Lease/Release: Architectural Support for Scaling Contended Data Structures

Syed Kamran Haider
University of Connecticut, USA *

Dan Alistarh
Microsoft Research, UK

Abstract

High memory contention is generally agreed to be a worst-case scenario for concurrent data structures. There has been a significant amount of research effort spent investigating designs which minimize contention, and several programming techniques have been proposed to mitigate its effects. However, there are currently few architectural mechanisms to allow scaling contended data structures at high thread counts.

In this paper, we investigate hardware support for scalable contended data structures. We propose Lease/Release, a simple addition to standard directory-based MESI cache coherence protocols, allowing participants to lease memory, at the granularity of cache lines, for a *short, bounded* period of time. Our analysis shows that Lease/Release can significantly reduce the overheads of contention for both non-blocking (lock-free) and lock-based data structure implementations, while ensuring that no deadlocks are introduced. We validate Lease/Release empirically on the Graphite multiprocessor simulator, on a range of data structures, including queue, stack, and priority queue implementations, as well as on transactional applications. Results show that Lease/Release consistently improves both throughput and energy usage, by up to 5x, both for lock-free and lock-based data structure designs.

1. Introduction

The last decade has seen a tremendous amount of research effort dedicated to designing and implementing concurrent data structures which are able to *scale*, that is, to increase their performance as more parallelism becomes available. Consequently, efficient concurrent variants have been proposed for most classic data structures, such as lists, e.g. [16, 26], hash tables, e.g. [6, 20, 26], skip lists, e.g. [14, 20], search trees, e.g. [10, 11, 31], queues [27], stacks [1, 40, 42], or priority queues, e.g. [3, 23].

One key principle for data structure scalability is avoiding *contention*, or *hotspots*, roughly defined as data items accessed concurrently by large numbers of threads. While for many *search* data structures, such as hash tables or search trees, it is possible to avoid contention and scale [6] thanks to their "flat" structure and relatively uniform access patterns, it is much harder to avoid hotspots in the case of data structures, such as queues, stacks, or priority queues. In fact, theoretical results [2, 12] suggest that such data structures may be *inherently contended*: in the worst case, it is impossible to avoid hotspots when implementing them, without relaxing their semantics.

Several software techniques have been proposed to mitigate the impact of contention, such as combining [18], elimination [1, 40], relaxed semantics [3, 19, 32], back-offs [7], or data-structure and architecture-specific optimizations [13, 29]. While these methods can be very effective in improving the performance of individual data structures, the question of maximizing performance under contention on current architectures is still a major challenge.

Contribution. In this paper, we investigate an alternative approach: providing *hardware support* for scaling concurrent data structures under contention. We propose Lease/Release, a simple addition to standard directory-based MESI cache coherence protocols, allowing a thread to lease memory, at the granularity of cache lines, for a *short, bounded* period of time, delaying incoming coherence requests for the lines during this interval. Our analysis shows that Lease/Release can significantly reduce the overheads of contention for both non-blocking (lock-free) and lock-based data structure implementations, while ensuring that no deadlocks are introduced. Our mechanism allows a core to lease either *single* or *multiple* cache lines at the same time, while preserving deadlock-freedom. We validate Lease/Release empirically on the Graphite multi-processor simulator [28], on a range of data structures, including queue, stack, and priority queue implementations, as well as on contended real-world applications. Results show that Lease/Release can improve throughput and energy by up to 5x for contended lock-free and lock-based programs.

The idea of leasing to mitigate contention has been explored before in the systems and networking literature, e.g. [33]. For cache coherence, references [35, 39], covered in detail in the next section, proposed transient delay mechanisms for single memory locations in the context of the Load-Linked/Store-Conditional (LL/SC) primitives, focusing on lock-based (blocking) data structures. By contrast, we propose a more general leasing mechanism which applies to both blocking and non-blocking concurrency patterns, and to a wider range of primitives. Moreover, we investigate multi-line leasing, and show that leases can significantly improve the performance of classic data structure designs.

An Example. To illustrate the ideas behind Lease/Release, let us consider Treiber's venerable stack algorithm [42], outlined in Figure 1, as a toy example. We start from a sequential design. To push a new node onto the stack, a thread first reads the current `head`, points its node's `next` pointer to it, and then attempts to `compare-and-swap (CAS)` the `head` to point to its new node, expecting to see the old `head` value. Under high concurrency, we can expect to have several threads contending on the cache line corresponding to the `head` pointer, both for reads and updates. At the level of cache coherence, the thread must first obtain *exclusive* ownership of the corresponding line. At this point, due to contention, its operation is likely to be delayed by concurrent ownership requests for the same line. Further, by the time the thread manages to re-

* Work performed while an intern at Microsoft Research Cambridge, UK

```
1: function STACKPUSH( Node *node )
2:     loop
           ▷ Take the lease on the head pointer
3:         Lease( & Head )
4:         h ← Head
5:         node→next ← h
6:         success = CAS ( & Head, node→next, node )
           ▷ Release the head pointer
7:         Release( &Head )
8:         if success then return
```

Figure 1: Illustration of leases on the Stack push operation. We lease the head pointer for the duration of the read-CAS interval. This ensures that the CAS validation is always successful, unless the lease on the corresponding line expires.

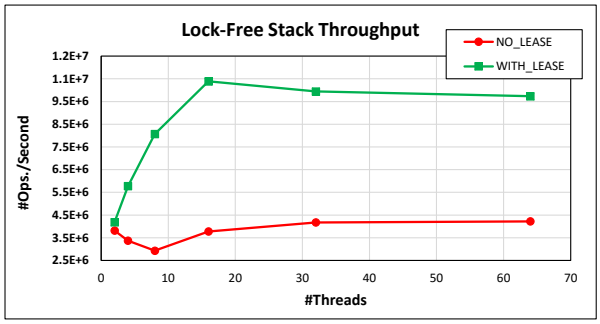

Figure 2: Throughput of the lock-free Treiber stack with and without leases, for 100% updates. The data points are at powers of 2.

obtain exclusive ownership, the memory value *may have changed*, which causes the CAS operation to fail, and the whole operation to be retried. The impact of contention on the performance of the data structure is illustrated in Figure 2.

Lease/Release is based on the following principle: *each time a core gets ownership of the contended line, it should be able to perform useful work*. We provide a `lease` instruction, which allows the core corresponding to the thread to *delay* incoming ownership requests on a line it owns for a *bounded* time interval. An incoming ownership request is queued at the core until either the line is released voluntarily by the thread (via a symmetric `release` instruction) or until the lease *times out*.[1] Crucially, the maximum time for which a lease can be held is upper bounded by a system-wide constant. This ensures that the Lease/Release mechanism may not cause deadlocks.

