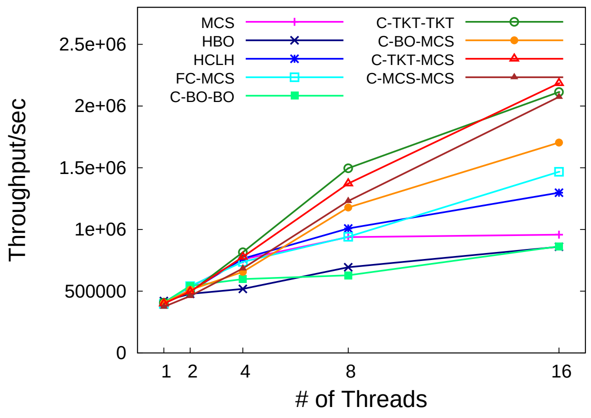

Figure 4. A closer look at the throughput results of Figure 2 for low contention levels (1 to 16 threads).

the best with scalability 60% better than FC-MCS, whereas C-TKT-MCS and C-MCS-MCS trail slightly behind C-BO-MCS.

In all the tests reported in this paper, the allowable maximum number of consecutive local lock handoffs for cohort locks was limited to a constant (64). As described in Section 3.7, this bound is necessary to avoid the deep unfairness that the basic cohort locks can possibly generate in an application. We conducted microbenchmark tests (not reported in this paper) on cohort lock versions without the local handoff limits and found that generally the deeply unfair versions out-scale the fair versions by about 10% during high contention loads. However, in our tests we found that, for LBench, the unfair versions typically led to local lock handoffs in the order of hundreds of thousands before the lock was acquired by a remote thread/cohort. Thus, we believe that the cost of 10% is small to avoid the potential problem of gross long-term unfairness and starvation.

4.1.2 Locality of Reference

Figure 3 provides the key explanation for the observed scalability results. Recall that each chip (a NUMA cluster) on our experimental machine has an L2 cache shared by all cores on that chip. Furthermore, the latency to access a cache block in the local L2 cache is much lower than the latency to access a cache block on a remote L2 cache (on our test machine, remote L2 access is approximately 4 times slower than local L2 access during light loads). The latter also involves bus transactions that can adversely affect the latency during high loads, further compounding the cost of remote L2 accesses. That is, remote L2 accesses always incur latency costs even if the interconnect is otherwise idle, but they can also induce interconnect channel contention if the system is under heavy load. Figure 3 reports the L2 coherence miss rates collected during the scalability experiment. These are the local L2 misses that were fulfilled by a remote L2, which represents the local to remote lock handoff events and related data movement.

MCS has a high L2 coherence miss rate because it is the fairest among all the locks, and does not prioritize local lock acquisition requests over remote requests. Interestingly, HCLH also has a high miss rate, which clearly explains its performance in Figure 2. We can attribute the high miss rate in HCLH to its complexity [7] and high rates of accesses to shared lock metadata, which translates to lower rate of batching of requests coming from the same NUMA cluster. HBO shows a very good miss rate until the number of threads is substantially high (64), after which, the miss rate deteriorates considerably. In our experiments with HBO, we observed that its backoff parameters are highly sensitive to the underlying workload; the HBO lock results discussed here are for a version whose backoff parameters were tuned for this microbenchmark (we will show later that these same backoff parameters hurt the performance of memcached). The L2 miss rate in FC-MCS degrades gradually, but consistently, with increasing thread count.

