Lock Cohorting: A General Technique for Designing NUMA Locks

DAVID DICE and VIRENDRA J. MARATHE, Oracle Labs
NIR SHAVIT, MIT

Multicore machines are quickly shifting to NUMA and CC-NUMA architectures, making scalable NUMA-aware locking algorithms, ones that take into account the machine's nonuniform memory and caching hierarchy, ever more important. This article 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 nonintrusive changes, into a scalable NUMA-aware spin-lock. 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

Additional Key Words and Phrases: Concurrency, NUMA, multicore, mutual exclusion, mutex, locks, hierarchical locks, spin locks

ACM Reference Format:
Dice, D., Marathe, V. J., and Shavit, N. 2014. Lock cohorting: A general technique for designing NUMA locks. *ACM Trans. Parallel Comput.* 1, 2, Article 13 (January 2015), 42 pages.
DOI: http://dx.doi.org/10.1145/2686884

1. INTRODUCTION

As microprocessor vendors aggressively pursue the production of bigger multicore multichip systems (Intel's Nehalem-based and Oracle's Niagara-based systems are typical examples), the computing industry is witnessing a shift toward distributed and cache-coherent Non-Uniform Memory Access (NUMA) architectures.[1] These systems contain multiple nodes where each node has locally attached memory, a fast local cache and multiple processing cores. A local cache is able to cache lines from both the local and

[1]We use the term NUMA broadly to include Non-Uniform Communication Architecture (NUCA) [Radović and Hagersten 2003] machines as well.

This work is an extension of an earlier conference paper, which appeared in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*.
Nir Shavit was supported in part by NSF grants CCF-1217921 and CCF-1301926, DoE ASCR grant ER26116/DE-SC0008923, and by grants from the Oracle and Intel corporations.
Author's address: D. Dice (corresponding author); email: dave.dice@oracle.com.

DOI: http://dx.doi.org/10.1145/2686884

remote nodes. The nodes are connected via a slower internode 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 node [Radović and Hagersten 2003]. Nodes are commonly implemented as individual processor chips. Such systems present a uniform programming model where all memory is globally visible and cache-coherent. The set of cache-coherent communications channels between nodes is referred to collectively as the interconnect. These internode links normally suffer from higher latency and lower bandwidth compared to the intranode channels. To decrease latency and to conserve interconnect bandwidth, NUMA-aware policies encourage intranode communication over internode communication.

Creating efficient software for NUMA systems is challenging because such systems present a naive uniform "flat" model of the relationship between processors and memory, hiding the actual underlying topology from the programmer. The programmer must study architecture manuals and use special system-dependent library functions to exploit the system topology. NUMA-oblivious multithreaded programs may suffer performance problems arising from long access latencies caused by internode coherence traffic and from interconnect bandwidth limits. Furthermore, internode interconnect bandwidth is a shared resource so coherence traffic generated by one thread can impede the performance of other unrelated threads because of queueing delays and channel contention. Concurrent data structures and synchronization constructs at the core of modern multithreaded applications must be carefully designed to adapt to the underlying NUMA architectures. This article presents a new class of NUMA-aware mutex locks – the *cohort locks* – that vastly improves performance over prior NUMA-aware and classic NUMA-oblivious locks.

1.1. Related Work

Dice [2003], and Radović and Hagersten [2003] were the first to identify the benefits of designing locks that improve locality of reference on 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 reduce the rate of *lock migration*. Lock migration occurs when a lock L is consecutively acquired by two threads running on different NUMA nodes. Lock migration leads to the transfer of cache lines – both for lines underlying the lock metadata as well as for lines underlying mutable data accessed in the critical section protected by the lock – from the cache associated with the first thread to the cache associated with the second thread. NUMA-aware locks promote node-level locality of reference for the lock and critical sections they protect and act to reduce the rate of write invalidations, coherence misses and cache-to-cache transfers that can arise when one node reads or writes to a cache line that is in *modified* [Sweazey and Smith 1986] state in another node's cache.

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 automatically adjusts the backoff delay period, so that when a thread T notices that another thread from its own local node owns the lock, T can reduce its delay and increase its chances of acquiring the lock relative to threads on remote nodes. This strategy reduces lock migration rates in a probabilistic fashion. 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 attempted modification of the shared global lock variable, which is especially costly on NUMA machines. Long backoff periods can mitigate the coherence costs arising from futile polling, but the delays also retard lock hand-over latency and responsiveness if the lock is released when the waiting threads happen to be using long backoff periods. Moreover, the dynamic adjustment of the backoff delay time introduces significant starvation and fairness issues: it becomes likely that two or more threads from the same node will repeatedly acquire a lock while threads from other nodes starve. (We note that some degree of short-term unfairness and non-FIFO admission order is actually desirable and required to reduce lock migration rates). Radović and Hagersten introduced an additional heuristic to combat this phenomena and improve long-term fairness: threads that waited too long on a remote node and thus could be potential victims of long-term starvation respond by becoming "impatient" and revert to a shortened backoff period. This approach introduces additional tuning parameters which invariably makes the lock's performance less reliable. The performance of HBO is sensitive to its tunable parameters: the optimal parameter values are platform-specific, and vary with system load, available concurrency, and critical section length. Extracting stable, reliable and portable performance from HBO is a challenge.

Luchangco et al. [2006] overcame these drawbacks by introducing HCLH, a hierarchical version of the CLH queue-lock [Craig 1993; Magnussen et al. 1994]. The HCLH algorithm collects requests on each node into a local CLH-style queue, and then has the thread at the head of the queue integrate each node's queue into a single global queue. This avoids the overhead of spinning on a shared location and eliminates fairness and starvation issues. HCLH intentionally inserts non work-conserving *combining delays* to increase the size of groups of threads to be merged into the global queue. While testing various lock algorithms the authors of this article discovered that HCLH required threads to be bound to one processor for the thread's lifetime. Failure to bind could result in exclusion and progress failures. Furthermore, as shown in Dice et al. [2011], FC-MCS (discussed below) dominates HCLH with respect to performance. We will not consider HCLH further.

More recently, Dice et al. [2011] showed that one could overcome the synchronization overhead of HCLH locks by collecting local queues using the *flat-combining* technique of Hendler et al. [2010], 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. Like HBO, FC-MCS required load-, platform- and workload-specific tuning to achieve performance beyond that of NUMA-oblivious locks.

Oyama et al. [1999] describe a mechanism that integrates a lock with a stack of pending critical section activations. Critical section invocations must be represented as closures [Wikipedia 2014a] that are pushed on the stack when the lock is found to be held. The lock status and a pointer to the top of the stack of closures are colocated in single word accessed via atomic compare-and-swap operators. Arriving threads either acquire the lock immediately, or if the lock is held they instead delegate responsibility for executing the critical section and push a closure representing their critical section onto the stack. When the owner eventually completes its critical section it notes the stack of pending operations, detaches the stack while retaining ownership of the lock, and then transiently takes on the role of a server, processing the pending operations on behalf of the other client threads that originally posted those requests. It then tries again to release the lock. If new requests arrived in the interim it retains the lock but again processes those operations. Threads that posted closures typically busy-wait on

a completion flag in the closure.[2] This form of delegated execution is cache-friendly and the activations tend to enjoy good temporal locality. Under contention, delegation tends to migrate the execution of the critical section toward caches that likely contain data to be accessed by code in the critical section. In a sense, the code—the critical section—moves toward the caches likely to contain the data, while traditional locking moves the data toward the execution of the code. While not explicitly designed for NUMA environments, delegation can be NUMA-friendly. As described by Oyama, if the arrival rate is sufficient, one thread could be forced to process pending requests indefinitely during its service session. It is simple to adapt the algorithm to bound how much back-to-back delegated work will be performed by a thread acting as a server. When the bound is reached the thread can pass responsibility for the pending activations to some other thread that is waiting for its critical section activation to complete. After a thread detaches a list of pending closures, it is also trivial to reverse the order of the list to provide first-come-first-served processing order. Reversing the list is optional. Even absent reversal, Oyama bounds bypass and provides strong progress properties – requests pushed by thread $T1$ can bypass (execute before) an older request pushed by $T2$ at most once before $T2's$ request is completed.

Subsequently, Fatourou and Kallimanis [2012] and Kallimanis [2013] developed delegation schemes that are explicitly NUMA-aware, specifically the H-Synch construct. Hendler et al. [2010] propose generalized flat-combining where the role of server is periodically handed over among client threads. Lozi et al. [2012] introduced "Remote Core Locking" which is another variation on delegated execution of critical sections where the server threads are dedicated. Calciu et al. [2013a] compared the performance of classic locks to delegation on x86 and SPARC-based NUMA platforms. Klaftenegger et al. [2014] introduced "Queue Delegation Locking", another variation on delegation.

While approaches that use delegation are appealing, it is often difficult or impossible to recast critical sections written to existing lock interfaces into closure or lambda-based form. Specific usage patterns that may preclude conversion of traditional critical sections to closure-based form include but are not limited to the following: stack-based security managers; nested locking; thrown exceptions; access to thread-local data within the critical section such as *errno*; use of thread identify values such as those returned by *pthread_self*; access to on-stack variables visible in the scope of the critical section; and asymmetric nonlexically balanced lock usage, such as hand-over-hand locking. Because of their lack of generality, we will not consider delegation-based forms any further. The approach we describe in this article does not require closure-based critical sections and can easily be used, for instance, to replace existing *pthread* mutex implementations, accelerating existing legacy binary applications. Our approach also permits the use *pthread_cond_t* condition variables and in

[2]Threads typically busy-wait for a response after posting a request, allowing the data structure representing the request and return value to be allocated on-stack and avoiding explicit allocation and deallocation. It is also possible for a thread to do useful work after posting a request, allowing additional parallelism via *futures* [Wikipedia 2014b]. This opportunity exists for most delegation-based mechanisms and even classic locks—see *asynchronous locking* in Garg et al. [2011]—and is not specific to Oyama. We expect most use cases to be fully synchronous, however, and unable to leverage this particular benefit. In some cases the posting thread does not need to wait for a response and can simply continue after posting a request. This asynchronous mode of operation may impose an additional memory management burden as the posted request data structures can not be reused until the operation completes. If explicitly allocated by the requester, these structures will tend to flow and circulate from posting threads to server threads. (Local free lists of requests may provide relief, but can become grossly imbalanced). These variations also require more elaborate programming interfaces that deviate even further from standard locking interfaces and will require more intrusive and error-prone changes to application code.

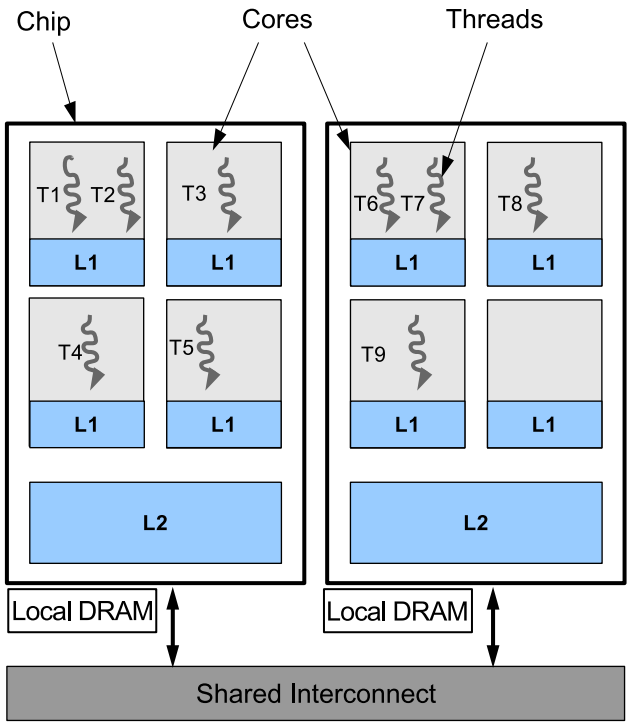

Fig. 1. An example multinode multicore NUMA system containing 2 nodes with 4 cores per node. Each node is an individual chip (socket). Each core can have multiple hardware thread contexts. Each core on a node has a private L1 cache and all cores on a chip share an L2 cache. Interthread communication through the local L1 and L2 caches is significantly faster than between remote caches as the latter entails coherence messages over the interconnect.

general imposes no new restrictions or constraints on the code allowed in the critical section body.

We note that the CC-Synch mechanism of Fatourou et al. can be transformed into a classic lock by choosing a value of 1 for the h parameter to avoid delegation, and using the prologue of the CC-Synch method as the lock operator and the epilogue as the unlock operator. The result is a queue-based spin-lock that can tolerate general critical sections and does not use delegated execution. Furthermore, if we start with Fatourou's H-Synch and (a) choose h to be 1 and (b) have each combiner increment a variable and release the global lock when this variable reaches a bound, then the result is tantamount to a cohort lock constructed from underlying CLH locks.

Suleman et al. [2009] proposed a hardware-based solution that allows critical sections to be migrated to fast processors, but no such implementations are generally available.

In summary, the HBO lock has the benefit of being conceptually simple, but is unfair and requires significant application-, load- 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.

1.2. Contributions

Figures 1 and 2 depict a scenario where NUMA-aware mutex locks can greatly improve application performance. The scenario illustrates a multithreaded application

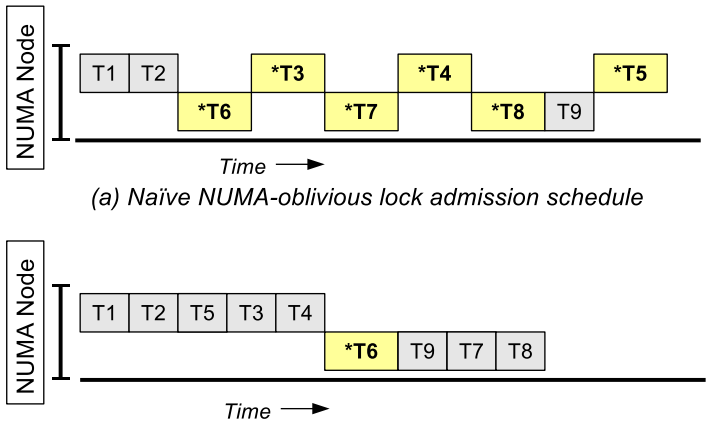

Fig. 2. Examples of lock admission schedules. Threads T1, T2, T3, T4 and T5 reside on NUMA node 0, while threads T6, T7, T8, and T9 reside on NUMA node 1. All are waiting to acquire a given lock. (a) reflects a NUMA-oblivious schedule, and (b) reflects a NUMA-friendly schedule that might be afforded by cohort locks. Critical section executions marked with "*" indicate that the designated thread incurred lock migration and consequently took longer to execute the critical section.

running 9 threads, all of which attempt to acquire a shared lock at about the same time. Figure 2 illustrates that biasing lock handoffs—transfer of ownership—to node-local threads can greatly improve performance by reducing cross-node communication.