Returning to the stack example, notice that the natural point to set the lease on the head pointer is before the read on line 4 of Figure 1. In our implementation, this populates a *lease table* at the core with an entry corresponding to the cache line and the lease timeout. On the next instruction accessing that line, the core will automatically perform an exclusive request, and start the lease as soon as ownership is granted. The natural release point is after the (probably successful) CAS operation. In the common case when the length of the read-CAS pattern does not exceed the lease interval, any incoming ownership request gets queued until the release, allowing the thread to complete its operation without delay. Please see Figure 2 for the relative throughput improvement.

Non-Blocking Patterns. We have investigated lease usage for a wide range of data structures. First, we found that most *non-blocking* data structures have natural points where leases should be inserted. Specifically, many non-blocking update operations are based on a optimistic scan-and-validate pattern, usually mapping to a load, followed by a later read-modify-write instruction on the same cache line. Thus, it is natural to lease the corresponding line on the scan, releasing it after the read-modify-write. We provide detailed examples in the later sections.

Lock-Based Patterns. It is interesting to examine how leasing can help in the case of contended *lock-based* data structures. Consider the case of a single highly-contended lock variable, implemented via a standard `test&test&set (TTS)` pattern. A thread p first acquiring the lock incurs a cache miss when first gaining ownership of the corresponding cache line. While executing the critical section, since the lock is contended, it is likely that the core corresponding to thread p loses ownership of the lock's cache line. This will induce unnecessary overhead for thread p when releasing the lock (since it needs to re-gain ownership), but also generates useless traffic between cores (since threads getting the line will not be able to acquire the lock).

Leasing the line corresponding to the lock at p's core for the duration of the critical section eliminates both of these overheads. First, the lock holder will not incur a second cache miss when releasing the lock, as it retains ownership. Second, looking at other threads requesting the lock, the first such thread (in the order of request arrival at the directory) will be queued at p's core, waiting for the line to be released. All the other threads will be queued at the directory. Thus, the communication cost of the critical section is significantly reduced, and the lease mechanism ensures an efficient "sequentialization" through a contended critical section.

We note that several software [5, 8, 24, 25] and hardware [15, 21, 36] solutions for reducing the communication overhead of contended locks are known, and that similar behaviour will occur for locks implemented using LL/SC with IQOLB [35].

Multiple Leases. In more complex data structures, such as trees, or in software transactional memory (STM), it appears beneficial to be able to lease a small set of cache lines (corresponding to a set of nodes, or to a set of locks), at the same time. We show that the lease mechanism can be extended to allow a core to lease multiple cache lines for a fixed interval, while still ensuring that no deadlocks occur.

A closer examination of this multi-line mechanism for several classic concurrent patterns leads to the following conclusions. The first is perhaps surprising: for many data structures, multiple leases are *not necessary*, and may in fact introduce unnecessary overhead. The intuition is that, in the case of concurrent lists or trees, data access is often *linear*: it is sufficient to lease only a predecessor node to prevent access to its successors. In turn, this ensures that a sequence of operations involving the predecessor and a small set of successors will probably succeed if the predecessor is leased.

The second observation is that, in scenarios where the operation requires ownership of two or more arbitrary locations, such as in transactional or multi-word CAS semantics, joint leases on these locations can be very beneficial for performance.

Experimental Results. We implemented the Lease/Release mechanism in the Graphite multi-processor simulator [28], and experimented with several fundamental data structures, such as queues, stacks, priority queues, hash tables and binary trees, as well as locks, and transactional algorithms. We found that using leases on top of contended data structures such as queues, stacks, and priority queues, can improve throughput by up to 5x. Using single leases, the relatively simple classic data structure designs such as the Treiber stack match or even improve the performance of highly

[1] It is important to note that, in current directory-based cache coherence protocols, given multiple concurrent requests on the same line, only *one* may reach the owning core, while the others are queued at the directory, and are processed serially.

optimized, complex implementations. Similarly, multi-leases improve the throughput of transactional algorithms which need to jointly acquire small sets of objects, by up to 5x. In scenarios with no contention, leases do not affect overall throughput in a discernible way; for low contention data structures (such as hash tables and trees), improvements are more modest ($\leq 5\%$).

An notable benefit of using leases is *reduced energy usage*. Specifically, leases reduce coherence traffic by up to 5x; consequently, energy consumption is reduced by a similar amount, as modeled by our simulator.

Summing up, one can split the cost of concurrent operations into *sequential* cost (time spent executing operations locally), *traffic* cost (time spent waiting for coherence requests), and *retry* cost (time spent because of failed lock acquisitions or CAS operations). As illustrated above, leases allow the programmer to minimize both *traffic* and *retry* costs for both lock-based and lock-free programming patterns.

Roadmap. The rest of the paper proceeds as follows. We cover related work in Section 2. We specify detailed semantics for both single-line and multi-line Lease/Release, as well as implementation details, in Section 3. We cover detailed usage examples in Section 4, and provide empirical validation in Section 5. We discuss our findings in Section 6.

2. Related Work

We focus on software techniques and hardware mechanisms for mitigating contention in concurrent data structures. Several software methods have been proposed to build efficient contended data structures. For instance, the *elimination* technique [40] proposes to directly match producer and consumer operations, such as `push` and `pop` for a stack, as a way to avoid memory hotspots, and has found several applications [1, 40]. *Combining* proposes to reduce the contention overhead of data structures by "shipping" operations directly to a chosen thread (the "combiner") which can apply them to a local version of the data structure, taking advantage of data-structure-specific optimizations [18]. Several data structure designs, e.g. [3, 19, 32, 37], aim to avoid hotspots by *relaxing ordering semantics*: for instance, by returning a top-k element instead of the absolute top.

Finally, for almost all fundamental data structures, implementations exist which achieve good performance through careful data-structure-specific or architecture-specific design. This is the case for queues [13, 27, 29], stacks [1, 42], and priority queues [23]. In our simulation, we obtain scalability trends comparable or exceeding those of highly optimized implementations of these data structures by just adding leases to the classic designs of Treiber [42], Michael–Scott [27], and Lotan–Shavit [23], respectively.

Lock cohorting [8] is a software mechanism for improving the performance of lock-based synchronization in NUMA systems, optimizing the average "travel time" of a lock by assigning ownership in a topology-aware fashion. Leases do not interfere with the pattern of lock ownership, and are hence compatible with cohorting.

Several hardware mechanisms have been proposed to simplify the design of scalable data structures. Perhaps the best known is hardware transactional memory (HTM). Current HTM implementations appear to suffer from high abort rates under contention [30], and are probably not good candidates for contended data structure implementations. QOLB (also known as QOSB) [15, 21] is a hardware queue mechanism for efficiently executing a contended critical section, similar in spirit to a queue lock. QOLB has been shown to speed up contended lock-based applications by up to an order of magnitude [35], but requires complex protocol support, new instructions, and re-compilation of the application code [35].

