NUMA-Aware Reader-Writer Locks

Irina Calciu
Brown University
irina@cs.brown.edu

Dave Dice
Oracle Labs
dave.dice@oracle.com

Yossi Lev
Oracle Labs
yossi.lev@oracle.com

Victor Luchangco
Oracle Labs
victor.luchangco@oracle.com

Virendra J. Marathe
Oracle Labs
virendra.marathe@oracle.com

Nir Shavit
MIT
shanir@csail.mit.edu

Abstract

Non-Uniform Memory Access (NUMA) architectures are gaining importance in mainstream computing systems due to the rapid growth of multi-core multi-chip machines. Extracting the best possible performance from these new machines will require us to revisit the design of the concurrent algorithms and synchronization primitives which form the building blocks of many of today's applications. This paper revisits one such critical synchronization primitive – the reader-writer lock.

We present what is, to the best of our knowledge, the first family of reader-writer lock algorithms tailored to NUMA architectures. We present several variations which trade fairness between readers and writers for higher concurrency among readers and better back-to-back batching of writers from the same NUMA node. Our algorithms leverage the *lock cohorting* technique to manage synchronization between writers in a NUMA-friendly fashion, binary flags to coordinate readers and writers, and simple distributed reader counter implementations to enable NUMA-friendly concurrency among readers. The end result is a collection of surprisingly simple NUMA-aware algorithms that outperform the state-of-the-art reader-writer locks by up to a factor of 10 in our microbenchmark experiments. To evaluate our algorithms in a realistic setting we also present performance results of the kccachetest benchmark of the *Kyoto-Cabinet* distribution, an open-source database which makes heavy use of pthread reader-writer locks. Our locks boost the performance of kccachetest by up to 40% over the best prior alternatives.

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

General Terms Algorithms, Design, Performance

Keywords NUMA, hierarchical locks, mutual exclusion, reader-writer locks

1. Introduction

As microprocessor vendors aggressively pursue the production of bigger multi-core multi-chip 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 local cache and multiple processing cores. 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 inter-node links normally suffer from higher latency and lower bandwidth compared to the intra-node channels. To decrease latency and to conserve interconnect bandwidth, NUMA-aware policies encourage intra-node communication over inter-node 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 inter-node coherence traffic and from interconnect bandwidth limits. Furthermore, inter-node 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. One key synchronization construct is the reader-writer (RW) lock.

A RW lock relaxes the central property of traditional mutual exclusion (mutex) locks by allowing multiple threads to hold the lock simultaneously in *read mode*. A thread may also acquire the lock in *write mode* for exclusive access. RW locks are used in a wide range of settings including operating system kernels, databases, high-end scientific computing applications and software transactional memory implementations [6].

RW locks have been studied extensively for several decades [1, 2, 11, 13–16], with proposals ranging from simple counter- or semaphore-based solutions [2], to solutions leveraging centralized wait-queues [14, 16], to solutions that use more sophisticated data structures such as Scalable Non-Zero Indicators (SNZI) [15]. Of these, all but the SNZI-based solutions rely on centralized structures to coordinate threads, and thus encounter scalability impediments [15]. The SNZI-based algorithms keep track of readers – threads acquiring the RW lock in read mode – with each reader arriving at a leaf in the "SNZI tree". Readers can be made NUMA-aware by partitioning the leaves of the SNZI-tree among the NUMA nodes, with threads arriving at SNZI leaves associated with their node. Writers, however, remain NUMA-*oblivious*, which can impair scalability.

PPoPP'13, February 23–27,2013,Shenzhen, China.

[1] We use the term NUMA broadly to include Non-Uniform Communication Architecture (NUCA) [17] machines as well.

Hsieh and Weihl [11] and Vyukov [20] independently suggested a simple *distributed*[2] approach to building scalable RW locks. Each distributed RW lock contains N RW locks where N is the number of processors in the system. Each reader is mapped to a single RW lock, and must acquire that lock in read mode in order to execute its critical section. A writer must acquire *all* the underlying RW locks in write mode to execute its critical section. Deadlocks between writers are avoided by forcing a specific locking order. The approach can be made NUMA-aware by restricting N to the number of NUMA nodes in the system, and mapping each reader to the lock dedicated to its node. This variant algorithm which we call DV (representing the initials of Vyukov), is partially NUMA-aware, just like the SNZI-based RW locks. Absent any writers, readers on different nodes can obtain and release read permission without generating any inter-node write coherence traffic. However, every writer incurs the overhead of acquiring write permission for the RW lock of every node, potentially generating significant coherence traffic. Thus, the performance of DV plummets with increased writer activity. Also, because of the canonical locking order used to avoid deadlock, readers on nodes that appear late in the order may enjoy an unfair performance advantage over readers running on nodes that appear earlier.

In this paper we present a novel family of RW locks that are designed to leverage NUMA features and deliver better performance and scalability than any prior RW lock algorithm. We take a three-pronged approach in our lock designs. First, similar to DV, we maintain a distributed structure for the readers metadata such that readers denote their intent by updating only locations associated with their node. By localizing updates to read indicators we reduce coherence traffic on the interconnect. Second, writers preferentially hand off access permission to blocked writers on the same node, enhancing reference locality in the node's cache for both the lock metadata and data accessed in the critical section it protects. Finally, our algorithms maintain tight execution paths for both readers and writers, reducing latency of the lock acquisition and release operations.

Our RW lock algorithms build on the recently developed *lock cohorting* technique [7], which allows for the construction of NUMA-aware mutual exclusion locks. Briefly, writers use a cohort lock to synchronize with each other and to maintain writer-vs-writer exclusion. Using the cohort locking approach, a writer releasing the lock generally prefers to transfer ownership to a pending local writer (if there is one), thus reducing *lock migrations*[3] between nodes.

Our RW locks also contain distributed implementations of *read indicators*, a data structure that tracks the existence of readers [15]. Readers "arrive" at these read indicators during lock acquisition and "depart" from them during lock release. Writers query the read indicators to detect concurrently active readers. Because of the distributed nature of our read indicators, the readers need to access just the node-specific metadata of the lock. We additionally use simple flags and checks for coordination between readers and writers. The result is a family of surprisingly simple algorithms that push the performance envelope of RW locks on NUMA systems far beyond the prior state-of-the-art algorithms.

Our various RW locks can be differentiated on the basis of the fairness properties they provide as recognized by Courtois et al. [2]. In particular, we present locks exhibiting different "preference" policies: reader-preference, writer-preference, and neutral-preference. The reader-preference policy dictates that readers should acquire (be granted) the lock as early as possible, regardless of arrival order, whereas the writer-preference policy has a symmetric bias towards writers. More concretely, these preference policies allow readers or writers to "bypass" prior pending writers or readers (respectively) in the race to acquire the lock. The preference policies—except for the neutral policy—may lead to starvation of threads engaged in the non-preferred lock acquisition operation. We avoid such situations by allowing the lock mechanism to temporarily override the preference policy so as to allow forward progress of starving threads. Starving threads become "impatient" and transiently change the preference policy.