This article presents *lock cohorting* [Dice et al. 2012a, 2012c], 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—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.

We report the performance benefits of cohort locks on 3 widely different systems, 2 of which are classic multichip NUMA systems (one x86-based and one SPARC-based) and one of which is a modern single-chip system. On the single-chip system, cohort locks exploit the on-chip cache hierarchies and varying proximity of caches to cores. This particular system reflects a design trend where single-chip processors are themselves becoming more nonuniform.

1.3. Lock Cohorting in a Nutshell

Say we have a spin-lock of type G that is *thread-oblivious*, that is, G 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 nonempty cohort of threads concurrently attempting to acquire the lock.

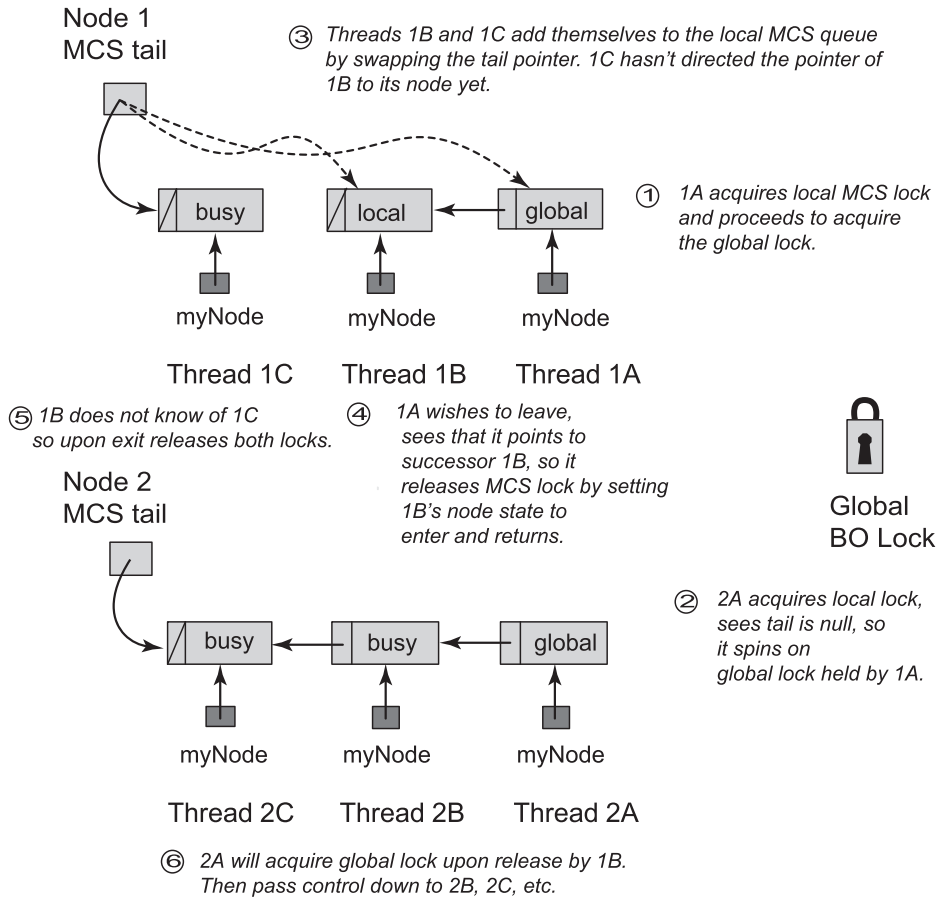

Fig. 3. A NUMA-aware C-BO-MCS lock for two nodes. A thread spins if its node state is *busy*, and can enter the critical section if the state is *local release*. A thread attempts to take the global lock if it sees the state set to *global release* or if it is added as the first in the queue (setting a null *tail* pointer to its own record). Notice that thread 1B does not notice that thread 1C is present at the MCS lock since 1C has not yet set the pointer in 1B's node that will refer to 1C's node. This is an example of a false-positive cohort-detection test in which alone? returns true despite the fact that the cohort is nonempty.

We convert a collection of NUMA-oblivious locks S and G into an aggregate NUMA-aware lock C as follows. A NUMA-aware cohort lock C consists of a single thread-oblivious global lock G and a set of local locks S_i, one for each NUMA node i. The local locks S_i must provide the cohort detect property. We say a cohort lock C 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 [Mellor-Crummey and Scott 1991] and G a simple *test-and-test-and-set backoff lock* [Agarwal and Cheritan 1989; Anderson 1990] (BO) as depicted in Figure 3. To access the critical section a thread must hold both the local lock S_i of its node, and the global lock G. Critically, given the special properties of S and G, once some thread in a node acquires G, ownership of the cohort lock C can be passed in a deadlock-free manner to another thread in the node using the local lock S_i, without releasing the global lock. Transferring ownership of S_i within a node also logically passes ownership of G and thus the cohort lock C itself. 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 node S_j to take ownership of the lock.

Broadly, the top-level global lock reflects which node owns the lock, and the node-level local locks identify which thread on that node holds the lock.

In more detail, each thread attempting to enter a critical section protected by a cohort lock C first acquires C's 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 nonempty 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 reacquire G before it can enter the critical section. The cohort detection property is therefore necessary in order to prevent progress failure in which a thread releases 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 node gives up the global lock. A simple node-local policy is to cede the global lock after an allowed number of consecutive local accesses. Duration-based bounds are also viable. 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 increased short-term unfairness.

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 node has acquired the global lock, control of the global lock is subsequently passed among contending threads within the node—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.

The local locks serve to moderate the arrival rate at the top-level lock G. Our approach allows ownership of the top-level lock, and thus the cohort lock, to be logically transferred between threads on a node in a very efficient manner, avoiding activity on the top-level lock. Actual acquire and release operations and subsequent transfers of ownership on the contended global lock are much most costly than are logical intranode transfers of the global lock via the local locks.

We use a naming convention of C-*G*-*S* to identify cohort locks. C-TKT-MCS, for instance, is a cohort lock formed from a top-level global ticket lock (TKT) and node-level local MCS locks.

1.4. Cohort Lock Designs and Performance

It is easy to find efficient locks that are thread-oblivious: BO or ticket locks [Mellor-Crummey and Scott 1991] have this property and can easily serve as the global locks. With respect to the cohort detection property, locks such as the MCS queue lock of Mellor-Crummey and Scott [1991] provide cohort detection by virtue of their design: each spinning thread's record in the queue has a pointer installed by its successor. Locks like BO, however, 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. Specifically, an indication of the existence of waiting threads needs to be visible to threads as they are about to release the local lock. Even if a lock implementation is completely opaque, a developer can still provide the cohort detection property by simply "wrapping" the acquisition operations of that lock with an atomic increment and decrement of a supplemental variable that tracks the number of contending threads.

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 inherently abortable) and the abortable CLH lock [Scott 2002] into abortable NUMA-aware cohort locks, which to our knowledge, are the first set of NUMA-aware abortable queue-locks.

We report results of our experimental evaluation of cohort locks on three distinct systems: (i) an Oracle SPARC T5440 system, (ii) an Oracle T4-1 SPARC system, and (iii) an Oracle Sun Fire X4800 system (with Intel x86 processors). These systems provide very different points in the NUMA architecture landscape, and help us get better insights into performance characteristics of the various NUMA-aware cohort locks. Our microbenchmark tests show that cohort locks outperform all prior best alternative algorithms (e.g. FC-MCS [Dice et al. 2011]) by significant margins: up to 60% on the T5440 and T4-1 systems, and 3X on the X4800 system. Our novel abortable NUMA-aware lock, the first of its kind, outperforms the HBO lock (abortable by definition) and the abortable CLH lock [Scott 2002] by about a factor of 6.

Our libc allocator experiments demonstrate how cohort locks can directly benefit multithreaded programs and significantly boost their node-level reference locality both for accesses by the allocator to allocation metadata under the lock and, more surprisingly, for accesses by the application to allocated memory blocks outside the critical section. In experiments conducted on a memory allocator stress test benchmark [Dice and Garthwaite 2002], cohort locks allow the benchmark to scale up to nearly 20X on the T5440 system, 10X on the T4-1 system, and up to 4X on the X4800 system.

Finally, our experiments with memcached [memcached.org 2013] demonstrate that while cohort locks do not impact performance of read-heavy workloads, they provide improvements to up to 5% in workloads with a moderate amount of writes, and in write-heavy workloads, they can improve performance by over 20%.

We describe our construction in detail in Section 2, both our general approach and five specific example lock transformations. We provide an experimental evaluation in Section 5.

2. THE LOCK COHORTING TRANSFORMATION

In this section, we describe our new lock cohorting transformation in detail. Assume that the system is organized into nodes of computing cores, each of which has a large cache that is shared among the cores local to that node, so that internode communication is significantly more expensive than intranode communication. We use the term *node* 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 node has a unique *node id* known to all threads running on the node.

Lock cohorting is a technique to compose NUMA-aware mutex locks from NUMA-oblivious mutex locks. It leverages two key properties of mutex lock implementations: (i) *cohort detection*, where a lock owner can determine whether there are additional threads waiting to acquire the lock; and (ii) *thread-obliviousness*, where the lock can be acquired by one thread and released by any other thread. Thread-obliviousness means that lock ownership can be safely transferred from one thread to another. Cohort locks are hierarchical in structure, with one top-level lock and multiple locks at the second level, one for each node in the NUMA system. The top-level lock must be thread-oblivious and the second-level locks must have the property of cohort detection. A cohort lock is said to be owned by a thread when that thread owns the top-level lock. Ownership of the top-level lock can be obtained directly by explicitly acquiring that lock, or implicitly, by logical transfer of ownership from a sibling via the local cohort lock.

To acquire the cohort lock, a thread must first acquire ownership of the local lock assigned to its node and then acquire ownership of the top-level lock. After executing its critical section, the cohort lock owner uses the cohort detection property of the node-local lock to determine if there are any local successors, and hands off ownership of the local lock to a successor. With this local lock handoff, the owner also implicitly passes ownership of the top-level lock to that same successor. Crucially, ownership of the top-level lock is logically transferred without physically accessing the top-level lock. This strategy avoids internode coherence traffic. If the lock owner determines that there are no local successors then it must release the top-level lock. The top-level lock's thread-obliviousness property comes into play here; the lock's ownership can be acquired by one thread from a node, then implicitly circulated among several threads in that node, and eventually released by some (possibly different) thread in that node.

To avoid starvation and provide long-term fairness, cohort lock implementations typically bound the number of back-to-back local lock transfers. (Unless otherwise stated we used a bound of 100 in all our experiments described in this article). Our algorithm intentionally trades strict short-term first-come-first-served FIFO/FCFS fairness for improved aggregate throughput. Smaller bounds are fairer over the short-term while large bounds are less fair over the short term, but provide higher potential throughput by virtue of reduced lock migration rates. Specifically, we leverage unfairness, where admission order deviates from arrival order, in order to reduce lock migrations and improve aggregate throughput of a set of contending threads. Unfairness, applied judiciously, and leveraged appropriately, can result in reduced coherence traffic and improved cache residency.

2.1. Rationale

We assume a single-writer multiple-reader MESI-like model [Papamarcos and Patel 1984] for cache coherence. At any given instant, a given cache line can be (a): in *shared* state simultaneously in 0 or more caches, or (b) in *modified* state in at most one cache and *invalid* in all other caches. *Shared* state indicates read-only permission and *modified* state reflects read-write permission. Specifically, if the line is *modified* in any one cache then it must be *invalid* in all other caches. No other concurrent readers or writers are permitted—writers have exclusive access. Zero or more caches can simultaneously have read permission (read-read sharing) as long as are no caches with write permission.

The primary goal of cohort locks is to reduce interconnect coherence traffic and coherence misses. In turn the hit rate in the local cache improves. We presume that critical section invocations under the same lock exhibit *reference similarity*: acquiring lock L is a good predictor that the critical section protected by L will access data that was accessed by recent prior critical sections protected by L. After a local handoff, data

to be written by the next lock owner is more apt to be in the owner's local cache, already in *modified* coherence state, as it may have been written by the prior owner. As such, the critical section may execute faster than if the prior owner resided on a different node. Cohort locks provide benefit by reducing coherence traffic on both lock metadata and data protected by the locks. If a cache line to be read is in *modified* state in some remote cache then it must currently be *invalid* or not present in the local cache. The line must be transferred to the local cache from the remote cache via the interconnect and downgraded to *shared* state in the remote cache.[3] Similarly, if a cache line to be written is not already in *modified* state in the local cache, all remote copies must be invalidated, and, if the line is not in *shared* state, the contents must be transferred to the writer's cache. Read-read is the only form of sharing that does not require coherence traffic. We are less concerned with classic NUMA issues such as the placement of memory relative to the location of threads that will access that memory and more interested in which caches shared data might reside in, and in what coherence states. Cohort locking works to reduce write invalidation and coherence misses satisfied from remote caches and does not specifically address remote capacity, conflict, and cold misses, which are also satisfied by transfers over the interconnect.

We note that cohort locks provide the least performance gain for locks that protect immutable or infrequently written data, although some gain is possible because of reduction in intercache "sloshing" of lock metadata. Cohort locks provide the most benefit when (a) thread $T1$ reads or writes a cache line C in a critical section protected by lock L where C was most recently written by thread $T2$, or (b) thread T writes to line C where C was last read by $T2$. Cohorting increases the odds that $T1$ and $T2$ reside on the same node, thus decreasing internode coherence transfers of line C. The degree of benefit afforded by cohort locks for a given critical section is proportional to the number of such distinct cache lines C. Compared to NUMA-oblivious locks, cohort locks reduce coherence misses incurred within the critical section arising from accesses to shared data, thus reducing the cycles-per-instruction for memory accesses in the critical section and in turn accelerating the execution of the critical section and reducing lock hold times.

Modern operating system schedulers exacerbate lock migration through thread placement policies that disperse threads widely over the available nodes [Dice 2011d]. This policy is appropriate for unrelated threads – threads that are not expected to communicate with each other through shared memory – and helps avoid conflicts for shared node-level resources such as cache occupancy, DRAM channels and thermal or power headroom. It assumes, however, that communication is rare or less important than node-level local resource conflicts. We also note in passing that it is conceivable to explicitly and intentionally migrate a thread to the node where the lock was last held. That is, we can move the thread to the cache containing the data instead of vice-versa. Thread migration is generally a costly operation, however, and leaves the thread vulnerable to so-called *seance communication* [Mendelson and Gabbay 2001].

