BRAVO—Biased Locking for Reader-Writer Locks

Dave Dice and Alex Kogan, *Oracle Labs*

https://www.usenix.org/conference/atc19/presentation/dice

This paper is included in the Proceedings of the
2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

BRAVO – Biased Locking for Reader-Writer Locks

Dave Dice
Oracle Labs
dave.dice@oracle.com

Alex Kogan
Oracle Labs
alex.kogan@oracle.com

Abstract

Designers of modern reader-writer locks confront a difficult trade-off related to reader scalability. Locks that have a compact memory representation for active readers will typically suffer under high intensity read-dominated workloads when the "reader indicator" state is updated frequently by a diverse set of threads, causing cache invalidation and coherence traffic. Other designs use distributed reader indicators, one per NUMA node, per core or even per thread. This improves reader-reader scalability, but also increases the size of each lock instance and creates overhead for writers.

We propose a simple transformation, BRAVO, that augments any existing reader-writer lock, adding just two integer fields to the lock instance. Readers make their presence known to writers by hashing their thread's identity with the lock address, forming an index into a *visible readers table* and installing the lock address into the table. All locks and threads in an address space can share the same readers table. Crucially, readers of the same lock tend to write to different locations in the table, reducing coherence traffic. Therefore, BRAVO can augment a simple compact lock to provide scalable concurrent reading, but with only modest and constant increase in memory footprint.

We implemented BRAVO in user-space, as well as integrated it with the Linux kernel reader-writer semaphore (`rwsem`). Our evaluation with numerous benchmarks and real applications, both in user and kernel-space, demonstrate that BRAVO improves performance and scalability of underlying locks in read-heavy workloads while introducing virtually no overhead, including in workloads in which writes are frequent.

1 Introduction

A reader-writer lock, also known as a shared-exclusive lock, is a synchronization primitive for controlling access by multiple threads (or processes) to a shared resource (critical section). It allows shared access for read-only use of the resource, while write operations access the resource exclusively. Such locks are ubiquitous in modern systems, and can be found, for example, in database software, file systems, key-value stores and operating systems.

Reader-writer locks have to keep track of the presence of active readers before a writer can be granted the lock. In the common case, such presence is recorded in a shared counter, incremented and decremented with every acquisition and release of the lock in the read mode. This is the way reader-writer locks are implemented in the Linux kernel, POSIX pthread library and several other designs [3, 35, 42]. The use of a shared counter lends itself to a relatively simple implementation and has a compact memory representation for a lock. However, it suffers under high intensity read-dominated workloads when the "reader indicator" state is updated frequently by a diverse set of threads, causing cache invalidation and coherence traffic [7, 17, 20, 31].

Alternative designs for reader-writer locks use distributed reader indicators, for instance, one per NUMA (non-uniform memory access) node as in cohort locks [6], or even one lock per core as in the Linux kernel brlock [10] and other related ideas [26, 31, 39, 46]. This improves reader-reader scalability, but also considerably increases the size of each lock instance. Furthermore, the lock performance is hampered when writes are frequent, as multiple indicators have to be accessed and/or modified. Finally, such locks have to be instantiated dynamically, since the number of sockets or cores can be unknown until the runtime. As a result, designers of modern reader-writer locks confront a difficult trade-off related to the scalability of maintaining the indication of the readers' presence.

In this paper, we propose a simple transformation, called BRAVO, that augments any existing reader-writer lock, adding just two integer fields to the lock instance. When applied on top of a counter-based reader-writer lock, BRAVO allows us to achieve, and often beat, the performance levels of locks that use distributed reader indicators while maintaining a compact footprint of the underlying lock. With BRAVO, readers make their presence known to writers by hashing their thread's identity with the lock address, forming an index into a *visible readers table*. Readers attempt to install the lock address into the element (slot) in the table identified by that index. If successful, readers can proceed with their critical section *without* modifying the shared state of the underlying lock. Otherwise, readers resort to the acquisition path of the underlying lock. Note that the visible readers table is shared by all locks and threads in an address space. Crucially, readers of the same lock tend to write to different locations in the table, reducing coherence traffic and thus resulting in a NUMA-friendly design. At the same time, a writer always uses the acquisition path of the underlying lock, but also scans the readers table and waits for all readers that acquired that lock through it. A simple mechanism is put in place to limit the overhead of scanning the table for workloads in which writes are frequent.

We implemented BRAVO and evaluated it on top of several locks, such as the POSIX pthread_rwlock lock and the PF-Q reader-writer lock by Brandenburg and Anderson [3]. For our evaluation, we used numerous microbenchmarks as well as `rocksdb` [40], a popular open-source key-value store. Furthermore, we integrated BRAVO with `rwsem`, a read-write semaphore in the Linux kernel. We evaluated the modified kernel through kernel microbenchmarks as well as several user-space applications (from the Metis suite [33]) that create contention on read-write semaphores in the kernel. All our experiments in user-space and in the kernel demonstrate that BRAVO is highly efficient in improving performance and scalability of underlying locks in read-heavy workloads while introducing virtually no overhead, even in workloads in which writes are frequent.

The rest of the paper is organized as following. The related work is surveyed in Section 2. We present the BRAVO algorithm in Section 3 and discuss how we apply BRAVO in the Linux kernel in Section 4. The performance evaluation in user-space and the Linux kernel is provided in Sections 5 and 6, respectively. We conclude the paper and elaborate on multiple directions for future work in Section 7.

2 Related Work

► **Reader-Indicator Design** Readers that are *active* – currently executing in a reader critical section – must be visible to potential writers. Writers must be able to detect active readers in order to resolve read-vs-write conflicts, and wait for active readers to depart. The mechanism through which readers make themselves visible is the *reader indicator*. Myriad designs have been described in the literature. At one end of the spectrum we find a centralized reader indicator implemented as an integer field within each reader-writer lock instance that reflects the number of active readers. Readers use atomic instructions (or a central lock) to safely increment and decrement this field. Classic examples of such locks can be found in the early work of Mellor-Crummey and Scott [35] and more recent work by Shirako et al. [42]. Another reader-writer lock algorithm having a compact centralized reader indicator is Brandenburg and Anderson's Phase-Fair Ticket lock, designated `PF-T` in [3], where the reader indicator is encoded in two central fields. Their Phase-Fair Queue-based lock, `PF-Q`, uses a centralized counter for active readers and an MCS-like central queue, with local spinning, for readers that must wait. We refer to this latter algorithm as "BA" throughout the remainder of this paper. Such approaches are compact, having a small per-lock footprint, and simple, but, because of coherence traffic, do not scale in the presence of concurrent readers that are arriving and departing frequently [7, 17, 20, 31].

To address this concern, many designs turn toward distributed reader indicators. Each cohort reader-writer lock [6], for instance, uses a per-NUMA node reader indicator. While distributed reader indicators improve scalability, they also significantly increase the footprint of a lock instance, with each reader indicator residing on its own private cache line or sector to reduce false sharing. In addition, the size of the lock is variable with the number of nodes, and not known at compile-time, precluding simple static preallocation of locks. Writers are also burdened with the overhead of checking multiple reader indicators. Kashyap et al. [27] attempt to address some of those issues by maintaining a dynamic list of per-socket structures and expand the lock instance on-demand. However, this only helps if a lock is accessed by threads running on a subset of nodes.

At the extreme end of the spectrum we find lock designs with reader indicators assigned per-CPU or per-thread [10, 26, 31, 39, 46]. These designs promote read-read scaling, but have a large variable-sized footprint. They also favor readers in that writers must traverse and examine all the reader-indicators to resolve read-vs-write conflicts, possibly imposing a performance burden on writers. We note that there are a number of varieties of such distributed locks: a set of reader-indicators coupled with a central mutual exclusion lock for writer permission, as found in cohort locks [6]; sets of mutexes where readers must acquire one mutex and writers must acquire all mutexes, as found in Linux kernel brlocks [10]; or sets of reader-writer locks where readers must acquire read permission on one lock, and writers must acquire write permission on all locks. To reduce the impact on writers, which must visit all reader indicators, some designs use a tree of distributed counters where the root element contains a sum of the indicators within the subtrees [30].