Implicit QOLB (IQOLB) [35] is a technique which delays servicing requests on lock variables for a finite time, to reduce both the overhead on the lock holder and the overall communication cost under contention. The delay induces an *implicit* queue of requests, as described in the previous section. For lock-based programs, the use of IQOLB is virtually identical to the use of leases on the lock variable, that is, Lease/Release can be used to implement IQOLB. Reference [35] implements IQOLB via LL/SC-based locks, by changing the LL instruction automatically to a *deferrable ownership request*. IQOLB was shown to improve performance by up to an order of magnitude in contended workloads on the SPLASH-02 benchmark (within 1% of QOLB), and introduces no new instructions, but requires hardware structures for predictors, and a misspeculation recovery mechanism.

Compared to QOLB and IQOLB, Lease/Release introduces new instructions, but allows for more flexibility, as it can be used in both lock-based and lock-free patterns. Lease/Release does not require predictor or recovery structures, although it could benefit from such mechanisms. Further, Lease/Release allows multiple concurrent leases. We also examine the applicability of this mechanism on a range of programming patterns.

In [39], Shalev and Shavit propose similar transient blocking semantics in the context of snoopy cache coherence, to implement `Load&Lease` and `Store&Unlease` instructions, and discuss the interaction between this mechanism and hardware transactional memory. Lease/Release provides slightly more general semantics, and also allows for multiple leases. Further, we illustrate and evaluate leases in the context of modern data structures.

Several protocols have been recently proposed as alternatives to MESI-based coherence, e.g. [4, 43], which show significant promise in a range of scenarios. By comparison, Lease/Release has a narrower focus, but is compatible with current protocols, and would, arguably, have lower implementation costs.

3. Memory Leases

3.1 Single-Line Memory Leases

The single-line leasing mechanism consists of two instructions: `Lease(addr, time)`, which leases the cache line corresponding to the address `addr` for `time` consecutive core cycles, and `Release(addr)`, which voluntarily releases the address. Further, the system defines a constant `MAX_LEASE_TIME`, which is an upper bound on the maximum length of a lease, and `MAX_NUM_LEASES`, an upper bound on the maximum number of leases that a core may hold at any given time. The core also maintains a *lease table* data structure, with the rough semantics of a key-value store, where each key is associated with a (decreasing) time counter and with a boolean flag. The pseudocode for these operations is given in Algorithm 1.

Specifically, whenever a `Lease(addr, time)` instruction is first encountered on address `addr`,[2] the system creates a new entry corresponding to the cache line in the lease table. It populates it with a `started` bit, initially set to `false`, and starts the corresponding counter with length $\min($`time`$,$ `MAX_LEASE_TIME`$)$. On the next core access on the line, which is expected to directly follow the lease instruction, the core requests the line in *Exclusive* state from the directory. When the request is filled, the core sets the `started` bit in the lease table to `true`, which starts the decrementing counter.

Upon an incoming coherence probe on an address `req`, the core scans the lease table for lines matching `req`. If a match is found and

[2] We do not allow extending leases on an already-leased address, as this could break the `MAX_LEASE_TIME` bound. Also, leases which would exceed `MAX_NUM_LEASES` are not serviced.

Algorithm 1 Single-Line Lease/Release Pseudocode.

```
1:  function LEASE(addr, time)
        ▷ Check if lease is valid
2:      found ← Lease-Table.find( addr )
3:      num ← Lease-Table.num_elements( )
4:      if (!found) and (num ≤ MAX_NUM_LEASES − 1) then
5:          time ← min( time, MAX_LEASE_TIME )
6:          started ← false
7:          Lease-Table[addr] = [ started, time ]

8:  upon event ACCESS(addr) do
9:      found ← Lease-Table.find( addr )
10:     if found and Lease-Table[addr].started = false then
11:         Request line corresponding to addr in Exclusive state
        ▷ Upon receipt
12:         Lease-Table[addr].started = true

13: function RELEASE(addr)
        ▷ Same procedure called when a lease counter expires
        ▷ Clear entry corresponding to the line
14:     Lease-Table.delete( addr )
        ▷ A single queued request may exist per line
15:     req ← coherence requests queued for this address
16:     if req then
17:         Fulfill req as per the cache coherence protocol
18: upon event CLOCK-TICK do
19:     for Each addr in Lease-Table with started = true do
20:         Decrement counter down to 0
21: upon event ZERO-COUNTER(addr) do
22:     RELEASE( addr )
23: upon event COHERENCE-PROBE(addr) do
24:     found = Lease-Table.find( addr )
25:     if found then Queue probe until lease on addr expires
```

Algorithm 2 MultiLease/MultiRelease Pseudocode.

```
1:  function MULTILEASE(num, time, addr1, addr2, ...)
        ▷ Check if lease is valid
2:      count ← Lease-Table.num_elements( )
3:      time ← min( time, MAX_LEASE_TIME )
4:      group_id ← unique group id
5:      if (count + num ) ≤ MAX_NUM_LEASES) then
        ▷ Lease all addresses with the same group id
6:          for each addr in list do
7:              LEASE(addr, time, group_id) ▷ Extend to include group_id
8:  upon event ACCESS(addr) do
9:      found ← Lease-Table.find( addr )
10:     if found and Lease-Table[addr].started = false then
11:         addr_list = Addresses with the group_id, in sorted order
12:         for each addr in addr_list do
13:             Request addr in Exclusive state
        ▷ Upon receipt
14:         Lease-Table[addr].started ← true
15: function RELEASEALL(addr)
        ▷ Clear entries corresponding all lines in the group
16:     addr_list = Addresses with the same group_id
17:     for each addr in addr_list do
18:         Lease-Table.delete( addr )
19:     for each addr in addr_list do
        ▷ A single queued request may exist per line
20:         Service any outstanding coherence requests for addr
```

the corresponding counter value is positive, the request is queued at the core. Otherwise, the request is serviced as usual. Upon every tick, the counter values corresponding to all *started* leases are decremented, until they reach zero. When this counter reaches zero, we say an *involuntary* release occurred. If the core calls `release` before this event, we say a *voluntary* release occurred.

In either case, the core preforms the following actions: it deletes the entry in the lease table, looks for any queued requests on the line, and fulfills them by downgrading its ownership and sending messages as specified by the cache coherence protocol.

3.2 Properties of Single-Line Leases

We now state some of the properties of the Lease/Release mechanism described above. First, we notice that, by the semantics of directory-based cache coherence protocols [41], only a single request for each line may be queued at a core. This bounds the number of queued requests per core by `MAX_NUM_LEASES`.

Proposition 1. *At any point during the execution of the protocol, a core may have a single outstanding request queued per each line.*

Proof. Recall that directory-based protocols [41] queue multiple requests for each line at the directory (in FIFO order). A request is not serviced by the directory until its predecessors in the queue are fully serviced. Therefore, at any point in time, at most a single request for a line may be queued at a core, while all other requests are queued at the directory. □

Next, since the lease interval is bounded, Lease/Release cannot introduce deadlocks.