We present an empirical evaluation of our RW locks, comparing them with each other and with prior RW lock implementations. Our evaluation, conducted on a 256-way 4-node Oracle SPARC T5440^{TM} server, shows that our locks significantly outperform all prior RW locks on a diverse set of workloads. In our microbenchmark experiments, our locks outperform the prior best RW lock (the SNZI-based *ROLL* lock [15]) by up to a factor of 10.

We discuss our RW lock design approach in Section 2. In Section 3, we present our lock algorithms in detail. We present our empirical evaluation in Section 4, and conclude in Section 5.

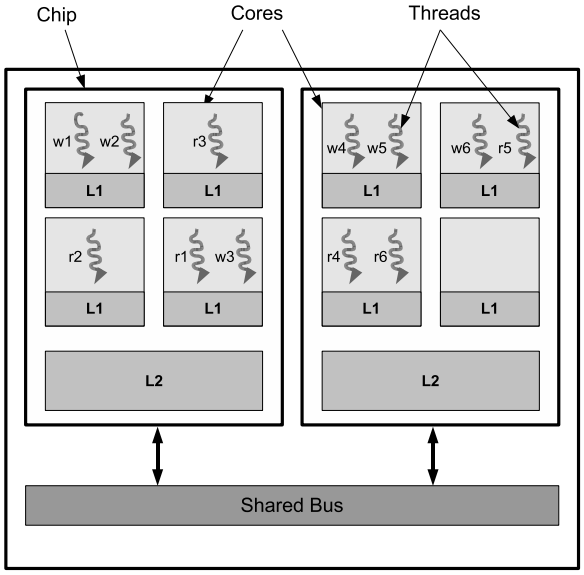


Figure 1. An example multi-core multi-chip NUMA system containing 2 chips with 4 cores per chip. Each chip is a NUMA node. Each core can have multiple hardware thread contexts (not shown in the figure). Each core has its individual $L1$ cache, and all cores on a chip share an $L2$ cache. Inter-thread communication via local caches ($L1$ and $L2$) is significantly faster than via remote caches because the latter involve coherence messages across the interconnect. In the figure, threads $r1..r6$ intend to acquire a RW lock in *read* mode, and threads $w1..w6$ intend to acquire the same lock in *write* mode.

2. Lock Design Rationale

NUMA-aware mutex locks have been explored in depth [3, 7, 17, 19]. However, to the best of our knowledge, there has been no prior effort toward constructing NUMA-aware RW locks. NUMA-aware mutex lock designs pursue only one goal – reduction of the lock

[2] The term "distributed" was coined by Vyukov for his algorithm [20], but this algorithm appears to be the same as Hsieh and Weihl's "static algorithm" [11]

[3] We say that *lock migration* occurs when the lock is consecutively acquired by threads residing on distinct NUMA nodes. On a cache-coherent NUMA system, 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 that of the second thread.

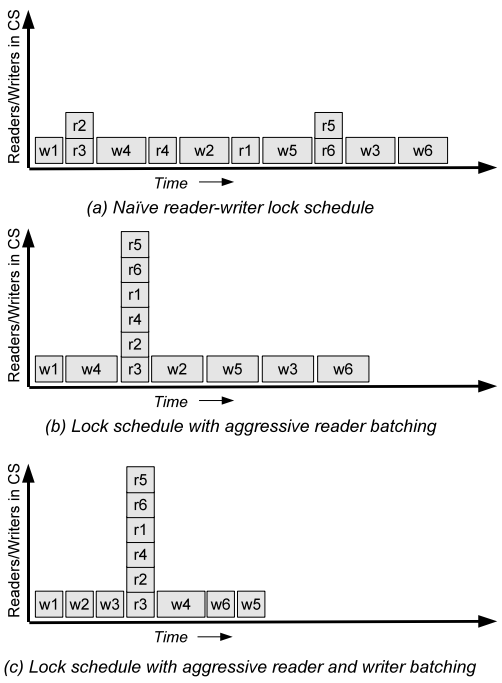


Figure 2. Execution scenarios depicting possible locking schedules for Figure 1

migration frequency so as to generate better node-local locality of reference for the lock and the critical section it protects. NUMA-aware mutex locks act to reduce the rate of write invalidations and coherence misses satisfied by cache-to-cache transfers from remote caches via the interconnect. We believe that, just like general RW locks, NUMA-aware RW lock designs must additionally consider the complementary goal of maximizing reader-reader concurrency.

We observe an interesting tension between these two goals: promoting concurrent reader-reader sharing across NUMA nodes tends to lead to designs that "spread" the lock metadata and critical section data across these nodes, whereas reducing the lock migration rate tends to significantly curtail this spread. However, this apparent contradiction between our goals can be effectively reconciled by using a policy that tries to reduce lock migrations only between writers while at the same time maximizing concurrency between readers. For this strategy to be most effective, we must aggressively "batch" the concurrent writer locking requests coming from a single NUMA node and maintain a high local writer-to-writer lock hand off rate. We note that this aggressive writer batching approach is not completely out-of-place. To the contrary, it nicely complements the goal of maximizing reader-reader concurrency because the latter can benefit significantly by aggressively aggregating (co-scheduling) reader locking requests. We illustrate the potential benefits of these design goals using an example.

Figure 1 depicts a NUMA system with six threads attempting to acquire a lock L in read mode, and six threads attempting to acquire L in write mode. We assume that the critical sections protected by the lock access the same data. Figure 2 shows possible critical section execution schedules for these readers and writers when the critical section is protected by different kinds of RW locks. Figure 2(a) shows a possible critical section execution schedule arbitrated by a naive RW lock that does not aggressively aggregate readers or provide back-to-back consecutive batching of writers from a given NUMA node. The schedule shows that the lock does not provide good reader-reader concurrency, and hence it takes more time to execute all the critical sections. Assuming a backlog of pending readers, higher rates of alternation between read and write modes yields lower levels of reader-reader concurrency. Figure 2(b) shows a scheduling policy that yields improved reader-reader concurrency. By aggressively aggregating read requests the lock successfully co-scheduled a large group of readers, allowing them to execute the critical section concurrently. However, the order of writers alternates between the two NUMA nodes from Figure 1. This leads to significant coherence traffic that slows down the writers. The width of the boxes reflects the relative time taken to complete a critical section invocation, with broader boxes showing the overheads associated with inter-node communication latencies. Figure 2(c) addresses this problem by batching together writers from the same NUMA node in a consecutive back-to-back fashion. As a result, writers $w2$, $w3$, $w5$, and $w6$ will incur fewer coherence misses during the execution of their critical sections. As we shall see in Section 4, these savings translate to significant performance gains for our locks.

3. RW Lock Algorithms

We use *lock cohorting* [7] as a starting point for our RW lock designs. Each of our RW lock instances contains a single central cohort mutual exclusion lock that is used to synchronize writers – we resolve writer-vs-writer conflicts via the cohort lock. Writers must first acquire this cohort lock in order to gain exclusive (write) ownership of the RW lock. Before executing the critical section, the writer owning the cohort lock must also reconcile reader-vs-writer conflicts by ensuring that there are no concurrent readers executing or about to execute their respective critical sections. The readers counterpart of our RW locks use distributed *read indicators* (ReadIndr for short). To acquire a RW lock in read mode, a reader must *arrive* at the lock's ReadIndr. ReadIndr is implemented as a distributed counter, with a counter per NUMA node. Each reader increments its local counter during arrival and decrements the local counter during *depart*. Crucially, writers update the central lock but only query and do not update the multiple reader indicators.

