Lease/Release: Architectural Support for Scaling Contended Data Structures

Syed Kamran Haider
University of Connecticut *

William Hasenplaugh
MIT

Dan Alistarh
Microsoft Research

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 MSI cache coherence protocols, allowing participants to lease memory, at the granularity of cache lines, by delaying coherence messages 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. [17, 26], hash tables, e.g. [8, 20, 26], skip lists, e.g. [15, 20], search trees, e.g. [12, 31], queues [27], stacks [39, 41], or priority queues, e.g. [4, 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 thanks to their "flat" structure and relatively uniform access patterns, e.g., [7, 8], it is much harder to avoid hotspots in the case of data structures such as queues, stacks, or priority queues. In fact, theoretical results [3, 13] 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 [39], relaxed semantics [4, 19], back-offs [9], or data-structure and architecture-specific optimizations [14, 29]. While these methods can be very effective in improving the performance of individual data structures, the general 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 MSI cache coherence protocols, allowing a core to lease memory, at the granularity of cache lines, by delaying incoming coherence requests 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. Importantly, this mechanism should not require the revalidation of the protocol logic, since it only introduces finite delays. Lease/Release 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 decrease communication 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. [32]. For cache coherence, references [34, 38], 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 [41], 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

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

PPoPP '16 March 12-16, 2016, Barcelona, Spain
Copyright © 2016 ACM 978-1-4503-4092-2/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2851141.2851155

```
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.

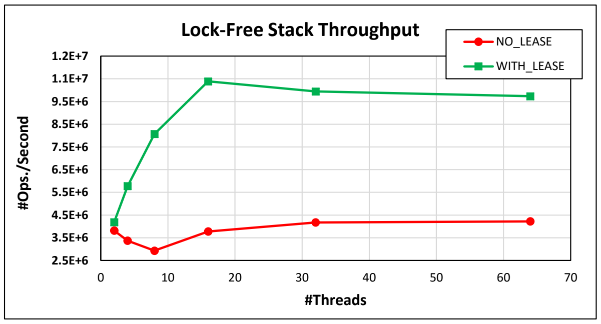

Figure 2: Throughput of the lock-free Treiber stack with and without leases, for 100% updates. The data points are at powers of 2.

`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-obtain exclusive ownership, the memory value *may have changed*, which causes the `CAS` instruction 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*. 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, a natural point to set the lease on the head pointer is before the read on line 4 of Figure 1. This populates a *lease table* structure at the core, adding an entry corresponding to the cache line and the lease timeout, and brings the line to the core in exclusive state. The lease starts as soon as ownership is granted. The natural release point is after the (probably successful) CAS operation. If 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. See Figure 2 for the relative throughput improvement.

Of note, leasing fits easily into protocols which support arbitrary but bounded delays for coherence messages, or negative acknowledgments. Intuitively, coherence messages delayed by leases can be thought of as having been delayed by the system by a quantum upper bounded by `MAX_LEASE_TIME`. Since this additional delay is bounded, the protocols remain correct.

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 an 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 [6, 10, 24, 25] and hardware [16, 21, 35] 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 [34].

Multiple Leases. In more complex data structures, such as trees, 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 a small set of 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 transactional semantics, joint leases on these locations can be beneficial for performance.

Experimental Results. We implemented Lease/Release 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 under contention, without affecting performance in the uncontended case. Using single leases, the relatively simple classic data structure designs such as the Treiber stack match or improve the performance of optimized, complex implementations. Using leases inside a Pagerank implementation with contended locks [] improves throughput by 8x, and allows the application to scale. 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 ($\leq 5\%$).

A notable benefit of using leases is *reduced coherence traffic*. Specifically, leases reduce coherence traffic by up to 5x; consequently, we believe that leases should also reduce the energy cost of contended programming patterns.

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). Leases allow the programmer to minimize both *traffic* and *retry* costs for both lock-based and lock-free programming patterns.

The rest of our paper develops as follows. We discuss related work in Section 2. We specify detailed semantics for both single-location and multi-location Lease/Release, as well as implementation details, in Section 3. We cover detailed usage examples in Section 6, and provide empirical validation in Section 7. We discuss our findings and the feasibility of a hardware implementation in Section 8.

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 [39] proposes to directly match producer and consumer operations, such as `push` and `pop` for a stack, as a way to avoid memory hotspots. *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. [4, 19, 36], aim to avoid hotspots by *relaxing ordering guarantees*..

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 [14, 27, 29], stacks [1, 41], 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 [41], Michael–Scott [27], and Lotan–Shavit [23], respectively.

Lock cohorting [10] 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 change the lock ownership pattern, and should hence be 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) [16, 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 [34], but requires complex protocol support, new instructions, and re-compilation of the application code [34].

Implicit QOLB (IQOLB) [34] 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 [34] 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 [38], 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 MSI-based coherence, e.g. [5, 42], which show 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. Single-Location 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 two constants `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 queue, where each key is associated with a (decreasing) time counter and with a boolean flag. High-level pseudocode for these operations is given in Algorithm 1.

Specifically, whenever a `Lease(addr, time)` instruction is first encountered on address `addr`,[1] the system creates a new entry corresponding to the cache line in the lease table. If the size of the table is already the maximum `MAX_NUM_LEASES`, then the new address replaces the oldest leased address (in FIFO order), which is automatically released. The core then requests the cache line corresponding to `addr` in exclusive state. When this request is fulfilled, the core sets a corresponding counter with length $\min(\texttt{time}, \texttt{MAX\_LEASE\_TIME})$, and starts decrementing this counter. Note that the operation does not bind the corresponding

[1] We do not allow extending leases on an already-leased address, as this could break the `MAX_LEASE_TIME` bound.

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 ) then
5:         time ← $\min$( time, MAX_LEASE_TIME )

6:         if ( num = MAX_NUM_LEASES ) then
             ▷ Evict oldest existing lease
7:             oldest ← Oldest existing lease, in FIFO order
8:             RELEASE( oldest )

9:         Lease-Table[addr] ← $time$
10:        Request line corresponding to addr in Exclusive state

11: upon event ZERO-COUNTER( addr ) do
12:     RELEASE( addr )
13: upon event COHERENCE-PROBE( addr ) do
14:     found = Lease-Table.find( addr )
15:     if found then Queue probe until lease on addr expires
16: upon event CLOCK-TICK do
17:     for each addr in Lease-Table with started = true do
18:         Decrement counter by 1, down to 0
19: function RELEASE( addr )
        ▷ Delete entry, returning true if entry exists
20:     found ← Lease-Table.delete( addr )
21:     if found then
22:         req ← coherence requests queued for this address
23:         Fulfill req as per the cache coherence protocol
24:     return found
```

