Exploring Time-Space trade-offs for *synchronized* in Lilliput

Dave Dice
Oracle Labs

Alex Kogan
Oracle Labs

Abstract

In the context of project *lilliput*, which attempts to reduce the size of object header in the HotSpot Java Virtual Machine (JVM), we explore a curated set of synchronization algorithms. Each of the algorithms could serve as a potential replacement implementation for the "synchronized" construct in HotSpot. Collectively, the algorithms illuminate trade-offs in space-time properties.

The key design decisions are *where* to locate synchronization metadata (monitor fields), *how* to map from an object to those fields, and the lifecycle of the monitor information.

The readers is assumed to be familiar with current HotSpot implementation of "synchronized" as well as the Compact Java Monitors (CJM) design [13] and Project Lilliput [1].

2012 ACM Subject Classification Software and its engineering → Multithreading; Software and its engineering → Mutual exclusion; Software and its engineering → Concurrency control; Software and its engineering → Process synchronization

Keywords and phrases Synchronization, Locks, Mutual Exclusion, Scalability, Java Virtual Machines, HotSpot

1 Algorithms

All the candidate locking algorithms below are FIFO/FCFS-fair, unless stated otherwise, and are able to support the full gamut of `synchronized` operators. All are space-conserving in the sense that acquiring a monitor requires that a thread contribute a *lock record* to some set of records associated with the object, and releasing that monitor reclaims that same lock record, avoiding the *objectMonitor* accretion concerns attendant in the existing HotSpot monitor implementation.

1.1 HashChains

In this variant, `synchronized` operations do not access the object header. The header word is never displaced by synchronization and no bits in the header are reserved for synchronization. This approach is appealing as avoids it multiplexing and overloading of the header word, simplifying the encoding thereof and reducing couplings between the synchronization subsystem and other components on the JVM.

The JVM maintains a shared hash table of synchronization buckets. Each bucket contains a simple mutex lock, and a pointer to the head of a linked list of *lock records*. Similar constructs in the linux kernel are sized at startup time, based on the number of logical processors. For all experiments reported herein, we used 4096 buckets. If necessary the hash tables could be resized at runtime.

To acquire a monitor, a thread first allocates and constructs a lock record, and then identifies the bucket associated with the object. We can hash the virtual address of the object to map to the bucket, although this would require rehashing in the event of moving garbage collections. To avoid that, we could instead hash on the object's identity hashCode value, although this would necessitate assigning identity hashCode values on objects that participate in synchronization. Our native C++ implementation hashes on the virtual address. The thread next acquires the bucket lock, and emplaces its lock record on the chain. The bucket locks can be simple `pthread` mutex locks, or any other simple native lock, such as MCS locks [20]. Our implementation uses *hemlock*[15, 16]. The thread then determines if there are any conflicting lock records on the chain – lock records inserted by other threads that refer to the same object. Subsequently, the thread releases the bucket lock. If no conflicting elements were observed, then the thread acquired the monitor without contention and is the owner of the monitor, and as such, may enter the critical section without waiting. Otherwise the thread parks, waiting on a field in its lock record to change state indicating that it has been granted ownership.

The lock record posted by a thread continues to reside on the chain while the thread executes in the critical section. There is no singular central memory location that encodes if an object is locked, but rather the *locked-ness* is determined by the presence or absence of conflicting lock records on the object's hash chain. Lock records include a "state" field which indicates if the associated thread holds the lock, is waiting on the lock, or is in `wait()`. The set of lock records resident in the bucket and associated with a given object form the *entry set* for that object.

To release a monitor, a thread again acquires the bucket lock and removes the element it originally posted, moving that element to a thread-local free list, allowing for subsequent re-use. As threads typically hold a very small number of locks concurrently, we can use thread-local free lists (stacks) of free lock records. The thread also checks for and identifies a successor from the set of lock records associated with the object. If no successor is found, the thread simply drops the bucket lock, otherwise it marks the successor's lock record as being the current owner, releases the bucket lock, and unparks the successor.

To allow for efficient `IllegalMontorStateExceptions` checks, and for nested locking, and automatic unlocking of monitors, each thread maintains a list of lock records reflecting the monitors it currently holds.

In our implementation, instead of a simple unstructured "bag" of lock records on the bucket chain, we use a *spine-and-rib* design where the spine elements reflect the current owner, and all remaining threads waiting on that object are linked as ribs off that spine element, in order to reduce traversal times and thus reduce hold times for the bucket locks.

Lock records could be implemented as native C++ constructs, or as first-class Java objects. In the case of the latter, much of the synchronization subsystem could be shifted into pure Java code.

No type-stable memory (TSM) or safe memory reclamation (SMR) is required. Furthermore the design places tight bounds on the amount of memory required for synchronization. No explicit deflation step is required. Trimming of thread-local free lists of lock records, if it is every needed – which is unlikely – is a strictly thread-local decision.

While simple, this approach entails a number of performance challenges. To acquire and release an uncontended monitor, we need to acquire and release the bucket lock twice, once to post the lock record to the chain, and another to extract it. This results in poor latency compared to other approaches. The traffic on the chain locks arising from such "double locking" also impinges on scalability. (Locks that are implemented with an "inner" or "meta" lock to protect their queues are usually inferior in performance locks that avoid inner locks). In addition, we're exposed and vulnerable to *false contention* on the bucket locks because of hash collisions. Even absent bucket lock collisions, we can incur false sharing and costly coherence traffic when multiple unrelated synchronization operations interleave accesses on a given bucket. Accordingly, both latency and scalability suffer.