Dice et al. [19] devised *read-write byte-locks* for use in the *TLRW* software transactional memory infrastructure. Briefly, read-write byte-locks are reader-writer locks augmented with an array of bytes, serving as reader indicators, where indices in the array are assigned to favored threads that are frequent readers. These threads can simply set and clear these reader indicators with normal store operations. The motivation for read-write byte-locks was to avoid atomic read-modify-write instructions, which were particularly expensive on the system under test. The design, as described, is not NUMA-friendly as the byte array occupies a single cache line.

In addition to distributing or dispersing the counters, individual counters can themselves be further split into constituent `ingress` and `egress` fields to further reduce write sharing. Arriving readers increment the ingress field and departing readers increment the egress field. Cohort reader-writer locks use this approach [6].

BRAVO takes a different approach, opportunistically representing active readers in the shared global visible readers table. The table (array) is fixed in size and shared over all threads and locks within an address space. Each BRAVO lock has, in addition to the underlying reader-writer lock, a boolean flag that indicates if reader bias is currently enabled for that lock. Publication of active readers in the array is strictly optional and best-effort. A reader can always fall back to acquiring read permission via the underlying reader-writer lock. BRAVO's benefit comes from reduced coherence traffic arising from reader arrival. Such coherence traffic is particularly costly on NUMA systems, consuming shared interconnect bandwidth and also exhibiting high latency. As such, BRAVO is naturally NUMA-friendly. However, unlike most other NUMA-aware reader-writer locks, it does not need to understand or otherwise query the system topology, further simplifying the design and reducing dependencies[1].

► **Optimistic Invisible Readers** Synchronization constructs such as *seqlocks* [9,23,29] allow concurrent readers, but forgo the need for readers to make themselves visible. Critically, readers do not write to synchronization data and thus do not induce coherence traffic. Instead, writers update state – typically a modification counter – to indicate that updates have occurred. Readers check that counter at the start and then again at the end of their critical section, and if writers were active or the counter changed, the readers self-abort and retry. An additional challenge for seqlocks is that readers can observe inconsistent state, and special care must be taken to constrain the effects and avoid errant behavior in readers. Often, non-trivial reader critical sections must be modified to safely tolerate optimistic execution. Various hybrid forms exist, such as the `StampedLock` [36] facility in `java.util.concurrent`, which consists of a reader-writer lock coupled with a seqlock, providing 3 modes: classic pessimistic write locking, classic pessimistic read locking, and optimistic reading.

To avoid the problem where optimistic readers might see inconsistent state, transactional lock elision [16,18,24,28,38] based on hardware transactional memory can be used. Readers are invisible and do not write to shared data. Such approaches can be helpful, but are still vulnerable to indefinite abort and progress failure. In addition, the hardware transactional memory facilities required to support lock elision are not available on all systems, and are usually best-effort, without any guaranteed progress, requiring some type of fallback to pessimistic mechanisms.

[1]While BRAVO is topology oblivious, it does require high-resolution low-latency means of reading the system clock. We further expect that reading the clock is scalable, and that concurrent readers do not interfere with each other. On systems with modern Intel CPUs and Linux kernels the `RDTSCP` instruction or `clock_gettime(CLOCK_MONOTONIC)` fast system call suffice.

► **Biased Locking** BRAVO draws inspiration from *biased locking* [14,22,37,41,44]. Briefly, biased locking allows the same thread to repeatedly acquire and release a mutual exclusion lock without requiring atomic instructions, except on the initial acquisition. If another thread attempts to acquire the lock, then expensive *revocation* is required to wrest bias from the original thread. The lock would then revert to normal non-biased mode for some period before again becoming potentially eligible for bias. (Conceptually, we can think of the lock as just being left in the locked state until there is contention. Subsequent lock and unlock operations by the original thread are ignored – the unlock operation is deferred until contention arises). Biased locking was a response to the CPU-local latencies incurred by atomic instructions on early Intel and SPARC processors and to the fact that locks in Java were often dominated by a single thread. Subsequently, processor designers have addressed the latency concern, rendering biased locking less profitable.

Classic biased locking identifies a preferred thread, while BRAVO identifies a preferred access mode. That is, BRAVO biases toward a mode instead of thread identity. BRAVO is suitable for read-dominated workloads, allowing a fast-path for readers when reader bias is enabled for a lock. If a write request is issued against a reader-biased lock, reader bias is disabled and revocation (scanning of the visible readers table) is required, shifting some cost from readers to writers. Classic biased locking provides benefit by reducing the number of atomic operations and improving latency. It does not improve scalability. BRAVO reader-bias, however, can improve both latency and scalability by reducing coherence traffic on the reader indicators in the underlying reader-writer lock.

3 The BRAVO Algorithm

BRAVO transforms any existing reader-writer lock A into *BRAVO-A*, which provides scalable reader acquisition. We say A is the *underlying* lock in *BRAVO-A*. In typical circumstances A might be a simple compact lock that suffers under high levels of reader concurrency. *BRAVO-A* will also be compact, but is NUMA-friendly as it reduces coherence traffic and offers scalability in the presence of frequently arriving concurrent readers.

Listing 1 depicts a pseudo-code implementation of the BRAVO algorithm. BRAVO extends A's structure with a new `RBias` boolean field (Line 2). Arriving readers first check the `RBias` field, and, if found set, then hash the address of the lock with a value reflecting the calling thread's identity to form an index into the visible readers table (Lines 12–13). (This readers table is shared by all locks and threads in an address space. In all our experiments we sized the table at 4096 entries. Each table element, or *slot*, is either `null` or a pointer to a reader-writer lock instance). The reader then uses an atomic compare-and-swap (CAS) operator to attempt to change the element at that index from `null` to the address of the lock, publishing its existence to potential writers (Line 14). If the CAS is successful then the reader rechecks the `RBias` field to ensure it remains set (Line 18). If so, the reader has successfully gained read permission and can enter the critical section (Line 19). Upon completing the critical section the reader executes the complementary operation to release read permission, simply storing `null` into that slot (Lines 29–31). We refer to this as the *fast-path*. The fast-path attempt prefix (Lines 11-23) runs in constant time. Our hash function is based on the *Mix32* operator found in [43].

If the recheck operation above happens to fail, as would be the case if a writer intervened and cleared `RBias` and the reader lost the race, then the reader simply clears the slot (Line 21) and reverts to the traditional *slow-path* where it acquires read permission via the underlying lock (Line 24). Similarly, if the initial check of `RBias` found the flag clear (Line 12), or the CAS failed because of collisions in the array (Line 14) – the slot was found to be populated – then control diverts to the traditional slow-path. After a slow-path reader acquires read permission from the underlying lock, it enters and executes the critical section, and then at unlock time releases read permission via the underlying lock (Line 33).

Arriving writers first acquire write permission on the underlying reader-writer lock (Line 36). Having done so, they then check the `RBias` flag (Line 37). If set, the writer must perform *revocation*, first clearing the `RBias` flag (Line 40) and then scanning all the elements of the visible readers table checking for conflicting fast-path readers (Lines 42–44). If any elements match the lock, the writer must wait for that fast-path reader to depart and clear the slot. If lock *L* has 2 fast-path active readers, for instance, then *L* will appear twice in the array. Scanning the array might appear to be onerous, but in practice the sequential scan is assisted by the automatic hardware prefetchers present in modern CPUs. We observe a scan rate of about 1.1 nanoseconds per element on our system-under-test (described later). Having checked `RBias` and performed revocation if necessary, the writer then enters the critical section (Line 50). At unlock-time, the writer simply releases write permission on the underlying reader-writer lock (Line 51). Therefore the only difference for writers under BRAVO is the requirement to check and potentially revoke reader bias if `RBias` was found set.