NUMA-aware cohort locks take the NUMA location of waiting threads—*where* the threads resides in the NUMA topology relative to other queued thread and to the current lock holder—into account when determining lock admission schedule and entry order. Specifically, the scheduling policy and criteria incorporates NUMA location to reduce lock migration in the admission schedule. In contrast, a classic NUMA-oblivious lock such as MCS, takes only arrival time—*when* the thread arrived—into account. Our cohort design intentionally allows threads from one node to transiently dominate the lock and to preferentially pass ownership of the lock among contenting threads on

[3]or *other* state under MOESI cache coherence

that node. The design incorporates an explicit tunable trade-off between short-term fairness and overall aggregate throughput expressed via the local handoff bound.

In comparison to Oyama's technique of delegated execution, cohort locks need more contention before they provide benefit. Cohort locks yield benefit only when there are multiple contending threads on a node that can form a cohort. Oyama yields benefit–by keeping execution on one node while processing the stack of pending operations–when there is *any* contention in the system as a whole. But, as noted previously, it is not always possible to transform existing critical sections into closures as is required by Oyama. Furthermore, if the critical section happens to access thread-private data, then that data will itself be subject to cache line migration when using Oyama's technique. Finally, the integrated lock/stack location in Oyama constitutes a centralized coherence hot-spot.

2.2. Designing a Cohort Lock

We describe lock cohorting in the context of spin-locks, although it could be as easily applied to locks that block threads or use hybrid spin-then-block waiting strategies. We assume the standard model of shared memory based on execution histories [Herlihy and Shavit 2008].

A *lock* is an object providing mutual exclusion with *lock* and *unlock* methods, implemented in shared memory, and having the requisite safety and liveness properties Herlihy and Shavit [2008]. 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 2.1. 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 2.2. 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 conservative definition is intended to allow for very relaxed and efficient implementations of *alone?*. A degenerate implementation of *alone?* might always return true. This would be tantamount to using just the top-level lock, and there would be no performance benefits from cohort locking. In practice we expect that if *alone?* is called repeatedly while no other thread executes a lock operation, then it will eventually return true.

We construct a NUMA-aware cohort lock by having each node 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 nodes in the NUMA system), and G can be of different types, for example, the S_i could be slight modifications of MCS queue-locks [Mellor-Crummey and Scott 1991] and G a simple *test-and-test-and-set backoff lock* [Agarwal and Cheritan 1989] (BO) as depicted in Figure 3.

The *lock* method of a thread in node 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 node or on a count of the number of times the local lock was acquired in succession in a *local release* state. It defines a trade-off 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 node 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 top-level 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 node has acquired the global lock, ownership is passed among the node's threads with the efficiency of a local lock. This reduces overall cross-node communication and increases intranode locality of reference when accessing data within the critical section.

Figure 4 illustrates the operation of a cohort lock in high-level pseudo-code. The *may-pass-local* predicate is represented by the PassLocally flag. In our cohort locks algorithms, we set the consecutive local lock handoff bound to 100 unless otherwise stated, after which the top-level lock is released by the local lock releaser.

3. SPECIFIC COHORT LOCK DESIGNS

Though most locks can be used in the cohort locking transformation, we briefly explain a set of specific constructions. The first three embodiments, C-TKT-TKT, C-PTL-TKT, and C-BO-MCS are non-abortable locks – they do not support timeouts [Scott and Scherer 2001]. C-TKT-TKT consists of a top-level global ticket lock and node-level local ticket locks [Mellor-Crummey and Scott 1991]. C-PTL-TKT consists of a top-level global *partitioned ticket lock* [Dice 2011b] and node-level local ticket locks. C-BO-MCS uses a test-and-test-and-set backoff lock [Agarwal and Cheritan 1989] as the top-level lock and local MCS locks [Mellor-Crummey and Scott 1991] per NUMA node. 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, which uses backoff locks at the top-level and local node-levels. We then present an abortable cohort lock comprising an abortable global BO lock and abortable local CLH locks [Scott 2002], which we call the A-C-BO-CLH lock.

```
1  // TL : top-level global lock
2  // LL : node-level local lock
3
4  // acquire the cohort lock
5  lock() :
6     LL = IdentifyLocalLock()      // find node-specific local lock instance
7     LL->Lock()
8     if LL->PassLocally :          // TL handed off by last local lock owner
9        LL->PassLocally = 0
10       return
11    TL->Lock()
12    MemoizedLL = LL               // record for use in subsequent unlock()
13
14 // release the cohort lock
15 unlock() :
16    LL = MemoizedLL
17    if LL->alone() || LL->localHandoffLimitReached() :
18       LL->resetLocalHandoffCount()
19       TL->Unlock()
20       LL->Unlock()
21       return
22    LL->advanceLocalHandoffCount()
23    LL->PassLocally = 1           // pass ownership of global to sib in local cohort
24    LL->Unlock()
```

Fig. 4. Cohort Lock pseudocode sketch.

3.1. The C-TKT-TKT Lock

The C-TKT-TKT uses ticket locks (TKT) [Mellor-Crummey and Scott 1991] as both the local locks as well as the top-level global lock. A traditional ticket lock consists of two counters: *request* and *grant*. A thread intending to acquire the lock first performs an atomic fetch-and-increment operation on the *request* counter and then spins, busy-waiting for the *grant* counter to contain the value returned by the fetch-and-increment. A thread subsequently releases the lock by incrementing the *grant* counter.

We say a spin-lock algorithm provides *private spinning* if at most one thread busy-waits polling on a given variable at any one time.[4] Depending on the system and interconnect architecture, such private spinning may confer a performance advantage as the store instruction that updates that polled variable needs to be communicated only from the unlocking thread to the spinning thread, and not necessarily broadcast over the system. At most one cache should be participating in polling the location and have a valid copy of the line underlying the variable, so when the owner releases the lock and writes to the location, only one cache must suffer invalidation. To be more precise, we wish to minimize the number of distinct nodes that have threads polling a given location: the number of caches participating in polling. Locks that employ private spinning may also enable the efficient use of instructions such as MONITOR-MWAIT. Under private spinning when a lock is released at most one thread will wake from MWAIT. A lock uses *global spinning* if all threads waiting for that lock spin on just one location. Classic ticket locks employ global spinning.[5]

Ticket locks are trivially thread-oblivious; a thread can increment the *request* and another thread can correspondingly increment the *grant* counter. Cohort detection is also easy in ticket locks; threads simply need to determine if the *request* and *grant*

[4]We use the term *private spinning* instead of *local spinning*. Local spinning has been variously defined to mean: (a) at most one thread spins on a given location at any one time; (b) the spin loop is read-only; (c) the thread and the busy-wait variable are colocated on the same node; (d) the busy-wait variable can reside in the local cache of the spinning thread; and (e) combinations thereof.

[5]In addition to the number of threads or caches that can concurrently spin on a variable, the placement of that variable relative to the threads can impact performance. Depending on the architecture, there may be an advantage if the thread or threads spinning on a location are colocated on the same node as the memory underlying the location.

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 TKT lock, and then the global TKT 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 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 resides in the local TKT lock.[6] It then releases the local TKT lock by incrementing the *grant* counter. If the *request* and *grant* counters are the same, the owner releases the global TKT lock and then the local TKT lock (without setting the *top-granted* field).

We note that David et al. [2013] independently rediscovered C-TKT-TKT, which they refer to as *hticket*.

3.2. The C-PTL-TKT Lock

We now introduce a cohort lock that uses a TKT lock [Mellor-Crummey and Scott 1991] for the NUMA node-local locks and a partitioned ticket lock (PTL) [Dice 2011b] for the top-level lock. We call this lock C-PTL-TKT. Figure 5 depicts the C-PTL-TKT lock algorithm in a schematic fashion. We assume the counters are sufficiently wide so as to obviate issues related to ticket overflow and wrap-around. In practice, 64-bit fields suffice.

PTL is similar to the TKT in that it uses the *request* and *grant* fields to coordinate lock acquisition. However, PTL disperses contention on the *grant* field by replacing it with an array of *grant* slots. The thread that intends to acquire the PTL lock first atomically increments the *request* counter and then spin-waits on the *grant* slot (computed using a simple mod operation: line 38) corresponding to the value returned from the atomic increment of *request*. A thread releases the lock by writing an incremented value of *grant* in the "next" grant slot (line 60). Relative to TKT, PTL reduces the number of threads concurrently spinning on a given location when the lock is contended; the update of a *grant* slot by a release operation causes fewer futile invalidations in the caches of the busy-waiting threads. PTL provides *semi-private spinning*. When the number of contending threads is less than the number of slots, the lock provides purely private spinning. As we shall demonstrate in Section 5, both these benefits play a key role in delivering excellent performance for C-PTL-TKT on widely varying NUMA architectures. Like TKT, PTL is trivially thread-oblivious.

3.3. The C-BO-MCS Lock

The design of the C-BO-MCS lock C-BO-MCS 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 a classic 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 an atomic *swap* operation 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. A thread spin-waits on a field in the queue node that it installed, and is informed by its predecessor thread that is has become the lock owner. Thereafter, the thread can

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

```
1  // data structures for lock types
2  TktLock :                                  // ticket lock : TKT
3     volatile int Request                    // next ticket
4     volatile int Grant                      // now serving
5     volatile int TopGrant                   // pass-locally handoff flag
6     volatile int BatchCount                 // tally of consecutive local handoffs (counts down)
7
8  PTLock :                                   // Partitioned ticket-lock : PTL
9     volatile int Request
10    volatile int Grants[Slots]              // for performance, Slots should be >= # of NUMA nodes
11    volatile int OwnerTicket
12
13 // C-PTL-TKT instance fields :
14 CPTLTKTLock :
15    PTLock TopLock                          // top-level global lock
16    TktLock * volatile TopHome              // local node with ownership of top-level lock
17    TktLock LocalLock[NumaNodes]            // set of node-level local locks : ensemble
18
19 // acquire the C-PTL-TKT lock
20 void lock (CPTLTKTLock * C) :
21    TktLock * L = &C->LocalLock[MyNode()]   // identify node-specific local lock instance
22
23    // acquire local ticket lock : gateway followed by waiting phase
24    int t = FetchAndIncrement(&L->Request)  // atomic
25    while L->Grant != t :                   // spin-wait
26       Pause()                              // busy-wait politely
27
28    // This thread now owns the local lock
29    // Must now get ownership of top-level lock : either passively by transfer from sibling or actively by physically acquiring
30    if L->TopGrant :
31       // claim : top-level lock also owned and C->TopHome == L !
32       // This thread inherits ownership of the top-level - conveyed from previous local owner
33       L->TopGrant = 0
34       return
35
36    // acquire top-level TopLock : physically and explicitly acquire
37    // The waiting phase for partitioned ticket locks is MONITOR-MWAIT friendly
38    t = FetchAndIncrement(&C->TopLock.Request)   // atomic
39    while C->TopLock.Grants[t mod Slots] != t :  // spin-wait
40       Pause()                              // busy-wait politely
41
42    C->TopLock.OwnerTicket = t              // record for subsequent release of top-level
43    C->TopHome = L                          // record for subsequent unlock
44
45 // release the C-PTL-TKT lock
46 void unlock (CPTLTKTLock * C) :
47    TktLock * L = C->TopHome
48    int G = L->Grant + 1
49    if L->Request != G :                    // Check "Alone?" predicate
50       // cohort detection : local lock has waiters - contended
51       // The existence of a local cohort is a stable property as long as the local lock remains held
52       if -- L->BatchCount >= 0 :
53          // forward ownership - hand off to next thread in local cohort
54          L->TopGrant = 1                  // logically pass & transfer ownership of TopLock
55          L->Grant = G                     // release node-level local lock to sibling
56          return
57       L->BatchCount = BatchLimit          // Reset local handoff limit - bound
58
59    // abdicate both - safe to release locks in either order
60    int T = C->TopLock.OwnerTicket + 1      // release top-level lock : physically
61    C->TopLock.Grants[T mod Slots] = T
62    L->Grant = G                            // release node-level lock
```

Fig. 5. C-PTL-TKT pseudocode.

enter the critical section, and release the lock by transferring lock ownership to its queue node's successor. MCS locks provide private spinning. (In subsequent sections we discuss the suitability of MCS locks as a sole primary lock, and in cohort locks for use as top-level locks and node-level local locks).

The MCS lock can be easily adapted to enable cohort detection as follows. We implement the *alone?* method by simply checking if a thread's record has a nonnull 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 BO lock and modified MCS locks can be integrated into the cohort lock protocol to deliver a NUMA-aware lock.

3.4. Abortable Cohort Locks

The property of *abortability* [Scott and Scherer 2001] 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.

3.4.1. The A-C-BO-BO Lock. The A-C-BO-BOlock is based on the C-BO-BO lock, which we now describe. 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. Figure 6 shows a schematic version of the A-C-BO-BO algorithm.

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 (atomic compare-and-swap) 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, intranode write-sharing typically enjoys low latency, mitigating any ill-effects of contention on cache lines that might be modified by threads on the same node. These observations are confirmed in our empirical evaluation.