In this section we describe the cohort lock used to provide write-write exclusion and then present three RW lock algorithms, each of which implements one of the three preference policies: neutral-, reader- or writer-preference. We first present these algorithms at a high level (Sections 3.2 through 3.4). We then make an important observation that implementors can substitute almost any type of mutex lock and reader indicator mechanism into their implementation of our RW locks. Finally, we describe the scalable read indicator implementations used in our RW locks.

3.1 The Writer Cohort Lock

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. 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 is 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.

To acquire the cohort lock, a thread must first acquire ownership of the 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 hand off, the owner also implicitly passes ownership of the top-

level lock to that same successor. 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. (We used a bound of 64 in all our experiments described in this paper). Our algorithm intentionally trades strict short-term FIFO/FCFS fairness for improved aggregate throughput. 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.

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 hand off, 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 transfered to the local cache from the remote cache via the interconnect and downgraded to *shared* state in the remote cache. 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 transfered to the writer's cache. Read-read is the only form of sharing that does not require coherence communication. 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.

For use in our RW locks, we have developed a new cohort lock that uses classic ticket locks [12] for the NUMA node-local locks and a partitioned ticket lock [5] for the top-level lock. We call this lock C-PTL-TKT, short for *Partitioned-Ticket-Ticket* cohort lock. We expose a new isLocked interface that allows readers to determine if the write lock is held. This function is implemented by comparing the *request* and *grant* indices of the top-level partitioned ticket lock. We elected to use C-PTL-TKT in our RW locks as it is competitive with the best of the cohort locks, avoids the node management overheads inherent in classic queue-based locks such as MCS but still provides local spinning. The top-level and node-level locks are FIFO although the resultant C-PTL-TKT lock is not itself necessarily FIFO.

3.2 The Neutral-Preference Lock

Our neutral-preference lock, called C-RW-NP (Cohort; Read-Write; Neutral-Preference) for short, attempts to ensure fairness between readers and writers. By fairness here we mean that the readers or writers do not get any preferential treatment over the writers or readers, respectively. To do so, all threads – including readers and writers – are "funnelled" through the central cohort lock, an approach that has been used in the past [15, 16]. Figure 3 depicts the high-level pseudo-code of C-RW-NP. Each thread must first acquire CohortLock. The reader uses the central lock to obtain permission to arrive at ReadIndr (the implementation details of which appear in Section 3.6), then immediately releases the lock, and proceeds to execute its critical section. The fact that readers execute their critical sections *after* releasing CohortLock enables the potential for reader-reader concurrency. After acquiring the cohort lock, the writer must ensure that there are no concurrent conflicting readers. This is done by spinning on ReadIndr (lines 9 and 10) waiting for any readers to depart. This algorithm is clearly very simple and also ensures neutral preference since both the readers and the writers have to acquire the cohort lock. However, requiring readers to acquire the cohort lock can be detrimental to the scalability of C-RW-NP, and also increases the latency of each read acquisition request. C-RW-NPpreserves some cache locality benefits for accesses to the lock metadata and the critical section because all operations funnel through the central cohort lock.

```
 1:  reader:
 2:      CohortLock.acquire()
 3:      ReadIndr.arrive()
 4:      CohortLock.release()
 5:      <read-critical-section>
 6:      ReadIndr.depart()

 7:  writer:
 8:      CohortLock.acquire()
 9:      while NOT(ReadIndr.isEmpty())
10:          Pause
11:      <write-critical-section>
12:      CohortLock.release()
```

Figure 3. The Neutral-Preference Lock (C-RW-NP). The top half is executed by a reader and the bottom half by a writer. For simplicity, the pseudo-code lists the entirety of lock acquisition, critical section execution, and lock release operations in sequential order. In their lock acquisition steps, both readers and writers acquire the cohort lock, while readers also arrive at the ReadIndr. ReadIndr arrival and departures must be atomic.

```
 1:  reader:
 2:      while RBarrier != 0
 3:          Pause
 4:      ReadIndr.arrive()
 5:      while CohortLock.isLocked()
 6:          Pause
 7:      <read-critical-section>
 8:      ReadIndr.depart()

 9:  writer:
10:      bRaised = false // local flag
11:    start:
12:      CohortLock.acquire()
13:      if NOT(ReadIndr.isEmpty())
14:          CohortLock.release()
15:          while NOT(ReadIndr.isEmpty())
16:              Pause
17:              if RanOutOfPatience AND ~bRaised
18                   // erect barrier to stall readers
19:                  atomically increment RBarrier
20:                  bRaised = true
21:          goto start
22:      if bRaised
23:          atomically decrement RBarrier
24:      <write-critical-section>
25:      CohortLock.release()
```

Figure 4. The Reader-Preference Lock (C-RW-RP).

We note that C-RW-NP does not guarantee FIFO semantics. Rather, admission ordering is determined by the prevailing policy imposed by the underlying CohortLock.

3.3 The Reader-Preference Lock

As noted above, C-RW-NP has a crucial drawback arising from the requirement that readers are forced to acquire the central CohortLock. Acquiring the CohortLock incurs extra path length and over-

heads for read operations, even if the cohort lock is uncontended. Under load, contention on the central lock can result in extra coherence traffic and contention for available interconnect bandwidth although this bottleneck is mitigated to some degree by our choice of lock cohorting which acts to reduce inter-node coherence traffic. Furthermore, the extra serialization related to the CohortLock critical section in the read path – albeit very brief – can constitute a scalability bottleneck. Finally, the algorithm's ordering of reader and writer requests based on the cohort lock acquisition order restricts the achievable degree of reader-reader concurrency. In the worst case, there will be no reader-reader concurrency if readers and writers alternate in the cohort lock acquisition order. We overcome both these problems in our reader-preference lock algorithm (called C-RW-RP for short).

Intuitively it makes sense to more aggressively aggregate reader lock acquisition requests to maximize reader-reader concurrency for better scalability. This, however, requires the ability to allow newly arriving readers to bypass writers that arrived earlier but that are still waiting to acquire the lock. This observation was made in the earliest work on RW locks by Courtois et al. [2], and then followed by other works that make the same trade off between fairness and scalability [11, 15, 16]. Our C-RW-RP algorithm also entails the same trade off.

Figure 4 depicts the pseudo-code of C-RW-RP. Readers and writers interact with each other in a way that is reminiscent of the classic Dekker locking idiom [8, 10], where each first declares its existence to the other, and then checks for the other's status. To detect and resolve conflicts, readers must be visible to writers, and writers visible to readers and other potential writers. C-RW-RP readers do not acquire the cohort lock. Instead, they directly arrive at the lock's ReadIndr (line 4). However, each reader can make forward progress only when there are no "active" writers queued on the cohort lock (lines 5–6) Thereafter readers can execute their critical sections and release the lock by departing from ReadIndr.