1.2 HashChains+3

HashChains+3 builds on **HashChains** but also requires 3 bits to be reserved for synchronization in the object's header word. The bits encode *Locked, WaitersExist, and Impatient* indicators. Our implementation does not currently use *Impatient*, but we reserve the bit to allow Fissile Locks [12, 14] to provide bounded bypass, if desired, to improve contended performance. As expected *Locked* indicates the lock is held. *WaitersExist* is a conservative indication – false-positives are allowed but never false-negatives – that contending threads exist on the associated hash chain. The indicator allows fast-path uncontended unlock operations when there are no waiting threads.

Arriving threads first attempt to acquire the lock by using CAS to toggle the *Locked* bit from 0 to 1. Unlike **HashChains**, the owner's lock record does not reside on the chain. Failing to acquire via that fast-path, threads ensure the *WaitersExist* bit is set and emplace on the chain, under the bucket lock, and then park in the usual fashion, waiting on the "state" field in its own lock record. The chain contains only waiting threads, and never the owner.

To release the monitor the thread first consults the *WaitersExist* bit. If clear, the thread attempts to use CAS to clear the lock bit, while ensuring the *WaitersExists* bit remains clear. If the CAS was successful, no further actions are required. Otherwise, if *WaitersExists* was set, the thread acquires the bucket lock and detaches a successor, if any. If no additional conflicting threads (beyond the successor) are present on the chain, it also clears the *WaitersExist* bit. The thread then drops the bucket lock, marks the successor's lock record as being the new owner, and unparks the successor, passing ownership directly to the successor. The *Locked* bit remains held continuously while ownership is conveyed. If no successor was found on the chain, the thread clears both the *Locked* and *WaiterExist* bits and drops the bucket lock.

Critically, uncontended acquire and release operations do not need to access the hash chains, improving latency. The hash chains are needed only under contention. Related ideas worthy of note can be found in `WebKit` [21].

1.3 CJM

Compact Java Monitors (CJM) [13] are based on the Compact NUMA-Aware Locks (CNA) algorithm, but forego the *NUMA-Aware* property and focus instead on the *Compact* aspect. CNA is itself a variation on the gold-standard MCS (Mellor-Crummey Scott) [20] queue-based lock algorithm [1].

[1] If you're unfamiliar with CJM, please see [13]. That document also includes a description of the MCS algorithm in an appendix. Briefly, MCS uses atomic `SWAP` and `CAS` primitives to construct a lock-free queue – implemented as a singly linked list – of elements, where each element serves to represent an allocation request by the thread that posted that element. The head element is the current owner and the MCS lock word points to the tail.

CNA was published in EuroSys 2019 [11, 7] and is being integrated into the Linux kernel as a replacement for the existing low-level *qspinlock* construct [4, 19], which is itself based on MCS. One of the key ideas in CNA is propagating values of interest from the MCS owner's queue element into the successor, which allows the lock body to remain compact – just one word. Specifically, fields that would normally appear in the body of a lock are instead maintained in the owner's queue element and, at unlock-time, conveyed to the successor in the queue.

In the context of the current discussion we're not interested in NUMA-aware aspects of CNA, where we propagate the list of remote queue elements through the MCS chain (avoiding extra fields in the lock body), but instead leverage the fact that the lock body is compact. Taken to the extreme, CJM shifts *all* the fields that would normally reside in the classic HotSpot *objectMonitor* construct into the MCS queue elements, so we can represent the abstract Java monitor with just a single pointer to the MCS tail.

When an object is locked, the implementation relocates the entire header word into a *displaced header* which is conveyed through the chain. The identity hashCode value will be assigned on the first synchronization operation on a given instance, in order to avoid mutating the displaced header. Relatedly, we assume the encoded class (*klass*) information in the header is immutable, or at least rarely mutated. And finally, we assume that the GC *age* bit field in the header word is also mutated infrequently [2]. See Section 6 of [13] for details.

Accessing the header fields (identity hashCode value, class information, etc) is relatively simple under CJM. First, if the object is not engaged in synchronization, those fields reside in their usual "home" position in the object header. If the object happens to be locked by the thread attempting to fetch the header, which is easily determined, the caller can quickly extract the displaced header value from the queue element (lock record) it originally posted to the MCS chain. Finally, if the object is locked by some other thread, which we expect to be rare, we can use the access protocol described in [13] Section 6 to extract the header word value. This final mode uses a chase-and-capture idiom that enjoys obstruction-free progress properties.

Briefly, CJM provides the following desirable properties:

- Eliminates stack-locking, resulting in a simple unified encoding in the header, and only one flavor of locking.
- Performance on-par with the existing subsystem.
- Extremely simple with fewer lines of code, and easily maintainable.
- FIFO-fair admission policy, whereas the existing system admits unbounded bypass and starvation.
- Reduced interactions and dependencies on other subsystems : safepoint, GC, etc.
- Avoids accretion of objectMonitors and deferred deflation to recover of those monitors. Threads contribute one queue element to a monitor's queue when they acquire the lock, and recover that same element when they release the lock, resulting in space-conserving *self-cleaning* pay-as-you-go memory use with extremely tight bounds. This property is a key desideratum and improvement over the existing `synchronized` implementation.

[2] The object header fields typify Conway's Law [23].

– As described, CJM is FIFO, but to gain more performance we can allow bounded bypass using the *Fissile Locks* technique [12, 14].

2 Waiting Policies

The choice of waiting policy – how a thread waits to acquire a contented lock – has subtle ramifications.

First, when executing in native C++ code we need to respect HotSpot's "no loitering" invariant. Threads must not busy-wait indefinitely. Bounded spinning is acceptable, but threads waiting for some condition to be satisfied must be prepared to resort to parking after some period. In addition, indefinite polling via operating system provided primitives such as `sched_yield` is not sufficient. Polling via timed sleeping is also not acceptable, as operating system timers don't scale and can also prevent systems from entering low power states.