We note that writers only scan the visible reader table, and never write into it. Yet, this scan may pollute the writer's cache. One way to cope with it is to use non-temporal loads, however, exploring this idea is left for the future work. Note that revocation is only required on transitions from reading to

```
 1 class BRAVOLock<T> :
 2   int RBias
 3   Time InhibitUntil
 4   T Underlying
 5
 6 ## Shared global :
 7 BRAVOLock * VisibleReaders [4096]
 8 int N = 9   # slow-down guard
 9
10 def Reader(BRAVOLock * L) :
11   BRAVOLock * * slot = null
12   if L.RBias :
13     slot = VisibleReaders + Hash(L, Self)
14     if CAS(slot, null, L) == null :
15       # CAS succeeded
16       # store-load fence required on TSO
17       # typically subsumed by CAS
18       if L.RBias :        # recheck
19         goto EnterCS      # fast path
20       *slot = null        # raced - RBias changed
21     slot = null
22   # Slow path
23   assert slot == null
24   AcquireRead (L.Underlying)
25   if L.RBias == 0 and Time() >= L.InhibitUntil :
26     L.RBias = 1
27   EnterCS:
28   ReaderCriticalSection()
29   if slot != null :
30     assert *slot == L
31     *slot = null
32   else :
33     ReleaseRead (L.Underlying)
34
35 def Writer(BRAVOLock * L) :
36   AcquireWrite (L.Underlying)
37   if L.RBias :
38     # revoke bias
39     # store-load fence required on TSO
40     L.RBias = 0
41     auto start = Time()
42     for i in xrange(VisibleReaders) :
43       while VisibleReaders[i] == L :
44         Pause()
45     auto now = Time()
46     # primum non-nocere :
47     # limit and bound slow-down
48     # arising from revocation overheads
49     L.InhibitUntil = now + ((now - start) * N)
50   WriterCriticalSection()
51   ReleaseWrite (L.Underlying)
```

Listing 1: Simplified Python-like implementation of BRAVO

writing and only when `RBias` was previously set.

In summary, active readers can make their existence public in one of two ways : either via the visible readers table (fast-path), or via the traditional underlying reader-writer lock (slow-path). Our mechanism allows both slow-path and fast-path readers simultaneously. Absent hash collisions, concurrent fast-path readers will write to different locations in the visible readers table. Collisions are benign, and impact performance but not correctness. Writers resolve read-vs-write conflicts against fast-path readers via the visible readers table and against slow-path readers via the underlying reader-writer lock.

One important remaining question is how to set RBias. In our early prototypes we set RBias in the reader slow-path based on a low-cost Bernoulli trial with probability $P = 1/100$ using a thread-local Marsgalia XOR-Shift [34] pseudo-random number generator. While this simplistic policy for enabling bias worked well in practice, we were concerned about situations where we might have enabled bias too eagerly, and incur frequent revocation to the point where *BRAVO-A* might be slower than *A*. Specifically, the worst-case scenario would be where slow readers repeatedly set RBias, only to have it revoked immediately by a writer.

The key additional cost in BRAVO is the revocation step, which executes under the underlying write lock and thus serializes operations associated with the lock[2]. As such, we measure the latency of revocation and multiply that period by N, a configurable parameter, and then inhibit the subsequent setting of bias in the reader slow-path for that period, bounding the worst-case expected slow-down from BRAVO for writers to $1/(N+1)$ (cf. Lines 41-49). Our specific performance goal is *primum non nocere* – first, do no harm, with *BRAVO-A* never underperforming *A* by any significant margin on any workload[3]. This tactic is simple and effective, but excessively conservative, taking into account only the worst-case performance penalty imposed by BRAVO, and not accounting for any potential benefit conferred by the BRAVO fast-path. Furthermore, measuring the revocation duration also incorporates the waiting time, as well as the scanning time, yielding a conservative over-estimate of the revocation scan cost and resulting in less aggressive use of reader bias. Despite these concerns, we find this policy yields good and predictable performance. For all benchmarks in this paper we used $N = 9$ yielding a worst-case writer slow-down bound of about 10%. Our policy required adding a second BRAVO-specific timestamp field InhibitUntil (Line 3), which reflects the earliest time at which slow readers should reenable bias[4]. We note that for safety, readers can only set RBias while they hold read permission on the underlying reader-writer lock, avoiding interactions with writers (cf. Lines 25–26).

In our implementations revoking waiters busy-wait for readers to depart. There can be at most one such busy-waiting thread for a given lock at any given time. We note, however, that it is trivial to shift to a waiting policy that uses blocking.

BRAVO acts as an accelerator layer, as readers can always fall back to the traditional underlying lock to gain read access. The benefit arises from avoiding coherence traffic on the centralized reader indicators in the underlying lock, and instead relying on updates to be diffused over the visible readers table. Fast-path readers write only into the visible readers table, and not the lock instance proper. This access pattern improves performance on NUMA systems, where write sharing is particularly expensive. We note that if the underlying lock algorithm *A* has reader preference or writer preference, then *BRAVO-A* will exhibit that same property. Write performance and the scalability of read-vs-write and write-vs-write behavior depends solely on the underlying lock. Under high write intensity, with write-vs-write and write-vs-read conflicts, the performance of BRAVO devolves to that of the underlying lock. BRAVO accelerates reads only. BRAVO fully supports the case where a thread holds multiple locks at the same time.

BRAVO supports *try-lock* operations as follows. For read *try-lock* attempts an implementation could try the BRAVO fast path and then fall back, if the fast path fails, to the slow path underling *try-lock*. An implementation can also opt to forgo the fast path attempt and simply call the underlying *try-lock* operator. We use the former approach when applying BRAVO in the Linux kernel as detailed in the next section. We note that if the underlying *try-lock* call is successful, one may set RBias if the BRAVO policy allows that (e.g., if the current time is larger than InhibitUntil). For write *try-lock* operators, an implementation will invoke the underlying *try-lock* operation. If successful, and bias is set, then revocation must be performed following the same procedure described in Lines 37–49.

As seen in Listing 1, the *slot* value must be passed from the read lock operator to the corresponding unlock. null indicates that the slow path was used to acquire read permission. To provide correct errno values in the POSIX pthread environment, a thread must be able to determine if it holds read, write, or no permission on a given lock. This is typically accomplished by using per-thread lists of locks currently held in read mode. We leverage those list elements to pass the slot. We note that the Cohort read-write lock implementation [6] passed the reader's NUMA node ID from lock to corresponding unlock in this exact fashion.

4 Applying BRAVO to the Linux Kernel `rwsem`

In this section, we describe prototype integration of BRAVO in the Linux kernel, where we apply it to rwsem. Rwsem is a read-write semaphore construct. Among many places inside the kernel, it is used to protect the access to the virtual memory area (VMA) structure of each process [11], which makes it a

[2]Additional costs associated with BRAVO include futile atomic operations from collisions, and sharing or false-sharing arising from near-collisions in the table. Our simplified cost model ignores these secondary factors. We note that the odds of collision are equivalent to those given by the "Birthday Paradox" [48] and that the general problem of deciding to set bias is equivalent to the classic "ski-rental" problem [47].

[3]Our approach conservatively forgoes the potential of better performance afforded by the aggressive use of reader bias in order to limit the possibility of worsened performance [49].

[4]We observe that it is trivial to collapse RBias and InhibitUntil into just a single field. For clarity, we did not do so in our implementation.

source of contention for data intensive applications [8, 11].

On a high level, `rwsem` consists of a counter and a waiting queue protected by a spin-lock. The counter keeps track of the number of active readers, as well as encodes the presence of a writer. To acquire the `rwsem` in the read mode, a reader atomically increments the counter and checks its value. If a (waiting or active) writer is not present, the read acquisition is successful; otherwise, the reader acquires the spin-lock protecting the waiting queue, joins the queue at the tail, releases the spin-lock and blocks, waiting for a wake-up signal from a writer. As a result, when there is no reader-writer contention, the read acquisition boils down to one atomic counter increment. On architectures that do not support an atomic increment instruction, this requires acquisition (and subsequent release) of the spin-lock. Even on architectures that have such an instruction (such as Intel x86), the read acquisition of `rwsem` creates contention over the cache line hosting the counter.