Algorithm 2 MultiLease/MultiRelease Pseudocode.

```
1: function MULTILEASE(num, time, addr1, addr2, ...)
         ▷ Release all currently held leases
2:     RELEASEALL( )
         ▷ Check if lease is valid
3:     count ← Lease-Table.num_elements( )
4:     time ← $\min$( time, MAX_LEASE_TIME )
5:     if (count + num ) $\leq$ MAX_NUM_LEASES) then
         ▷ Lease all addresses within the group
6:         for each addr in list, in fixed order do
7:             LEASE(addr, time)

8: function RELEASEALL( )
         ▷ Clear entries corresponding all lines in the group
9:     addr_list = Addresses currently leased

10:    for each addr in addr_list do
11:        Lease-Table.delete( addr )
12:    for each addr in addr_list do
         ▷ A single queued request may exist per line
13:        Service any outstanding coherence requests for addr
```

value to a register. This may occur on the next instruction on the line, which typically follows the lease instruction.

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 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. (A `release` on a line not present in the lease table does nothing.) If the core calls `release` before this event, we say a *voluntary* release occurred. Optionally, the `release` instruction may return a boolean signaling whether the target line was released voluntarily or involuntarily.

For a voluntary release, the core performs 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.

3.1 Properties of Single-Location Leases

We now state some of the properties of the Lease/Release mechanism described above. We will make the following observation about the implementation of directory-based cache coherence protocols: at any given time, only a single request for each line may be queued at a core. This upper bounds the number of queued requests per core in the single-line mechanism by *one*.

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

Proof. This observation is based on the fact that directory-based protocols queue multiple requests for each line at the directory, in FIFO order [40] . (It is possible that requests for *distinct* lines may be mapped to the same FIFO queue at the directory.) A request for line x is not serviced by the directory until its predecessor requests in the queue for line x are fully serviced.

Therefore, at any given point in time, of all the requests for line x, at most a single request for a specific line may be queued at a core, that is, the one being currently serviced by the directory. All other requests for line x are queued in the directory queue corresponding to the cache line. □

We next prove that, since the lease interval is bounded, Lease/Release may only introduce a bounded amount of delay for any coherence request. In the following, we call *the owning core* the core which currently has a given cache line in Exclusive state.

Proposition 2. *Let M be a coherence message sent by a coherence protocol $\mathcal{C}$. If T is an upper bound on the maximum delay before M is processed in $\mathcal{C}$, then ($T +$ MAX_LEASE_TIME) will be an upper bound on the number of time steps by which M will be processed in the version of $\mathcal{C}$ which implements Lease/Release.*

Proof. For the proof of this claim, first notice that only coherence messages which encounter a lease at a core will be delayed beyond the standard delay of $\mathcal{C}$. In particular, with Lease/Release, a message corresponding to a coherence request which reached the top of the directory queue and is in the process of being serviced may be delayed by at most `MAX_LEASE_TIME` additional time steps, the waiting time at the owning core if the line happens to be leased.

After this, the message is guaranteed to be processed. Therefore, $(T + \texttt{MAX\_LEASE\_TIME})$ is an upper bound on the total waiting time of the request. □

We build on this claim to obtain that Lease/Release preserves the correctness of coherence protocols which support arbitrary bounded message delays.