In pure Java code, indefinite spinning is permissible but unwise. Say, for instance, that a lock implemented in pure Java spins indefinitely without parking. Furthermore say that we have more ready threads than logical CPUs, so operating system preemption (involuntary preemption) is in play. If the lock holder is preempted then we have to wait for spinning threads to be preempted and the CPU to be passed to the lock holder to achieve succession and progress. Similar stalls can occur if the lock holder transfers ownership to a preempted successor. Involuntary preemption (time slicing) operates on a relatively long time frame (1 to 10 msecs, which is effectively geologic time) and as such, throughput over the contented lock will suffer.

The current `synchronized` implementation uses adaptive spinning where we try to use the success or failure rate of recent spin attempts on a given object to better inform the spin-park decision and spin duration. `ReentrantLock` does not spin and immediately parks on contention.

Our CJM implementation uses a simple spin-then-park STP waiting policy. Simple STP using a bounded spin duration of half the voluntary context switch "round trip" latency is *2-competitive* versus the ideal. That is, STP with a fixed spin duration of half the context switch latency is never more than 2x worse (performance) than an idealized schedule where we have an "oracle" with perfect foreknowledge – a so-called *offline* algorithm – that tells us in advance the waiting time, and thus lets us decide immediately and with perfect certainty whether to spin or park when a thread needs to wait for a lock. As we don't have such foreknowledge, the spin-versus-park decision is considered an *online* problem [3]. The spin phase uses the PAUSE instruction on x86, although MONITOR-MWAIT would be a better choice if available. (PAUSE loops executed in virtual machines trigger VM exits, which inform the VM that busy-waiting is taking place, and inform the VM that gang coscheduling may be called for).

Parking, instead of unbounded spinning, also reduces the number of ready threads, and acts to forestall the onset of preemption.

The CJM and HashChain implementations reported herein used a maximum (bounded) spin duration of 2500 PAUSE instructions before they devolve to parking, rechecking the

[3] The spin-park problem is equivalent to the *Ski Rental Problem* [24].

condition of interest after every PAUSE instruction. Specifically, we execute a loop where each iteration checks (polls) the condition of interest. If satisfied, we return immediately. Otherwise we execute a single PAUSE instruction. After 2500 iterations if the condition is not satisfied, we revert to parking the waiting thread. Spinning is used solely as an optimistic "bet" to avoid parking and the overheads of context switching. (Unfortunately the typical latency of a PAUSE instruction varies by a factor of 10 between steppings on Intel CPUs). All spinning in our implementations is *local* with at most one thread busy-waiting on a given location at any given time in order to reduce coherence traffic. In contrast, simple test-and-set and ticket locks use *global* spinning.

Busy-waiting or spinning should be performed in a *polite* fashion, which minimizes the performance impact of spinning on other threads in the system. Impolite naive spinning can impede the performance of other coscheduled threads. Crucially, we don't want the spinning thread to compete for resources such as power and thermal caps, turbo-mode enablement, cache residency, interconnect or memory bandwidth, pipelines, etc. On x86, for instance, the PAUSE instruction can be used for polite spinning. Altruistic spinning may even benefit the spinning thread, as the lock owner may execute its critical section and relinquish the lock in a more timely fashion, avoiding the so-called *tragedy of the commons*. Note that we intentionally avoid using `sched_yield` on linux as, with the advent of per-CPU dispatch queues, it is advisory and has no particular semantics. Yielding is also not polite to sibling threads co-executing on the system.

The **succession policy** is convolved with waiting policies. Succession describes how a lock algorithm conveys ownership to waiting successors.

Direct handoff transfers ownership directly from the outgoing thread to the designated successor. Examples include CJM, MCS, CNA, the linux kernel's `qspinlock` and Java's `ReentrantLock` in fair (FIFO) mode. All FIFO/FCFS lock algorithms use direct handoff but direct handover locks are not necessarily FIFO. Direct handover perform poorly when preemption (involuntary context switching – oversubscription) is in play, as would be the case when the number of ready participating threads exceeds the number of logical CPUs. In particular, direct handoff allows ownership to be transferred to preempted thread. Direct succession also performs poorly when parking is in use as the overheads incumbent in voluntary context switching are subsumed into the critical section, greatly increasing the effective length of critical sections. If we unpark a successor, it typically takes more than 10000 cycles for the wakee to resume and return from park (and even more if the wakee is dispatched onto a CPU that was previously idle in deep sleep states). Using a spin-then-park waiting policy can provide some relief against that problem. Unfortunately, if we use a spin-then-park waiting policy with a FIFO admission policy, the immediate successor will be the thread that has waiting longest, and is thus most likely to have consumed its spin allotment and resorted to parking. This can result in a rather abrupt drop in performance in certain areas of the parameter space, violating the *principle of least surprise*.

All the `HashChains` forms are ostensibly FIFO. But, as they use an additional internal lock to protect their queues, it is possible that thread $T1$ arrives before $T2$ in when calling $Lock(L)$ but $T2$ beats $T1$ to the inner lock and enqueues before $T1$. In practice this is not a significant concern.

Competitive handoff releases ownership and, if necessary, "pokes" a potential successor – the heir apparent – to retry the lock and ensure progress. Competitive handoff is also known as *barging*, *bypass*, *pouncing* or *renouncement*. After the owner releases ownership in `unlock()`, newly arriving threads in `lock()` can pounce on the lock, and bypass (barge past) other threads that have waited longer. Example of algorithms and implementations that use competitive handoff include Java's `synchronized`, `ReentrantLock`, `pthread_mutex` primitives, test-and-set locks, etc. Competitive handoff admits long-term unfairness and starvation. In addition, the degree of unfairness is influenced by the fairness of the platform's cache arbitration policy, and thus varies. Perversely, simple test-and-set locks are NUMA-friendly on x86 platforms, as threads on the same node are more likely to acquire ownership of a just released lock.