Writers first acquire CohortLock (line 12) and then verify that there are no concurrent "active" readers. If there are any concurrent readers (indicated by the ReadIndr), the writer releases CohortLock (lines 13–14) and then waits for the readers to drain (line 15). Note that there is a danger of starvation of the writers if they simply wait for no readers to be present but there is a steady stream of arriving readers. To avoid this problem, we have introduced a special reader barrier (called RBarrier) that lets the writer temporarily block all new readers from acquiring read ownership of C-RW-RP. Lines 17–20 show the writer raising the barrier (which is then lowered on line 23), and lines 2–3 show the new readers being blocked by the barrier[4]. The reader barrier is implemented as a single central counter. The writer waits for a pre-determined amount of time before running out of patience on line 17. The writer's "patience threshold" is fairly long so that the barrier is raised rarely and as a result we do not expect it to become a contention bottleneck. In our experiments we used a writer patience threshold of 1000 iterations of the busy-wait loop. The threshold is a tunable parameter. After the writer raises the barrier, the readers steadily drain and when all readers have departed the writer may execute its critical section (line 24) and then finally relinquish write permission by simply releasing CohortLock (line 25).

While the above algorithm is simple, it has a significant performance flaw because of an interesting interaction between contending readers and writers and the succession policy of the CohortLock: Consider an execution scenario where N writers W_1, W_2, W_3, ..., W_N are queued on the cohort lock. W_1 is the lock owner but it has not yet reached line 13. This means that the isLocked function called on line 5 will return true, and block all the readers. A multitude of readers arrive at that time, atomically incrementing ReadIndr, and then spin-wait for isLocked to return false. Next, W_1 executes line 13, detects concurrent readers, and releases CohortLock on line 14. In the process, W_1 hands off CohortLock to W_2, which in turn similarly hands off CohortLock to W_3, and so on. All this while, CohortLock remains in the *locked* state though the lock owner keeps changing, and isLocked returns true for all the readers spinning on it. This circulation of CohortLock ownership between the writers leads to superfluous coherence activity on the lock metadata as well as long and unnecessary waiting periods for readers. In our experiments we have observed that this undesirable interaction between readers and writers leads to significant performance degradation. Furthermore, circulation voids any ordering imposed between writers by the underlying CohortLock.

To avoid this problem we add a new WActive field to C-RW-RP that reflects the *logical* state of the CohortLock. We modify the reader-writer conflict detection logic in line 5 of Figure 4 to spin while WActive is true, instead of spinning on CohortLock. Meanwhile, for the writers, the code between lines 11 and 21 changes to the following:

```
CohortLock.acquire()
loop:
  while NOT(ReadIndr.IsEmpty())
    if RanOutOfPatience AND ~bRaised
      // erect barrier to stall readers
      atomically increment RBarrier
      bRaised = true
  WActive = true // set flag for readers to spin
  if NOT(ReadIndr.IsEmpty())
    // there exist some active readers
    WActive = false // reset the flag
    goto loop
```

Writers acquire CohortLock in the usual fashion and then enter a loop. The code in the loop first waits for ReadIndr to show that there are no pending or active readers, optionally erecting RBarrier if the writer becomes impatient. After ReadIndr indicates that there are no active readers, the code sets WActive to true, and then validates that there are no active or pending readers. If this is the case then control exits the loop and passes into the write critical section. If ReadIndr indicates the existence of readers, however, the code sets WActive to false and passes control back to the top of the the loop which again waits for extant readers to depart. The writer continues to hold CohortLock while it waits for the readers to vacate, avoiding superfluous lock hand offs between writers. After completing its critical section the writer releases the lock by setting WActive to false and then releasing CohortLock. Readers can be blocked by writers only in the brief window where the writer sets WActive and then resets it after detecting the pending readers. We refer to this form as C-RW-RP-opt. Notice that WActive is modified only under CohortLock, and reflects the lock's state: true if CohortLock is acquired, and false otherwise. There is no analogous writer-preference "-opt" form as readers can efficiently rescind publication of their intent to take read permission and then defer to pending writers.

3.4 The Writer-Preference Lock

Conventional wisdom suggests that the reader-preference policy would perform better than both the writer and the neutral-preference policies. Since the application developer has selected a RW lock instead of a mutual exclusion lock, we expect the workload to be read-dominated. The intuition is that packing together as many readers as possible generally leads to better reader-reader concurrency, and hence better throughput. Though we agree with the intuition, we contend that, assuming that a RW lock is acquired by threads in read mode most of the time, the writer-preference policy indirectly leads to the same result – packing together large numbers of reader requests. This is because preferential treatment

[4] There may be another pathology that lets readers starve in the case where writers continuously keep raising the reader barrier and do not allow any readers to make forward progress. We consider such a situation to be even more rare than the rare case where writers run out of patience and raise the reader barrier, and as a result, do not address it in our algorithm.

```
 1:  reader:
 2:      bRaised = false // local flag
 3:    start:
 4:      ReadIndr.arrive()
 5:      if CohortLock.isLocked()
 6:          ReadIndr.depart()
 7:          while CohortLock.isLocked()
 8:              Pause
 9:              if RanOutOfPatience AND ~bRaised
10:                  atomically increment WBarrier
11:                  bRaised = true
12:          goto start
13:      if bRaised
14:          atomically decrement WBarrier
15:      <read-critical-section>
16:      ReadIndr.depart()

17:  writer:
18:      while WBarrier != 0
19:          Pause
20:      CohortLock.acquire()
21:      while NOT(ReadIndr.isEmpty())
22:          Pause
23:      <write-critical-section>
24:      CohortLock.release()
```

Figure 5. The Writer-Preference Lock (C-RW-WP).

of writers leads to a build up of pending reader requests which are then granted en masse when all the writers complete their critical sections. Furthermore, we have observed that the reader-preference policy actually leads to an interesting performance pathology, which we describe in Section 4.3, that can seriously undermine the lock's scalability potential.

Figure 5 depicts the pseudo-code for our writer-preference lock, which we call C-RW-WP. Our C-RW-WP algorithm is clearly symmetric to our C-RW-RP algorithm, the only difference being that the roles of readers and writers in their interactions are switched. Readers arrive at the lock's ReadIndr (line 4), check for a writer (line 5), and if there is one, they depart from the ReadIndr and wait for the writers to drain. If a reader runs out of patience – which is a tunable parameter set to 1000 in our experiments – it can raise a writer barrier (line 10) to block out new writers from acquiring CohortLock (lines 18–19). Writers first verify that the writer barrier has not been raised (line 18–19), then acquire CohortLock (line 20) and ensure that there are no concurrent readers (lines 21–22) before executing the critical section.

3.5 RW Lock Generalization

We observe that our RW lock algorithms are oblivious of the underlying read indicator (ReadIndr) and mutex lock (CohortLock) implementations. Our RW locks require the read indicator data structure to provide just the *arrive*, *depart*, and *isEmpty* operations, and the mutex lock to provide *acquire*, *release*, and *isLocked* operations. Any read indicators and mutex locks that support these operations can be trivially plugged into our algorithms. Furthermore we expect that most implementations of read indicators and mutex locks can support all these operations with at most trivial modifications.