The A-C-BO-BO lock is very similar to the C-BO-BO lock that we described above, 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

```
1  // data structure for node-local BO lock
2  BOLock :
3     volatile int tasBackoff      // encoded as FREE, LOCKED, FORWARED
4     volatile bool succ           // successor indicator
5
6  // top-level A-C-BO-BO lock
7  ACBOBOLock :
8     volatile int TopLock         // encoded as FREE, LOCKED
9     BOLock * volatile TopHome ;
10    BOLock LocalLocks[NumaNodes]
11
12 // acquire the A-C-BO-BO lock
13 bool lock (ACBOBOLock * C, TimeUnit patience) :
14    TimeUnit startTime = CurrentTime()
15    BOLock * L = &C->LocalLocks[MyNode()]
16    int retSwap
17
18    L->succ = TRUE               // making waiting thread visible to threads in unlock
19    // A more precise approach would be to implement succ as a counter incremented and decremented by atomic operations
20    for :
21       while L->tasBackoff == LOCKED :
22          if (CurrentTime() - startTime) > patience :
23             L->succ = FALSE                      // abandon attempt - retract intent ; ST-LD memory fence needed
24             if L->tasBackoff == FORWARDED :      // local lock forwarded
25                break
26             return FALSE                         // lock acquire failed
27          if L->succ == FALSE :
28             L->succ = TRUE
29          backoff();
30
31       // acquire local BO lock
32       retSwap = SWAP (&L->tasBackoff, LOCKED)    // atomic swap instruction
33       if retSwap != LOCKED :
34          break                                   // local BO lock acquired; break out of loop
35
36    // local BO lock is acquired at this point
37    if retSwap == FORWARDED :
38       return TRUE                                // lock forwarded locally; we have it now, return
39
40    // need to acquire TopLock
41    while SWAP (&C->TopLock, LOCKED) != FREE :
42       while C->TopLock != FREE :
43          if CurrentTime() - startTime > patience :
44             L->tasBackoff = FREE
45             return FALSE                         // lock acquire failed
46          backoff()
47
48    // TopLock acquired
49    C->TopHome = L
50    return TRUE
51
52 // release the A-C-BO-BO lock
53 void unlock (ACBOBOLock * C) :
54    BOLock * L = C->TopHome
55
56    if L->succ == TRUE :
57       // apparent successor - attempt to forward lock
58       L->tasBackoff = FORWARDED                  // ST-LD memory fence
59       // validate and ratify continued existence of some successor
60       if L->succ == FALSE
61          // possible race with aborting thread; inopportune interleaving - detect and recover
62          if CompareAndSwap(&L->tasBackoff, FORWARDED, FREE) == FORWARED :
63             // released local BO lock; now release TopLock
64             C->TopLock = FREE
65             return
66       // local BO lock already acquired by someone local
67       return
68
69    // no successor detected; release TopLock and local BO lock
70    C->TopLock    = FREE
71    L->tasBackoff = FREE
```

Fig. 6. A-C-BO-BO pseudocode.

releaser checks to see if the *successor-exists* flag was set (which indicates that there exist threads in the local node 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.[7]

[7]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).

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.

3.4.2. 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 [Scott 2002] (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 [2002]. 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 nodes. However, since the CAS targets memory that is likely to already be resident in cache of the local node 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.

4. CORRECTNESS SKETCH

The following simple informal correctness arguments can be readily turned into formal proofs of mutual exclusion (safey) and deadlock freedom (eventual progress).

To see that the *mutual exclusion property* of any cohort lock is guaranteed, assume by way of contradiction that there are two threads in the critical section. They cannot be from separate nodes because, by the algorithm, when a thread from a given node enters the critical section, either it acquired the global lock itself or a thread from its node must have released the local lock with a *local release* state. In both cases the global lock is locked. By the mutual exclusion property of the global lock only one thread, and therefore one node, can own the global lock at a time. Thus, the two threads that are by way of contradiction in the critical section must be from the same node. However, the local lock guarantees that there can be at most one owner thread of the lock at any given time. Since a thread must acquire its local lock in order to enter the critical section, there cannot be two threads from the same node entering the critical section, a contradiction.

To see that a cohort lock provides *deadlock freedom*, recall that according to the standard definition [Herlihy and Shavit 2008], being deadlocked implies that some thread is not managing to successfully complete a *lock* method call, yet other threads are not entering the critical section infinitely often. Assume by way of contradiction that we have a deadlock. This implies the thread must be spinning forever on either the local or global lock, and other threads are not entering the critical section infinitely often. If a thread is spinning forever on the local lock, then since it is a deadlock free lock, it must be that some thread is holding the local lock and spinning on the global lock, never acquiring it (because otherwise it would enter the critical section and eventually release the local lock, possibly in addition to releasing the global lock, upon exiting the critical section). Thus we are reduced to the case of a thread spinning forever on the global lock. Since the global lock is a deadlock free lock, this can only happen if there is no thread in the critical section. However, the only way a thread (from some node) can leave the critical section without releasing the global lock, is if its call to *alone?* returned false and it released the nodes's local lock, which implies, given the deadlock freedom of the local lock, that some thread from this node will enter the critical section, contradicting the possibility of a deadlock.

The *starvation freedom* and *fairness* of a cohort lock transformation depend on the starvation-freedom and fairness of the locks chosen for the implementation and on the choice of the *may-pass-local* method.

5. 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 these locks in several ways. This gives us a good insight into the performance characteristics of the locks. Second, we modified the libc memory allocator to study the effects of cohort locks on allocation intensive multithreaded applications; we present results of experiments on a microbenchmark [Dice and Garthwaite 2002]. Third, we integrated these locks in memcached, a popular key-value data store application, to study their impact on real-world workload settings.

Our microbenchmark evaluation clearly demonstrates that cohort locks outperform all prior locks by significant margins. Additionally, the abortable cohort locks scale vastly better (by a factor of 6) than the state-of-the-art abortable locks. Furthermore, our libc allocator experiments demonstrate that simply replacing the lock used by the default Solaris allocator with a cohort lock can significantly boost node-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 multithreaded application that make heavy use of memory allocation services. Finally, cohort locks improved the performance of memcached by up to 20% for write-heavy workloads.

In our evaluation, we compare the performance of our nonabortable 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-TKT-TKT, C-BO-MCS and C-PTL-TKT cohort locks. We compare these with multiple NUMA-oblivious locks: the ticket lock (TKT) [Mellor-Crummey and Scott 1991], the partitioned ticket lock (PTL) [Dice 2011b] and the MCS lock [Mellor-Crummey and Scott 1991]; and other NUMA-aware locks, namely, HBO [Radović and Hagersten 2003], and FC-MCS [Dice et al. 2011]. 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 [Scott 2002].

In addition to MCS we also evaluated CLH [Craig 1993; Magnussen et al. 1994] for use as a sole primary lock, but opted to present only MCS as the two locks use structurally similar queue-based implementations and yielded similar performance over our suite of benchmarks. Both provide private spinning. Crucially, both MCS and CLH provide strict FIFO/FCFS admission order – admission order simply reflects arrival order. As such, they are *NUMA-oblivious*. Scott [2013] notes that MCS may be preferable to CLH in some NUMA environments as the queue "node" structures on which threads busy-wait will migrate between threads under CLH but do not circulate under MCS.

For the libc allocator and memcached experiments, we implemented all the above locks within LD_PRELOAD interposition libraries that expose the standard POSIX pthread_mutex_t programming interface. The pthreads interface gives the implementor considerable latitude and license as to admission schedule and fairness. Using LD_PRELOAD interposition allows us to change lock implementations by varying the LD_PRELOAD environment variable and without modifying the application code that uses locks. We implemented all of the lock algorithms and benchmarks in C and C++ compiled with GCC 4.7.1 at optimization level -O3 in 32-bit mode. As required, we inserted memory fences to support the memory model on x86 [Sewell et al. 2010] and SPARC where a store and load in program order can be reordered by the architecture. While not shown in our pseudo-code, padding and alignment were added in the data structures to avoid false sharing. We 64 bytes on SPARC and 128 bytes on x86. The unit of cache coherence is 64 bytes on our x86 processors, but Intel explicitly recommends 128 bytes, and 128 bytes serves to reduce false sharing arising from the processor's *adjacent sector prefetch* facility.

We use the Solaris *schedctl* interface to efficiently query the identity of the CPU on which a thread is running, requiring just two load instructions on SPARC and x86 platforms. In turn the CPU number may be trivially and efficiently converted to a NUMA node number. The RDTSCP instruction may be a suitable alternative to schedctl on other x86-based platforms where the kernel has arranged to return the CPUID. In our implementation a thread queries its NUMA node number each time it tries to acquire a lock. Involuntary thread migration that might occur after the thread has queried its NUMA node but before the thread releases the lock is rare and benign with respect to the safety and correctness of our cohort algorithms.

In order to adhere as much as possible to real-world execution environments we do not artificially bind threads to hardware contexts, and instead rely on the default Solaris kernel scheduler to manage placement of the benchmark's threads in the NUMA system [Dice 2011d]. Our cohort locks tolerate ambient thread placement and migration and do not require explicit binding of threads to processors.

We note that all our current cohort lock designs are work-conserving. While our lock policies may schedule admission to reduce lock migration, they never insert artificial delays to encourage or promote cohort formation at the expensive of denying entry when there are arriving or eligible waiting threads elsewhere in the system.

We report the results of several experiments we conducted on three systems, all of which exhibit interesting NUMA characteristics.

— An Oracle T5440 system which consists of 4 T2+ chips, each chip containing 8 cores, and each core containing 4 hardware thread contexts per pipeline, for a total of 256 hardware thread contexts. Each T2+ chip is a distinct NUMA node and the nodes are connected via a central coherence hub. Internode cache coherence is maintained via a MOESI [Sweazey and Smith 1986] protocol. Each T2+ chip has 4MB local L2 cache, locally connected memory, and runs at a 1.4 GHz clock frequency. Each core has an 8KB L1 cache shared by the pipelines on that core. The system was running Solaris 10.
— An Oracle T4-1 SPARC processor [Oracle Corporation 2012]. The T4-1 consists of a single SPARC T4 processor chip running at 2.8 GHz, having 8 cores with 2 pipelines per core. When only one thread is active on a core, both pipelines can be fused to accelerate the execution of that one thread. Consequently, purely cycle-bound applications with independent threads that should be ideally scalable will see no improvement in aggregate performance as we increase the number of threads beyond 8. The system supports a total of 64 hardware contexts. Each core has private L1 and L2 data caches. The L2 caches are write-back and 128KB each. The shared L3 cache is banked 8 ways and accessed over a common crossbar that connects the cores and banks. A bit field in the physical address selects a specific L3 bank where a line will be cached. The 8 L3 banks are divided into 2 groups, with each group attached to a unique DRAM channel. The L2 uses a standard MESI [Papamarcos and Patel 1984] cache coherence protocol for on-chip coherence with cache-to-cache transfers over the crossbar between local L2 caches. The inclusive L3 is shared by all cores and employs MOESI coherence. The system was running Solaris 11.0.
  The T4-1 is *not* a classic NUMA system – DRAM and L3 are equidistant from all cores – but by treating each of the 8 cores as a NUMA node for the purposes of cohorting, we are able to minimize L2 cache-to-cache transfers and improve performance. This approach is not profitable for the T2+ processors – such as those used in the T5440 – where there is just a small core-private write-through L1 cache.
— An Oracle Sun Fire X4800 system [Oracle Corporation 2010] consisting of 8 Intel Xeon X7560 x86 chips, each running at 2.2 GHz with 8 cores per chip and 2 hardware contexts per core. The system supports a total of 128 hardware contexts. Each chip is a NUMA node, and the 8 chips are connected in a glueless "twisted ladder" Mobius configuration via chip-to-chip QuickPath Interconnect (QPI) coherence links [Intel Corporation 2009]. The system uses QPI version 1.0 with source-based snooping and the MESIF [Goodman and Hum 2009] coherence protocol, a variation on MESI. For a given chip, 3 other chips are 1 hops distant and 4 chips are 2 hops distant. The system was running Solaris 11.1.

We note that on the multichip systems (T5440 and X4800), the Solaris thread scheduler tends to uniformly distribute the threads of a process across all the chips (NUMA nodes) of the system. For instance, for a process with 32 threads running on the T5440 system, which contains 4 64-way T2+ chips, the Solaris scheduler will tend to schedule 8 threads of the process on each NUMA node on the system. For the same process, the scheduler will tend to schedule 4 threads on each chip of the X4800 system, which contains 8 16-way Intel Xeon X7560 x86 chips. The Solaris scheduler is work-

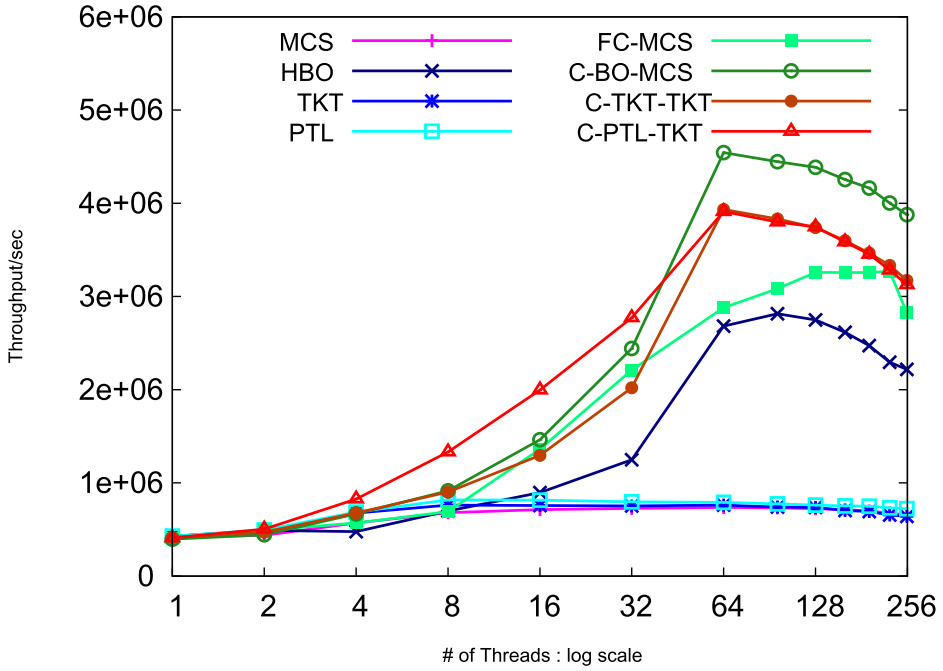

Fig. 7. T5440 : Scalability results of the LBench experiment.

conserving, but to maintain cache residency, tries to avoid involuntary migration of threads. Thread migration was observed to be minimal for all our experiments.

We present scalability results of LBench and mmicro on all the above systems, and further analyze other performance aspects (e.g. fairness, reference locality, etc.) of LBench on the T5440 system in greater detail. We present the scalability results of our memcached experiments on just the T5440 system. Unless explicitly stated otherwise, results reported were averaged over 3 test runs.

5.1. Microbenchmark Evaluation

5.1.1. Scalability. We constructed what we consider to be a reasonable representative microbenchmark, LBench [Dice et al. 2012a], to measure the scalability of lock algorithms. LBench launches a specified number of identical concurrent 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 noncritical work phase of about 4 microseconds. The critical section reads and writes shared variables residing on two distinct cache lines. At any point in the measurement interval a given thread is either waiting for the central lock, executing in the critical section, or executing in the noncritical section. At most one thread will be in the critical section at any given moment, although multiple threads can be executing concurrently in their noncritical sections. 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 and to report iterations by node, providing a measure of internode fairness.