In our integration of BRAVO on top of `rwsem`, we make a simplifying assumption that the semaphore is always released by the same thread that acquired it for read. This is not guaranteed by the API of `rwsem`, however, this is a common way of using semaphores in the kernel. This assumption allows us to preserve the existing `rwsem` API and limits the scope of changes required, resulting in a patch of only three files and adding just a few dozens lines of code. We use this assumption when determining the slot into which a thread would store the semaphore address on the fast acquisition path, and clear that slot during the release operation[5].

While we have not observed any issue when running and evaluating the modified kernel, we note that our assumption can be easily eliminated by, for example, extending the API of `rwsem` to allow an additional pointer argument for read acquisition and release functions. In case the acquisition is made on the fast path, this pointer would be used to store the address of the corresponding slot; later, this pointer can be passed to a (different) releasing thread to specify the slot to be cleared. Alternatively, we can extend the API of `rwsem` to include a flag explicitly allowing the use of the fast path for read acquisition and release. This flag would be set only in call sites known for high read contention (such as in functions operating on VMAs), where a thread that releases the semaphore is known to be the one that acquired it. Other call sites for semaphore acquisition and release can be left untouched, letting them use the slow path only.

We note that the default configuration of the kernel enables a so-called spin-on-owner optimization of `rwsem` [32]. With this optimization, the `rwsem` structure includes an `owner` field that contains a pointer to the `current` struct of the owner task when `rwsem` is acquired for write. Using this field, a reader may check whether the writer is currently running on a CPU, and if so, spin rather than block [32]. While writers do not use this field to decide whether they have to spin (as there might be multiple readers), in the current `rwsem` implementation a reader updates the `owner` field regardless, storing there its `current` pointer along with a few control bits (that specify that the lock is owned by a reader). These writes by readers are for debugging purposes only, yet they create unnecessary contention on the `owner` field. We fix that by letting a reader set only the control bits in the `owner` field, and only if those bits were not set before, i.e., when the first reader acquires that `rwsem` instance after a writer. Note that all subsequent readers would read, but not update the `owner` field, until it is updated again by a writer.

[5]We determine the slot by hashing the task struct pointer (`current`) with the address of the semaphore.

5 User-space Evaluation

All user-space data was collected on an Oracle X5-2 system. The system has 2 sockets, each populated with an Intel Xeon E5-2699 v3 CPU running at 2.30GHz. Each socket has 18 cores, and each core is 2-way hyperthreaded, yielding 72 logical CPUs in total. The system was running Ubuntu 18.04 with a stock Linux version 4.15 kernel, and all software was compiled using the provided GCC version 7.3 toolchain at optimization level "-O3". 64-bit code was used for all experiments. Factory-provided system defaults were used in all cases, and *turbo mode* [45] was left enabled. In all cases default free-range unbound threads were used.

We implemented all 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 reader-writer locks. This same framework was used to implement Cohort reader-writer locks [6].

In the following figures "BA" refers to the Brandenburg-Anderson PF-Q lock [3]; "Cohort-RW" refers to the C-RW-WP lock [6]; "Per-CPU" reflects a lock that consists of an array of BA locks, one for each CPU, where readers acquire read-permission on the sub-lock associated with their CPU, and writers acquire writer permission on all the sub-locks (this lock is similar to the Linux kernel brlock construct [10]); "pthread" is a default Linux POSIX "pthread_rwlock" read-write lock mechanism; "BRAVO-BA" reflects BRAVO implemented on top of BA and "BRAVO-pthread" is BRAVO implemented on top of pthread_rwlock.

We also experimented with several other reader-writer locks. In particular, we took data on the Brandenburg-Anderson PF-T lock and the BRAVO form thereof. PF-T implements the reader indicator via a central pair of counters, one incremented by arriving readers and the other incremented by departing readers. Waiting readers busy-wait on a dedicated *writer present* bit encoded in the reader arrival counter. In PF-Q active readers are tallied on a central pair of counters in the same fashion as PF-T, but waiting readers enqueue on an MCS-like queue. In both PF-T and PF-Q, arriving readers update the central reader indicator state, generating more coherence traffic than would be the case for locks that use distributed reader indicators or BRAVO. Waiting readers in PF-T use global spinning, while waiting readers in PF-Q use local spinning on a thread-local field in the enqueued element. PF-T enjoys slightly shorter code paths but also suffers from lessened scalability because of the global spinning. We found that PF-T and PF-Q offer broadly similar performance, with PF-T having a slight advantage when the arrival rate is high, the number of waiting threads is low, and the waiting period is shorter. PF-T is slightly more compact having just 4 integer fields, while PF-Q has 2 such fields and 4 pointers. For brevity, we do not include PF-T results. We also found that "fair lock with local only spinning" by Mellor-Crummey and Scott [35] yielded performance similar to or slower than that of PF-Q.

We note that the default pthread read-write lock implementation found in our Linux distribution provides strong reader preference, and admits indefinite writer starvation[6]. The reader indicator is centralized and the lock has a footprint of 56 bytes for 64-bit programs. Waiting threads block immediately in the kernel without spinning. While this policy incurs overheads associated with voluntary context switching, it may also yield benefits by allowing "polite" waiting by enabling turbo mode for those threads making progress. Except where otherwise noted, we plot the number of concurrent threads on the X-axis, and aggregate throughput on the Y-axis, and report the median of 7 independent runs for each data point.

We use a 128 byte sector size on Intel processors for alignment to avoid false sharing. The unit of coherence is 64 bytes throughout the cache hierarchy, but 128 bytes is required because of the adjacent cache line prefetch facility where pairs of lines are automatically fetched together. BA requires just 128 bytes – 2 32-bit integer fields plus 4 pointers fields with the overall size rounded up to the next sector boundary. BRAVO-BA adds the 8-byte `InhibitUntil` field, which contains a timestamp, and the 4-byte `RBias` field. Rounding up to the sector size, this still yields a 128 byte lock instance. Per-CPU consists of one instance of BA for each logical CPU, yielding a lock size of 9216 bytes on our 72-way system. Cohort-RW consists of one reader indicator (128 bytes) per NUMA node, a central location for state (128 bytes) and a full cohort mutex [21] to provide writer exclusion. In turn, the cohort mutex requires one 128-byte sub-lock per NUMA node, and another 256 bytes for central state. Thus, the total size of the Cohort-RW lock on our dual-socket system is 896 bytes. (While our implementation did not do so, we note that a more space aggressive implementation of Cohort-RW could colocate the per-node reader indicators with the mutex sub-locks, and the central state for the read-write lock with its associated cohort mutex, yielding a size of 512 bytes).

[6]The pthread implementation allows writer preference to be selected via a non-portable API. Unfortunately this feature currently has bugs that result in lost wakeups and hangs: https://sourceware.org/bugzilla/show_bug.cgi?id=23861.

As noted above, the size of the pthread read-write lock is 56 bytes, and the BRAVO variant adds 12 bytes. The size of BA, BRAVO-BA, pthread, and BRAVO-pthread are fixed, and known at compile-time, while the size of Per-CPU varies with the number of logical CPUs, and the size of Cohort-RW varies with the number of NUMA nodes. Finally, we observe that BRAVO allows more relaxed approach toward the alignment and padding of the underlying lock. Since fast-path readers do not mutate the underlying lock fields, the designer can reasonably forgo alignment and padding on that lock, without trading off reader scalability.

The size of the lock can be important in concurrent data structures, such as linked lists or binary search trees, that use a lock per node or entry [4, 12, 25]. As Bronson at el. observe, when a scalable lock is striped across multiple cache lines to avoid contention in the coherence fabric, it is "prohibitively expensive to store a separate lock per node" [4].

BRAVO also requires the visible readers table. With 4096 entries on a system with 64-bit pointers, the additional footprint is 32KB. The table is aligned and sized to minimize the number of underlying pages (reducing TLB footprint) and to eliminate false sharing from variables that might be placed adjacent to the table. We selected a table size 4096 empirically but in general believe the size should be a function of the number of logical CPUs in the system. Similar tables in the linux kernel, such as the *futex* hash table, are sized in this fashion [5].

BRAVO yields a favorable performance trade-off between space and scalability, offering a viable alternative that resides on the design spectrum between classic centralized locks, such as BA, having small footprint and poor reader scalability, and the large locks with high reader scalability.