Corollary 1. *Any coherence protocol that is correct for arbitrary* bounded *message delays will remain correct if bounded single-location leases are implemented.*

Proof. Consider a cache coherence protocol which is correct for any finite bounded message delay $B > 0$. Let T be an upper bound on the maximum delay supported by the protocol. By Proposition 2, $(T + \texttt{MAX\_LEASE\_TIME})$ is an upper bound on the message delay in the version with Lease/Release. Since the protocol remains correct if the message delay is $(T + \texttt{MAX\_LEASE\_TIME})$, the claim follows. □

4. Multi-Location Memory Leases

The multi-location leasing mechanism gives a way for a core to *jointly* lease a set of memory locations for a fixed, bounded period of time. To simplify the exposition, we will begin by sketching the hardware implementation of joint leases, and then describe a software emulation. Importantly, we do not allow concurrent use of single-location and multi-location leases. We discuss implementation and usage issues in Section 5.

The interface to joint leases 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. 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.

MultiLeases will enforce the following two assumptions. First, the `MultiLease` call will first release all currently held leases. Second, a `MultiLease` request that causes the `MAX_NUM_LEASES` bound to be exceeded is ignored. The pseudocode for these instructions is given in Algorithm 2.

The procedure uses the same `Lease-Table` data structure as for single leases. The `MultiLease` procedure simply performs `Lease` calls on each of the addresses in the argument list, using the same `time`. This procedure does not actually bind the address values to registers. This occurs on accesses by the core on the corresponding lines in the group, which typically follow the lease instruction.

On a `MultiLease` call, 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*, waiting for each ownership request to be granted before moving to the next one. 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, e.g. [11, 22]. In this way, the acquisition time is bounded, although it may increase because of concurrent leases. A release on any address in the group causes all the other leases to be canceled. The rest of the events are identical to single-line leases, and are therefore omitted.

Software Implementation. While MultiLeases appear intuitively straightforward, they may be cumbersome to implement in hardware. (Please see Section 5 for details.) MultiLeases can be simulated in software, with weaker semantics, on top of a the single-location mechanism, as follows. The (software) MultiLease instruction requests leases on addresses in the group in sorted order, using the single-location instructions, maintaining group id information in software. Additionally, the instruction can adjust the lease timeout to maximize the probability that the leases are held jointly for `time` time steps, by requesting the jth outer lease for an interval of $(\texttt{time} + jX)$ units, where X is a parameter approximating the time it takes to fulfill an ownership request. For instance, when jointly leasing two lines A and B, the lease on A is taken for $(\texttt{time} + X)$ time units, whereas the lease on B is taken for `time` time units. Notice that this mechanism does not guarantee that leases will be held jointly.

4.1 Properties of Multi-Line Memory Leases

In this section, we prove that the hardware MultiLease protocol ensures deadlock-freedom, under assumptions on the coherence protocol. Notice that the *software* MultiLease protocol is correct, since the protocol is a composition of single-location leases. We make the following assumption on the cache coherence protocol, which we discuss in detail in Section 5.

Assumption 1. *Requests for each cache line are queued independently in a FIFO queue at the directory. In particular, a coherence request on a line A may not be queued behind a request corresponding to a distinct line B.*

Based on Assumption 1, we can prove that MultiLeases may only introduce bounded message delays. It therefore follows that Corollary 1 applies to MultiLeases as well.

Proposition 3. *Let M be a coherence message sent by a coherence protocol $\mathcal{C}$. If there exists a finite bound on the maximum delay before M is processed in $\mathcal{C}$, then there exists a finite bound bound on the maximum number of time steps until M will be processed in the version of $\mathcal{C}$ which implements Lease/Release.*

Proof Sketch. As in the previous section, we will bound the maximum delay for a coherence request once it reaches the top of the directory queue corresponding to the requested cache line.

Consider a request r_0, on cache line R_0, by core p_0, which is currently being processed by the directory. Assume for contradiction that request r_0 is indefinitely delayed in the protocol variant implementing MultiLeases. Without loss of generality, let us assume that the line R_0 is in modified (M) state. Since r_0 has reached the top of the directory request queue, it must hold that the directory forwards an invalidation request to the core owning R_0, which we call p_1. We have two cases.

In the first case, the owning core p_1 is executing outside lines 12–14 of the MultiLease protocol. Then, the invalidation request is guaranteed to be processed within finite time, by protocol correctness. In the second case, the core is executing a multiple lease sequence, and may delay incoming requests indefinitely during this period, if itself is indefinitely delayed. Specifically, request r_0 is indefinitely delayed only if there exists a request r_1, on a line R_1, part of p_1's MultiLease sequence which is infinitely delayed. First, notice that $R_1 \neq R_0$, since by construction p_1 already owns R_0. Similarly, $p_1 \neq p_0$. Second, notice that p_1 must have acquired R_0 as part of its current MultiLease call, since this call released all currently held addresses. This implies that cache line R_0 must come *before* R_1 in the global order of addresses, since p_1 has already acquired it.

By Assumption 1, the requests r_0 and r_1 are mapped to distinct directory queues, therefore r_0's progress does not affect r_1's progress. We can therefore assume without loss of generality that r_1 is at the top of its corresponding directory queue. By iterating the same argument, we obtain that request r_1 can be indefinitely delayed only if line R_1 is held by a core p_2, which is executing lines 12–14 of the MultiLease protocol, which is indefinitely delayed on a request r_2 on a line R_2.