Proposition 2. *The single-line Lease/Release mechanism does not introduce deadlocks.*

Proof. Consider an arbitrary deadlock-free application, which we run with Lease/Release. The proof relies on two claims. The first is that the running time of the application is bounded by its worst-case running time in the lease-free case, multiplied by `MAX_LEASE_TIME`. This follows since, at each step, in the worst case, each fulfilled probe is delayed by `MAX_LEASE_TIME` additional time steps. Next, assume for contradiction that a deadlocked execution, having infinite running time, exists in the application using leases. The first claim, bounding the running time, implies that the *lease-free* version also has infinite running time, which leads to a contradiction with our original deadlock-free assumption. □

3.3 Multi-Line Memory Leases

The multi-line leasing mechanism gives a way for a core to *jointly* lease a set of memory locations for a fixed, bounded period of time. It consists of two instructions: first, `MultiLease( num, time, addr1, addr2, ... )` defines a joint lease on `num` addresses, for an interval of `time` cycles. More precisely, `MultiLease` defines a *group* of addresses which will be leased together as soon as one is accessed. Second, when `MultiRelease( addr )` is called on one address in the group, all leases on addresses in the group are

released at the same time. The pseudocode for these instructions is given in Algorithm 2. Importantly, we assume and enforce that each address corresponds to a *distinct* cache line.

The procedure uses the same `Lease-Table` data structure as for single leases, with the addition of a `group_id` field. The `MultiLease` procedure simply performs `Lease` calls on each of the addresses in the argument list, using the same `time` and `group_id`. Recall that this procedure does not actually start the lease. This occurs on the first access by the core on a line in the group, which is expected to follow the lease instruction.

On this event, the following occurs. We sort the addresses in the group according to some fixed, global comparison criterion. We then request Exclusive ownership for these addresses *in sorted order*, setting the *started* bit for each in the `Lease-Table` once ownership is granted. Notice that the *fixed global order* of ownership requests is critical: otherwise, if two cores request the same two lines A and B in *inverted* orders, the system might deadlock since the core delays incoming ownership requests during the lease acquisition phase. In this way, the acquisition time is bounded, although it may increase because of concurrent leases. The rest of the events are identical to single-line leases, and are therefore omitted.

Software-Only Implementation. MultiLease could be implemented in software, with relaxed semantics, on top of a hardware single-lease mechanism. This could be done by requesting leases in a fixed order, and adjusting lease timeouts.

3.4 Properties of Multi-Line Memory Leases

The key property of the multi-lease mechanism is that it ensures deadlock-freedom.

Proposition 3. *The multi-line Lease/Release mechanism does not introduce deadlocks.*

Proof Sketch. We split the proof into two claims. First, let us assume that lines 12–14 of the MultiLease protocol occur atomically. This assumption implies that the core may not hold ownership of one cache line in `addr_list`, but not another, while receiving a coherence message. Notice that, at this point, the proof of deadlock-freedom in this case reduces to the single-line argument, which concludes this case.

To complete the proof, we need to show that deadlocks may not occur while cores are requesting ownership in lines 12–14 of the protocol. The key observation behind this claim is that, since lines are requested in a fixed order, it is impossible for a dependency cycle to occur at this level of the protocol. This argument is folklore: it is sketched by several references in the context of transactional contention management, e.g. [9], and is given in full in [22]. □

3.5 Hardware Implementation

An actual hardware implementation of Lease/Release can significantly simplify the above algorithmic description by leveraging the existing L1 cache controller logic and tag array. In particular, the `started` and `group_id` fields of the `Lease-Table` can be implemented by reserving $1 + \log_2(\texttt{MAX\_NUM\_LEASES})$ bits in the L1 cache tag array. Further, for the `time` field, an array of `MAX_NUM_LEASES` down-counters is required where each counter is $\log_2(\texttt{MAX\_LEASE\_TIME})$ bits wide. Whenever a lease request is served by the memory system and the corresponding cache line is stored in the L1 cache, the `started` bit of this cache line in the tag array is set and an *available* counter from the counter array is allocated to this leased cache line. The index of the corresponding counter is stored in the `group_id` field in the tag array, which is later used to link the counter to the cache line. The counter counts down the clock cycles starting from an initial value equivalent to the the requested lease time. This design optimizes for space, at the cost of decrementing counters in every cycle. A more time-efficient, but less space-efficient, alternative would be to simply store the lease expiry time for each lease, instead of the counter.

For multi-line leasing, the lease time of all the addresses in the group starts and ends at the same time and hence only one counter is required per group lease. This can be accessed by the `group_id` of the corresponding group lease. For a leased cache line, if the core clock value has not yet reached the lease expiration time (i.e. the corresponding counter is not zero), we can now hold off (or NACK—force a retry at the requestor) any probe to this address. When the core clock finally reaches the expiration time or if a voluntarily release is issued by the core itself before the expiration time, the `started` bit is cleared, the corresponding counter becomes available for subsequent leases, and any outstanding probes for this address (at most one, by Proposition 1) can be honored.

4. Detailed Examples

In this section, we illustrate the Lease/Release mechanism through some examples. We start with a simple scenario, using single-line leases to reduce coherence traffic in the context of try-locks. We then illustrate single-line leases in the context of the classic Michael-Scott queue [27], one of the most popular concurrent implementations for this data structure. Finally, we illustrate multi-line leases through a relaxed priority queue implementation, the MultiQueue [37].

4.1 Leases for TryLocks

Description. We assume a lock implementation which provides `try_lock` and `unlock` primitives, which can be easily implemented via `test&set`, `test&test&set`, or `compare&swap`.

The basic idea is to take advantage of leases to prevent wasted coherence traffic while a thread executes its critical section, by *leasing the lock variable* while the lock is owned. The thread leases the lock variable before attempting to acquire it, and maintains the lease for the duration of the critical section. If the native `try_lock` call fails, the thread will immediately drop the lease, as holding it may delay other threads.

This procedure can be very beneficial for the performance of contended locks (see Figure 3). Notice that, if leases held by the thread in the critical section do not expire involuntarily, the execution maintains the invariant that, whenever a thread is granted ownership of the line corresponding to the lock, the lock is unlocked and ready to use. Further, we obtain the performance benefit of the implicit queuing behavior on the lock, described in Section 1. On the other hand, if several involuntary releases occur, the lock may travel in locked state, which causes unwanted coherence traffic.

4.2 Leases for the Non-Blocking Michael-Scott Queue

Description. For completeness, we give a brief description of the non-blocking version of this classic data structure, adapted from [27, 38]. Its pseudocode is given in Figure 3. (Our description omits details related to memory reclamation and the ABA problem, which can be found here [38].)

We start from a singly-linked list, and maintain pointers to its head and tail. The head of the list is a "dummy" node, which precedes the real items in the queue. A successful `dequeue` operation linearizes at the `CAS` operation which moves the `head` pointer; an unsuccessful one linearizes at the point where n is read in the last loop iteration.

For *enqueue*s, two operations are necessary: one that makes the `next` field of the previous last element point to the new node, and one that swings the tail pointer to the new node. Operations are linearized at the `CAS` which updates the `next` pointer.