5.1 Sensitivity to Inter-Lock Interference

As the visible readers array is shared over all locks and threads within an address space, one potential concern is collisions that might arise when multiple threads are using a large set of locks. Near collisions are also of concern as they can cause false sharing within the array. To determine BRAVO's performance sensitivity to such effects, we implemented a microbenchmark program that spawns 64 concurrent threads. Each thread loops as follows: randomly pick a reader-writer lock from a pool of such locks; acquire that lock for read; advance a thread-local pseudo-random number generator 20 steps; release read permission on the lock; and finally advance that random number generator 100 steps. At the end of a 10 second measurement interval we report the number of lock acquisitions. No locks are ever acquired with write permission. Each data point is the median of 7 distinct runs. We report the results in Figure 1 where the X-axis reflects the number of locks in the pool (varying through powers-of-two between 1 and 8192) and the Y-axis is the number of acquisitions completed by BRAVO-BA divided by the number completed by a

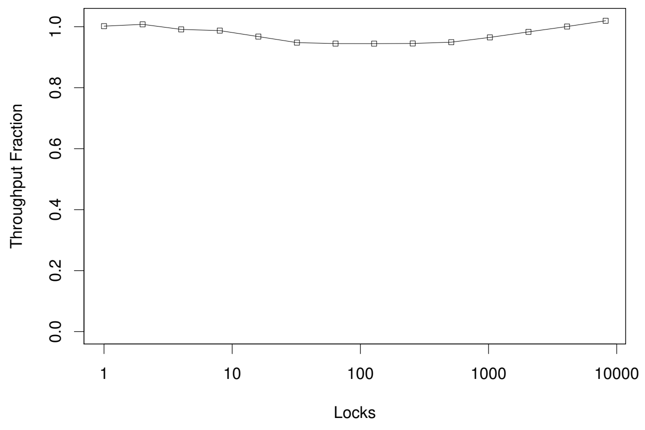


Figure 1: Inter-lock interference

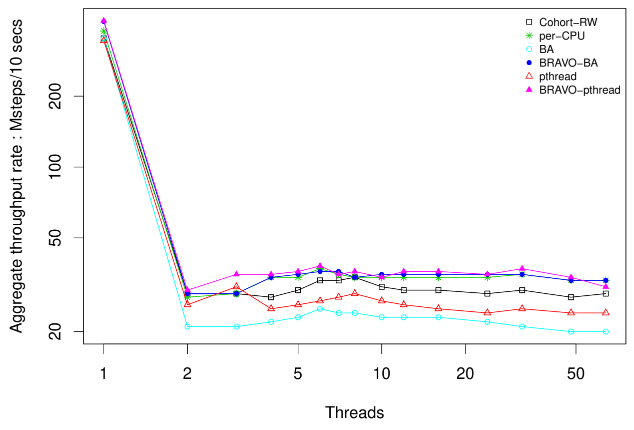


Figure 2: Alternator

specialized version of BRAVO-BA where each lock instance has a private array of 4096 elements. This fraction reflects the performance drop attributable to inter-lock conflicts and near conflicts in the shared array, where the modified form of BRAVO-BA can be seen as an idealized form that has a large per-instance footprint but which is immune to inter-lock conflicts[7]. The worst-case penalty arising from inter-lock interference (the lowest fraction value) is always under 6%.

5.2 Alternator

Figure 2 shows the results of our `alternator` benchmark. The benchmark spawns the specified number of concurrent threads, which organize themselves into a logical ring, each waiting for notification from its "left" sibling. Notification is accomplished via setting a thread-specific variable with a store instruction and waiting is via simple busy-waiting. Once notified, the thread acquires and then immediately releases read permission on a shared reader-writer lock. Next the thread notifies its "right" sibling and then again waits. There are no writers, and there is no concurrency between readers. At most one reader is active at any given moment. At the end of a 10 second measurement interval the program reports the number of notifications.

The BA lock suffers as the lines underlying the reader indicators "slosh" and migrate from cache to cache. In contrast BRAVO-BA readers touch different locations in the visible readers table as they acquire and release read permissions. BRAVO enables reader-bias early in the run, and it remains set for the duration of the measurement interval. All locks experience a significant performance drop between 1 and 2 threads due to the impact of coherent communication for notification. Crucially, we see that BRAVO-BA outperforms the underlying BA by a wide margin, and is competitive with the much larger Per-CPU lock. In addition, the performance of BA can be seen to degrade as we add threads, whereas the performance of BRAVO-BA remains stable. The same observations are true when considering BRAVO-pthread and pthread locks.

Since the hash function that associates a read locking request with an index is deterministic, threads repeatedly locking and unlocking a specific lock will enjoy temporal locality and reuse in the visible readers table.

5.3 test_rwlock

We next report results from the `test_rwlock` benchmark described by Desnoyers et al. [13][8]. The benchmark was designed to evaluate the performance and scalability of reader-writer locks against the RCU (Read-Copy Update) synchronization mechanism. We used the following command-line: `test_rwlock T 1 10 -c 10 -e 10 -d 1000`. The benchmark launches 1 fixed-role writer thread and T fixed-role reader threads for a 10 second measurement interval. The writer loops as follows: acquire a central reader-writer lock instance; execute 10 units of work, which entails counting down a local variable; release writer permission; execute a non-critical section for 1000 work units. Readers loop acquiring the central lock for reading, executing 10 steps of work in the critical section, and then release the lock. (The benchmark has no facilities to allow a non-trivial critical section for readers). At the end of the measurement interval the benchmark reports the sum of iterations completed by all the threads. As we can see in Figure 3, BRAVO-BA significantly outperforms BA, and even the Cohort-RW lock at higher thread counts. Since the workload is extremely read-dominated, the Per-CPU lock yields the best performance, albeit with a very

[7]We note that as we increase the number of locks, cache pressure constitutes a confounding factor for the specialized version of BRAVO-BA. For full discussion, see the extended version of this paper available at https://arxiv.org/abs/1810.01553.

[8]obtained from https://github.com/urcu/userspace-rcu/blob/master/tests/benchmark/test_rwlock.c and modified slightly to allow a fixed measurement interval.

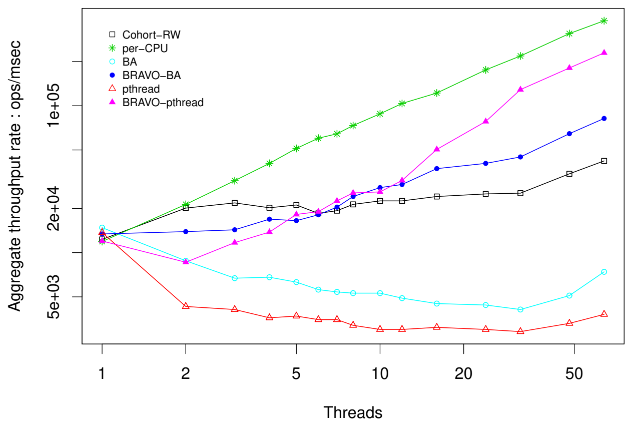


Figure 3: test_rwlock

large footprint and only because of the relatively low write rate. For that same reason, and due to its default reader preference, BRAVO-pthread easily beats pthread, and comes close to the performance level of Per-CPU.

5.4 RWBench

Using `RWBench` – modeled on a benchmark of the same name described by Calciu et al. [6] – we evaluated the reader-write lock algorithms over a variety of read-write ratios, ranging from write-intensive in Figure 4a (9 out of every 10 operations are writes) to read-intensive in Figure 4f (1 out of every 10000 operations are writes), demonstrating that BRAVO inflicts no harm for write-intensive workloads, but improves performance for more read-dominated workloads. `RWBench` launches T concurrent threads for a 10 second measurement interval. Each thread loops as follows: using a thread-local pseudo-random generator, decide to write with probability P via a Bernoulli trial; writers acquire a central reader-write lock for write permission and then execute 10 steps of a thread-local C++ `std::mt19937` random number generator and then release write permission, while readers do the same, but under read permission; execute a non-critical section of N steps of the same random-number generator where N is a random number uniformly distributed in $[0,200)$ with average and median of 100. At the end of the measurement interval the benchmark reports the total number of top-level loops completed.