We now make two observations. The first is that $p_2 \notin \{p_0, p_1\}$. The fact that $p_2 \neq p_1$ follows again by Assumption 1. Assume for contradiction that $p_2 = p_0$. This would imply that p_0 is holding a lease on R_1 and requesting $R = R_2$ as part of a MultiLease operation, while p_1 is holding a lease on R and requesting R_1 as part of a MultiLease operation. However, this contradicts the *sorted* order property of MultiLeases. Hence, $p_2 \notin \{p_0, p_1\}$. Similarly, $R_2 \notin \{R_0, R_1\}$, and R_2 must come *after* R_0 and R_1 in the global sorting order of addresses.

We iterate this argument to obtain that, for request r_0 to be indefinitely delayed, there must exist a chain $p_1, p_2, \ldots$ of distinct cores, where core p_i owns line R_{i-1}, and is delayed on request r_i on line R_i. Further, it must hold that line R_i comes *after* lines $R_0, R_1, \ldots, R_{i-1}$ in the global sort order of addresses. Since the cores are distinct and their number is finite, the chain is finite. Let us examine the last core in this chain, p_k. If line R_k is not currently leased, then p_k's request must progress, by the correctness of the original cache coherence protocol. If the line R_k is currently leased, it cannot be leased as part of an ongoing MultiLease operation: it could not be leased by a core outside $p_1, \ldots, p_k$ (since p_k is the last core in the chain), and none of the cores $p_0, p_1, \ldots, p_{k-1}$ may have requested line R_k during their current MultiLease operation, since these requests must be performed in order. Hence, the line R_k can only be leased as part of a lease operation *that has already completed*. It follows that the corresponding request r_k will be serviced within finite time, a contradiction.

It therefore follows that request r_0 will be serviced within finite time, which in turn implies that every request is serviced within finite time. Finally, we note that the argument that transactional requests performed in fixed sorted order do not deadlock is folklore, and is given in full in [22]. $\square$

5. Implementation Details

Core Modifications. A hardware implementation of Lease/Release could be built by modifying the core structure which keeps track of outstanding cache misses, which we call the *load buffer*, and by adding a *lease table* structure, which keeps track of lease timers.

We add two new line states to the load buffer: *lease* and *transition to lease*. A line is in *lease* state if the data is already in the cache, and the lease on the line is still valid. This state serves to catch incoming coherence requests for the line, which will be queued until either the lease expires, or the line is voluntarily released. Here, incoming requests exploit fact that the load buffer is content-addressable. The *transition to lease* state corresponds to an *outstanding* coherence request on which the core will take a lease once the request is serviced. When the request is completed, the entry is retained, and is transitioned to the *lease* request type.

The second component is the *lease table*, which *mirrors* the load buffer in that each position in the lease table maps to the corresponding position in the load buffer. The lease table contains a collection of `MAX_NUM_LEASES` counters, counting down from `MAX_LEASE_TIME` at each cycle. The lease table need not be content addressable, and should therefore be cheaper to implement. When a counter reaches 0, or the lease is voluntarily released, the corresponding entry in the load buffer is deleted, and the counter is made available. This causes any outstanding coherence requests on the line to be honored.

MultiLeases require the counters for the corresponding cache lines to be correlated. For this, whenever a coherence request that is part of a MultiLease is completed, the request transitions to *lease* state, and checks whether all the other requests in the lease group have also completed. (Recall that a single MultiLease group may be active at a time.) If this is the case, then all corresponding counters are allocated and started. When one of the entries is released, or the counters reach 0, the counters are freed and the load buffer entries are removed.

Directory Structure and Queuing. The MultiLease logic assumes independent progress per cache line, by mapping each line to a distinct directory queue. This may not be the case for protocol implementations where multiple cache lines map to the same request queue at the directory. In particular, the following scenario may occur: core C_1 holds a lease on line A, and requests a lease on line B. Concurrently, Core C_2 is requesting line A. If lines A and B map to the same directory request queue, and C_2's request is ahead, then C_1 is waiting for its own lease on A to expire before it can make progress on B.

We can avoid this issue by modifying the directory structure to prioritize queued requests which are not waiting for a leased line (effectively, this would allow C_1's request on B to be treated before C_2's request on A, overriding FIFO order). We note that such delay scenarios may also be an issue in the variant of the protocol without leases, although we are not aware of implementations of such priority-based solutions.

We note that this issue does not occur in the version of the protocol where a core may only have *a single* outstanding lease, i.e. where `MAX_NUM_LEASES` $= 1$. We believe that the leasing mechanism can be implemented without any changes to the directory structure in this case.

Another potential issue regarding directory structure is that leases may increase the maximum queuing occupancy over time, and may thus require the directory to have larger queues. However, in the average case, leases enable the system to make more forward progress at the application level, reducing the number of repeated coherence requests, and therefore reducing system load.