All our cohort locks have significantly lower (by a factor of two or greater – note that the Y-axis is in log scale) L2 miss rates than all other reported locks. This is because the cohort locks consistently provide long sequences of successive accesses to the lock from the same NUMA cluster (cohort), which accelerates the critical section performance by reducing inter-core coherence transfers for data accessed within and by the critical section. There are two reasons for longer cohort sequences (or, more generally, *batches*) in cohort locks, compared to prior NUMA-aware locks, such as FC-MCS. First, the simplicity of cohort locks streamlines the instruction sequence of batching requests in the cohort, which makes the batching more efficient. Second, and more importantly, a cohort batch can dynamically "grow" during its execution without the interference by threads from other cohorts. As an example, consider a cohort of 3 threads $T1$, $T2$, and $T3$ (ordered in that order in a local MCS queue M in a C-BO-MCS lock). Let $T1$ be the owner of the global BO lock. So $T1$ can enter its critical section, and handoff the BO lock (and the local MCS lock) to $T2$ on exit. The BO lock will eventually be forwarded to $T3$. However, note that in the meantime, $T1$ can return and post another request after $T3$'s request in M. If $T3$ holds the lock during this time, it ends up handing it off to $T1$ after it is done executing its critical section. This dynamic growth aspect of our cohort locks can significantly boost local handoff rates when there is sufficient cluster-local contention. This dynamic batch growth aspect in cohort locks contrasts with the more "static" approach of other NUMA-aware locks, which in turn gives more power to cohort locks to enhance the locality of reference of the critical section. In our experiments, we have observed that the batching rate in all the locks is inversely proportional to the lock migration rate and observed coherence traffic reported in Figure 3, and that the batching rate in cohort locks increases more quickly with contention compared to other locks.

4.1.3 Low Contention Performance

We believe that for a highly scalable lock to be practical, it must also perform efficiently at low or zero contention levels. At face value, the hierarchical nature of cohort locks appears to suggest that they will be expensive at low contention levels. That is because each thread must acquire both the local and the global locks, which become a part of the critical path in low contention or contention free scenarios. To understand this cost we took a closer look at

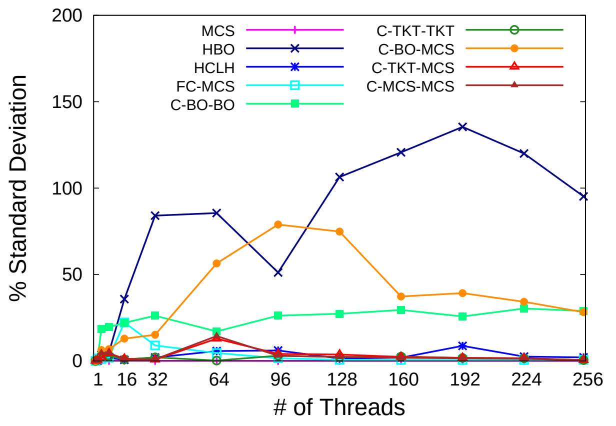

Figure 5. The graph shows the standard deviation in percentage points of per-thread throughput from the average throughput reported in Figure 2 (the lower the standard deviation, the more fair the lock is in practice).

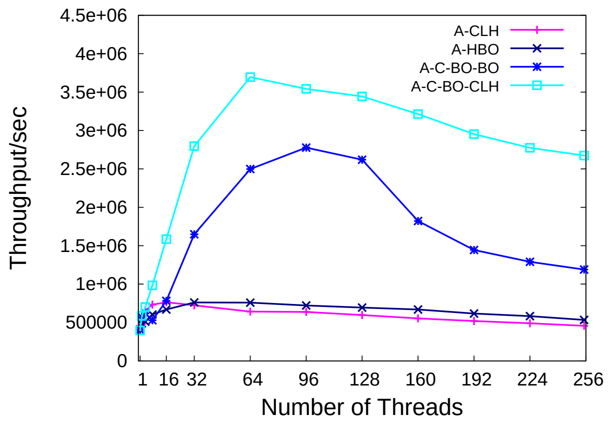

Figure 6. Abortable lock average throughput in terms of number of critical and non-critical sections executed per second. The critical section accesses two distinct cache blocks (increments 4 integers counters on each block), and the non-critical section is an idle spin loop of up to 4 micro seconds.

the scalability results reported in Figure 2 with an eye toward performance at low contention levels. Figure 4 zooms into that part of Figure 2. Interestingly, we observed that the performance of all the cohort locks was competitive with all other locks that do not need to acquire locks at multiple levels in the hierarchy (viz. MCS, HBO, and FC-MCS). On further reflection, we note that the extra cost of multi-level lock acquisitions in cohort locks withers away as background noise in the presence of non-trivial work in the critical and non-critical sections. In principle, one can devise contrived scenarios (for example, where the critical and non-critical sections are empty) to show that cohort locks might perform worse than other locks at low contention levels. However, we believe that such scenarios are most likely unrealistic or far too rare to be of any consequence. Even if one comes up with such a scenario, we can add the same “bypass the local lock” optimization that was employed in FC-MCS to minimize the flat combining overhead at low thread counts.