In Figure 4a we see poor scalability over all the locks by virtue of the highly serialized write-heavy nature of the workload. Per-CPU fairs poorly as writes, which are common, need to scan the array of per-CPU sub-locks. Cohort-RW provides some benefit, while BRAVO-BA (BRAVO-pthread) tracks closely to BA (pthread, respectively), providing neither benefit nor harm. The same behavior plays out in Figure 4b ($P = 1/2$) and Figure 4c ($P = 1/10$), although in the latter we see some scaling from Cohort-RW. In Figure 4d ($P = 1/100$) we begin to see BRAVO-BA outperforming BA at higher thread counts. Figure 4e ($P = 1/1000$) and Figure 4f ($P = 1/10000$ – extremely read-dominated) are fairly similar, with BRAVO-BA and BRAVO-pthread yielding performance similar to that of Per-CPU, Cohort-RW yielding modest scalability, and BA and pthread yielding flat performance as the thread count increases.

5.5 `rocksdb` readwhilewriting

We next explore performance sensitivity to the reader-writer lock in the `rocksdb` database [40]. We observed high frequency reader traffic arising in the readwhilewriting benchmark from calls in `::Get()` to `GetLock()` defined in `db/memtable.cc`[9]. In Figure 5 we see the performance of BRAVO-BA and BRAVO-pthread tracks that of Per-CPU and always exceeds that of Cohort-RW and the respective underlying locks.

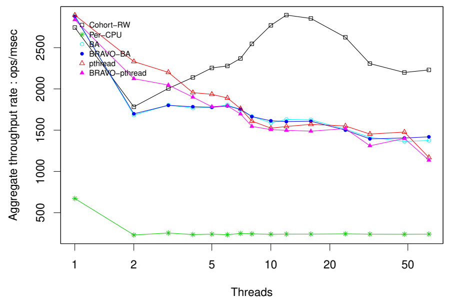
5.6 `rocksdb` hash_table_bench

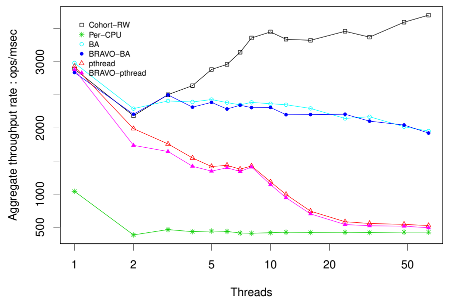
`rocksdb` also provides a benchmark to stress the hash table used by their persistent cache[10]. The benchmark implements a central shared hash table as a C++ `std::unordered_map` protected by a reader-writer lock. The cache is pre-populated before the measurement interval. At the end of the 50 second measurement interval the benchmark reports the aggregate operation rate – reads, erases, insertions – per millisecond. A single dedicated thread loops, erasing random elements, and another dedicated thread loops inserting new elements with a random key. Both erase and insertion operations require write access. The benchmark launches T reader threads, which loop, running lookups on randomly selected keys. We vary T on the X-axis. All the threads execute operations back-to-back without a delay between operations. The benchmark makes frequent use of malloc-free operations in the `std::unordered_map`. The default malloc allocator fails to fully scale in this environment and masks any benefit conferred by improved reader-writer locks, so we instead used the index-aware allocator by Afek el al. [1].

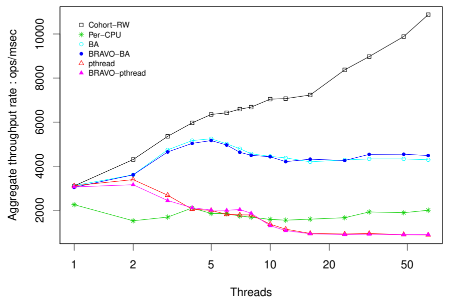
The results are shown in Figure 6. Once again, BRAVO enhances the performance of underlying locks, and shows substantial speedup at high thread counts.

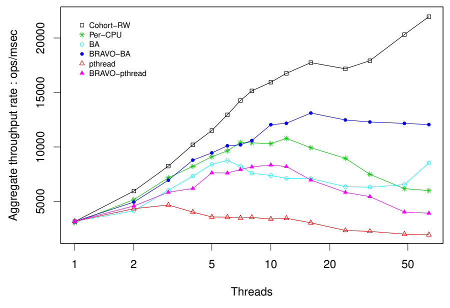
[9] We used `rocksdb` version 5.13.4 with the following command line: `db_bench -threads=T -benchmarks=readwhilewriting -memtablerep=cuckoo -duration=100 -inplace_update_support=1 -allow_concurrent_memtable_write=0 -num=10000 -inplace_update_num_locks=1 -histogram -stats_interval=10000000`

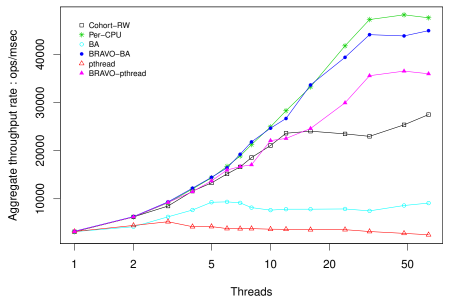
[10] `https://github.com/facebook/rocksdb/blob/master/utilities/persistent_cache/hash_table_bench.cc` run with the following command-line: `hash_table_bench -nread_thread=`T `-nsec=50`


(a) RWBench with 90% writes (9/10)


(b) RWBench with 50% writes (1/2)


(c) RWBench with 10% writes (1/10)


(d) RWBench with 1% writes (1/100)


(e) RWBench with .1% writes (1/1000)

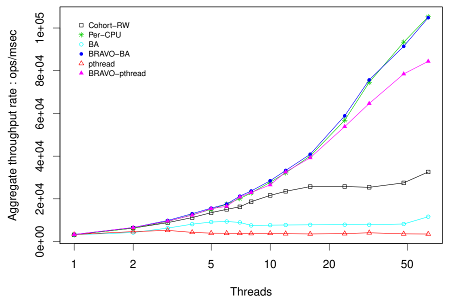

(f) RWBench with .01% writes (1/10000)

Figure 4: RWBench

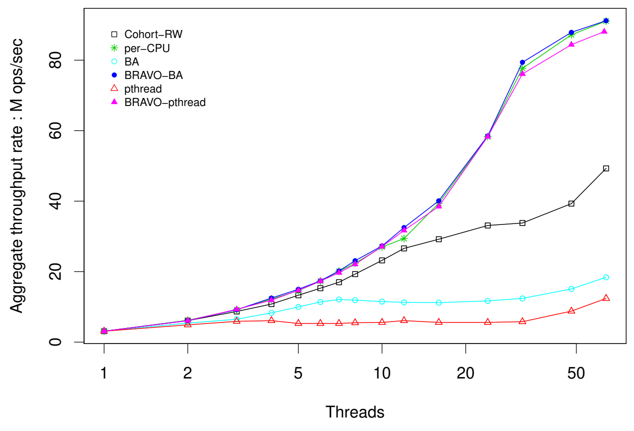


Figure 5: `rocksdb` readwhilewriting

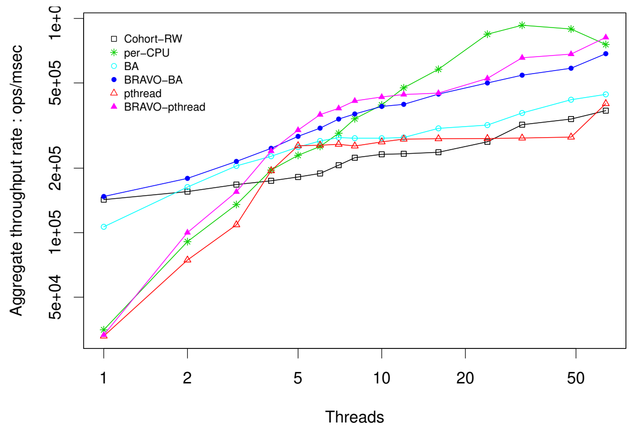


Figure 6: `rocksdb` hash_table_bench with `std::unordered_map`

6 Linux Kernel Experiments

All kernel-space data was collected on an Oracle X5-4 system. The system has 4 sockets, each populated with an Intel Xeon CPU E7-8895 v3 running at 2.60GHz. Each socket has 18 cores, and each core is 2-way hyperthreaded, yielding 144 logical CPUs in total. The patch was applied on top of a recent Linux version 4.20.rc4 kernel, which we refer to as *stock*. We refer to the kernel version modified to use BRAVO simply as *BRAVO*.