The design flexibility afforded by our RW locks grants programmers significant leverage to build RW locks that are best suited for their applications. For instance, in this paper, we have proposed NUMA-aware RW locks that leverage known NUMA-aware mutex locks and scalable read indicators. In our empirical evaluation (Section 4), we also present performance results of a RW lock that uses distributed counters in the read indicator, and the MCS lock [12] for writer-writer mutual exclusion. Such a lock may be appropriate for applications where writing is exceptionally rare.

3.6 Tracking Readers

Lev et al. [15] observed that readers of a RW lock can be tracked with just the *read indicator* abstraction. Writers checking for the existence of conflicting readers do not need an exact count of readers, but instead need only determine if there are any extant readers. This read indicator can be implemented as a simple counter, updated atomically, which tracks the number of readers that are executing or intend to execute their respective critical sections. However, a simple counter does not scale on a NUMA system. Having made this observation, Lev et al. proposed a SNZI-based [9] solution in their RW locks [15]. The SNZI-based solution significantly scales the read indicator, however the algorithm is complex and readers incur significant overheads at low and moderate contention levels (as we shall see in Section 4). As a result, we have adopted a simple strategy where we "split" a logical counter into multiple physical counters, one per NUMA node. The main goal of our approach is to have a solution that has low latency at low to moderate read arrival rates and scales well at high arrival rates.

A reader thread always manipulates its node-local reader counter. This ensures that counter manipulations do not lead to inter-node coherence traffic. However, after acquiring the internal cohort lock, the writer must peruse through all the reader counters of the RW lock to determine if it is safe to proceed executing the critical section. This adds overhead to the writer's execution path. There is a clear trade off here, and assuming that a RW lock will be acquired in read mode most often, we have opted to simplify the reader's execution path (which involves an increment of just the local reader counter) at the cost of making the writer's execution path longer. Furthermore, most multi-core multi-chip systems available today have relatively small number of NUMA nodes (4 in the system used in our experiments), and we believe the overhead on the writer's execution path is not a major performance concern on these systems. Future NUMA systems with larger numbers of nodes may pose a problem, but we leave the exploration of possible solutions to future work.

The decentralized split-counter can itself be implemented in multiple ways. We discuss two approaches. First is the trivial split-counter, where each node-specific counter is an integer counter. Each reader atomically increments the counter assigned to the reader's node during lock acquisition (arrival), and atomically decrements that same counter during lock release (departure). Using alignment and padding, each node-specific counter is sequestered on its own cache line to avoid false sharing. Each writer, during lock acquisition, verifies that each node-specific counter is 0, and spin-waits on any non-zero counter.

The simple split-counter approach, though effective in reducing inter-node coherence traffic for readers, still admits intra-node contention. Our second approach reduces this contention by employing a pair of *ingress* and *egress* counters in place of each node-specific counter. A reader atomically increments the ingress counter during lock acquisition, and atomically increments the egress counter during lock release. By splitting the logical node-level counter into two variables we divide contention arising from rapid intra-node arrival and departure of readers. On a given node, arriving threads can update ingress independently of concurrently departing threads that are incrementing egress. The ingress and egress counters for a given node may reside on the same cache line. The counter is logically 0 when ingress and egress are equal. Interestingly, the approach of using per-node counters or per-node split ingress-egress counters appears to outperform SNZI-based reader counters, at least for the platforms on which we have taken performance data. We believe this outcome to be platform-specific. Given a sufficiently large number of nodes, the burden of work required to scan those nodes by writers when resolving reader-vs-writer conflicts could become prohibitive. But on current platforms split ingress-egress counters are our preferred implementation for reader counters.

During lock acquisition, a writer verifies that each node-specific ingress-egress pair is equal. This cannot be done atomically, and special care needs to be taken to avoid any races with concurrent readers that are manipulating the counters. More specifically, in our C-RW-WP algorithm, the writer must first read the egress counter

and then the ingress counter in order to correctly determine if the two are equal. Note that both these counters are monotonically increasing, and it is always guaranteed that $egress \leq ingress$ at any given time.

4. Empirical Evaluation

We now present the empirical evaluation of our NUMA-aware RW locks, comparing them with each other and also with other state-of-the-art RW locks. We first present scalability results of these locks on a synthetic microbenchmark. Our results cover a wide range of configurations on varying critical and non-critical section lengths and distributions of read-only and read-write critical sections. We also report what we believe to be a fundamental performance pathology in reader-preference locks. We thereafter show how read indicator implementations affect the scalability of our RW locks. Finally, we present performance results of the kccachetest benchmark of the Kyoto-Cabinet open-source database package, when used with different RW locks. Our empirical evaluation shows that our NUMA-aware RW locks deliver far superior performance than all prior RW locks.

We present performance results of all our locks: the C-RW-NP lock, both the variants of the C-RW-RP lock (the basic C-RW-RP lock, and its optimized form, C-RW-RP-opt, which eliminates the writer ownership circulation problem), and the C-RW-WP lock. Unless specified otherwise, we use the split ingress-egress counter for ReadIndr in our algorithms. We compare our locks with the SNZI-based ROLL lock, the distributed RW lock (DV), and the recently published NUMA-oblivious RW lock by Shirako et al. [18]. Since our locks are built on top of cohort locks, we add a simple mutual-exclusion cohort lock in the mix to understand the benefits our RW locks give above and beyond cohort locks. Finally, to quantify the benefits of using a cohort lock in our RW locks, we compare them with a variant of C-RW-WP that uses an MCS lock for writer-writer exclusion. We call this the DR-MCS lock (short for Distributed Readers, MCS writers).

We implemented all of the above algorithms in C compiled with GCC 4.4.1 at optimization level -O3 in 32-bit mode. The experiments were conducted on an Oracle T5440 series system which consists of 4 Niagara T2+ SPARC chips, each chip containing 8 cores, and each core containing 2 pipelines with 4 hardware thread contexts per pipeline, for a total of 256 hardware thread contexts, running at a 1.4 GHz clock frequency. Each chip has locally connected memory, a 4MB L2 cache, and each core has a shared 8KB L1 data cache. Each T2+ chip is a distinct NUMA node, and the nodes are connected via a central coherence hub. The Solaris 10 scheduler is work-conserving and to maintain cache residency will try to avoid migrating threads. Thread migration was observed to be minimal for all our experiments. While not shown in our pseudo-code, explicit memory fences were inserted as necessary.

We implemented all the above locks within LD_PRELOAD interposition libraries that expose the standard POSIX *pthread_rwlock_t* programming interface. This allows us to change lock implementations by varying the LD_PRELOAD environment variable and without modifying the application code that uses RW locks.

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. In our implementation a thread queries its NUMA node number each time it tries to acquire a lock. We record that number and ensure that readers depart from the same node. The RDTSCP instruction may be a suitable alternative to schedctl on other x86-based platforms where the kernel has arranged to return the CPUID.