Competitive succession, however, tends to be more tolerant of preemption, as the successor, by definition, was on a CPU when it acted to acquire the lock, avoiding the scenario where we pass ownership to a preempted thread as can manifest with direct handoff.

As a general observation, locks that use competitive handover will yield better aggregate throughput than those that use direct handover. This drives the use of competitive handover for common locks such as `ReentrantLock`, default `pthread_mutexes` and Java's `synchronized`. Competitive succession, however, allows unbounded unfairness.

Approaches such as *Concurrency Restriction* [9, 10] and *Malthusian Locks* [6, 5] act to intentionally reduce the number of threads circulating over a lock in some period, and improving the performance of various lock algorithms, particularly those that use direct handoff.

We can capture the best attributes of competitive handoff and direct handoff with hybrid schemes such as Fissile Locks [12, 14], which allow competitive succession over the short term, but if waiting threads become *impatient* we switch briefly to direct handover to avoid starvation. The aggressiveness with which waiting threads determine they have become impatient acts as a tunable "knob" to strike an explicit trade-off between throughput and fairness, and reflects the inherent tension between those properties. (Infinite patience is equivalent to full competitive succession, and *no* patience makes Fissile locks equivalent to directed handoff). The tactic provides *bounded bypass*. CJM-By (below) employs this approach. The implementation used in this paper inhibits bypass after the longest waiting thread has waited more than 1 millisecond.

Addition discussion on the topic of waiting and succession policies can be found in Section 5.1 of [5].

3 Empirical Results

Unless otherwise noted, all data was collected on an Oracle X5-2 system. The system has 2 sockets, each populated with an Intel Xeon E5-2699 v3 CPU running at 2.30GHz. Each socket has 18 cores, and each core is 2-way hyperthreaded, yielding 72 logical CPUs in total (2x18x2). The system was running Ubuntu 20.04 with a stock Linux version 5.4 kernel, and all software was compiled using the provided GCC version 9.3 toolchain at optimization level "-O3". 64-bit C++ code was used for all experiments. Factory-provided system defaults were used in all cases, and Turbo mode [22] was left enabled. In all cases default free-range unbound threads were used (no pinning of threads to processors).

All the underlying native C++ locking algorithms support the full gamut of synchronization and hashCode operations. This framework – the locks, exposing a common API, and the associated benchmarks – serve as a useful and faithful in-vitro model for the performance of `synchronized` activities in HotSpot. The locking algorithms are implemented via portable C++ `std::atomic` primitives and a park-unpark interface based on per-thread mutex-condvar pairs. By using `std::atomic`, no explicit memory fences or barriers were required. To reduce false sharing, each lock record was sized at 128 bytes and aligned accordingly.

We collected performance data with a synthetic `MuteBench` micro-benchmark, which measures performance under lock contention. When we transliterate the benchmarks from C++ to Java, we see that the C++ CJM algorithms yields about the same performance as the existing HotSpot `synchronized` implementation. We note that the existing HotSpot `synchronized` implementation is superior to equivalent `pthreads` code for both contended and uncontended operations, a property expected by developers and which we want to retain. [4].

Note that ideal scalability, even absent and communication or contention, is elusive because of conflicts for shared resources, caps on energy, and thermal constraints. See Section 7 *Maximum Ideal Scalability* of [8].

The `MutexBench` benchmark spawns T concurrent threads. We have a global pool of NL shared locks. NL is configured as 1 in all runs reported below. Threads iterate as follows. Each thread selects NA random locks, without replacement, from the set of NL, forming a lockset. (The locks are shared, but the locksets are thread-private). To avoid deadlock, we sort the lockset by address. We also configure NA as 1 for all runs reported below [5]. Each thread acquires all the elements of its lockset, and then executes a critical section of CSL steps of a shared C++ `std::mt1993` pseudo-random number generator. The thread then immediately releases the lockset elements in reverse order, mimicking Java's usual last-acquired-first-released pattern. The thread then executes a non-critical phase where it first computes a uniform random value in the range $[0, NCSL * 2)$ and then executes that many steps of a a thread-local `std::mt1993` random number generator. The average and median duration of the non-critical phase is $NCSL$ steps of the thread-local random number generator. We intentionally randomize the non-critical section duration to avoid *entrainment* where threads would otherwise tend to enter the critical section in a relatively stable cyclic order, which can persist for long periods, and where the number of NUMA node transitions inherent in that schedule is determinative of performance. $NCSL$ and

[4] Say we have group of threads contending on a single high-traffic `pthread` mutex, where threads arrive and depart frequently. The user-mode `pthread` mutex implementation in `glibc` does not use spinning, so contended threads immediately resort to blocking in the kernel via the *futex* mechanism. The linux kernel spin lock that protects the futex hash chain associated with that user-mode mutex address will itself become highly contended, often to the point where most of the waiting time is for that spinlock, instead of "normal" waiting for a wakeup notification. When we implemented Compact NUMA-Aware locks (see section 7 of [7]) we observed that improved kernel spin locks (those protecting the futex chains) sped up the futex operations and in turn sped up apps with highly contended pthread mutexes. Equivalent contended `synchronized` code running in the HotSpot JVM saw no such benefit as each thread already blocked on its own parking construct, and therefore we avoided hot futex chains. Put differently, the approach used by the JVM where each thread has private parking constructs serves to diffuse accesses over the set of kernel futex hash chains, avoiding hotspots and conferring an advantage. The decay in scalability exhibited by contended pthreads mutex operations in Figure 1, below, arises from this secondary contention.