Figures 7, 8, and 9 depict the performance of the nonabortable locks on all three systems. We vary the thread-count in the X-axis and report throughput on the Y-axis

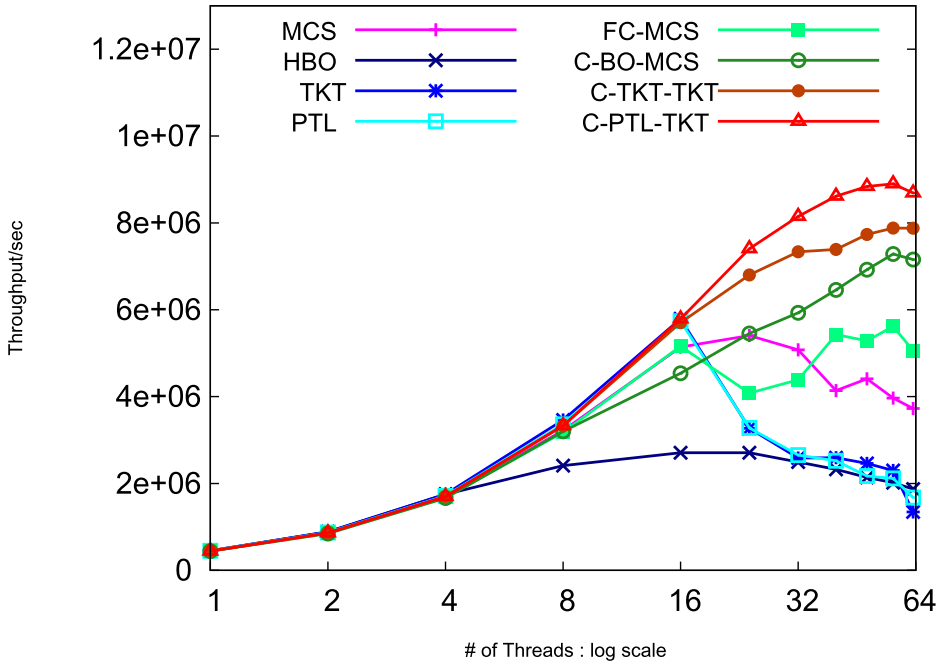

Fig. 8. T4-1 : Scalability results of the LBench experiment.

as aggregate iterations per millisecond. As a baseline to compare against, we measured the throughput of the MCS lock, which is a classic queue-based spin-lock. As expected, the lock does not scale as well as the NUMA-aware locks because it does not leverage reference locality, which is critical for good performance on NUMA architectures. Since the T4-1 machine is a single-chip multicore system, MCS does appear to scale up to 16 threads, but tapers off thereafter.

TKT and PTL exhibit performance patterns similar to that of MCS on the T5440 system. On the other two systems, TKT and PTL [Dice 2011b; Mellor-Crummey and Scott 1991] exacerbate contention across chip boundaries even further, and hence perform worse than MCS at high threading levels. Among all three systems, internode cache-to-cache transfers are most expensive on the X4800 system. On the X4800 we see that PTL maintains a margin over TKT for moderate thread counts. TKT uses global spinning and has just a single *grant* field which can be a coherence hot-spot, whereas PTL has semi-private spinning and disperses polling and updates over the multiple slots in its *grant* array. (We used 8 slots for PTL in all our experiments). When the thread count is below the number of slots, PTL provides purely private spinning and the performance remains above that afforded by TKT. But as the thread count passes the number of slots, the slots become shared by multiple threads and the performance of PTL on the X4800 deteriorates and falls toward that of TKT. On the SPARC systems, however, PTL and TKT yield similar results.

The HBO and FC-MCS locks scale somewhat better on the T5440 system, but do not perform well on the other two systems. We have observed that, contrary to cohort locks, both HBO and FC-MCS are highly sensitive to the underlying workload and system, and need to be finely tuned for most settings in order to get performance improvements beyond the NUMA-oblivious locks. Tuning and selection of backoff parameters for HBO convolves issues of fairness and interconnect communication costs, as well as

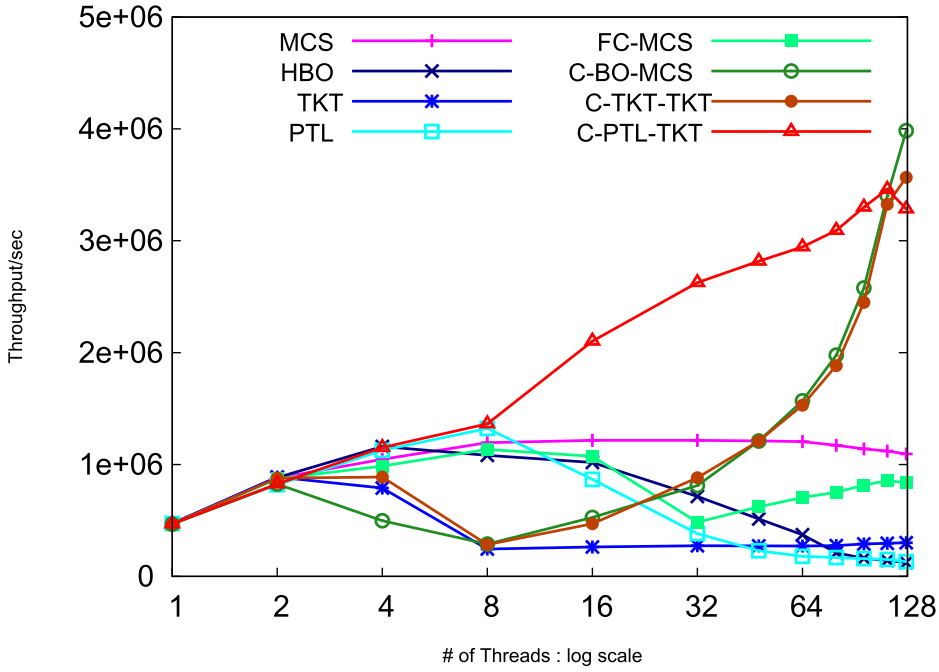

Fig. 9. X4800 : Scalability results of the LBench experiment.

available concurrency levels and critical section duration. For all experiments we used HBO and FC-MCS tuning values that yielded optimal performance on LBench running on the T5440. Those tunings are not portable over other applications or platforms. In fact, the tunable parameter settings that might yield the best results in one context might yield relatively poor performance in another.

The cohort lock results on the three systems provide interesting insights into the interactions of these lock designs and the underlying NUMA architectures. First, on the T5440 system, we find C-BO-MCS to be the best performing lock, followed by C-TKT-TKT and C-PTL-TKT, which perform comparably at high threading levels.

The T4-1 system is not a classic NUMA architecture. However, core-to-core crossbar latencies are substantially higher than core-local cache accesses. That is, access to a line that must be satisfied from another core's L2 via the crossbar is more expensive than accessing a line in a core's own local L2. As a result, the cohort locks scale substantially better (up to 60%) than all other locks. However, the relative performance of the cohort locks is significantly different from what we observed with the T5440 system. In particular, the top-level lock design appears to be the dominating factor for scalability on the T4-1 system. Since BO is least scalable at the top level, C-BO-MCS performs the worst of the cohort locks. Since, compared to the BO lock, the TKT lock distributes update traffic between its *Request* and *Grant* fields, C-TKT-TKT performs better that C-BO-MCS. C-PTL-TKT is more scalable than C-TKT-TKT because its top-level PTL lock disperses updates between its ticket partitions (8 in our experiments) at the top-level.

Results on the X4800 system are equally interesting. As depicted in Figure 9, C-BO-MCS and C-TKT-TKT pay a heavy price for having any contention what so ever on the top-level lock. However, since PTL tends to disperse contention across its multiple

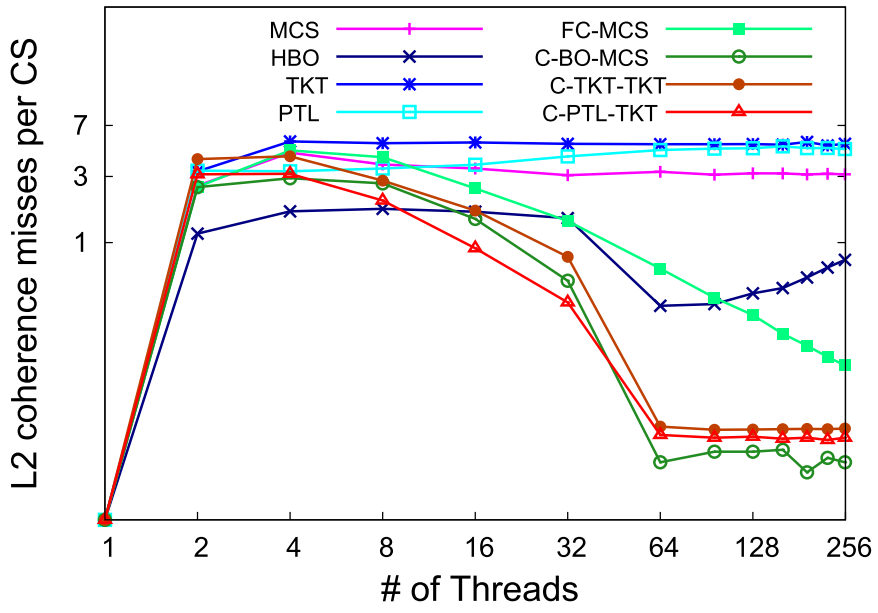

Fig. 10. The graph shows the average number of L2 cache coherence misses per critical section execution for the experiment in Figure 7 (lower is better). The X-axis and Y-axis are in log scale.

partitions, it does not deteriorate in performance at low thread counts. As the thread count increases beyond 8 we see the results of the build-up of threads at each NUMA node for the cohort locks. This build-up—cohort formation—leads to frequent local lock handoffs, which in turn leads to better performance.

Note that on all platforms, composite cohort lock constructions outperformed their constituent underlying lock types. C-PTL-TKT, for instance, is more scalable than either PTL or TKT.

5.1.2. Locality of Reference. To further establish etiology and support our claimed mode of benefit, we analyze coherence miss rates under various lock algorithms, showing that low coherence miss rates strongly agree with higher throughput. Figure 10 provides the key explanation for the observed scalability results on the T5440 system. Recall that each chip (a NUMA node) on the T5440 system 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 owned by a remote L2 cache (on the T5440 system, remote L2 access is approximately 4 times slower than local L2 access during light loads). The latter also involves interconnect 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 and congestion if the system is under heavy load. Figure 10 reports the L2 coherence miss rates collected during the LBench experiments. 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. As previously noted, LBench can report lock migration counts. Those counts correlate strongly with L2 cache coherence misses, and higher rates correlate with lower aggregate throughput.

TKT, PTL, and MCS are the fairest among all the locks, and hence experience high L2 coherence miss rates and comparatively low aggregate throughput. We believe the

differences shown by these 3 locks in Figure 10 are attributable to different ways in which they access their lock metadata. TKT uses global spinning and displays the highest miss rates. PTL disperses grant access over across all 8 the partitions, and hence experiences slightly lower L2 coherence misses at lower thread counts, but experiences equally high misses at high thread counts, where there is enough contention at each partition. Classic MCS does better than TKT and PTL because of its local and private spinning properties. A contended acquire-release pair under TKT updates 2 central global variables (*ticket* and *grant*). Under MCS, acquire will update the central *tail* field and release will update a flag in the successor's queue node. PTL updates the central *ticket* field, but diffuses polling and release updates over the array of *grant* slots.

HBO shows a very good coherence miss rate until the number of threads is substantially high (64), after which, the miss rate deteriorates considerably. The L2 miss rate under FC-MCS degrades gradually, but consistently, with increasing thread count.

All our cohort locks have significantly lower—by a factor of two or greater—L2 miss rates than all other locks. (Note that the Y-axis is in log scale). This is because the cohort locks consistently provide long sequences of successive accesses to the lock from the same NUMA node (cohort), which accelerates the critical section performance by reducing intercore 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 important, 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 $T3$ (placed 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. $T1$ 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 of local lock requests in cohort locks can significantly boost local handoff rates when there is sufficient node-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 10, and that the batching rate in cohort locks increases more quickly with contention compared to other locks.

Note that, at high threading levels, the L2 coherence miss rates of C-BO-MCS are noticeably better than the ones observed in C-TKT-TKT and C-PTL-TKT. This results in the performance edge that C-BO-MCS has over the latter two locks in Figure 7. We delve into the reason for this apparent advantage in the following section.

5.1.3. 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 7 on the relative standard deviation of per-thread throughput from the average throughput of all the threads. The results from the T5440 are shown in Figure 11. These results give us a sense of the distribution of work completed over the set of threads during a one minute execution of the test run.

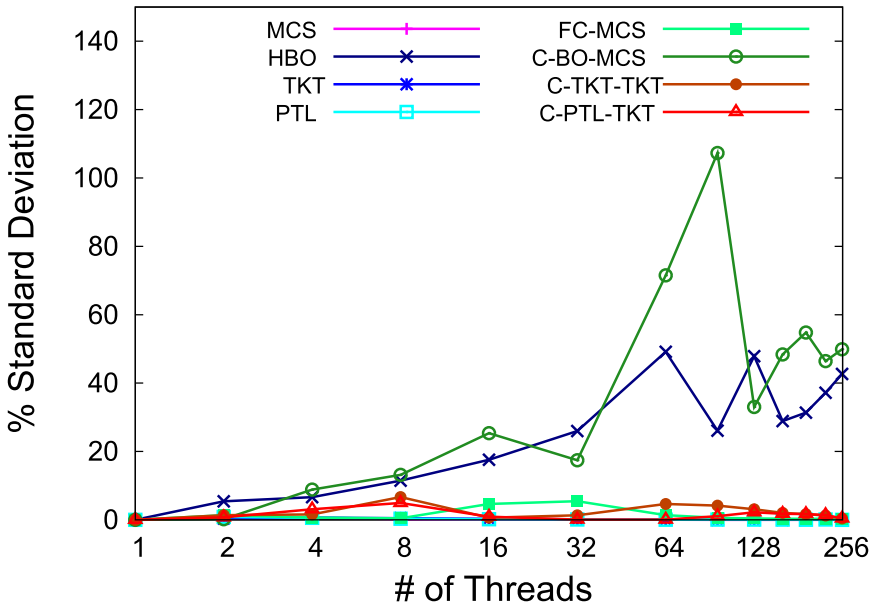

Fig. 11. The graph shows the relative standard deviation (RSTD) of per-thread throughput from the average throughput reported in Figure 7. The X-axis is in log scale. The higher the relative standard deviation, the less fair the lock is in practice.