Algorithm 3 Michael-Scott Queue [27] with Leases.

```
1:  type node { value v, node* next }
2:  class queue { node* head, node* tail }

3:  function ENQUEUE( value v )
4:      node* w ← new node ( v )
5:      node* t, n
6:      while true do
7:          Lease( & tail, MAX_LEASE_TIME )
8:          t ← tail
9:          n ← t→next
10:         if t = tail then
11:             if n = NULL then ▷ tail pointing to last node
12:                 if CAS( & t→next, n, w ) then ▷ add w
13:                     CAS( & tail, t, w ) ▷ swing tail to inserted node
14:                     Release( & tail )
15:                     return                                  ▷ Success
16:             else ▷ tail not pointing to last node
17:                 CAS( & tail, t, n )                         ▷ Swing tail
18:         Release( & tail )

19: function DEQUEUE( )
20:     node* h, t, n
21:     while true do
22:         Lease( & head, MAX_LEASE_TIME )
23:         h ← head
24:         t ← tail
25:         n ← h→next
26:         if h = head then ▷ are pointers consistent?
27:             if h = t then
28:                 if n = NULL then
29:                     Release( & head )
30:                     return NULL                             ▷ empty queue
31:                 CAS( & tail, t, n )                         ▷ tail fell behind, update it
32:             else
33:                 ret ← p→v
34:                 if CAS( & head, t, n ) then ▷ swing head
35:                     Release( & head )
36:                     break                                   ▷ success
37:         Release( & head )
38:     return ret
```

Algorithm 4 MultiQueues [37] with Leases.

```
1:  type priority_queue, lock_ptr
2:  class MultiQueue { priority_queue MQ[M]
3:  lock_ptr Locks }                          ▷ Locks[i] points to lock i

4:  function DELETEMIN( )
5:      int i, k
6:      while true do
7:          i = random(1, M)
8:          k = random(1, M)                  ▷ k can be chosen ≠ i
9:          MultiLease(2, MAX_LEASE_TIME, Locks[i], Locks[k] )
10:         if try_lock ( Locks[i] ) then
11:             if try_lock (Locks[ k ] ) then
12:                 i ← queue containing higher priority element
13:                 k ← index of the other queue
14:                 unlock( Locks[k] )
15:                 ReleaseAll( )
16:                 rtn ← MQ[ i ].deleteMin()                   ▷ Sequential
17:                 unlock( Locks[i] )
18:                 return rtn
19:             else
                ▷ Failed to acquire Locks[k]
20:                 unlock( Locks[i] )
21:                 ReleaseAll( )
22:         else
            ▷ Failed to acquire Locks[i]
23:             ReleaseAll( )

24: function INSERT( value v )
25:     node* w ← new node ( v )
26:     while true do
27:         i = random(1, M)
28:         Lease( Locks[i], MAX_LEASE_TIME )
29:         if try_lock ( Locks[i] ) then
30:             MQ[ i ].insert( w )                             ▷ Sequential
31:             unlock( Locks[i] )
32:             Release( Locks[i] )
33:             return i
34:         else
35:             Release( Locks[i] )
```

Using Leases. There are several ways of employing leases in the context of the Michael-Scott queue. One natural option is to lease the `head` and `tail` pointers at the beginning of the `while` loop, and releasing them either on a successful operation, or at the end of the loop. This usage is illustrated in Algorithm 3.

This option has the advantage of cleanly "ordering" the enqueue and dequeue operations, since each needs to acquire the line corresponding to the tail/head before proceeding. Let us examine the common path for each operation in this scenario. For `Dequeue`, the lease will likely not expire before the `CAS` operation on line 34 (assuming the probable case where the head and tail pointers do not clash), which ensures that the `CAS` operation is successful, completing the method call. For `Enqueue`, the same is likely to hold for the `CAS` on line 12, but for a more subtle reason: it is unlikely that another thread will acquire and modify the next pointer of the last node, as the tail is currently owned by the current core.

This usage has two apparent drawbacks. First, it may appear that it reduces parallelism, since two threads may not hold one of the sentinel (head/tail) pointers at the same time, and for instance "helper" operations, such as the swinging of the tail in the `Enqueue`, have to wait for release. However, it is not clear that the slight increase in parallelism due to multiple threads accessing one of the ends of the queue is helpful for performance, as the extra `CAS` operations introduce significant coherence traffic. Experimental results appear to validate this intuition. A second issue is that, in the case where head and tail point to the same node, the `CAS` on the tail in line 31 of `Dequeue` may have to wait for a release of the tail by a concurrent `Enqueue`. We note that this case is unlikely.

Results. The throughput comparison for the queue with and without leases is given in Figure 3. We have also considered alternative uses of Lease/Release, such as leasing the `next` pointer of the tail for the enqueue before line 9, or leasing the head and tail

Table 1: System Configuration

Parameter	Value
Core model	1 GHz, in order core
L1-I/D Cache per tile	32 KB, 4-way, 1 cycle
L2 Cache per tile	256 KB, 8-way, Inclusive, Tag/Data: 3/8 cycles
Cacheline size	64 Bytes
Coherence Protocol	MSI (Private L1, Shared L2 Cache hierarchy)

nodes themselves, instead of the sentinel pointers. The first option increases parallelism, but slightly decreases performance since threads become likely to see the tail trailing behind, and will therefore duplicate the `CAS` operation swinging the tail. The second option leads to complications (and severe loss of performance) in the corner cases when the head and the tail point to the same node.

4.3 Leases for MultiQueues

Description. MultiQueues [37] are a recently proposed method for implementing a relaxed priority queue. The idea is to share a set of M *sequential* priority queues, each protected by a `try_lock`, among the threads. To `insert` a new element, a thread simply selects queues randomly, until it is able to acquire one, and then inserts the element into the queue and releases the lock. To perform a `deleteMin` operation, the thread repeatedly tries to acquire locks for *two* randomly chosen priority queues. When succeeding, the thread pops the element of *higher* priority from the two queues, and returns this element after unlocking the queues. This procedure provides *relaxed* priority queue semantics, with the benefit of increased scalability, as contention is distributed among the queues.

MultiLeases on MultiQueues. We use leases in the context of MultiQueues [37] as described in Algorithm 4. On `insert`, we lease the lock corresponding to the queue, releasing it on unlock, as described in Section 4.1. On `deleteMin`, we MultiLease on the locks corresponding to the chosen queues, before attempting to acquire them. The thread then attempts to acquire both locks. If successful, the thread compares the top priority values. Let i be the index of the queue with the top value, and k be the index of the other queue. As soon as the comparison is done, the thread *unlocks* queue k, and releases *both* of the leases on the locks. The thread then completes its operation, removing the top element from queue i, unlocking the queue, and returning the element.

It is tempting to hold the lease on queue i until the unlock point at the end of the operation. As we have seen in Section 4.1, this reduces useless coherence traffic for threads reading an owned lock. However, this traffic is *not useless* in the case of MultiQueues: it allows a thread to stop waiting on a locked queue, to get a new random choice, and make progress on another set of queues. Since the operations on the sequential priority queue can be long, allowing for fast retries brings a performance benefit. Please see Figure 4 for the throughput comparison.