[5] We took data over a wide range of NA and NL values but opted to report on runs where both values were set to 1.

CSL are specified as command-line arguments. At the end of a 10 second measurement interval the benchmark reports the total number of aggregate iterations completed by all the threads. We report the median of 7 independent runs. (To assist in detecting potential safety "exclusion" errors, after the measurement interval, the benchmark resets the shared PRNG state and then, in single-threaded execution, advances the PRNG by the total number of iterations completed, and then ensures the final state agrees with the state of the PRNG at the end of the measurement interval).

In the figures below the X-axis reflects the number of concurrently executing threads contending for the lock, and the Y-axis reports aggregate throughput. For clarity and to convey the maximum amount of information to allow a comparison of the algorithms, the X-axis is offset to the minimum score and the Y-axis is logarithmic.

To facilitate comparison, we also include performance data on a version of the `MutexBench` benchmark transliterated to java, using `synchronized`, `ReentrantLock`, and `ReentrantLock` in FIFO mode. We used JDK version 16.0.2 with the following command-line arguments `-server -XX:-UseBiasedLocking -Xmx10G -Xms10G -XX:+EnableContended`. As we also collected data with `ReentrantLock`, we sized the heap at 10Gb as that locking algorithms allocates a new queue element for each acquisition [18] and puts pressure on the allocator and collector. We employed `@Contended` to avoid false sharing between `ReentrantLock` instances and other data elements. To deal with variance, each benchmark process executes 7 sub-runs of 10 seconds each. An external script runs the benchmark 3 times in sequence, with completely independent processes, for total of 21 runs of 10 seconds each. We report the median value of those 21 runs for aggregate throughput.

To allow comparison against simple `pthreads` we also included a form of `MutexBench` where each object contained an full embedded `pthread_mutex` instance.