Factory-provided system defaults were used in all cases. In particular, we compiled the kernel in the default configuration, which notably disables the lock performance data collection mechanism (aka `lockstat`[11]) built into the kernel for debugging lock performance. As we mention below, this mechanism was useful to gain insights into usage patterns of kernel locks in various applications. However, we kept it disabled during

[11] https://www.kernel.org/doc/Documentation/locking/lockstat.txt

performance measurements as it adds a probing effect by generating stores into shared variables, e.g., by keeping track of the last CPU on which a given lock instance, `rwsem` included, was acquired. These stores hamper the benefit of techniques like BRAVO that aim to reduce updates to the shared state during lock acquisition and release operations.

Each experiment is repeated 7 times, and the reported numbers are the average of the corresponding results. Unless noted, the reported results were relatively stable, with variance of less than 5% from the average in most cases.

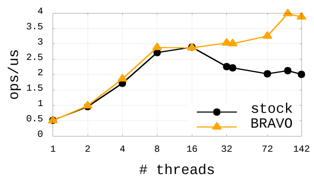
6.1 will-it-scale

`will-it-scale` is an open-source collection of microbenchmarks[12] for stress-testing various kernel subsystems. `will-it-scale` runs in user-mode but is known to induce contention on kernel locks [15]. Each microbenchmark runs a given number of tasks (that can be either threads or processes), performing a series of specific system calls (such as opening and closing a file, mapping and unmapping memory pages, raising a signal, etc.). We experiment with a subset of microbenchmarks that access the VMA structure and create contention on `mmap_sem`, an instance of `rwsem` that protects the access to VMA [11]. In particular, the relevant microbenchmarks are `page_fault` and `mmap`. The former continuously maps a large (128M) chunk of memory, writes one word into every page in the chunk (causing a page fault for every write), and unmaps the memory. The latter simply maps and unmaps large chunks of memory. (Each of those benchmarks has several variants denoted as `page_fault1`, `page_fault2`, etc.)

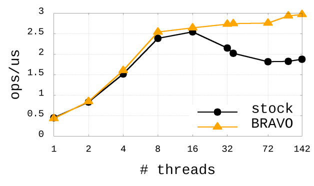
Page faults require the acquisition of `mmap_sem` for read, while memory mapping and unmapping operations acquire `mmap_sems` for write [8]. Therefore, the access pattern for `mmap_sem` is expected to be read-heavy in the `page_fault` microbenchmark and more write-heavy in `mmap`. We confirmed that through `lockstat` statistics. We note that BRAVO is not expected to provide any benefit for `mmap`, yet we include it to evaluate any overhead BRAVO might introduce in write-heavy workloads.

Figure 7 presents the results of our experiments for `page_fault` and `mmap`, respectively. In `page_fault`, the BRAVO version performs similarly to stock as long as the latter scales. After 16 threads, however, the throughput of the stock version decreases while the BRAVO version continues to scale, albeit at a slower rate. At 142 threads, BRAVO outperforms stock by up to 93%. At the same time, `mmap` shows no significant difference in the performance of BRAVO vs. stock, suggesting that BRAVO does not introduce overhead in scenarios where it is not profitable.

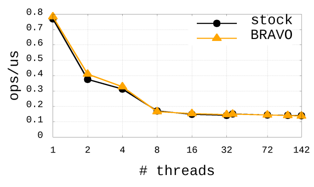
[12] https://github.com/antonblanchard/will-it-scale



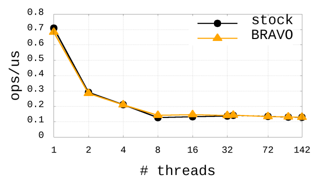
(a) page_fault1_threads



(b) page_fault2_threads



(c) mmap1_threads



(d) mmap2_threads

Figure 7: `will-it-scale` results

6.2 Metis

Metis is an open-source MapReduce library [33] used in the past to assess the scalability of Linux kernel locks [2, 8, 27]. Metis is known for a relatively intense access to VMA through the mix of page-fault and mmap operations [27]. By collecting lock performance statistics with `lockstat`, however, we found that only few of Metis benchmarks have both a large number of `mmap_sem` acquisitions and a large portion of those acquisitions is for read. We note that like in all other kernel benchmarks, `lockstat` was disabled when measuring performance numbers reported below.

Tables 1 and 2 present the performance results, respectively, for `wc`, a map-reduce based word count, and `wrmem`, which allocates a large chunk of memory and fills it with random "words", which are fed into the map-reduce framework for inverted index calculation. The BRAVO version can achieve speedups of over 30%. We note that some of the data, particularly for `wc`, was noisy; we print values with variance larger than 5% from the mean in italics. (All values have variance of 19% or less). We also note that BRAVO did not create significant overhead for any other Metis benchmark, although some benchmarks produced noisy results similarly to `wc`.

7 Conclusion and Future Work

BRAVO easily composes with existing locks, preserving desirable properties of those underlying locks, and yielding a composite lock with improved read-read scalability. We specifically target read-dominated workloads with multiple concurrent threads that acquire and release read permission at a high rate. The approach is simple, effective, and yields improved performance for read-dominated workloads compared to commonly used compact locks. The key trade-off inherent in the design is the benefit accrued by reads against the potential slow-down imposed by revocation. Even in mixed or

#threads	stock	BRAVO	speedup
1	*18.059*	17.957	0.6%
2	12.189	12.090	0.8%
4	10.045	*9.652*	3.9%
8	*8.880*	*7.976*	10.2%
16	*13.321*	11.325	15.0%
32	23.654	*19.281*	18.5%
72	*119.018*	98.054	17.6%
108	132.147	111.139	15.9%
142	143.217	125.160	12.6%

Table 1: `wc` runtime (sec)

#threads	stock	BRAVO	speedup
1	279.553	279.423	0.0%
2	137.271	136.706	0.4%
4	68.989	69.013	0.0%
8	36.161	36.210	-0.1%
16	22.647	21.985	2.9%
32	18.332	14.707	19.8%
72	*48.397*	31.868	34.2%
108	58.529	*36.910*	36.9%
142	58.994	42.544	27.9%

Table 2: `wrmem` runtime (sec)

write-heavy workloads, we limit any slow-down stemming from revocation costs and bound harm, making the decision to use BRAVO simple. BRAVO incurs a very small footprint increase per lock instance, and also adds a shared table of fixed size that can be used by all threads and locks. BRAVO's key benefit arises from reducing coherence cost that would normally be incurred by locks having a central reader indicator. Write performance is left unchanged relative to the underlying lock. BRAVO provides read-read performance at, and often above, that of the best modern reader-writer locks that use distributed read indicators, but without the footprint or complexity of such locks. By reducing coherence traffic, BRAVO is implicitly NUMA-friendly.