To our surprise, we found C-BO-MCS to be the least fair lock even though we limit local lock handoffs in all cohort locks to a maximum of 100 at a time in our experiments. Examining the distribution of work completed by threads, we saw that the threads that made the most progress were resident on just one node. We initially suspected the role of backoff delay when releasing the top-level lock. Threads on other nodes might be stalled on a long backoff period, allowing another thread from the same node as the previous owner to immediately reacquire the top-level lock. As a transient experiment we forced a 0-length delay for the BO lock, but still observed significant unfairness. On further reflection, the reason for this unfairness is clear – the global BO lock in C-BO-MCS is highly unfair. A thread T on node N releases the global BO lock with a simple store operation. That store causes the line underlying the lock word to go into modified state in the releaser's cache and invalid in the caches of other nodes. Other contending threads on N acquire the local lock and then immediately attempt to acquire the global BO lock. Very often that attempt will succeed. Busy-waiting threads on other nodes need to upgrade the line from invalid to shared, and if they see the lock is not held, from shared to modified as they attempt to acquire the lock. Hence unfairness arises from unfairness in cache coherence arbitration. This phenomena allows one node to dominate the lock for long periods.[8] Moreover, once a thread on another node notices the lock is available, it must then branch to pass control to the atomic operation that attempts to acquire the lock. That conditional branch is typically predicted to stay in the busy-wait loop, thus the thread incurs a mispredict stall, increasing the length of the window where a thread on the original node N might arrive at the top-level lock and regain ownership.

[8]As an experiment on x86 we used the *CLFLUSH* instruction after the store that releases the lock in order to expel the cache line underlying the lock word from the cache hierarchy. This improved fairness but degraded overall throughput.

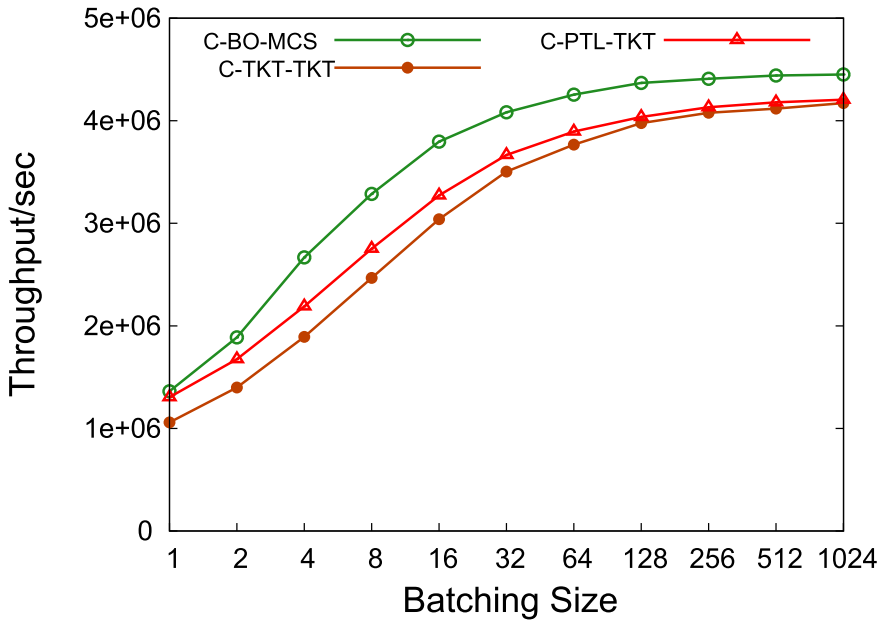

Fig. 12. The influence of the batching size in our cohort locks. All the test were conducted by fixing the number of threads to 64.

On the other hand, since C-TKT-TKT and C-PTL-TKT are composed from fair locks, these locks provide excellent long-term fairness. HBO is the second least fair lock. All other locks are provide excellent long-term fairness properties.

5.1.4. Batching Size Effect. To further illustrate that the mode of benefit from cohort locks arises from a reduction in lock migration, we provide a sensitivity analysis in Figure 12, with batching limits on the X-axis and aggregate throughput on the Y-axis. Data was collected with LBench on the SPARC-based T5440 system where the number of threads was fixed at 64. As is evident in the graph, increasing the batch size to modest levels can yield significant improvements in throughput. Furthermore, the improvements diminish significantly beyond a certain point (batch size > 100). When the batching size is 1, the performance of the cohort lock is little different than the performance affording by using the top-level lock alone. At that point the cohort lock is effectively a degenerate version of the top-level lock and the local locks serve no useful purpose.

Throughput is a function of a batching size as well as other factors, such as platform-specific cache coherent communication costs and application-specific factors such as the number of distinct cache lines written in the critical section. A limit of 100 is both practical and reasonable, however.

5.1.5. 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 the scalability results reported in Figure 7 with an eye toward performance at low contention levels.

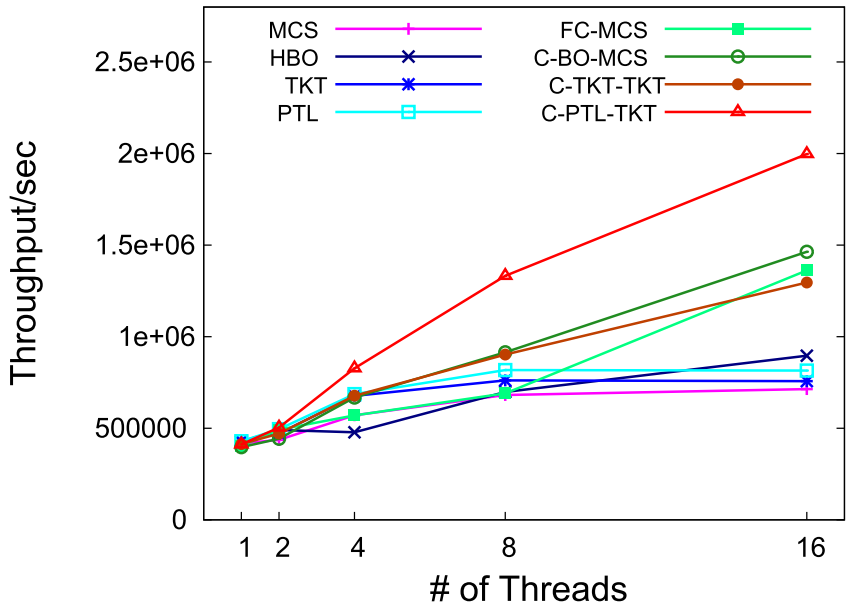

Fig. 13. A closer look at the throughput results of Figure 7 for low contention levels (1 to 16 threads).

Figure 13 zooms into that part of Figure 7. Interestingly, we observed that the performance of all the cohort locks was better than or competitive with all other locks that do not need to acquire locks at multiple levels in the hierarchy (viz. MCS, HBO, and FC-MCS). Acquiring the local lock tends to be a relatively cheap operation as the cache lines underlying that lock's metadata are likely to be resident in the local cache. On further reflection, we note that the extra cost of multilevel lock acquisitions in cohort locks withers away as background noise in the presence of nontrivial work in the critical and noncritical sections. In principle, one can devise contrived scenarios (for example, where the critical and noncritical 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. Specifically, threads can try to directly acquire the top-level lock, avoiding the local lock. If successful they can enter and execute the critical section and, when done, release the top-level lock. If the top-level lock was contended, however, they can simply revert to normal cohort locking. Adaptive variations that track recent contention and either try the top-level lock directly or use normal cohort locking are also possible. We previously stated that a thread must hold both its local lock and the top-level lock in order to enter a critical section. That constraint can be relaxed, however, to allow this fast-path optimization where the thread holds just the top-level lock.

5.1.6. Abortable Lock Performance. Our abortable lock experiments in Figure 14 make an equally compelling case for cohort locks. Our cohort locks (A-C-BO-BO and A-C-BO-CLH) outperform a state-of-the-art abortable lock (A-CLH [Scott 2002]) and an abortable variant of HBO (called A-HBO in Figure 14, 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

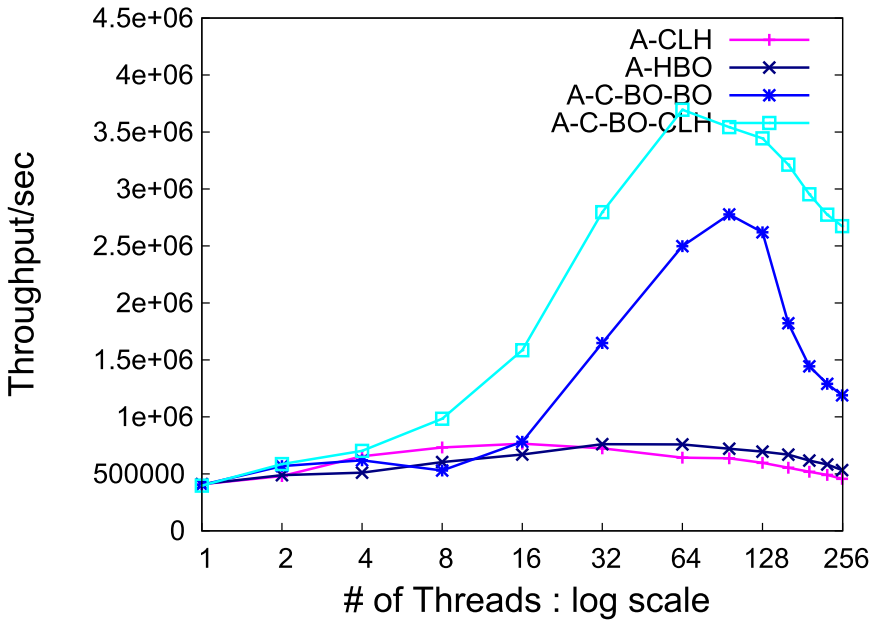

Fig. 14. Abortable lock average throughput in terms of number of critical and noncritical sections executed per second. The critical section accesses two distinct cache blocks (increments 4 integers counters on each block), and the noncritical section is an idle spin loop of up to 4 micro seconds.

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 node 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 for most workload settings.)

5.2. 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 multithreaded 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 used the mmicro benchmark [Dice and Garthwaite 2002] to test lock algorithms via the interpose library. In the benchmark, each thread repeatedly allocates a 64-byte block of memory, initializes it by writing to the first 4 words, 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 Figures 15 through 17, showing that cohort locks outperform all the other locks.

There are two reasons for this impressive scalability of cohort locks: First, they tend to more effectively batch requests coming from the same NUMA node, thus improving

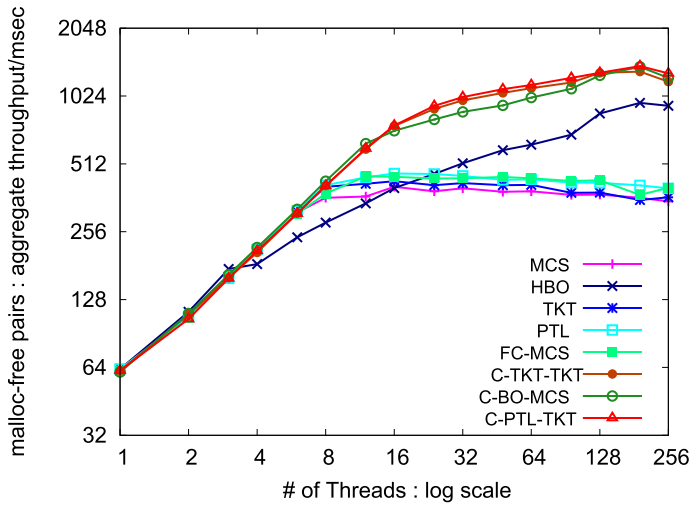

Fig. 15. T5440 : Scalability results of the malloc experiment (in terms of malloc-free pairs executed per millisecond).

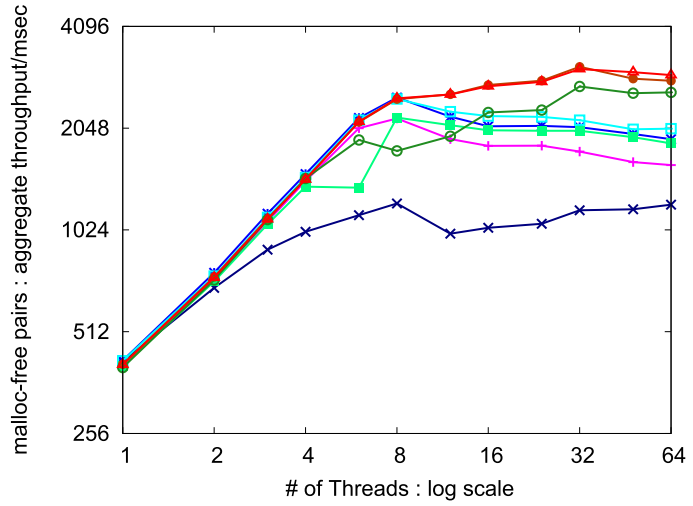

Fig. 16. T4-1 : Scalability results of the malloc experiment (in terms of malloc-free pairs executed per millisecond).

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* [Sleator and Tarjan 1985] of free memory blocks ordered by size. Allocation requests are serviced by a best-fit policy. The allocator also maintains segregated free lists, implemented as simple singly linked lists, for certain common small block sizes less than 40 bytes. Since mmicro requests 64 byte blocks, all the requests go to the splay tree. The allocator uses the free blocks themselves as the splay tree nodes. A newly inserted free block always goes to the root of the tree, and as a result, the most recently deallocated memory blocks tend to be reallocated more often in a last-in-first-out fashion. Thus a small number of blocks are continuously circulated between threads.

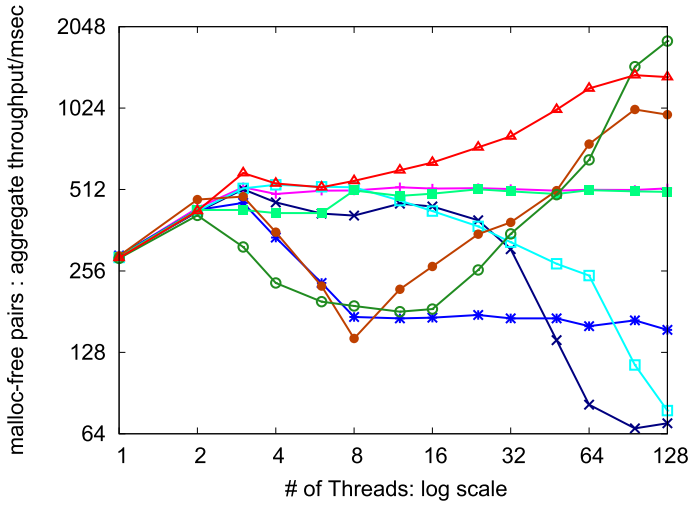

Fig. 17. X4800 : Scalability results of the malloc experiment (in terms of malloc-free pairs executed per millisecond).