4.1 RWBench

To understand the performance characteristics of our locks, and compare them with other locks, we implemented a synthetic multi-threaded microbenchmark that stresses a single RW lock by forcing threads to repeatedly execute critical sections in read or write mode. The microbenchmark, which we call RWBench, is a flexible framework that lets us experiment with various workload characteristics such as varying critical and non-critical section lengths, distribution of read and write mode operations, number of distinct cache lines accessed, etc., all of which are configurable parameters. It uses the pthreads RW lock interface to acquire and release the lock, and we use our LD_PRELOAD interposition library to select various lock implementations.

RWBench spawns the configured number of concurrent threads, each of which loops continuously for 10 seconds. Each top-level iteration starts by casting a biased Bernoulli dice via a thread-local random number generator to determine if the particular iteration should execute a read-only or read-write critical section. The probability with which each loop iteration selects read-write critical sections is a configurable parameter. The critical section touches a single shared array of 64 integers which is protected by a single global RW lock instance. The read-only operation iterates through an inner loop for RCSLen times (a configurable parameter) where each iteration fetches 2 randomly selected integers from the shared array. The read-write operation iterates through an inner loop for WCSLen times (another configurable parameter) where each iteration selects two integers from the shared array, and adds a random value to one integer and subtracts that same value from the other integer. The non-critical section of the main loop similarly updates another thread-private array of 64 integers for NCSLen iterations. At the conclusion of the 10 second run the benchmark verifies that the sum of all the values in the shared array is 0.

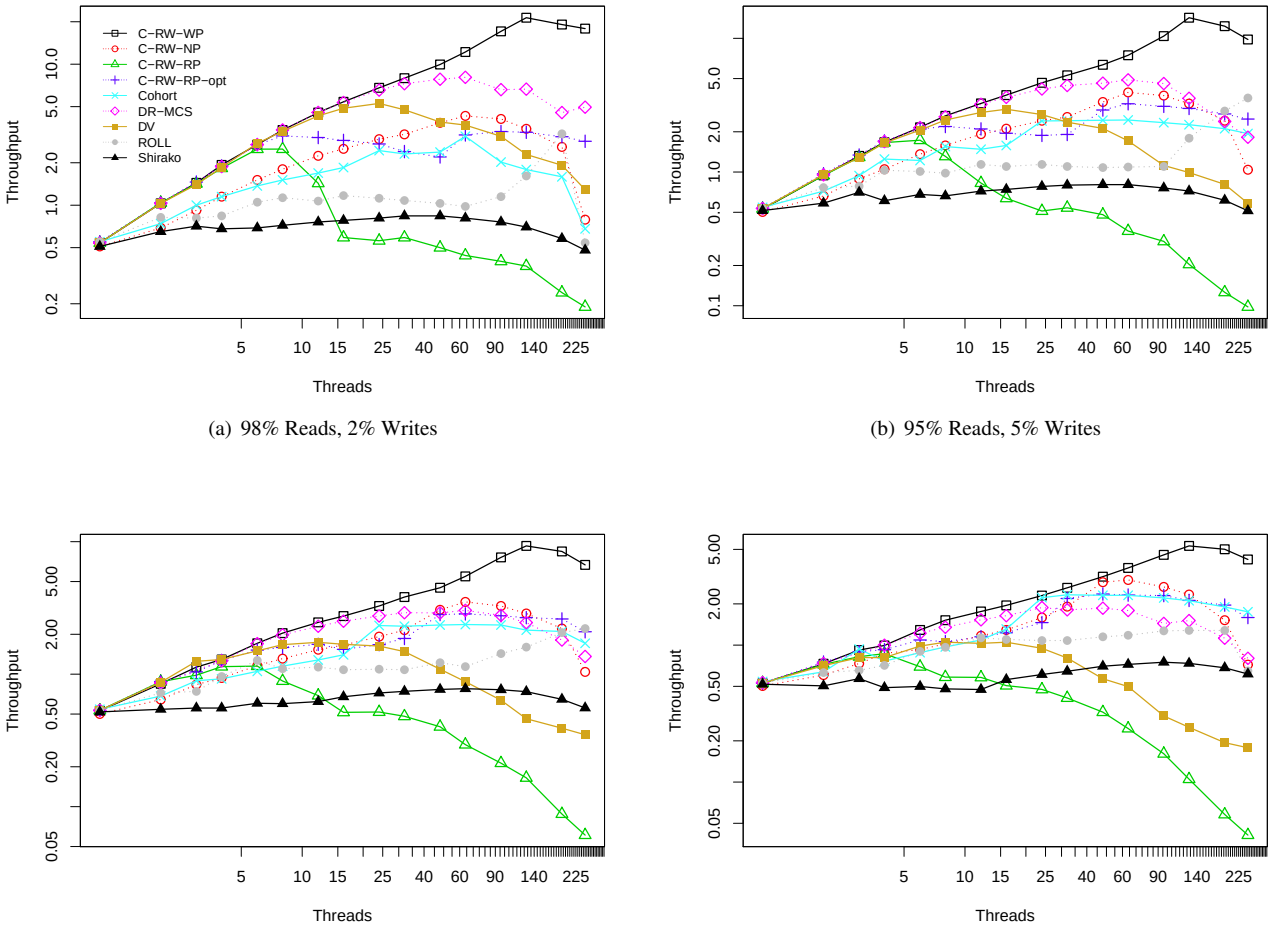
The benchmark reports aggregate throughput at the end of a 10 second run, expressed as iterations of top-level loop executed by the worker threads. We ran 3 trials for each configuration and report their median result. The observed variance was extremely low. In order to adhere as much as possible to 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 [4]. Unlike some other NUMA-aware locks, our locks tolerate ambient thread placement instead of requiring explicit binding.

4.2 RWBench Scalability Test

Figure 6 reports throughput of RWBench with the different locks, and different read/write percent distributions. We believe the read-write distribution mix covers a broad swath of workloads that appear in real application settings. We collected data for higher write percent configurations (up to 50% writes), and found the results to be qualitatively similar to that of Figure 6 (d). The critical and non-critical section sizes in these experiments were deliberately kept small to help us better understand the behavior of all the RW locks under high arrival rates.

First, C-RW-WP is clearly the best performer across the board. Interestingly, DR-MCS performs the second best at 2% writes, but deteriorates considerably with increasing write load. This is because the writes start to play an increasingly important role in performance and DR-MCS experiences excessive coherence traffic due to the NUMA-oblivious queuing of writers on its internal MCS lock. DV is competitive at low thread counts, but deteriorates significantly with high contention, presumably because writers must acquire all the NUMA-node RW locks, which increases both the coherence traffic and delays in lock acquisition in both read and write modes.

ROLL initially scales slowly with increasing thread count. This is because the threads in our test harness are dispersed by the



(a) 98% Reads, 2% Writes

(b) 95% Reads, 5% Writes

(c) 90% Reads, 10% Writes

(d) 80% Reads, 20% Writes

Figure 6. RWBench scalability results of various RW locks for varying read/write distributions: 2%, 5%, 10%, and 20% of write mode lock acquisitions. All graphs are in *loglog* scale. Here RCSLen $= 4$, WCSLen $= 4$, and NCSLen $= 32$. We vary the number of threads from 1 to 255 on the X-axis. Y-axis throughput is expressed in terms of aggregate million loop iterations performed per second.