► **Future directions** We identify a number of future directions for our investigation into BRAVO-based designs:

- Dynamic sizing of the visible readers table based on collisions. Large tables will have reduced collision rates, but larger scan revocation overheads.
- The reader fast-path currently probes just a single location and reverts to the slow-path after a collision. We plan on using a secondary hash to probe an alternative location. In that vein, we note that while we currently use a hash function to map a thread's identity and the lock address to an index in the table, there is no particular requirement that the function that associates a read request with an index be deterministic. We plan on exploring other functions, using time or random numbers to form indices. While this will be less beneficial in terms of cache locality for the reader, it might be helpful in case of temporal contention over specific slots.
- Accelerate the revocation scan operation via SIMD instructions such as AVX. The visible readers table is usually sparsely populated, making it amenable to such optimizations. As already noted, non-temporal non-polluting loads may also be helpful for the scan operation.
- As noted, our current policy to enable bias is conservative, and leaves untapped performance. We intend to explore more sophisticated adaptive policies based on recent behavior and to use a more faithful cost model.
- An interesting variation is to implement BRAVO on top of an underlying mutex instead of a reader-writer lock. Slow-path readers must acquire the mutex, and the sole source of read-read concurrency is via the fast path. We note that some applications might expect the reader-write lock implementation to be fully work conserving and *maximally admissive* – always allowing full read concurrency where available. For example, an active reader thread $T1$, understanding by virtue of application invariants that no writers are present, might signal another thread $T2$ and expect that $T2$ can enter a reader critical section while $T1$ remains within the critical section. This progress assumption would not necessarily hold if readers are forced through the slow path and read-read parallelism is denied.
- In our current implementation arriving readers are blocked while a revocation scan is in progress. This could be avoided by adding a mutex to each BRAVO-enhanced lock. Arriving writers immediately acquire this mutex, which resolves all write-write conflicts. They then perform revocation, if necessary; acquire the underlying reader-vs-write lock with write permission; execute the writer critical section; and finally release both the mutex and the underlying reader-writer lock. The underlying reader-writer lock resolves read-vs-write conflicts. The code used by readers remains unchanged. This optimization allows readers to make progress during revocation by diverting through the reader slow-path, mitigating the cost of revocation. This also reduces variance for the latency of read operations. We note that this general technique can be readily applied to other existing reader-writer locks that employ distributed reader indicators, such as Linux's brlock [10].

Acknowledgments

We thank Shady Issa for useful discussions about revocation and the cost model. We also thank the anonymous reviewers and our shepherd Yu Hua for providing insightful comments.

References

[1] Yehuda Afek, Dave Dice, and Adam Morrison. Cache index-aware memory allocation. In *Proceedings of the International Symposium on Memory Management*. ACM, 2011.

[2] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of Linux scalability to many cores. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 1–16, 2010.

[3] B. B. Brandenburg and J. H. Anderson. Spin-based reader-writer synchronization for multiprocessor real-time systems. In *Real-Time Systems Journal*, 2010.

[4] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 257–268, 2010.

[5] Davidlohr Bueso. futexes and hash table collisions. https://blog.stgolabs.net/2014/01/futexes-and-hash-table-collisions.html, 2014. Accessed: 2019-05-14.

[6] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. NUMA-aware reader-writer locks. In *Proceedings of ACM PPoPP*, pages 157–166. ACM, 2013.

[7] Bryan Cantrill and Jeff Bonwick. Real-world concurrency. *ACM Queue*, 6(5):16–25, 2008.

[8] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using RCU balanced trees. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 199–210, 2012.

[9] Jonathan Corbet. Driver porting: mutual exclusion with seqlocks. http://lwn.net/Articles/22818, 2003. Accessed: 2018-04-20.

[10] Jonathan Corbet. Big reader locks. https://lwn.net/Articles/378911, 2010. Accessed: 2018-04-20.

[11] Jonathan Corbet. The LRU lock and mmap_sem. https://lwn.net/Articles/753058, 2018. Accessed: 2019-01-10.

[12] Tyler Crain, Vincent Gramoli, and Michel Raynal. A speculation-friendly binary search tree. In *Proceedings of ACM PPoPP*. ACM, 2012.

[13] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems*, 2012.

[14] Dave Dice. Biased locking in HotSpot. https://blogs.oracle.com/dave/biased-locking-in-hotspot, 2006.

[15] Dave Dice and Alex Kogan. Compact NUMA-aware locks. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2019.

[16] Dave Dice, Alex Kogan, and Yossi Lev. Refined transactional lock elision. In *Proceedings of ACM PPoPP*, 2016.

[17] Dave Dice, Yossi Lev, and Mark Moir. Scalable statistics counters. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2013.

[18] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[19] Dave Dice and Nir Shavit. TLRW: Return of the read-write lock. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2010.

[20] David Dice, Danny Hendler, and Ilya Mirsky. Lightweight contention management for efficient compare-and-swap operations. In *Proceedings of the International Conference on Parallel Processing (EuroPar)*. Springer-Verlag, 2013.

[21] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: A general technique for designing NUMA locks. *ACM Trans. Parallel Comput.*, 2015.

[22] David Dice, Mark Moir, and William N. Scherer III. Quickly reacquirable locks – US Patent 7,814,488, 2002.

[23] William B. Easton. Process synchronization without long-term interlock. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1971.

[24] Pascal Felber, Shady Issa, Alexander Matveev, and Paolo Romano. Hardware read-write lock elision. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2016.

[25] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *Proceedings of the 9th*

International Conference on Principles of Distributed Systems, OPODIS'05. Springer-Verlag, 2006.

[26] W. C. Hsieh and W. E. Weihl. Scalable reader-writer locks for parallel systems. In *Proceedings Sixth International Parallel Processing Symposium*, 1992.

[27] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Scalable NUMA-aware blocking synchronization primitives. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[28] Andi Kleen. Lock elision in the GNU C library. https://lwn.net/Articles/534758, 2013. Accessed: 2019-05-13.

[29] Christoph Lameter. Effective synchronization on Linux/NUMA systems. In *Gelato Conference*, 2005.

[30] Yossi Lev, Victor Luchangco, and Marek Olszewski. Scalable reader-writer locks. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2009.

[31] Ran Liu, Heng Zhang, and Haibo Chen. Scalable read-mostly synchronization using passive reader-writer locks. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2014.

[32] Waiman Long. locking/rwsem: Enable reader optimistic spinning. https://lwn.net/Articles/724384/, 2017. Accessed: 2019-01-24.

[33] Yandong Mao, Robert Morris, and Frans Kaashoek. Optimizing MapReduce for multicore architectures. Technical report, MIT, 2010.

[34] George Marsaglia. Xorshift rngs. *Journal of Statistical Software, Articles*, 2003.

[35] John M. Mellor-Crummey and Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of ACM PPoPP*, 1991.

[36] Oracle. Api documentation for java.util.concurrent.locks.stampedlock. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/StampedLock.html, 2012.

[37] Filip Pizlo, Daniel Frampton, and Antony L. Hosking. Fine-grained adaptive biased locking. In *Proceedings of the International Conference on Principles and Practice of Programming in Java (PPPJ)*, 2011.

[38] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2001.

[39] Andreia Craveiro Ramalhete and Pedro Ramalhete. Distributed cache-line counter scalable RW-lock. http://concurrencyfreaks.blogspot.com/2013/09/distributed-cache-line-counter-scalable.html, 2013.

[40] rocksdb.org. A persistent key-value store for fast storage environments. rocksdb.org, 2018.

[41] Kenneth Russell and David Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2006.

[42] Jun Shirako, Nick Vrvilo, Eric G. Mercer, and Vivek Sarkar. Design, verification and applications of a new read-write lock algorithm. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2012.

[43] Guy L. Steele, Jr., Doug Lea, and Christine H. Flood. Fast splittable pseudorandom number generators. In *Proceedings of the ACM Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 453–472, 2014.

[44] N. Vasudevan, K. S. Namjoshi, and S. A. Edwards. Simple and fast biased locks. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[45] U. Verner, A. Mendelson, and A. Schuster. Extending Amdahl's law for multicores with turbo boost. *IEEE Computer Architecture Letters*, 2017.

[46] D. Vyukov. Distributed reader-writer mutex. http://www.1024cores.net/home/lock-free-algorithms/reader-writer-problem/distributed-reader-writer-mutex, 2011.

[47] Wikipedia contributors. Ski rental problem — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Ski_rental_problem&oldid=813551905, 2017. [Online; accessed 8-August-2018].

[48] Wikipedia contributors. Birthday problem — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Birthday_problem&oldid=853622452, 2018. [Online; accessed 8-August-2018].

[49] Wikipedia contributors. Loss aversion. https://en.wikipedia.org/wiki/Loss_aversion, 2018.