5. Empirical Evaluation

5.1 Experimental Setup

We use Graphite [28], which simulates a tiled multi-core chip, for all our experiments. The hardware configuration is listed in Table 1. We run the simulation in *full mode*, which ensures accurate modeling of the application's stack and instructions. We have implemented Lease/Release in Graphite on top of a directory-based MSI protocol for private L1 and shared L2 cache hierarchy. In particular, we extended the L1 cache controller logic (at the cores) to implement memory leases. As such, the directory did not have to be modified in any way.

For validation, we have compared the behavior of some of the base (lease-less) implementations on the simulator and on a real Intel processor with similar characteristics. The scalability trends are similar, with the note that the real implementations appear to incur more CAS failures than the simulation ($\leq 20\%$).

5.2 Results

Experiments. We have tested leases for a range of classic concurrent data structure implementations, including the Treiber stack [42], the Michael-Scott queue [27], the Lotan-Shavit skiplist-based priority queue [3, 23], the Java concurrent hash table, the Harris lock-free list [16], and skiplist implementations [14, 34]. We also compared lock throughput against optimized hierarchical ticket locks [6] and CLH queue locks [5, 24]. We tested multiple leases on queues, lists, MultiQueues [37], the TL2 transactional algorithm [9], and the software implementation of MCAS [17]. Some implementations use code from the ASCYLIB library [6]. Using Lease/Release usually entailed modifying just a few lines of code in the base implementation.

Scalability under Contention. Figure 3 shows the effect of using leases in the context of highly contended shared structures (lock-based counter, queue, priority queue), while Figure 2 showed results for the Treiber stack. Specifically, the counter benchmark is a contended lock protecting a counter variable. The baseline Lotan-Shavit priority queue is based on a fine-grained locking skiplist design by Pugh [34]. The lease-based implementation relies on a global lock. As we are interested in high contention, the benchmarks are for 100% update operations. We illustrate both throughput (operations per second) and energy (nanoJoules per operation). We also recorded the number of coherence messages, and the number of cache misses. The messages and cache misses are well correlated with energy results, and we therefore only display the latter. The `MAX_LEASE_TIME` variable is set to 20K cycles, corresponding to 20 microseconds.

The key finding from these graphs is that using leases can increase throughput by up to 7x on lock-free data structures, and by up to 20x for the lock-based counter, when compared to the base implementations. Further, it reduces energy usage by up to 10x (in the case of the counter). We believe the main reason for this improvement is that leases keep both cache misses and coherence messages per operation close to *constant* as contention grows. For instance, average cache misses per operation for the stack are constant around 2.1 from 4 to 64 threads; on the base implementation, this parameter increases by 5x at 64 threads. The same holds if we record average coherence messages per operation (constant around 9.5 for the stack), and even if we decrease `MAX_LEASE_TIME` to 1K. Results are similar for the queue, with different constants.

Throughput decreases with concurrency for the skiplist-based priority queue (although the lease-based implementation is still superior), since the number of cache misses per operation increases with concurrency, due to the structure of the skiplist. (The increase in messaging with contention is also apparent in the energy graph.)

In some of these data structures, there is potential for using multiple leases. For instance, in the Michael-Scott enqueue, we could potentially lease both the `tail` pointer and the `next` pointer of the last element, to further reduce retries. In general, we found that using multiple leases for "linear" data structures such as lists, queues, or trees, does improve upon the base implementation, but has inferior performance to simply using a lease on the predecessor of the node we are trying to update. The queue graph in Figure 3 provides results for both single and multiple leases. The relative difference comes from the additional overhead of multiple leases, coupled with the fact that, in such structures, leasing the predecessor node makes extra cache misses on successors unlikely.

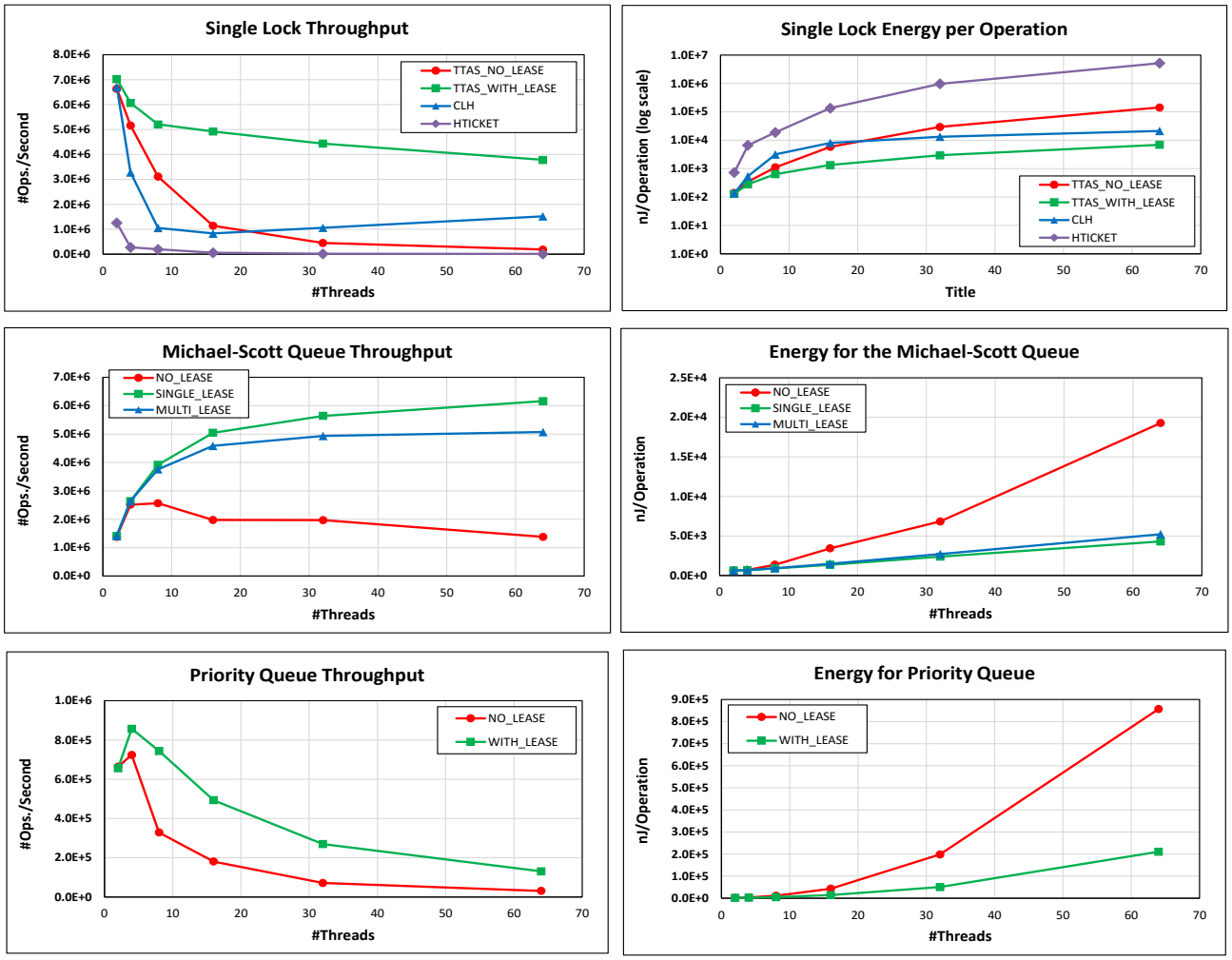

Figure 3: Throughput and energy results for lock-based counter, queue, and skip-list-based priority queue. We tested for 2, 4, 8, 16, 32, 64 threads/cores.