Protocol Correctness. As shown in Propositions 2 and 3, the lease mechanism fits well with protocols which support arbitrary but bounded delays for coherence messages, or negative acknowledgments (NACKs). For single-location leases, if the protocol is correct for any any finite message bound B, then we can simulate the process by which a message is queued on the lease by an additional delay of `MAX_LEASE_TIME` units on the incoming message, after which the request is immediately fulfilled, as in the standard version of the protocol. This simulation step leads to a protocol in which messages can be delayed by at most $(\texttt{MAX\_LEASE\_TIME} + B)$ time steps. This protocol is also correct, since this maximum delay is also bounded. The same argument applies for protocols supporting NACKs, by simulating the lease delay using a finite number of NACK messages.

Out of Order Execution. Modern architectures may alter program order to gain efficiency. In particular, reads may be "hoisted" before a preceding write, to reduce the overhead of clearing the core's write buffer after every load. From this point of view, the `lease` instruction is almost identical to a standard `prefetch` instruction. It can be moved before preceding instructions without impacting correctness. However, over-eager reordering may extend the lease interval artificially, by introducing extra instructions inside the lease interval. This reordering may impact performance if performed aggressively, since it may cause leases to expire.

The `release` instruction should not be reordered, since artificially delaying the release may cause performance issues. (Please see Section 7 for examples.) Thus, we propose that this instruction should have memory fence semantics. While the overhead of memory fences is usually significant, we believe that it is reduced in this case, since the release usually follows an instruction with fence semantics, such as CAS, which already clears the store buffer.

Prioritization. One potential optimization is to split coherence requests into "regular" requests (loads, stores, etc.) and "lease" requests, where lease requests are given *lower* priority. More precisely, a "regular" request automatically breaks an existing lease, while a lease request is queued at the core. This optimization requires an extra "priority" bit in the coherence message, but can improve performance in practice.

Speculative Execution. In complex cores, Lease/Release will need to recover from misspeculation. For instance, a branch misprediction which acquires a lease will need to release it automatically. Since lease usage is advisory, i.e., early release does not affect program correctness, releasing all currently held leases is always a valid strategy for recover from misspeculation. Moreover, we believe that Lease/Release could significantly benefit from a speculative mechanism which keeps track of leases which cause frequent involuntary releases, and ignores the corresponding lease. More precisely, such a mechanism could track the program counter of the lease, and count the number of involuntary releases or the average number of cycles between the lease and the corresponding release. If these numbers exceed a set threshold, the lease is ignored.

Cheap Snapshots. We also note that the variant of Lease/Release which returns a boolean signaling whether the `release` was voluntary can be used to provide inexpensive lock-free snapshots, as follows. The snapshot operation first leases the lines corresponding to the locations, reads them, and then releases them. If all the releases are *voluntary*, the values read form a correct snapshot. Otherwise, the thread should repeat the procedure. This procedure may be cheaper than the standard double-collect snapshot.

6. Detailed Examples

Leases for TryLocks. 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.

The Non-Blocking Michael-Scott Queue. For completeness, we give a brief description of the non-blocking version of this classic data structure, adapted from [27, 37]. Its pseudocode is given in Figure 3. (Our description omits details related to memory reclamation and the ABA problem, which can be found in [37].)

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.

There are several ways of employing leases in the context of the Michael-Scott queue. One natural option is to lease the `head` pointer (for a `Dequeue`) and `tail` pointer (for an `Enqueue`) at the beginning of the corresponding `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.

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 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.

Leases for MultiQueues. MultiQueues [36] 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.

We use leases in the context of MultiQueues as described in Algorithm 4. On `insert`, we lease the lock corresponding to the queue, releasing it on unlock, as described in Section 6. 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 6, 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

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

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

function ENQUEUE( value v )
    node* w ← new node ( v )
    node* t, n
    while true do
        Lease( & tail, MAX_LEASE_TIME )
        t ← tail
        n ← t→next
        if t = tail then
            if n = NULL then ▷ tail pointing to last node
                if CAS( & t→next, n, w ) then ▷ add w
                    CAS( & tail, t, w ) ▷ swing tail to inserted node
                    Release( & tail )
                    return                                  ▷ Success
            else ▷ tail not pointing to last node
                CAS( & tail, t, n )                         ▷ Swing tail
        Release( & tail )

function DEQUEUE( )
    node* h, t, n
    while true do
        Lease( & head, MAX_LEASE_TIME )
        h ← head
        t ← tail
        n ← h→next
        if h = head then ▷ are pointers consistent?
            if h = t then
                if n = NULL then
                    Release( & head )
                    return NULL                       ▷ empty queue
                CAS( & tail, t, n )        ▷ tail fell behind, update it
            else
                ret ← n→v
                if CAS( & head, h, n ) then ▷ swing head
                    Release( & head )
                    break                                 ▷ success
        Release( & head )
    return ret
```

Algorithm 4 MultiQueues [36] with Leases.