4.1.4 Fairness

Given that cohort locks are inherently unfair (which is the key “feature” that all NUMA-aware locks harness and leverage to enhance locality of reference for better performance), we were interested in quantifying that unfairness. To that end, we report more data from the experiment reported in Figure 2 on the standard deviation of per-thread throughput from the average throughput of all the threads. The results are shown in Figure 5. These results give us a sense of how far each thread progressed during its one minute of execution in a test run.

We found HBO to be the least fair lock, where some threads executed only a handful of critical sections, while others completed millions of critical sections. The next most unfair lock, to our surprise, was C-BO-MCS. (Recall that the cohort locks contain a constant limit of 64 local handoffs after which a lock releaser must release the global and local locks.) On further reflection, the reason for this unfairness is clear – the global BO lock in C-BO-MCS is unfair. After a thread releases the global BO lock, causing the cache block of the BO lock to be invalidated from other caches, and go to modified state in the releaser’s cache, it immediately releases the local lock, which is also quickly detected by the next thread waiting to acquire the local lock. This next local thread, identifying that it must acquire the global BO lock, almost instantly attempts to do so, and usually succeeds because the BO lock’s cache block is in its (and the last releaser’s) L1 or L2 cache. Hence the obvious unfairness arises from unfairness in cache coherence arbitration.

We also observe that the standard deviation for C-TKT-MCS and C-MCS-MCS are low, because their global locks (ticket and MCS) are comparatively fair.

C-BO-BO is fairer than C-BO-MCS because the interval between a lock releaser releasing the global lock and the next local lock acquirer attempting to acquire the global BO lock is inflated because the acquirer must first acquire the local BO lock. The extended interval increases the window in which a remote thread/cohort can acquire the global BO lock.

All other locks, MCS, HCLH, FC-MCS, and C-TKT-TKT are also fair, as expected, with the standard deviation well under 5% (the deviation of FC-MCS spikes to about 20% at 16 threads, but is reasonably low at other concurrency levels).

4.1.5 Abortable Lock Performance

Our abortable lock experiments in Figure 6 make an equally compelling case for cohort locks. Our cohort locks (A-C-BO-BO and A-C-BO-CLH) outperform the best prior abortable lock (A-CLH) and an abortable variant of HBO (called A-HBO in Figure 6, where a thread aborts its lock acquisition by simply returning a failure flag from the lock acquire operation) by up to a factor of 6. Since lock handoff is a “local” operation in A-C-BO-CLH involving just the lock releaser (that uses a CAS to release the lock) and the lock acquirer (just like a CLH lock), A-C-BO-CLH scales significantly better than A-C-BO-BO, where threads incur significant contention with other threads on the same NUMA cluster to acquire the local BO lock. (For these and other unreported experiments, the abort rate was lower than 1%, which we believe is a reasonable rate.)

4.2 Memcached

Memcached [2] is a popular open-source, high-performance, distributed in-memory key-value data store that is typically used as a caching layer for databases in high-performance applications. Memcached has several high profile users including Facebook, LiveJournal, Wikipedia, Flickr, Youtube, Twitter, etc.

In memcached, the key-value pairs are stored in a huge hash table, and all server threads access this table concurrently. Access to the entire table is mediated through a single lock (called the cache_lock). The cache_lock is known to be a contention bottleneck [11], which we believe makes it a good candidate for the evaluation of cohort locks. Among other things, the memcached API contains two fundamental operations on key-value pairs: get (that returns the value for a give key) and set (that updates the value of

#	pthread locks	Fib-BO	MCS	HBO	HBO (tuned)	FC-MCS	C-BO-BO	C-TKT-TKT	C-BO-MCS	C-TKT-MCS	C-MCS-MCS
1	1.00	0.89	0.99	0.83	1.01	0.83	0.99	0.82	0.81	0.77	0.95
4	3.06	3.17	3.15	1.58	3.37	2.70	3.11	3.09	3.09	3.05	2.99
8	4.37	4.48	4.47	1.96	4.43	4.25	3.48	4.46	4.45	4.49	4.45
16	4.55	4.59	4.60	2.55	4.58	4.47	2.56	4.56	4.60	4.58	4.53
32	4.47	4.57	4.53	3.05	4.53	4.18	3.03	4.54	4.57	4.56	4.55
64	4.40	4.54	4.45	3.37	4.51	4.26	2.98	4.52	4.49	4.51	4.44
96	4.39	4.50	4.46	3.37	4.52	4.32	2.97	4.48	4.50	4.52	4.46
128	4.39	4.49	4.47	3.39	4.52	4.28	2.98	4.46	4.49	4.53	4.46