Comparison with Backoffs and Optimized Implementations. We have also compared against variants of these data structures which use backoffs to reduce the overhead of contention. In general, we found that adding backoffs improves performance by a constant factor (up to 3x), but is considerably inferior to using leases. For instance, for the stack, we also compared against a highly optimized implementation with carefully chosen backoffs [13] (graph omitted due to space constraints). The implementation of [13] has superior performance to both flat-combining and elimination techniques. While it improves throughput by up to 3x over the base implementation, it is still 2.5x lower on average than simply using leases on the Treiber stack. Further, the ticket lock implementation in Figure 3 uses linear backoffs.

The performance difference between leases and backoffs is natural since backoffs also introduce "dead time" in which no operations are executed, and do not fully mitigate the coherence overhead of contention. As such, given hardware support for leases, we believe backoffs would be an inferior alternative.

Low Contention. We have also examined the impact of using leases in scenarios with low contention, such as lock-free linked lists [16], skiplists [14], binary trees [31], and lock-based hash tables, with 20% updates on uniform random keys and 80% searches. We found that throughput is the same on these structures, as they have little or no contention. Using leases slightly improves throughput ($\leq$5%) at high thread counts ($\geq$ 32).

MultiLease Examples. To test multiple leases, we have implemented MultiQueues [37], a variant of the TL2 STM algorithm [9], as well as the software MCAS algorithm of Fraser and Harris [17]. In the MultiQueue benchmark, threads alternate between `insert` and `deleteMin` operations, implemented as described in Section 4.3, on a set of eight queues. In the TL2 benchmark, transactions attempt to modify the values of two randomly chosen transactional objects out of a fixed set of ten, by acquiring locks on both. If an acquisition fails, the transaction aborts and is retried. Finally, we tested an MCAS-based skiplist [14] in a scenario where a small fraction of keys are contended. Figure 4 illustrates the results for MultiQueues and TL2.

For MultiQueues, the improvement is of about 50% (due to the long critical section), while in the case of TL2 the improvement is of up to 5x, as leases significantly decrease the abort rate. Leasing just the lock associated to the first object improves throughput only moderately. For the MCAS-based skiplist, throughput improvement is relatively modest ($\leq$ 20%), as the average contention is low. However, average latency for the 90th percentile of opera-

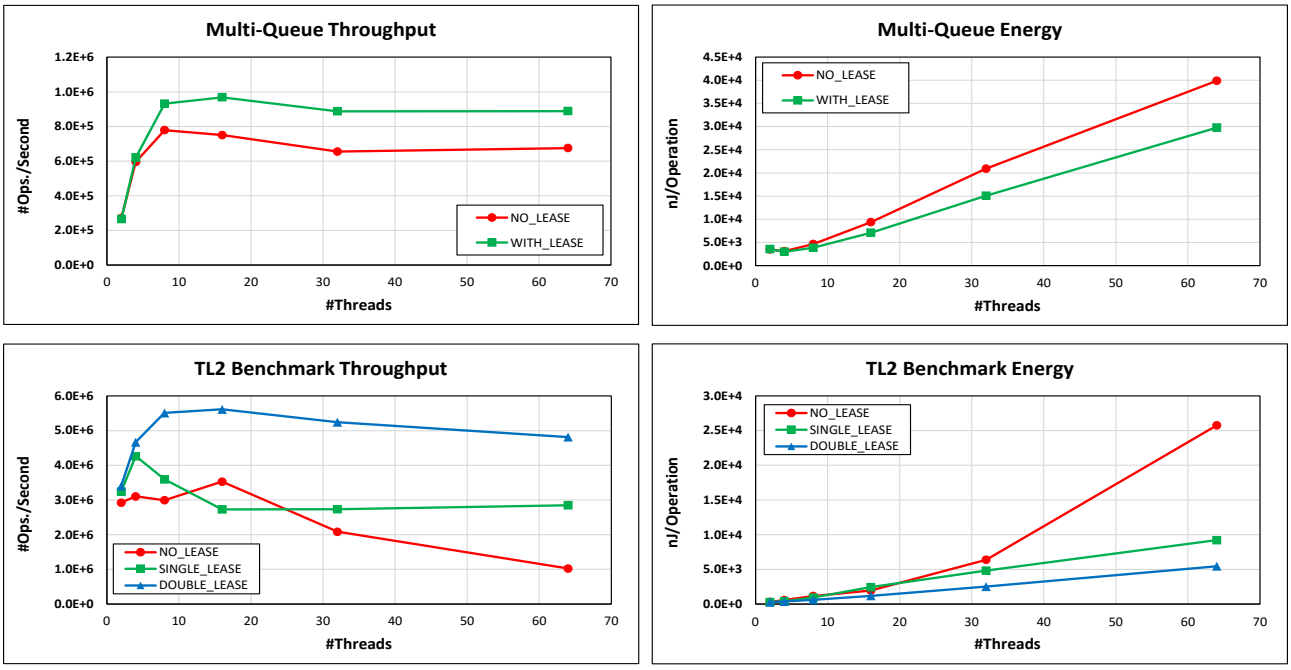

Figure 4: Throughput and energy graphs for MultiLease benchmarks. We tested for 2, 4, 8, 16, 32, 64 threads/cores.

tions is improved by up to 10x: this is because leases mostly eliminate the need for the costly helping mechanism called when an MCAS operation is contended.

Observations and Limitations. While the use of leases does not usually decrease performance when compared to the baseline, we did find that improper use can introduce overheads. For instance, not releasing a lock variable already acquired by another thread may slow down the application, since the owner thread is delayed while attempting to reset the lock. Further, in the case of data structures with linear access patterns, such as lists, leasing nodes close to the head can reduce performance, as concurrent traversals for other elements may be blocked behind the lease. As mentioned, multiple leases can be inferior to careful placement of single leases.

One potential complication is *false sharing*, i.e. inadvertently leasing multiple variables located on the same line. In the case of multiple leases, false sharing may even cause deadlock, as it can break the ordering property for lock acquisition. This behavior can be prevented via careful programming, and could be enforced automatically.