Additionally, the allocated memory blocks are also updated by the benchmark. as would commonly be the case in most applications that allocate blocks. Because all the cohort locks create large batches of consecutive requests coming from the same NUMA node, they manage to recycle blocks in the same node for extended periods. Surprisingly, cohort locks conferred benefits for memory accesses to allocated blocks executed outside the allocator critical sections. In contrast, for all other locks, a block of memory migrates more frequently between NUMA nodes, 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.

The benchmark reports aggregate throughput at the end of a 10 second run, expressed as iterations of top-level loop executed by the worker threads. We ran 3 trials for each configuration and report their median result. The observed variance was extremely low.

TKT, C-TKT-TKT and C-BO-MCS exhibit a local minimum on the X4800 at 8 threads because of global spinning and a high arrival rate at the top-level lock. Since the X4800 has 8 nodes, each of the 8 threads will reside on a distinct node as the sole occupant. As the thread count increases past 8 and cohort formation starts to occur, the arrival rate at the top-level lock decreases and performance rebounds for C-TKT-TKT and C-BO-MCS. In contrast, global spinning is tolerated well on the T4-1 and T5440 and no such trough is evident in the respective graphs. PTL performs reasonably well up to 8 threads on the X4800 but then fades because spinning becomes nonprivate. Again, no such scalablity fade occurs on the T4-1 or T5440. (Similar patterns of behavior are evident in the LBench results in Figure 9). The performance of HBO continues to be unstable. We note that HBO performance is sensitive to its tunable parameters. In contrast, cohort locks introduce only one new tunable—the bound on local handoffs—and 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, and avoids the need for empirical tuning.

5.3. Memcached

Memcached [memcached.org 2013] 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 (we use v1.4.5 in our experiments), 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 stats_lock protects updates to statistical counters. Both of these locks are known to be contention bottlenecks [Gunther et al. 2011; Pohlack and Diestelhorst 2011; Vyas et al. 2013] which we believe makes memcached 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 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 [libmemcached.org 2013]. 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 SPARC system. 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 I 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, HBO, TKT, PTL, FC-MCS, and all the nonabortable cohort locks discussed in Section 3.

For read-heavy loads (Table I (a)), the performance of all the locks is identical, with all locks enabling over 5X scaling. For loads with moderately high set ratios (Table I (b)), we observe that all the spin locks significantly outperform pthread locks. Interestingly, TKT and PTL scale noticeably better than HBO and FC-MCS. Our cohort locks perform about 5% better than TKT and PTL, and are competitive with each other. For

Table I.

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.

#	pthread locks	MCS	HBO	TKT	PTL	FC-MCS	C-BO-MCS	C-TKT-TKT	C-PTL-TKT
1	1.00	1.21	1.06	1.22	1.06	0.99	0.99	1.05	1.00
4	3.99	3.89	3.69	3.83	3.82	3.76	3.81	3.81	3.79
8	5.34	5.42	5.23	5.41	5.36	5.38	5.39	5.36	5.40
16	5.48	5.58	5.43	5.49	5.54	5.52	5.57	5.53	5.51
32	5.40	5.59	5.42	5.50	5.50	5.51	5.47	5.52	5.51
64	5.22	5.53	5.38	5.45	5.46	5.43	5.43	5.47	5.47
96	5.25	5.50	5.37	5.44	5.43	5.44	5.42	5.41	5.45
128	5.23	5.51	5.35	5.42	5.40	5.44	5.42	5.37	5.42

(a) 90% gets and 10% sets

1	1.00	1.00	1.07	1.21	1.00	0.99	1.18	1.20	1.05
4	2.93	3.18	2.98	3.29	3.22	3.15	3.23	3.24	3.16
8	3.48	4.71	3.97	4.81	4.77	4.67	4.83	4.80	4.78
16	3.58	5.11	4.48	5.22	5.16	5.05	5.35	5.30	5.25
32	3.45	5.05	4.69	4.91	5.02	4.94	5.28	5.31	5.25
64	3.33	4.80	4.64	4.74	4.76	4.59	5.14	5.14	5.13
96	3.31	4.80	4.66	4.81	4.74	4.28	5.11	5.11	5.14
128	3.35	4.76	4.67	4.86	4.78	4.60	5.13	5.12	5.06

(b) 50% gets and 50% sets

1	1.00	1.01	1.02	1.20	1.06	0.98	1.04	1.17	1.04
4	2.38	2.71	2.49	2.68	2.84	2.59	2.76	2.61	2.66
8	2.53	3.61	3.17	3.62	3.56	3.47	3.92	3.77	3.73
16	2.61	3.85	3.68	3.98	3.93	3.77	4.55	4.54	4.50
32	2.49	3.82	4.03	3.92	3.88	3.58	4.64	4.72	4.68
64	2.41	3.62	4.04	3.71	3.63	3.42	4.42	4.49	4.42
96	2.42	3.60	3.86	3.63	3.65	3.45	4.21	4.19	4.44
128	2.42	3.58	4.07	3.71	3.66	3.41	4.42	4.49	4.37

(c) 10% gets and 90% sets

write-heavy loads (Table I (c)), our cohort locks clearly out-scale the best of all other locks, HBO, by up to 20%.

6. DISCUSSION

Generally, most locks are suitable for use as node-level local locks as long as they provide the cohort detection property. Performance of a cohort lock is surprisingly insensitive to the choice of the local lock type. For instance, a ticket lock is simple, fair and convenient, and contention—waiting threads—can easily be detected by checking if the *request* field differs from the *grant* field. Presumably, node-local communication costs are moderate, so global spinning is permissible for local locks. Depending on the coherence architecture, the number of caches that are polling a location, and thus need invalidation by a write that releases the lock, may influence performance. In the case of a local cohort lock, even one that uses global spinning, write invalidation caused by the stores that release the lock do not need to propagate beyond the local cache.

The top-level lock must operate over a wide range of arrival rates. When the number of contending threads T is less than N, where N is the number of NUMA nodes, the top-level lock may encounter higher arrival rates. But as T exceeds N and cohort formation occurs, the arrival rate will typically decrease but the hold time will increase.

The number of threads contending for the top-level lock at any one time is typically bounded by N by virtue of the local locks.

The selection of a lock type for the top-level lock is critical. We consider a number of common lock types and their applicability as top-level cohort locks.

(1) *MCS*. A key property of MCS is *private spinning* [Mellor-Crummey and Scott 1991], where each stalled thread spin-waits on its own MCS queue node, and is informed by its predecessor thread that it 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 queue node is then available for reuse or to be deallocated.
When used as a top-level cohort lock, however, we found that MCS queue nodes can migrate in an undesirable fashion. As the top-level lock must be thread-oblivious, one thread may acquire the MCS lock and some other thread might release the lock. This can result in MCS queue nodes migrating from one thread to another. Because of this, and because the *pthreads* interface by which we expose our lock implementations does not require lexically scoped locking, we can not allocate MCS queue nodes on thread stacks. We initially used thread-local free lists to cache available MCS queue nodes but discovered gross levels of interthread queue node migration with some workloads while using MCS as a top-level cohort lock. This resulted in a large number of queue nodes in circulation and stress on the memory allocator. Returning nodes to a global pool keeps the number of nodes in check, but trades one concurrency problem for another. We determined that we could cache one available MCS queue node in the lock structure and, when a thread installed a node, after acquiring the lock, it could steal a replacement node from that cache. Complementary code would try to reprovision the cache when releasing the lock. The cache field—a pointer to a queue node—was protected by the top-level lock itself. While this approach sufficed to avoid rampant allocation and circulation, it artificially lengthened the critical section and caused more accesses to shared metadata, increasing cache-coherent communication traffic. Another work-around was to simply embed N MCS queue nodes in the cohort lock instance, one assigned to each NUMA node, and have each thread contending for the top-level lock use its designated embedded MCS node. Acquiring the local node-level lock confers ownership of the embedded MCS node, which in turn can be used in conjunction with the top-level MCS lock. While workable, this increases the footprint and complexity of the lock.

(2) *Ticket locks*. Ticket locks are simple and provide strict FIFO ordering. A classic ticket lock consists of *request* and *grant* fields. Arriving threads use an atomic fetch-and-add instruction to advance *request*, and then busy-wait until the *grant* field matches the value returned by the fetch-and-add operation. To release a lock, the owner simply increments the *grant* field. Platforms that have a true fetch-and-add instruction may enjoy performance benefits over those where fetch-and-add is emulated with CAS. And on platforms so enabled, concurrent algorithms that can make use of fetch-and-add may have a performance advantage over those that use CAS [Dice 2011a].
Unlike MCS, ticket locks use so-called global spinning, where all waiting threads spin (busy-wait) polling the *grant* field. We observed that platforms such as SPARC that use MOESI [Sweazey and Smith 1986] cache states for intersocket coherence are tolerant of global spinning;[9] there is little advantage to be had by using private spinning instead of global spinning. On modern Intel processors that use

[9]MOESI allows direct cache-to-cache transfers of dirty lines without the need for write-back to memory.

MESI [Papamarcos and Patel 1984] or MESIF [Goodman and Hum 2009] cache coherence states, however, we found that private spinning provides better performance than did global spinning. Modern AMD processors employ MOESI [AMD 2012; Conway et al. 2010; David et al. 2013] and, like SPARC processors, better tolerate global spinning.

On Intel MESI or MESIF systems, ticket locks can be enhanced with a delay in the busy-wait loop that is proportional to the difference between the thread's ticket in hand—the value returned by the fetch-and-add operation on *request*—and the last observed value of the *grant* field [Boyd-Wickizer et al. 2012]. This approach can provide some performance relief, but the delay factors are both platform- and application-specific. If the delay is too short, then we introduce futile checks of the *grant* field that can induce coherence traffic. If the delay is too long then we retard lock response time and hand-over latency, effectively increasing the critical section length. In our experiments we also found that proportional delay on MOESI-based systems either had no impact or negative impact on performance. And as noted above, the top-level cohort lock is expected to perform over a wide variety of hold-times and arrival rates. Broadly, our experience suggests that ticket locks without proportional delay are acceptable as top-level locks for MOESI systems, but they are not recommended for MESI-based systems.

(3) *Backoff locks*. Backoff test-and-test-and-set locks with randomized truncated binary exponential backoff are inherently unfair. While they might yield excellent aggregate throughput, it's not uncommon for a small number of threads or nodes to monopolize or dominate a lock for long periods, excluding other threads. This works to reduce lock migration and improve overall throughput, but can lead to starvation of some threads. Backoff locks where the delay increases over time are inherently unfair as threads that have been waiting longer are less likely to acquire the lock in unit time than are more recently arrived threads. Such a delay policy is anti-FIFO. Furthermore, backoff duration represents a trade-off between potentially futile coherence traffic arising from polling and lock hand-over efficiency. Long backoff periods can mitigate coherence costs, but also retard lock hand-over latency and responsiveness if the lock is released when the waiting threads happen to be using long backoff periods. The spinning threads thus take some time to notice that the lock was released before they attempt an atomic operation to acquire the lock. This "dead time" reduces scalability.

    These locks also use global spinning and are exposed to the MESI/MOESI issues described above. Tuning such locks results in parameters that are application- and platform-specific, as well as specific to the expected concurrency level. Such performance sensitivity to tunables makes it difficult to construct backoff locks that yield good performance over a wide range of loads and platforms. The degree of fairness achieved by backoff locks is also influenced by hardware fairness in arbitrating cache coherence states. These same concerns apply to HBO.

(4) *Partitioned Ticket Locks*. In partitioned ticket locks, threads arriving to acquire the lock atomically increment a central *request* field, but waiting threads use different *grant* "slots". Like ticket locks, they are fair and provide a FIFO/FCFS admission schedule and they also enjoy the benefits of fetch-and-add. When the number of contending threads is less then the number of slots they provide purely private spinning. Unlike MCS locks, there are no queue nodes to allocate or manage. Only one atomic operation is required to acquire, and none are needed to release the lock. In contrast to MCS locks, where the *tail* field might be accessed both by threads acquiring the lock and threads releasing the lock, under partitioned ticket locks threads increment one location to acquire the lock but polling operations and the updates to release the lock are decentralized and dispersed over the set of slots, so

acquiring and releasing threads tend not to generate as much mutual coherence interference.
When used as the top-level cohort lock, and when configured to have at least as many slots as there are NUMA nodes, partitioned ticket locks provide purely private spinning. Specifically, the node-level local locks moderate admission to the top-level partitioned ticket lock and ensure that there are no more than N threads contenting for the top-level partitioned ticket lock. As such, there is only private spinning in the top-level partitioned ticket lock, so the lock works well for both MESI- and MOESI-based platforms.

Our experience suggests that the top-level lock should provide private spinning in order to support both MESI-based and MOESI-based platforms. In particular, partitioned ticket locks are suitable as top-level cohort locks, avoiding many of the issues inherent in the other lock types.

When coupled with ticket locks used at the node-level, for instance, we have a cohort lock where the only tunable parameter is the bound on the number of local handoffs. This parameter directly reflects the trade-off between fairness and throughput improvement achieved from cohort formation. It is not platform-specific. Coupling a partitioned ticket lock at the top-level and classic ticket locks for node-level locks also demonstrates that a NUMA-friendly lock can be easily constructed from NUMA-oblivious underlying lock types.

Conceivably, the scalability benefits exhibited by cohort locks could arise in part from dispersing contention over a set of local locks and hence by reducing contention on the top-level lock. To demonstrate this was not the case we experimented with a variety of policies for assigning threads to local locks, including: the classic cohort policy, by NUMA node; randomized assignment; randomized assignment where we ensured the number of threads associated with each local lock was balanced and equitable; and intentionally pathological NUMA-unfriendly policies where each local lock was associated with threads residing on a wide range of NUMA nodes. The cohort lock always outperformed these experimental variations by a large margin. We also experimented with strategies that split a given physical node into two cohort nodes, although this variant delayed cohort formation by artificially increasing the number of nodes.