(a) 90% gets and 10% sets

#	pthread locks	Fib-BO	MCS	HBO	HBO (tuned)	FC-MCS	C-BO-BO	C-TKT-TKT	C-BO-MCS	C-TKT-MCS	C-MCS-MCS
1	1.00	1.04	1.15	0.97	0.93	0.95	1.14	1.00	0.92	1.12	1.11
4	2.84	3.21	3.30	1.45	3.23	2.73	2.98	3.10	3.19	2.98	3.16
8	3.55	4.63	4.51	1.73	4.75	4.00	2.46	4.53	4.51	4.47	4.32
16	3.56	4.95	4.93	2.17	5.18	4.51	2.59	5.08	5.05	5.03	4.97
32	3.42	4.93	4.77	2.57	5.10	3.92	2.79	4.99	5.04	4.95	4.94
64	3.29	4.81	4.45	2.86	5.08	3.93	2.67	4.88	4.88	4.79	4.74
96	3.32	4.80	4.47	2.84	5.07	3.94	2.69	4.84	4.86	4.78	4.71
128	3.32	4.81	4.24	2.84	5.09	3.90	2.68	4.87	4.85	4.71	4.68

(b) 50% gets and 50% sets

#	pthread locks	Fib-BO	MCS	HBO	HBO (tuned)	FC-MCS	C-BO-BO	C-TKT-TKT	C-BO-MCS	C-TKT-MCS	C-MCS-MCS
1	1.00	1.05	1.03	1.02	1.22	1.00	1.03	0.97	1.05	1.06	1.13
4	2.62	3.03	2.74	1.43	2.95	2.44	2.82	2.65	2.66	2.57	2.60
8	2.74	3.80	3.62	1.76	4.23	3.21	2.27	3.80	3.80	3.74	3.61
16	2.77	4.08	3.92	1.99	4.76	3.62	2.50	4.54	4.52	4.48	4.30
32	2.67	4.08	3.94	2.27	4.86	3.31	2.53	4.81	4.70	4.70	4.50
64	2.59	3.89	3.62	2.49	4.63	3.34	2.44	4.47	4.41	4.40	4.23
96	2.62	3.92	3.65	2.49	4.64	3.35	2.45	4.44	4.40	4.38	4.23
128	2.59	3.94	3.51	2.49	4.67	3.33	2.44	4.46	4.47	4.30	4.20

(c) 10% gets and 90% sets

Table 1. Scalability results (in terms of speedup over single thread runs that use pthread locks) for memcached for (a) read-heavy (90% get operations, and 10% set operations), (b) mixed (50% get operations, and 50% set operations), and (c) write-heavy (10% get operations, and 90% set operations) configurations.

the given key). These are the most frequently used API calls by memcached client applications.

To generate load for the memcached server, we use memaslap, a load generation and benchmarking tool for memcached. memaslap is a part of the standard client library (called libmemcached) for memcached [1]. memaslap generates a memcached workload consisting of a configurable mixture of get and set requests. We experimented with a wide range of get-set mixture ratio, ranging from configurations with 90% gets and 10% sets (representing read-heavy workloads) to configurations with 10% gets and 90% sets (representing write-heavy workloads). The read-heavy workloads are the norm for memcached applications. The write-heavy workloads, however uncommon, do exist. Examples of write-heavy workloads include servers that continuously collect huge amounts of sensor network data, or servers that constantly update statistical information on a large collection of items. These applications at times can exhibit bi-modal behavior, alternating between write-heavy and read-heavy phases, collecting and processing large amounts of data respectively.