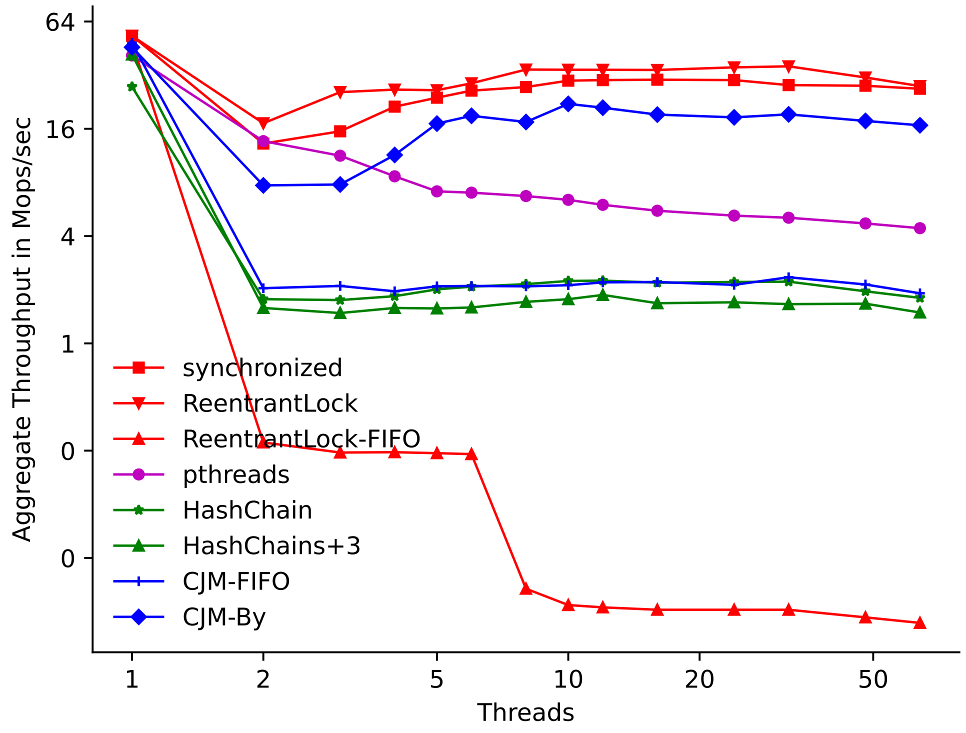

Figure 1 Maximum Contention

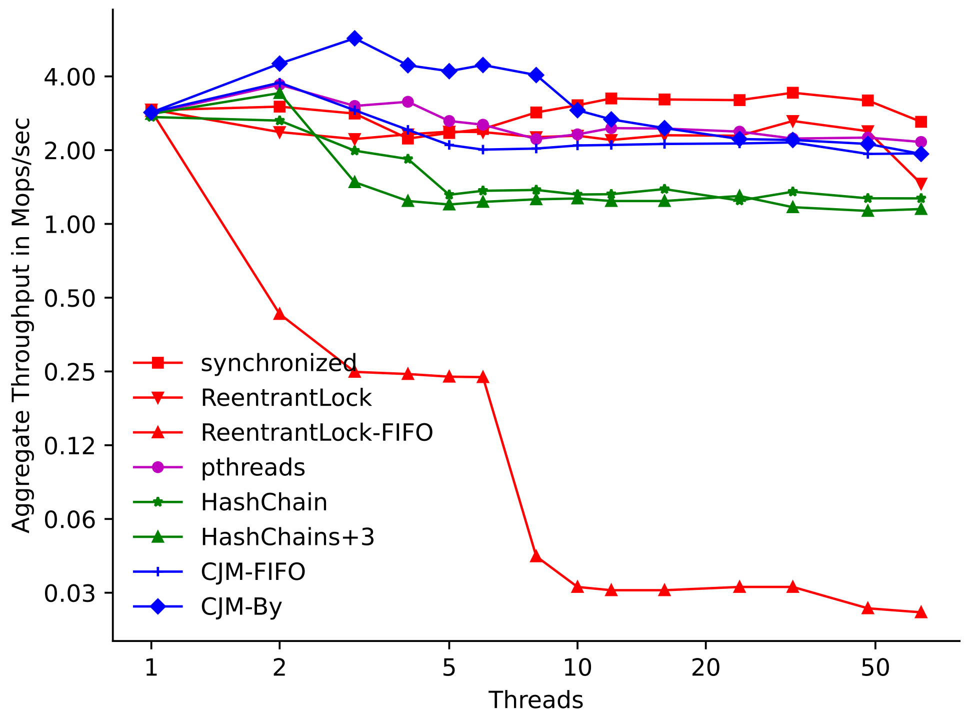

Figure 2 Moderate Contention

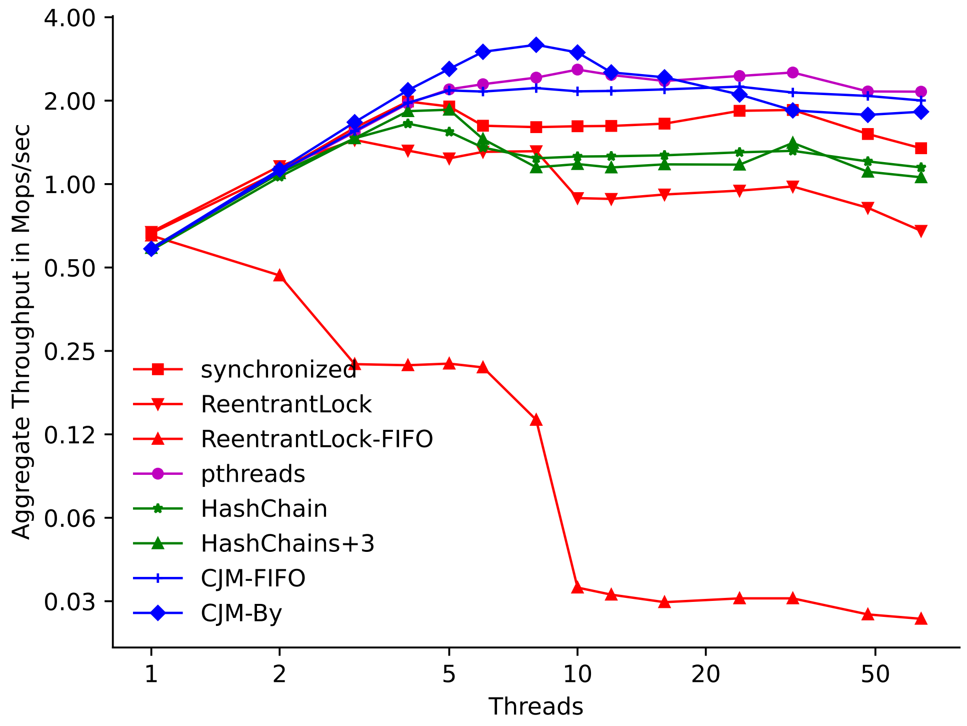

Figure 3 Light Contention

Figures 1, 2 and 3 report the aggregate throughput of contended locking operations with `MutexBench`. `synchronized`, `ReentrantLock` and `ReentrantLock-FIFO` reflect

MutexBench ported to Java. pthreads is a degenerate version of our native C++ locking framework where each object incorporates a pthread_mutex lock. HashChain and HashChains+3 are described above. CJM-FIFO is simple CJM and CJM-By is CJM with bounded bypass enabled.

Figure 1 shows extreme contention, with empty critical and non-critical sections, on a single "hot" lock. *CSL* and *NCSL* are both configured as 0. We see that ReentrantLock-FIFO scales poorly as the cost of park and unpark operations and the futex operations are subsumed into the effective critical section length. The other FIFO forms that use spin-then-block waiting (CJM-FIFO, HashChains, and HashChains+3) tend to cluster in a band. The spin duration is largely sufficient to avoid parking. As noted above, pthreads fades because of futex sleep chain kernel spin lock contention induces by user-level pthread contention. ReentrantLock provides the best performance closely followed by synchronized. ReentrantLock and synchronized admit long-term unfairness. Over a 10 second interval, it is not uncommon to see a 3x difference between the thread that accomplished the most iterations versus the thread that completed the least iterations. CJM-By with bounded bypass performs almost as well as synchronized and ReentrantLock but is much fairer over the measurement interval.

The data at 1 thread also serves as a good measure of uncontended latency. We can see, for instance, that HashChains exhibits increased latency, because of the longer paths needed to acquire and release the bucket locks.

Figures 2 (CSL=1 and $NCSL = 200$) and 3 ($CSL = 1$ and $NCSL = 1000$) also have a single hot lock, but use a very short critical section with longer non-critical sections, reflecting more likely real-world scenarios. Broadly, CJM-By, synchronized and ReentrantLock provide comparable performance.

CJM with bypass provides reasonable scaling – in keeping and competitive with the existing synchronized and ReentrantLock implementation – avoiding the performance collapses and retrograde scaling exhibited by some of the other locks. In addition, CJM with bounded bypass provides long-term fairness, in contrast to pthreads, synchronized and default ReentrantLock, but at the same time remains preemption tolerant, unlike the FIFO locks.