```
class MultiQueue { p_queue MQ[M]; lock_ptr Locks }
                                  ▷ Locks[i] points to lock i

function DELETEMIN( )
    int i, k
    while true do
        i = random(1, M)
        k = random(1, M)                       ▷ k can be chosen ≠ i
        MultiLease(2, MAX_LEASE_TIME, Locks[i], Locks[k] )
        if try_lock ( Locks[i] ) then
            if try_lock (Locks[ k ] ) then
                i ← queue containing higher priority element
                k ← index of the other queue
                unlock( Locks[k] )
                ReleaseAll( )
                rtn ← MQ[ i ].deleteMin()              ▷ Sequential
                unlock( Locks[i] )
                return rtn
            else
        ▷ Failed to acquire Locks[k]
                unlock( Locks[i] )
                ReleaseAll( )
        else
        ▷ Failed to acquire Locks[i]
            ReleaseAll( )

function INSERT( value v )
    node* w ← new node ( v )
    while true do
        i = random(1, M)
        Lease( Locks[i], MAX_LEASE_TIME )
        if try_lock ( Locks[i] ) then
            MQ[ i ].insert( w )                        ▷ Sequential
            unlock( Locks[i] )
            Release( Locks[i] )
            return i
        else
            Release( Locks[i] )
```

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.

7. Empirical Evaluation

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.

The Graphite simulator is loosely synchronized, which reduces the number of interleavings with respect to real hardware. 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 ($\leq 30\%$). Hence, our results may underestimate the benefits of adding leases on a real implementation.

Further, Graphite simulates simple in-order cores, therefore reordering effects are not simulated in our results, and only basic one-bit branch prediction is applied. The directory structure in Graphite implements a separate request queue per cache line, and hence effects due to collisions on the same queue are not simulated.

Experiments. We have tested leases for a range of classic concurrent data structure implementations, including the Treiber stack [41], the Michael-Scott queue [27], the Lotan-Shavit skiplist-based priority queue [4, 23], the Java concurrent hash table, the Harris

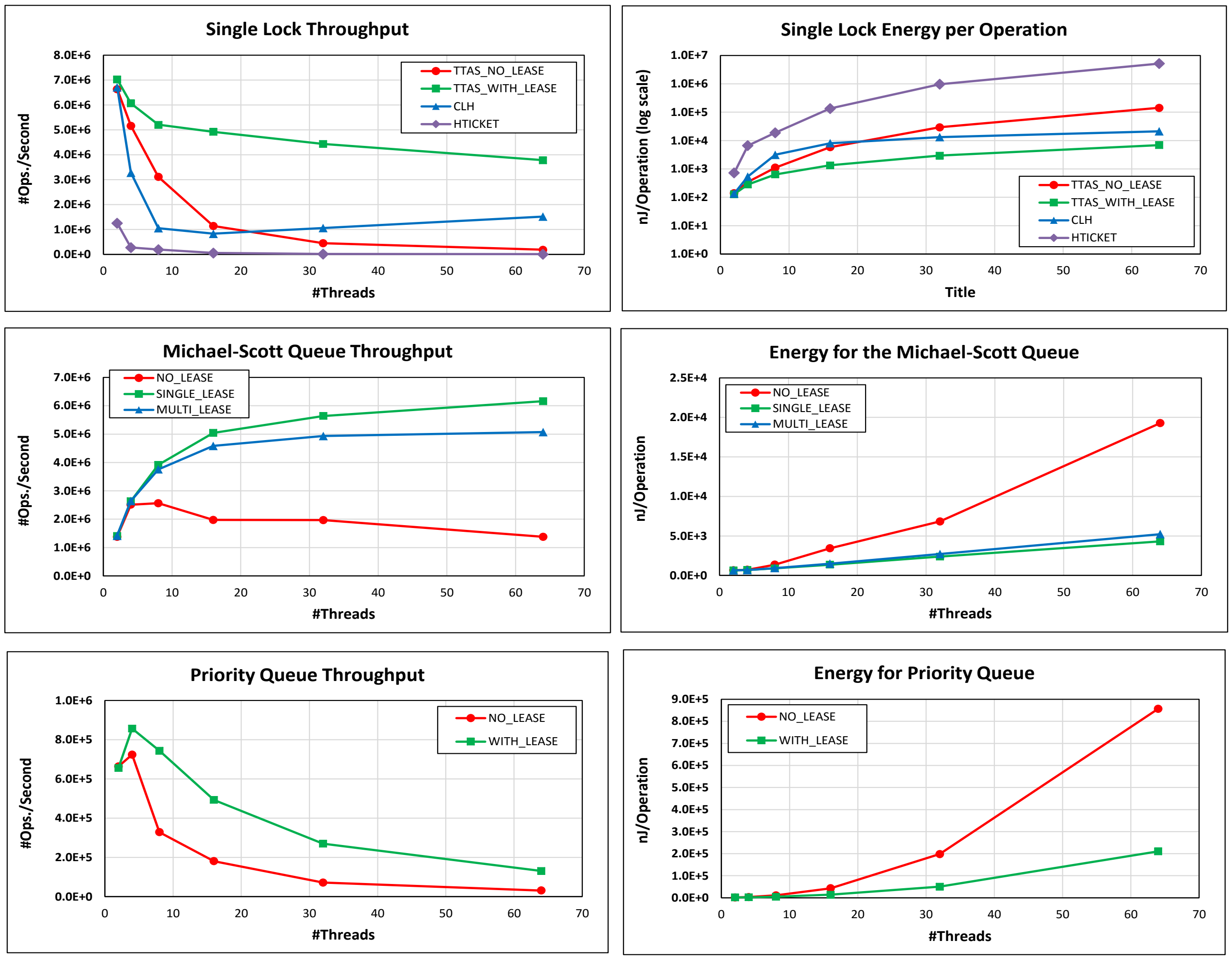

Figure 3: Throughput and energy results for lock-based counter, queue, and skip-list-based priority queue. We tested for 2, 4, 8, 16, 32, 64 threads/cores.

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)

lock-free list [17], and skiplist implementations [15, 33]. We also compared lock throughput against optimized hierarchical ticket locks [8] and CLH queue locks [6, 24]. We tested multiple leases on queues, lists, MultiQueues [36] and the TL2 transactional algorithm [11]. Some of the implementations are built on top of code from the ASCYLIB library [8]. Using Lease/Release usually entailed modifying just a few lines of code in the base implementation, similarly to the examples given in Algorithms 1–4.

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 [33]. 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 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 cycles. 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.