As discussed earlier, we used an interpose library to inject our locks under the pthreads API used by memcached. For our experiments, we ran an instance of memcached on the T5440 server, and an instance of memaslap on another 128-way Niagara II machine. We varied the thread count for memcached from 1 to 128 (the maximum number of threads permitted by memcached). We ran the memaslap client with 32 threads for all tests so as to keep the load generation high and constant. For each test, the memaslap clients were executed for one minute, after which the throughput, in terms of operations per second, was reported. Table 1 shows the relative performance of memcached while it was configured to use the different locks. The figure contains 3 tables for three different get-set proportions, each representing read-heavy, mixed, and write-heavy loads respectively. Each entry in each table is normalized to the performance of pthread locks at 1 thread.

The first column in all the tables represents the number of memcached threads used in the test. The second column reports the performance of pthread locks. The remaining columns report the performance of memcached when used with MCS, a test-and-test-and-set lock with Fibonnaci backoff (Fib-BO), the HBO lock (representing prior NUMA-aware locks), a tuned version of HBO (we had to add this version because the default version did not scale well), and all the non-abortable cohort locks discussed in Section 3. Note that the fine tuning for the HBO lock was done on the local and remote backoff parameters. The version that we first used in the experiments (column titled HBO) had the backoff parameters tuned for our microbenchmark experiments. As is clear from all three tables, these did not work well on memcached. This clearly demonstrates the instability of HBO's performance.

For read-heavy loads (Table 1 (a)), the performance of all the locks except HBO and C-BO-BO is identical, with all locks enabling over 5X scaling. For loads with moderately high set ratios (Table 1 (b)), we observe that all the spin locks except HBO and C-BO-BO significantly outperform pthread locks, and are generally competitive with each other. For write-heavy loads (Table 1 (c)), the NUMA-aware locks clearly out-scale the NUMA-oblivious locks by at least 20%. The untuned HBO and C-BO-BO locks scale poorly in all configurations. It appears that C-BO-BO suffers because of contention on the local BO locks, whereas HBO suffers with contention on the central lock. FC-MCS performs better than HBO and C-BO-BO, but worse than all other spinlocks.

4.3 malloc

Memory allocation and deallocation is a common operation appearing frequently in all kinds of applications. A vast number of C/C++ programs use libc's malloc and free functions for managing their dynamic memory requirements. These functions are thread-safe, but on Solaris the default allocator relies on synchronization via a single lock to guarantee thread safety. For memory intensive multi-threaded applications that use libc's malloc and free functions, this lock can quickly become a contention bottleneck. Consequently, we found it to be an attractive evaluation tool for cohort locks. We modified libc's malloc.c file to use pthread locks, and injected our locks in the code via the interpose library discussed previously.

We used the mmicro benchmark [6] to test various lock algorithms via the interpose library. In the benchmark, each thread repeatedly allocates a block of memory (size 64 bytes), initializes it

thrds	pthread locks	fib-BO	MCS	HBO	HBO (tuned)	FC-MCS	C-BO-BO	C-TKT-TKT	C-BO-MCS	C-TKT-MCS	C-MCS-MCS
1	198	211	197	206	206	190	197	195	191	191	183
2	197	237	224	204	231	231	223	220	214	218	208
4	125	258	271	206	300	288	253	326	252	307	249
8	145	294	307	230	382	322	320	486	326	456	432
16	151	318	307	244	420	327	483	592	513	576	564
32	149	323	307	248	291	329	783	839	941	827	814
64	149	302	303	259	151	328	883	1011	1183	1001	952
128	146	225	290	263	73	321	932	884	1120	863	822
255	142	139	277	257	38	264	926	695	961	682	651

Table 2. Scalability results of the malloc experiment (in terms of malloc-free pairs executed per millisecond.

(by writing to the first 4 words of it), and subsequently frees it. Each test runs for 10 seconds and reports the aggregate number of malloc-free pairs completed in that interval. We add an artificial delay after each of the calls to malloc and free functions. This delay is a configurable parameter; we injected a delay of about 4 microseconds, which enables some concurrency between the thread executing the critical sections (malloc or free), and the threads waiting in the delay loop. The results of the tests appear in Table 2, showing that cohort locks outperform all the other locks. While the other locks scale the benchmark's throughput by up to a factor of 2X, the scalability with cohort locks ranges between a factor of 5X and 6X.