Solaris scheduler over the entire system [4], and as a result, there are fewer threads dedicated to a SNZI-tree leaf node of the ROLL lock. The end result is that more threads tend to "climb" up the SNZI-tree and compete at the top level (root) node, which is shared across the entire system and consequently leads to an increase in coherence traffic. ROLL starts to scale more quickly at high thread counts, presumably because the load at each leaf of the SNZI-tree is sufficient to reduce the number of threads visiting the root node, thus reducing the coherence traffic over the system. However, this scaling is constrained even at 20% write load, where writers, which are NUMA-oblivious, begin to play a noticeable role in the scalability of the lock. The Shirako lock, which is another recent NUMA-oblivious lock, does not scale on our NUMA system.

Cohort exhibits interesting performance characteristics. Since it does not provide any reader-reader concurrency, Cohort does not scale as well as the best RW locks. However, as the write rate increases, Cohort starts to close its performance gap with our other RW locks, and is quite competitive with all but C-RW-WP at 20% write loads. This demonstrates that even at modestly high write loads, the writer-writer exclusion component of a RW lock becomes an important scalability factor, and since Cohort is extremely efficient and scalable at writer-writer exclusion on NUMA systems, it tends to be competitive with the best RW locks.

Relative to a simple mutual exclusion lock, RW locks usually have longer path lengths (latency) and access more shared metadata. The latter can detract from scalability. Comparing Cohort to true RW locks is interesting as the benefits of potential reader-reader concurrency do not necessarily overcome the additional overheads inherent in RW locks, particularly when critical sections are relatively short or the thread count is low. C-RW-NP appears to leverage additional reader-reader concurrency benefit only marginally over Cohort when the write load is low (2%). In all other cases, it performs similarly to Cohort because even the readers have to acquire the cohort lock in order to arrive at the ReadIndr.

Lastly, the difference in performance of C-RW-RP and C-RW-RP-opt clearly demonstrates the pitfalls of the superfluous writer ownership circulation problem in C-RW-RP. However, C-RW-RP-opt does not scale as well as C-RW-WP because of its reader-preference performance pathology, which we describe next.

4.3 Reader-Preference Performance Pathology

As is clear from Figure 6, our reader-preference locks perform considerably worse than our writer-preference lock. It is well-known that a *strict* reader-preference policy may allow writers to starve. However, we have observed that a reader-preference policy can also result in a secondary phenomenon where readers themselves may underutilize available concurrency.

Say we have a fixed set of threads that loop, deciding randomly whether to take a central RW lock for either reading (query) or writing (update). We will assume that reading is more frequent than writing – access to the RW lock is read-dominated. We will assume that most of the threads are initially readers. Over time

those readers will complete their operation and some fraction will evolve into writers, while the majority will remain readers. Those writers will block in deference to the extant readers given the reader-preference policy. Finally, when no readers remain active, a writer is allowed admission. But when that writer completes its operation it may quickly turn into a reader again, thus obstructing the large set of blocked writers. This undesirable mode can persist. At any given time most threads are blocked trying to acquire write permission while either one writer is active or a small number of readers are active. This results in underutilizing the system by failing to leverage potential reader-reader concurrency that could be realized with a different lock admission schedule. Even though our workload is read-dominated, we have sufficient threads, and we have a reader-preference lock policy, we achieve very low reader throughput and experience diminished reader-reader concurrency. The writer starvation effect that arises from the reader-preference policy prevents a sufficient number of writers from evolving back into readers, thus restricting reader-reader concurrency. While we observed this behavior in RWBench we note that a pool of server threads accepting query or update requests on a data structure protected by an RW lock are also exposed to this pathology.

One might wonder if the writer-preference policy could also lead to a similar pathology. Though incoming writers may obstruct concurrent readers, we do not expect the problem to be as severe with a writer-preference policy. We assume that RW locks will be predominantly acquired in read mode. Hence, even if a set of writers block concurrent readers, their threads will usually return to reacquire the lock in read mode. As a result the readers will be stalled for short intervals. Furthermore, a group of waiting readers can proceed concurrently once all the writers get out of their way.

Strict reader/writer-preference policies may lead to the starvation of the threads engaged in the non-preferred lock acquisition operation. Consequently, to avoid such issues, we believe that a general-purpose RW lock algorithm should detect and recover from policy-based starvation. If the base policy is writer-preference, for example, then if readers languish for too long then the lock can block incoming writers and allow the backlog of readers to enter. And in fact this mechanism can promote aggregation of readers, yielding increased reader-reader concurrency. Effectively, the lock can switch transiently from writer-preference to either neutral-preference or reader-preference. Similarly, if the base lock policy is reader-preference, when writers starve the lock could take remedial action to ensure that writers make progress by transiently shifting to neutral- or writer-preference. This mechanism is illustrated in the anti-starvation facility shown in Figure 5. As an alternative to erecting barriers to block the flow of incoming writers, we also experimented with having impatient readers acquire the CohortLock as shown in the neutral-preference lock in Figure 3. Readers first attempt to use the fast path, but if they fail to make reasonable progress, they instead use a more pessimistic approach and pass through the CohortLock. This approach also yields reasonable performance.

4.4 ReadIndr Implementations

As described in Section 3.6, ReadIndr can be implemented in multiple ways. We now compare the performance of each of the three implementations we discussed – a simple integer counter, a split counter with one integer counter per NUMA node, and a split ingress-egress counter with one pair of ingress-egress counters per NUMA node. Figure 7 depicts the performance of C-RW-WP when equipped with these counter implementations when run with 98% reads and 2% writes. C-RW-WP-1RC represents the version of C-RW-WP with a single central reader counter. This central counter is clearly a scalability bottleneck and results in significant performance deterioration with increasing thread count. C-RW-WP-4RC is a variant of C-RW-WP that uses 4 simple counters, one per

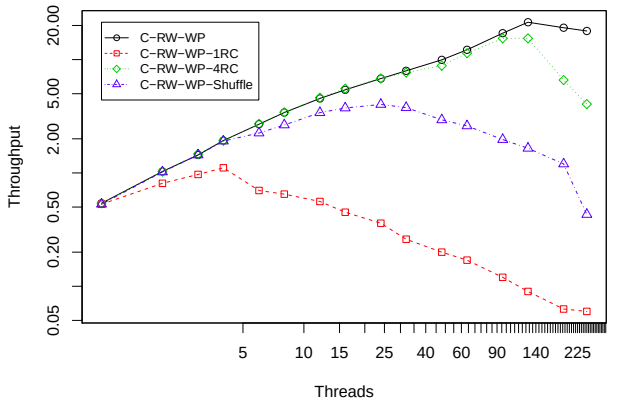


Figure 7. Scalability of the various reader counter implementations used in the same RWBench test configuration as Figure 6(a).

NUMA node on our experimental platform. The improvement with this split counter is significant and tracks the performance of C-RW-WP (which uses 4 ingress-egress counter pairs, one pair per NUMA node) to about 96 threads, but subsequently deteriorates because of excess contention on each of the node-local counters. The ingress-egress pair halves such contention, and scales better than the split counter implementation.