7. 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 [David et al. 2013; Eyerman and Eeckhout 2010; Johnson et al. 2010; Scott 2013]. This article tackles the most basic of the multicore synchronization algorithms, the lock, presenting a simple new lock design approach—*lock cohorting*—fit for NUMA machines. NUMA-aware cohort locks are easily constructed from existing NUMA-oblivous lock types. Our design encourages and preserves cache residency between critical section invocations, and acts to conserve interconnect bandwidth, which is proving to be a key resource in modern systems. The wide range of cohort locks we presented in the article, 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.

As our cohort locks retain standard programming interfaces, the performance of otherwise NUMA-oblivious applications can be improved by simply substituting our lock implementations.

Cohort locks have already proved useful in the construction of reader-writer locks, where write requests are moderated by a cohort lock [Calciu et al. 2013b].

In the future we hope to explore 3-level cohort locks having a top-level lock, node-level locks, and core-level locks. Such a mechanism may prove profitable on multichip systems that use the T4 processor, for instance, Currently, cohort lock implementations distinguish only between local and remote nodes. We hope to explore strategies that take interconnect distance metrics, such as hop count or proximity, into account for succession policies in the top-level lock. This approach might be used to reduce total lock migration *distance* in a time interval. Relatedly, it may be useful to treat a group of "close" NUMA nodes as a single synthetic node for the purposes of cohort locking: multiple physical NUMA nodes are aggregated into a single logical node. Having fewer but larger cohort nodes promotes cohort formation but at the cost of increased communication traffic. Zhang et al. [2014] recently explored similar ideas.

Early experiments suggest that explicitly placing local lock instances on pages residing on their respective home NUMA nodes may provide improved performance through local spinning. We are exploring this potential optimization.

We have assumed the main mode of benefit conferred by cohort locks arises from reduced internode "sloshing" of the cache lines underlying mutable lock metadata and mutable shared data accessed within critical sections. In the future we hope to explore alternative modes of benefit. Say, for instance, we have critical sections that access a large working set but write relatively few locations. Assuming the existence of unrelated threads that compete for cache residency and continuously decay that critical section working set, batching may provide benefit by virtue of increased cache residency relative to admission orders such as FIFO. The first thread in a batch may suffer significant misses as it loads the working set, but subsequent threads in that same batch may suffer fewer misses.

During our exploration we encountered a contrived workload on the T4-1 where cohort locks did not perform as well as simple NUMA-oblivious locks such as MCS. Upon investigation we determined that the cohort admission schedule tended to "pack" the critical section and multiple concurrent noncritical section executions on just the currently favored core while threads on other cores were more likely to be waiting for the lock. This consequently inhibited pipeline fusion on the T4-1 platform [Dice 2011c] for the favored core and caused the logical processors, which were running the critical and noncritical sections, on the favored core to run more slowly than would be the case under noncohort locks, where the critical noncritical executions might be more likely dispersed over the set of available cores. (The T4-1 has 2 pipelines per core. If only one thread is running on a core then both of the pipelines will automatically be fused to accelerate the execution of that thread). By concentrating and localizing execution on just one core at a given time by way of cohort formation, we induced contention for the resources local to that core. Specifically, the benefit arising from cohort locking and reduced intercore coherence traffic did not overcome the slowdown caused by increased intracore competition for available pipeline cycles. By "packing" execution, we inhibited pipeline fusion and in turn caused the critical section duration to increase as the thread executing the critical section had to compete for shared pipeline resources. The T4 is equipped with a rich variety of hardware performance counters that allow us to detect when a core is oversubscribed and threads are stalled, waiting for pipelines. We hope to use this information to inform the cohort lock mechanism so it can respond to the phenomena described above.

The same concern applies to Intel processors that support *turbo mode* [Dice et al. 2012b]. More generally, we observe that cohort locks may reduce coherence traffic, but at the same time cohort formation and the short-term "packing" of execution on cores or nodes may lead to excessive competition for shared core- or node-level compute resources such as pipelines, DRAM channels, and cache occupancy. The temporal and spatial clustering of critical section activations afforded by the cohort lock admis-

sion schedule acts to reduce coherent communications costs for shared data accessed under the lock in critical sections. But that same policy also increases the odds that noncritical sections will run concurrently on the favored node or core, causing contention for local compute resources. Programs using cohort locks, for example, may be vulnerable to destructive cache interference as a relatively large number of nonwaiting threads may be executing concurrently on the favored core or node, thus increasing cache pressure in the shared per-node or per-core cache. The response of a given application under cohort locks depends on the ratio of computation to communication, but recall that cohort formation only occurs when the lock is contended, so communication is already known to be present. Other NUMA-aware lock algorithms such as HCLH and HBO are similarly vulnerable to these effects. We plan on more deeply exploring these tensions and trade-offs in future work, as well investigating how cohort locks interact with *computational sprinting* [Raghavan et al. 2012]. Ideally, if a particular core was transiently sprinting, a lock algorithm might be enhanced to bias the admission schedule so as favor that core, improving scalability as the critical section would execute more quickly.

REFERENCES

A. Agarwal and M. Cheritan. 1989. Adaptive backoff synchronization techniques. *SIGARCH Comput. Archit. News 17*, 3, 396–406. DOI:http://dx.doi.org/10.1145/74926.74970.

AMD. 2012. *AMD64 Architecture Programmer's Manual: Vol. 2 System Programming*. http://support.amd.com/us/Embedded_TechDocs/24593.pdf.

T. E. Anderson. 1990. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst. 1*, 1, 6–16. DOI:http://dx.doi.org/10.1109/71.80120.

Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2012. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*.

Irina Calciu, Dave Dice, Tim Harris, Maurice Herlihy, Alex Kogan, Virendra J. Marathe, and Mark Moir. 2013a. Message passing or shared memory: Evaluating the delegation abstraction for multicores. In *Proceedings of the 17th International Conference on Principles of Distributed Systems*. Roberto Baldoni, Nicolas Nisse, and Maarten van Steen, Eds., Lecture Notes in Computer Science, vol. 8304, Springer, 83–97. http://dx.doi.org/10.1007/978-3-319-03850-6_7.

Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. 2013b. NUMA-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13)*. ACM, New York, 157–166. DOI:http://dx.doi.org/10.1145/2442516.2442532.

Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. 2010. Cache hierarchy and memory subsystem of the AMD Opteron processor. *IEEE Micro 30*, 2, 16–29. DOI:http://dx.doi.org/10.1109/MM.2010.31.

Travis Craig. 1993. Building FIFO and priority-queueing spin locks from atomic swap. Tech. Rep. TR 93-02-02. Department of Computer Science, University of Washington.

Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, New York, 33–48. DOI:http://dx.doi.org/10.1145/2517349.2522714.

David Dice. 2003. US Patent # 07318128: Wakeup affinity and locality. http://patft.uspto.gov/netacgi/nph-Parser?patentnumber=7318128.

David Dice. 2011a. Atomic fetch and add vs CAS. (2011). https://blogs.oracle.com/dave/entry/atomic_fetch_and_add_vs.

David Dice. 2011b. Brief announcement: a partitioned ticket lock. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'11)*. ACM, New York, 309–310. DOI:http://dx.doi.org/10.1145/1989493.1989543.

David Dice. 2011c. Polite busy-waiting with WRPAUSE on SPARC. https://blogs.oracle.com/dave/entry/polite_busy_waiting_with_wrpause.

David Dice. 2011d. Solaris Scheduling: SPARC and CPUIDs. (2011). https://blogs.oracle.com/dave/entry/solaris_scheduling_and_cpuids.

David Dice and Alex Garthwaite. 2002. Mostly lock-free malloc. In *Proceedings of the 3rd International Symposium on Memory Management (ISMM'02)*. ACM, New York, 163–174. DOI:http://dx.doi.org/10.1145/512429.512451.

David Dice, Virendra J. Marathe, and Nir Shavit. 2011. Flat-combining NUMA Locks. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'11)*. ACM, New York, 65–74. DOI:http://dx.doi.org/10.1145/1989493.1989502.

David Dice, Virendra J. Marathe, and Nir Shavit. 2012a. Lock cohorting: a general technique for designing NUMA locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*. ACM, New York, 247–256. DOI:http://dx.doi.org/10.1145/2145816.2145848.

David Dice, Nir Shavit, and Virendra J. Marathe. 2012b. US Patent Application 20130047011 - Turbo Enablement. http://www.google.com/patents/US20130047011.

David Dice, Nir Shavit, and Virendra J. Marathe. 2012c. US Patent US8694706 - Lock Cohorting. (2012). http://www.google.com/patents/US8694706.

Stijn Eyerman and Lieven Eeckhout. 2010. Modeling critical sections in Amdahl's law and its implications for multicore design. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, 362–370. DOI:http://dx.doi.org/10.1145/1815961.1816011.

Panagiota Fatourou and Nikolaos D. Kallimanis. 2012. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*. ACM, New York, 257–266. DOI:http://dx.doi.org/10.1145/2145816.2145849.

Nitin Garg, Ed Zhu, and Fabiano C. Botelho. 2011. Light-weight locks. *CoRR* abs/1109.2638 (2011). http://arxiv.org/abs/1109.2638.

J. R. Goodman and H. H. J. Hum. 2009. MESIF: A two-hop cache coherency protocol for point-to-point interconnects. https://researchspace.auckland.ac.nz/bitstream/handle/2292/11594/MESIF-2009.pdf.

Neil J. Gunther, Shanti Subramanyam, and Stefan Parvu. 2011. A methodology for optimizing multithreaded system scalability on multi-cores. *CoRR* abs/1105.4301 (2011).

Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*. 355–364. DOI:http://dx.doi.org/10.1145/1810479.1810540.

Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann.

Intel Corporation. 2009. An introduction to the Intel QuickPath Interconnect. http://www.intel.com/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf. Document Number: 320412-001US.

F. Ryan Johnson, Radu Stoica, Anastasia Ailamaki, and Todd C. Mowry. 2010. Decoupling contention management from scheduling. In *Proceedings of the 15th Conference on Architectural Support for Programming Languages and Operating System*. ACM, New York, 117–128. DOI:http://dx.doi.org/10.1145/1736020.1736035.

N. D. Kallimanis. 2013. Highly-Efficient synchronization techniques in shared-memory distributed systems. http://www.cs.uoi.gr/tech_reports//publications/PD-2013-2.pdf.

David Klaftenegger, Konstantinos Sagonas, and Kjell Winblad. 2014. Queue Delegation Locking. (2014). http://www.it.uu.se/research/group/languages/software/qd_lock_lib.

libmemcached.org. 2013. libmemcached. www.libmemcached.org.

Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. 2012. Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'12)*. USENIX Association, Berkeley, CA, 6–6. http://dl.acm.org/citation.cfm?id=2342821.2342827.

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

P. Magnussen, A. Landin, and E. Hagersten. 1994. Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing*. 165–171.

John Mellor-Crummey and Michael L. Scott. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Computer Syst. 9*, 1, 21–65.

memcached.org. 2013. memcached – a distributed memory object caching system. *www.memcached.org*. (2013).

Avi Mendelson and Freddy Gabbay. 2001. The effect of seance communication on multiprocessing systems. *ACM Trans. Comput. Syst. 19*, 2, 252–281. DOI:http://dx.doi.org/10.1145/377769.377780.

Oracle Corporation. 2010. Oracle's Sun Fire X4800 server architecture. http://www.oracle.com/technetwork/articles/systems-hardware-architecture/sf4800g5-architecture-163848.pdf.

Oracle Corporation. 2012. Oracle's SPARC T4-1, SPARC T4-2, SPARC T4-4, and SPARC T4-1B server architecture. http://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/o11-090-sparc-t4-arch-496245.pdf.

Y. Oyama, K. Taura, and A. Yonezawa. 1999. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of the International Workshop on Parallel and Distributed Computing For Symbolic And Irregular Applications (PDSIA'99)*. World Scientific, 182–204.

Mark S. Papamarcos and Janak H. Patel. 1984. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture (ISCA'84)*. ACM, New York, 348–354. DOI:http://dx.doi.org/10.1145/800015.808204.

Martin Pohlack and Stephan Diestelhorst. 2011. From lightweight hardware transactional memory to lightweight lock elision. In *Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing*.

Zoran Radović and Erik Hagersten. 2003. Hierarchical backoff locks for nonuniform communication architectures. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*. 241–252.

Arun Raghavan, Yixin Luo, Anuj Chandawalla, Marios Papaefthymiou, Kevin P. Pipe, Thomas F. Wenisch, and Milo M. K. Martin. 2012. Computational sprinting. In *Proceedings of the IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA'12)*. IEEE, 1–12. DOI:http://dx.doi.org/10.1109/HPCA.2012.6169031.

Michael L. Scott. 2002. Non-blocking timeout in scalable queue-based spin locks. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing (PODC'02)*. ACM, New York, 31–40. DOI:http://dx.doi.org/10.1145/571825.571830.

Michael L. Scott. 2013. Shared-memory synchronization. *Synthesis Lectures Comput. Architec. 8*, 2, 1–221. DOI:http://dx.doi.org/10.2200/S00499ED1V01Y201304CAC023.

Michael L. Scott and William Scherer. 2001. Scalable queue-based spin locks with timeout. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*. 44–52.

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM 53*, 7, 89–97. DOI:http://dx.doi.org/10.1145/1785414.1785443.

Daniel Dominic Sleator and Robert Endre Tarjan. 1985. Self-adjusting binary search trees. *J. ACM 32*, 3, 652–686. DOI:http://dx.doi.org/10.1145/3828.3835.

M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. 2009. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 253–264. DOI:http://dx.doi.org/10.1145/1508244.1508274.

P. Sweazey and A. J. Smith. 1986. A class of compatible cache consistency protocols and their support by the IEEE futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture (ISCA'86)*. IEEE, 414–423. http://dl.acm.org/citation.cfm?id=17407.17404.

Trilok Vyas, Yujie Liu, and Michael Spear. 2013. Transactionalizing legacy code: An experience report using GCC and memcached. In *Proceedings of the 8th ACM SIGPLAN Workshop on Transactional Computing*.

Wikipedia. 2014a. Closure (computer programming). http://en.wikipedia.org/wiki/Closure_(computer_programming).

Wikipedia. 2014b. Futures and promises. http://en.wikipedia.org/wiki/Futures_and_promises.

Benlong Zhang, Junbin Kang, Tianyu Wo, Yuda Wang, and Renyu Yang. 2014. A flexible and scalable affinity lock for the kernel. In *Proceedings of the 16th IEEE International Conference on High Performance Computing and Communications (HPCC'14)*.

Received April 2013; revised July 2014; accepted September 2014