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 up to 3x over the base implementation, but is considerably inferior to using leases. For instance, for the stack, we also compared against a highly optimized implementation with carefully chosen backoffs [14]. The implementation of [14] 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 [17], skiplists [15], 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 (≥ 32).

MultiLease Examples. To test multiple leases, we have implemented MultiQueues [36], and a variant of the TL2 STM algorithm [11]. In the MultiQueue benchmark, threads alternate between `insert` and `deleteMin` operations, implemented as described in Section 6, 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. Figure 4 illustrates the results.

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, although it suggests that single leases may be useful even in transactional scenarios.

Additional Experiments. Figure 5 (left) presents a comparison of software and hardware MultiLeases on the TL2 benchmark. Their performance is comparable; software MultiLeases incur a slight, but consistent performance hit because of the extra software operations, and because joint leasing is not guaranteed. Figure 5 (right) considers the lock-based Pagerank implementation of [2]. In this application, the variable corresponding to inaccessible pages in the web graph (around 25%) is protected by a contended lock. Protecting this critical section by a lease improves throughput by 8x at 32 threads, and allows the application to scale.

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. This issue can be mitigated by two mechanisms: a thread should immediately release a lock that is already owned, and the prioritization mechanism discussed in Section 5 ensures the lock owner's reset instruction has high priority, and automatically breaks an existing lease.

One potential complication is *false sharing*, i.e. inadvertently leasing multiple variables located on the same line. (For instance, in the MS queue example, the `head` and `tail` pointers may be located on the same cache line.) False sharing may significantly degrade performance by increasing contention, and inducing cyclic dependencies among lease requests. This behavior can be prevented via careful programming, and could be enforced automatically via the compiler, by ensuring that leased variables are allocated in a cache-aligned fashion.

8. 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.

Implementation Proposal. We have investigated several lease semantics. We find that the variant which allows a core to lease *a single line* at any given time provides a good trade-off between performance improvements, ease of verification, and the complexity of the hardware implementation. In particular, this variant should not require modification of the directory logic, and needs relatively small changes at the core. Empirical evidence suggests that single leases are sufficient to significantly improve the performance of contended data structures and applications.

Other Protocols. For simplicity, our presentation assumes a basic MSI coherence protocol. Lease/Release also applies to MESI and MOESI-type protocols, with the same semantics: a core leasing a line demands it in Exclusive state, and will delay incoming coherence requests on the line until the (voluntary or involuntary) release. (A leased line cannot be in *Owned* state, since this state implies that a coherence request has already been served since the point when the line was in Exclusive state.) Similarly, Lease/Release can be applied to non-MSI protocol types, such as Tardis [42], to delay messages which would downgrade the ownership level for a leased cache line. The protocol requirements for the applicability of Lease/Release are discussed in Section 5.