The scalable performance exhibited by C-RW-WP could arise in part by virtue of NUMA-locality via node-specific counters and in part by simply distributing accesses over the set of 4 reader indicator instances. C-RW-WP-Shuffle is a variation of C-RW-WP where we randomly shuffle reader indicator indices associated with threads. The number of threads accessing each counter remains the same, but those threads can reside on different nodes. As can be seen in the figure, a significant fraction of the benefit in C-RW-WP comes from NUMA-locality.

4.5 kccachetest Performance

kccachetest is provided with the Kyoto-Cabinet distribution, a popular open-source database package. It serves as a stress test and performance benchmark for the in-memory "cache hash" (CacheDB) database. CacheDB makes heavy use of standard pthread_rwlock_t operators, and performance of the benchmark is sensitive to the quality of the lock implementation. We ran the program with the "wicked" argument, which constructs an in-memory non-persistent database and then runs randomly selected transactions against that database. Both the benchmark and database reside in the same process and communicate via calls and shared memory. The benchmark is internally constrained to at most 64 threads. The number of threads can be specified on the command line, and each of the worker threads loops, selecting a random operation to be performed on the database. In some cases the operations are simple lookups or deletes, while others are more complex transactions. Each thread performs the same number of operations, and the program reports the interval between the time the first thread starts and the time the final thread completes. Unfortunately the size of the key range, and thus the footprint of the in-memory database, is a function of the number of threads. So as we increase the number of threads we also increase the key range and the footprint, which means that the results of runs with different numbers of threads are not easily comparable. As a result, we do not plot kccachetest results on a graph, but rather use a tabular representation.

Figure 8 shows the performance of kccachetest when used with the different RW locks. While all the locks are competitive at low threading levels, their performance diverges significantly with increasing thread counts. kccachetest contains a diverse mix of crit-

Locks	1T	2T	3T	4T	6T	8T	12T	16T	24T	32T	48T	64T
C-RW-WP	.510	1.20	1.78	1.95	3.08	4.24	6.99	9.24	14.5	18.0	26.7	37.7
C-RW-NP	.521	1.09	1.64	2.16	3.09	4.26	6.58	9.22	14.3	17.6	25.4	36.4
C-RW-RP	.550	1.12	1.77	2.23	3.31	4.53	7.09	10.4	13.3	18.9	34.5	52.5
C-RW-RP-opt	.531	1.15	1.76	2.19	3.30	4.54	7.45	10.5	13.5	17.9	27.0	41.2
Cohort	.516	1.25	1.74	2.35	3.41	4.36	7.03	8.55	13.3	18.2	29.6	44.3
DR-MCS	.531	1.13	1.58	1.94	3.17	4.19	6.79	10.2	17.6	26.7	49.6	77.1
DV	.511	1.18	1.75	2.14	3.18	4.12	6.59	9.88	12.8	21.3	52.0	56.8
ROLL	.547	1.25	1.55	1.99	3.32	4.50	7.79	11.3	20.2	29.0	46.4	63.3
Shirako	.554	1.16	1.61	2.07	3.33	4.53	7.30	10.7	18.5	27.0	35.5	55.2

Figure 8. Scalability results of the Kyoto Cabinet kccachetest benchmark (with the command line arguments: wicked -th Thrds -capsiz 2000000 100000). Each entry in the table reports wall clock time to completion in seconds.

ical sections, including short read-only and read-write ones, and long and complex read-write ones. Overall, the workload is dominated by read-write critical sections, where the threads acquire the RW locks in write mode. As a result, Cohort performs comparably to our NUMA RW locks, and much better than all other locks that contain NUMA-oblivious writers – DR-MCS, DV, ROLL, and Shirako. DR-MCS scales poorly because the underlying MCS lock acquired by writers forces the cache lines of the lock and the data it protects to bounce between NUMA nodes more often than other locks. Since Cohort significantly curtails lock migration, it performs much better. Our NUMA-aware RW locks, except C-RW-RP, further extend the *cohorting* advantage because of NUMA-friendly reader-reader concurrency. C-RW-RP succumbs to the superfluous writer ownership circulation performance problem described earlier, and, as a result, does not scale as well as our other locks. It does however scale better than all prior locks. Overall, C-RW-WP and C-RW-NP, which perform the best, outperform the best of the prior locks (DV and Shirako) by about 40%.

5. Conclusion

The rapid growth of multi-core multi-chip systems is making NUMA architectures commonplace, and fundamental data structures and synchronization primitives must be redesigned to adapt to these environments. We introduced a new family of surprisingly simple NUMA-aware reader-writer locks that outperform prior lock algorithms by a large margin. Writers use centralized lock metadata and readers use decentralized metadata. Microbenchmark experiments suggest that our best lock exceeds the performance of the prior state-of-the-art by up to a factor of 10, and our experiments on a real-world application, the Kyoto Cabinet database, show our locks can boost the application's performance by up to 40%.

Acknowledgments

We thank Doug Lea for useful discussions. Nir Shavit was supported in part by NSF grant 1217921.

References

[1] B. B. Brandenburg and J. H. Anderson. Spin-based Reader-Writer Synchronization for Multiprocessor Real-time Systems. *Real-Time Syst.*, 46(1):25–87, 2010.

[2] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, 1971.

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

[4] D. Dice. Solaris Scheduling: SPARC and CPUIDs. URL `https://blogs.oracle.com/dave/entry/solaris_scheduling_and_cpuids`.

[5] D. Dice. A Partitioned Ticket Lock. In *Proceedings of the 23rd ACM Aymposium on Parallelism in Algorithms and Architectures*, pages 309–310, 2011.

[6] D. Dice and N. Shavit. TLRW: Return of the Read-Write Lock. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 284–293, 2010.

[7] D. Dice, V. J. Marathe, and N. Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 247–256, 2012.

[8] E. W. Dijkstra. The origin of concurrent programming. chapter Cooperating sequential processes, pages 65–138. 2002.

[9] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: Scalable NonZero Indicators. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*, pages 13–22, 2007.

[10] E. Freudenthal and A. Gottlieb. Process coordination with fetch-and-increment. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 260–268, 1991.

[11] W. C. Hsieh and W. E. Weihl. Scalable Reader-Writer Locks for Parallel Systems. In *Proceedings of the Sixth International Parallel Processing Symposium*, 1991.

[12] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[13] J. M. Mellor-Crummey and M. L. Scott. Synchronization without Contention. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 269–278, 1991.

[14] O. Krieger, M. Stumm, R. Unrau, and J. Hanna. A Fair Fast Scalable Reader-Writer Lock. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages 201–204, 1993.

[15] Y. Lev, V. Luchangco, and M. Olszewski. Scalable Reader-Writer Locks. In *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures*, pages 101–110, 2009.

[16] J. M. Mellor-Crummey and M. L. Scott. Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 106–113, 1991.

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

[18] J. Shirako, N. Vrvilo, E. G. Mercer, and V. Sarkar. Design, verification and applications of a new read-write lock algorithm. In *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures*, pages 48–57, 2012.

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

[20] D. Vyukov. Distributed Reader-Writer Mutex. URL `http://www.1024cores.net/home/lock-free-algorithms/reader-writer-problem/distributed-reader-writer-mutex`.