There are two reasons for this impressive scalability of cohort locks: First, they tend to effectively batch requests coming from the same NUMA cluster, thus improving the lock handoff latency. The second reason has to do with the recycling of memory blocks deallocated by threads: The libc allocator maintains a single *splay tree* of free nodes of various sizes (it also maintains lists of small – 40 bytes or less – memory blocks used for small size requests). Since mmicro requests 64 byte blocks, all the requests go to the splay tree. A newly inserted node always goes to the root of the tree, and as a result, the most recently deallocated memory blocks tend to be reallocated more often (allocation is done by returning the first matching block in the splay tree). Thus a small number of tree nodes (and their respective memory blocks) are continuously circulated between threads. The tree node cache lines are updated on every delete (malloc) and insert (free). Additionally, the allocated memory blocks are also updated by the benchmark. All these writes play a crucial role in the performance of the underlying locks used by the allocator. Because all the cohort locks create large batches of consecutive requests coming from the same NUMA cluster, they manage to recycle blocks in the same cluster for extended periods. In contrast, for all other locks, a block of memory migrates more frequently between NUMA clusters, thus leading to greater coherence traffic, and the resulting performance degradation.

While highly scalable allocators exist and have been described at length in the literature, selection of such allocators often entails making tradeoffs such as footprint against scalability. In part because of such concerns the default on Solaris remains the simple single-lock allocator. By employing cohort locks under the default libc allocator we can improve the scalability of applications but without forcing the user or developer to confront the issues and decisions related to alternative allocators.

FC-MCS does not show any significant improvements over prior locks. The performance of HBO continues to be unstable: The first HBO column in Table 2 shows the libc allocator's performance with the backoff parameters picked from our earlier microbenchmark experiments, while the second HBO column, titled: HBO (tuned), uses the parameters tuned for good performance on memcached. In this case, the tuned version of HBO scales better than the untuned version up to modest levels of contention. However, the performance dramatically deteriorates with higher contention. In contrast, cohort locks are vastly more stable across a broad swath of workloads. This property of "parameter parsimony" makes cohort locks a significantly more attractive choice for deployment in real world applications.

5. Conclusion

The growing size of multicore machines is likely to shift the design space in the NUMA and CC-NUMA direction, requiring a significant rehash of existing concurrent algorithms and synchronization mechanisms. This paper tackles the most basic of the multicore synchronization algorithms, the lock, presenting a simple new lock design approach – *lock cohorting* – fit for NUMA machines. The wide range of cohort locks we presented in the paper, along with their empirical evaluation, demonstrates that lock cohorting is not only a simple approach to NUMA-aware lock construction, but also a powerful one that delivers locks that out-scale prior locks by significant margins, while remaining competitive at low contention levels.

References

[1] libmemcached. *www.libmemcached.org*.

[2] memcached – a distributed memory object caching system. *www.memcached.org*.

[3] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. *SIGARCH Comput. Archit. News*, 17:396–406, April 1989.

[4] T. Craig. Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, University of Washington, Dept of Computer Science, February 1993.

[5] D. Dice. *US Patent # 07318128: Wakeup affinity and locality*.

[6] D. Dice and A. Garthwaite. Mostly Lock Free Malloc. In *Proceedings of the 3rd International Symposium on Memory Management*, pages 163–174, 2002.

[7] D. Dice, V. Marathe, and N. Shavit. Flat Combining NUMA Locks. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, 2011.

[8] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat Combining and the Synchronization-Parallelism Tradeoff. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 355–364, 2010.

[9] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2007.

[10] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Computer Systems*, 9(1):21–65, 1991.

[11] M. Pohlack and S. Diestelhorst. From Lightweight Hardware Transactional Memory to LightWeight Lock Elision. In *Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing*, 2011.

[12] Z. Radović and E. Hagersten. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *HPCA-9*, pages 241–252, Anaheim, California, USA, Feb. 2003.

[13] M. Scott and W. Scherer. Scalable queue-based spin locks with timeout. In *Proc. 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 44–52, 2001.

[14] M. L. Scott. Non-blocking timeout in scalable queue-based spin locks. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 31–40, New York, NY, USA, 2002. ACM.

[15] Victor Luchangco and Dan Nussbaum and Nir Shavit. A Hierarchical CLH Queue Lock. In *Proceedings of the 12th International Euro-Par Conference*, pages 801–810, 2006.