6. Discussion

Summary. We have investigated an extension to standard cache coherence protocols which would allow the leasing of memory locations for short, bounded time intervals, and explored the potential of this technique to speed up concurrent data structures. Our empirical results show that Lease/Release can improve both throughput and energy efficiency under contention by up to 5x, while preserving performance in uncontended executions. Employing Lease/Release on classic, relatively simple, data structure designs compares well with complex, highly optimized software techniques for scaling the same constructs.

The key feature of Lease/Release is that it minimizes the coherence cost of operations under contention: on average, each operation pays a constant number of coherence messages for each contended cache line it needs to access; further, the number of retried operations is minimized. Thus, Lease/Release allows the programmer to improve throughput in the presence of bottlenecks, beyond what is possible with current software techniques.

Future Work. The Lease/Release mechanism is not without limitations. The semantics we propose require careful programming; for lock-free data structures, a basic understanding of the underlying mechanics is required for proper placement. Improper use is possible, and can lead to performance degradation. To address this, we plan to investigate *automatic* lease insertion, using compiler and hardware techniques. This has two goals: first, automatically identifying lease-friendly patterns would simplify programming, and reduce the likelihood of erroneous use. Second, it would allow automatic optimization of lease times.

A second topic for further investigation is the MultiLease mechanism in the context of transactional memory (TM). In particular, recent work suggests that hardware TM has limited performance under contention [30]; using joint leases inside short hardware transactions could reduce these costs. In general, the potential of leases in the context of transactional semantics is an interesting area for future work. Finally, our experimental study mostly focuses on classic data structures. It would be interesting to see if leases can be used to speed up other, more complex, applications, and whether it can inform new data structure designs which take explicit advantage of the leasing mechanism.

7. Acknowledgments

The authors would like to thank Richard Black, Miguel Castro, Aleksandar Dragojevic, William Hasenplaugh, Ant Rowstron, Nir Shavit, and Vasileios Trigonakis for fruitful discussions, helpful suggestions, and code support during the making of this paper.

References

[1] Y. Afek, M. Hakimi, and A. Morrison. Fast and scalable rendezvousing. *Distributed computing*, 26(4):243–269, 2013.

[2] D. Alistarh, J. Aspnes, K. Censor-Hillel, S. Gilbert, and R. Guerraoui. Tight bounds for asynchronous renaming. *Journal of the ACM (JACM)*, 61(3):18, 2014.

[3] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit. The spraylist: A scalable relaxed priority queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 11–20, New York, NY, USA, 2015. ACM.

[4] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou. Denovo: Rethinking the memory hierarchy for disciplined parallelism. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 155–166. IEEE, 2011.

[5] T. Craig. Building fifo and priorityqueuing spin locks from atomic swap. Technical report, Technical Report 93-02-02, University of Washington, Seattle, Washington, 1994.

[6] T. David, R. Guerraoui, and V. Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 631–644. ACM, 2015.

[7] D. Dice, D. Hendler, and I. Mirsky. Lightweight contention management for efficient compare-and-swap operations. In *Euro-Par 2013 Parallel Processing*, pages 595–606. Springer, 2013.

[8] D. Dice, V. J. Marathe, and N. Shavit. Lock cohorting: A general technique for designing numa locks. *ACM Trans. Parallel Comput.*, 1(2):13:1–13:42, Feb. 2015.

[9] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Distributed Computing*, pages 194–208. Springer, 2006.

[10] D. Drachsler, M. Vechev, and E. Yahav. Practical concurrent binary search trees via logical ordering. In *ACM SIGPLAN Notices*, volume 49, pages 343–356. ACM, 2014.

[11] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 131–140, New York, NY, USA, 2010. ACM.

[12] F. Ellen, D. Hendler, and N. Shavit. On the inherent sequentiality of concurrent objects. *SIAM J. Comput.*, 41(3):519–536, 2012.

[13] P. Fatourou and N. D. Kallimanis. A highly-efficient wait-free universal construction. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 325–334. ACM, 2011.

[14] K. Fraser. *Practical lock-freedom*. PhD thesis, PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579, 2004.

[15] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. *SIGARCH Comput. Archit. News*, 17(2):64–75, Apr. 1989.

[16] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, London, UK, UK, 2001. Springer-Verlag.

[17] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In D. Malkhi, editor, *Distributed Computing, 16th International Conference, DISC 2002, Toulouse, France, October 28-30, 2002 Proceedings*, volume 2508 of *Lecture Notes in Computer Science*, pages 265–279. Springer, 2002.

[18] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 355–364. ACM, 2010.

[19] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova. Quantitative relaxation of concurrent data structures. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 317–328, New York, NY, USA, 2013. ACM.

[20] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[21] A. Kägi, D. Burger, and J. R. Goodman. Efficient synchronization: Let them eat qolb. *SIGARCH Comput. Archit. News*, 25(2):170–180, May 1997.

[22] C. Leiserson. A simple deterministic algorithm for guaranteeing the forward progress of transactions. *Transact 2015*.

[23] I. Lotan and N. Shavit. Skiplist-based concurrent priority queues. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 263–268. IEEE, 2000.

[24] P. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Parallel Processing Symposium, 1994. Proceedings., Eighth International*, pages 165–171. IEEE, 1994.

[25] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. *SIGPLAN Not.*, 26(4):269–278, Apr. 1991.

[26] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82. ACM, 2002.

[27] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[28] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12. IEEE, 2010.

[29] A. Morrison and Y. Afek. Fast concurrent queues for x86 processors. In *ACM SIGPLAN Notices*, volume 48, pages 103–112. ACM, 2013.

[30] T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, and H. Tomari. Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and power8. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 144–157, New York, NY, USA, 2015. ACM.

[31] A. Natarajan and N. Mittal. Fast concurrent lock-free binary search trees. In *ACM SIGPLAN Notices*, volume 49, pages 317–328. ACM, 2014.

[32] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM.

[33] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[34] W. Pugh. Concurrent maintenance of skip lists. 1998.

[35] R. Rajwar, A. Kagi, and J. R. Goodman. Improving the throughput of synchronization by insertion of delays. In *High-Performance Computer Architecture, 2000. HPCA-6. Proceedings. Sixth International Symposium on*, pages 168–179. IEEE, 2000.

[36] R. Rajwar, A. Kägi, and J. R. Goodman. Inferential queueing and speculative push for reducing critical communication latencies. In *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS '03, pages 273–284, New York, NY, USA, 2003. ACM.

[37] H. Rihani, P. Sanders, and R. Dementiev. Brief announcement: Multiqueues: Simple relaxed concurrent priority queues. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 80–82, New York, NY, USA, 2015. ACM.

[38] M. L. Scott. *Shared-Memory Synchronization*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[39] O. Shalev and N. Shavit. Transient blocking synchronization. Technical report, Mountain View, CA, USA, 2005.

[40] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks: preliminary version. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 54–63. ACM, 1995.

[41] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011.

[42] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.

[43] X. Yu and S. Devadas. Tardis: Timestamp based coherence algorithm for distributed shared memory. *arXiv preprint arXiv:1501.04504*, 2015.