Not surprisingly, dedicating more header word state to synchronization yields better synchronization performance. The HashChain forms, which are appealing simple, put little demand on the header, but suffer relative to the variants which use a displaced header word.

It is worth remarking that our benchmarks used only a single object. If we use multiple objects, then the HashChain forms start to suffer from false contention in the hash chains, even for logically uncontended locking.

4 Additional Remarks

1. The CJM variant should be able, with some additional effort, to tolerate GC algorithms that employ *forwarding pointers*. Presumably the low-order tag bits in the single header word would encode the following possible states : *Normal, Displaced-For-Locking, Forwarded*.
2. If the JVM allows forwarding on-the-fly, by mutators, outside of safepoints, then an additional approach presents itself. The first time we synchronize on a object, we imme-

diately copy and forward the object to an instance that has the CJM synchronization word appended to the object body. At the same time, we change the object's type from T to $T+$ to indicate that the instance now supports the CJM word. Conceptually we're cloning the object and changing the type to a new derived class that contains the CJM word. For a given instance, the state transition is one way. Instead of relying on the type system, an additional bit in the header could also be reserved to indicate the *has-lockword* state. One concern inherent to this approach is that synchronization operations might trigger out-of-memory conditions, although most synchronization designs also admit this same problem.

3. It is worth comparing the idea above to the "tri-state" approach reportedly used by some of IBM's JVM implementations to reduce the size of the header. 2 bits are reserved in the header to support the identity hashCode, encoding 3 possible states : *neutral*, *hashed-by-address*, and *hashcode-appended*. All objects start in neutral state. On the 1st call to query the identity hashCode, the JVM computes the hashCode as a deterministic function of the object's heap address, and shifts the state from neutral to hashed-by-address. Subsequent calls to query the identity hashCode observe the hashed-by-address state and recompute and return the same hashCode value. If and when a hashed-by-address object is moved (copied) by the garbage collector, the collector changes the state to hashCode appended, and extends the object accordingly, computes the hashCode, and stores the identity hashCode value at the end of the object. Subsequent calls to query the hashCode notice the hashcode-appended state and extract the value from the new field appended to the object. An unfortunate consequence of this tactic is that the collector can exhaust memory extending objects during a garbage collection operation. See also `https://github.com/rkennke/lilliput/tree/compact-hashcode`.
   Related ideas can be found in IBM's J9 *lock nursery* [2] `https://blog.openj9.org/2019/04/02/lock-nursery/`

4. Instead of CJM, we could continue to use the existing synchronization subsystem – with all its inherent monitor lifecycle problems – and displace the new lilliput single header into a compact *displaced header word*, instead of the existing *displaced mark* word. See `https://mail.openjdk.java.net/pipermail/lilliput-dev/2021-July/000096.html`. CJM, however, lends itself to displaced headers far more gracefully than the existing synchronized system.

5. Both contended[3] and uncontended performance are critical quality-of-implementation (*QoI*) metrics for the design of a synchronization subsystem. Previously, in the era of *bus locking*, before cache locking, techniques such as *biased locking*[17] were used to improve the latency of uncontended operations. Thankfully, changes in processor architecture have obviated the need for biased locking, and its incumbent complexity. Contended performance is usually measure in terms of scalability – aggregate throughput and Uncontended performance is measured via simple latency. Between the extremes we also find so-called "promiscuous" objects, which are locked by various threads but suffer relatively little contention. In this case, coherence traffic usually dictates performance. Specifically, an implementation should act to minimize write invalidation. While not as actively studied in the literature, performance is promiscuous mode is also of importance.

6. For the hash-based variants we'd want to employ a secondary hash function, likely with a salted nonce, to reduce the threat of DoS attacks against the buckets.

7. Interactions with projects *loom* and *graal* have not been considered.
8. While not required, we assume to the extent reasonable and possible, that the object header will reside on the last word of a cache line, with the constituent object fields starting on the following line. This benefits SIMD and superword optimizations and also reduces the span – the number of cache lines – underlying particular object instances, improving spatial locality [6].
9. We assume a 64-bit JVM with a 64-bit header word.

References

1 Project lilliput, 2021. URL: https://wiki.openjdk.java.net/display/lilliput.

2 David F. Bacon, Stephen J. Fink, and David Grove. Space- and time-efficient implementation of the java object model, 2002. URL: https://doi.org/10.1007/3-540-47993-7_5.

3 Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Non-scalable locks are dangerous. *Ottawa Linux Symposium (OLS)*, 2012. URL: https://www.kernel.org/doc/ols/2012/ols2012-zeldovich.pdf.

4 Jonathan Corbet. MCS locks and qspinlocks. https://lwn.net/Articles/590243, March 11, 2014, 2014. Accessed: 2018-09-12.

5 Dave Dice. Malthusian locks. *CoRR*, abs/1511.06035, 2015. URL: http://arxiv.org/abs/1511.06035, arXiv:1511.06035.

6 Dave Dice. Malthusian locks. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, 2017. URL: http://doi.acm.org/10.1145/3064176.3064203.

7 Dave Dice and Alex Kogan. Compact numa-aware locks. *CoRR*, abs/1810.05600, 2018. URL: http://arxiv.org/abs/1810.05600.

8 Dave Dice and Alex Kogan. TWA - ticket locks augmented with a waiting array. *CoRR*, abs/1810.01573, 2018. URL: http://arxiv.org/abs/1810.01573, arXiv:1810.01573.

9 Dave Dice and Alex Kogan. Avoiding scalability collapse by restricting concurrency. *CoRR*, abs/1905.10818, 2019. URL: http://arxiv.org/abs/1905.10818, arXiv:1905.10818.