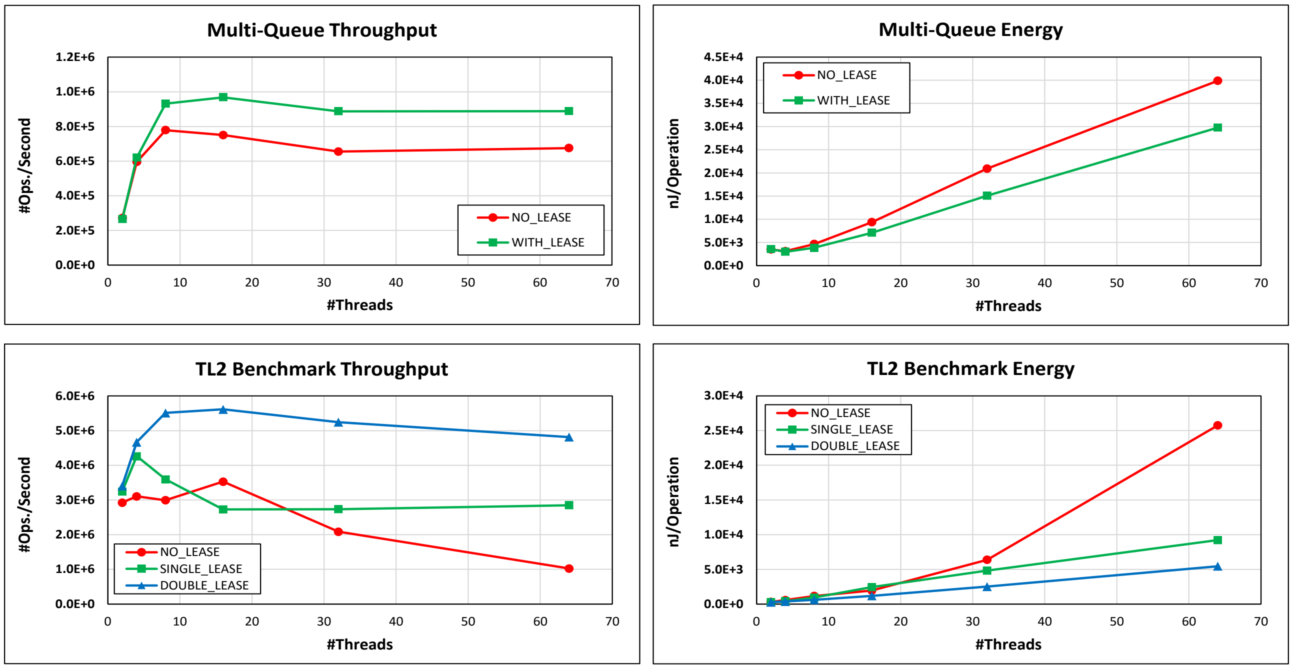

Figure 4: Throughput and energy graphs for MultiLease benchmarks. We tested for 2, 4, 8, 16, 32, 64 threads/cores.

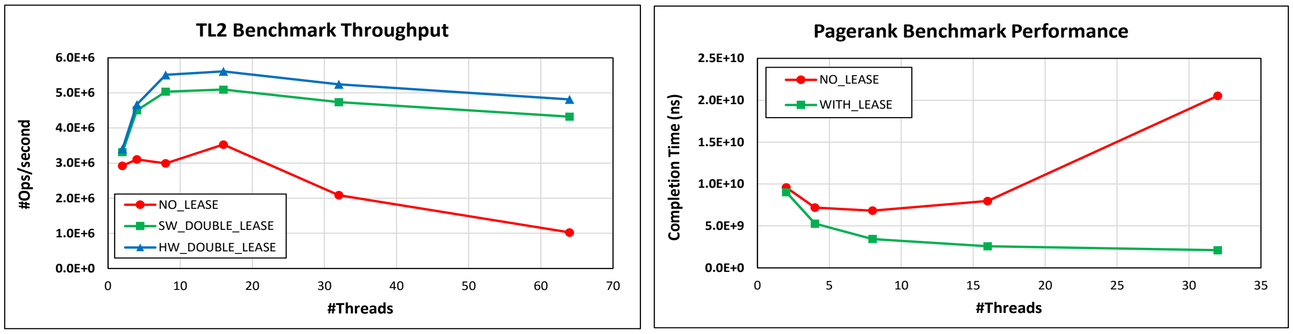

Figure 5: Experiments for Hardware versus Software Multi-Leases, and the Lock-based Pagerank Implementation of [2].

Future Work. Lease/Release is not without limitations. It requires careful programming; for lock-free data structures, a basic understanding of the underlying mechanics is required. Improper use can lead to performance degradation. To address this, we plan to investigate *automatic* lease insertion, using compiler and hardware techniques. The first goal is to automatically identify lease-friendly patterns, reducing the likelihood of erroneous use. Second, it would allow automatic optimization of lease times.

A second topic for investigation is leasing in the context of transactional memory (TM). In particular, recent work suggests that hardware TM has limited performance under contention [30]; using leases inside short hardware transactions could reduce these costs. In general, the usage of leases in the context of TM appears an interesting topic 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.

9. Acknowledgments

We would like to thank Richard Black, Miguel Castro, Dave Dice, Aleksandar Dragojevic, Maurice Herlihy, Ant Rowstron, Nir Shavit, and Vasileios Trigonakis, as well as the anonymous reviewers, for helpful suggestions during the development of this paper.

References

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

[2] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan. Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pages 44–55. IEEE, 2015.

[3] 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.

[4] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit. The spraylist: A scalable relaxed priority queue. In *Proceedings of the 20th ACM SIG-*

PLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, pages 11–20, New York, NY, USA, 2015. ACM.

[5] 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.

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

[7] T. Crain, V. Gramoli, and M. Raynal. A speculation-friendly binary search tree. *ACM SIGPLAN Notices*, 47(8):161–170, 2012.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

[14] 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.

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

[16] 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.

[17] 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.

[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] 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.

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

[34] 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.

[35] 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.

[36] 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.

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

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

[39] 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.

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

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

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