10 Dave Dice and Alex Kogan. Avoiding scalability collapse by restricting concurrency. In *Euro-Par 2019: Parallel Processing - 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26-30, 2019, Proceedings*, Lecture Notes in Computer Science. Springer, 2019. doi:10.1007/978-3-030-29400-7_26.

11 Dave Dice and Alex Kogan. Compact numa-aware locks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19. Association for Computing Machinery, 2019. doi:10.1145/3302424.3303984.

12 Dave Dice and Alex Kogan. Fissile locks, 2020. URL: https://arxiv.org/abs/2003.05025, arXiv:2003.05025.

13 Dave Dice and Alex Kogan. Compact java monitors. *CoRR*, abs/2102.04188, 2021. URL: https://arxiv.org/abs/2102.04188, arXiv:2102.04188.

14 Dave Dice and Alex Kogan. Fissile locks. In *International Conference on Networked Systems - NETYS*. Springer International Publishing, 2021. URL: https://doi.org/10.1007/978-3-030-67087-0_13.

15 Dave Dice and Alex Kogan. Hemlock : Compact and scalable mutual exclusion. *CoRR*, 2021. URL: https://arxiv.org/abs/2102.03863, arXiv:2102.03863.

[6] A large fraction of current `malloc` allocators get this precisely wrong, placing the header word on the first word of a cache line, even though we expect the header word to be accessed infrequently relative to the words in the allocated block.

16 Dave Dice and Alex Kogan. Hemlock : Compact and scalable mutual exclusion. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '21. Association for Computing Machinery, 2021. doi:10.1145/3409964.3461805.

17 David Dice, Mark Moir, and William N. Scherer III. Quickly reacquirable locks – us patent 7,814,488, 2002. URL: https://patents.google.com/patent/US7814488.

18 Doug Lea. The java.util.concurrent synchronizer framework. *Science of Computer Programming*, 58(3), 2005. Special Issue on Concurrency and synchonization in Java programs. URL: https://www.sciencedirect.com/science/article/pii/S0167642305000663, doi:https://doi.org/10.1016/j.scico.2005.03.007.

19 Waiman Long. qspinlock: Introducing a 4-byte queue spinlock implementation. https://lwn.net/Articles/561775, July 31, 2013, 2013. Accessed: 2018-09-19.

20 John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 1991. URL: http://doi.acm.org/10.1145/103727.103729.

21 Filip Pizlo. Locking in webkit, 2016. URL: https://webkit.org/blog/6161/locking-in-webkit/.

22 U. Verner, A. Mendelson, and A. Schuster. Extending amdahl's law for multicores with turbo boost. *IEEE Computer Architecture Letters*, 2017. URL: https://doi.org/10.1109/LCA.2015.2512982.

23 Wikipedia. Conway's law. URL: https://en.wikipedia.org/wiki/Conway%27s_law.

24 Wikipedia contributors. Ski rental problem — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Ski_rental_problem, 2017. [Online; accessed 8-August-2018].

A Appendix – Variations

HashChains with Fast-Path : To address the issue of uncontended latency – caused by the need to acquire and release the chain locks for both acquire and release operations on the monitor –we can modify **HashChains** by adding a fast-path that allows threads to insert and remove a lock record from the bucket chain using just an atomic `compare-and-swap` (CAS) operation in the case where the chain is otherwise empty, avoiding the need to acquire and release the bucket locks. We use a specialized encoding of the lock word for the bucket lock to indicate a singleton lock record is present. If additional threads (or objects) access the chain, the chain devolves to normal locking until the chain again becomes empty. The fast-path optimization is purely a ploy to improve uncontended latency [7].

CJM and Hashed Hybrid : A hybrid of CJM and the hashed forms is also viable. Briefly, in this form, we displace the header word as in CJM. But to reduce complexity, we protect the chain of waiting threads with internal locks instead of resorting to the lock-free techniques used in CJM. This allows the chain to be structured as a tail-anchored circular linked list (TACLL) which is convenient for our needs. The header word points to the tail element (the most recently arrived) and the tail points to head, which is the owner. We also maintain the definitive displaced header word value in the tail element.

The locks can be situated in either a shared global array hashed by the object's virtual address or hashCode, or we can site the locks in the chain elements. In the latter case, we

[7] We collected performance data on this variant, but to avoid clutter in the graphs opted to not report it below.

lock the chain by attempting to lock the tail element and then confirming that the element remains at the tail, "chasing" ownership as needed. (This approach has lock-free progress properties. If we find the tail changed then some other thread made forward progress). This variation requires safe memory reclamation for the queue elements, but avoids hashing and hash collisions (false contention). The same locking protocol is used to access the displaced header word, which we maintain in the tail element referred to by the header word. Inserting the initial element, for uncontended acquisition, is accomplished with an atomic compare-and-swap operation as there is not tail to lock. This also accelerates uncontended operations.

Access to the header word is accomplished as follows. When the object is not locked the header is not displaced, and can thus be accessed directly. Specifically, we can fetch the header word and examine the low order bits, which constitute a discriminated union or tag, to determine if the object is locked. If the object is locked and the accessor is the owner (which is easily determined) then the owner can simply find the displaced header word in the lock record it installed. If the object is locked and the accessor is not the owner, a situation we expect to be rare, then a more elaborate access protocol is required. In this situation the accessor locks the chain, as described above, to find the displaced header word in the tail element.

This approach is simple and effective, but suffers from the "double locking" performance concern. In addition, both monitor acquire and release operations access the header word, which is undesirable due to the increased coherence traffic on that location [8].

[8] Again, we implemented and collected data on this variant, but opted to omit the performance data in the evaluation section.