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 the Compact Java Monitors (CJM) design [8].

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

Page 1

All the locking algorithms below are FIFO/FCFS-fair 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. This approach is appealing as avoids it multiplexing (overloading) of the header word, simplifying the encoding thereof and reducing couplings between the synchronization subsystem and other components on the JVM. No information in the header word is displaced by synchronization.

The JVM maintains a shared hash table of synchronization buckets. Each bucket contains a 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. 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 lock, such as MCS locks. Our implementation used *hemlock*[10, 11]. The thread then determines if there are any conflicting lock records on the chain – lock records emplaced 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 without contention and is the owner of the lock, 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()`.

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 used an *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. In addition, we're exposed 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+F

HashChains+F builds on **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.

1.3 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 [7, 9] 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.

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.

1.4 CJM

Compact Java Monitors (CJM) [8] are based on the Compact NUMA-Aware Locks (CNA) algorithm, but ignoring the *NUMA-Aware* property and focusing on the *Compact* aspect. CNA is itself a variation on the gold-standard MCS (Mellor-Crummey Scott) [14] queue-based lock algorithm [1].

[1] If you're unfamiliar with CJM, please see [8]. That document also includes a description of the MCS algorithm in an appendix.

Underlying much of the following design is our approach from Compact NUMA-Aware Locks (CNA) which was published in EuroSys 2019 [6, 4] and is being integrated into the Linux kernel as a replacement for the existing low-level *qspinlock* construct [3, 13], which is itself based on MCS. CNA is itself a variation on classic 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. See Section 6 of [8] 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 [8] 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 few 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 deferred deflation and accretion of objectMonitors. Threads contribute one queue element to a monitor's queue when the 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.

2 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 [15] was left enabled. In all cases default free-range unbound threads were used (no pinning of threads to processors).

All the underlying 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).

We collected data with two synthetic micro-benchmarks : `BU` measures uncontended performance and `BC` measure 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.

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 [5].

The `BU` benchmark spawns T concurrent threads. Each thread allocates NL private and distinct locks (objects). Each thread loops as follows: Select NA locks randomly, with a uniform distribution, and with replacement, from the set of NL, constructing the *lockset*. NA and NL are command-line arguments and NA must be less than or equal to NL. As we select with replacement, duplicates are allowed in the lockset, admitting the possibility of recursive locking. Next, the thread acquires all the locks identified in the lockset. As the locks and lockset are private, there can be no contention and no risk of deadlock. The thread then immediately releases the lockset elements in reverse order, mimicing Java's usual last-acquired-first-released pattern. 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.

The `BC` 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 [2]. Each thread acquires all the elements of its lockset, and then executes a critical section of CSL steps of a thread-local C++ `std::mt1993` pseudo-random number generator. The thread then releases the locks in reverse order, and then executes a non-critical phase of $NCSL$ steps of that same random number generator. $NCSL$ and CSL are specified as

[2] 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.

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.

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.

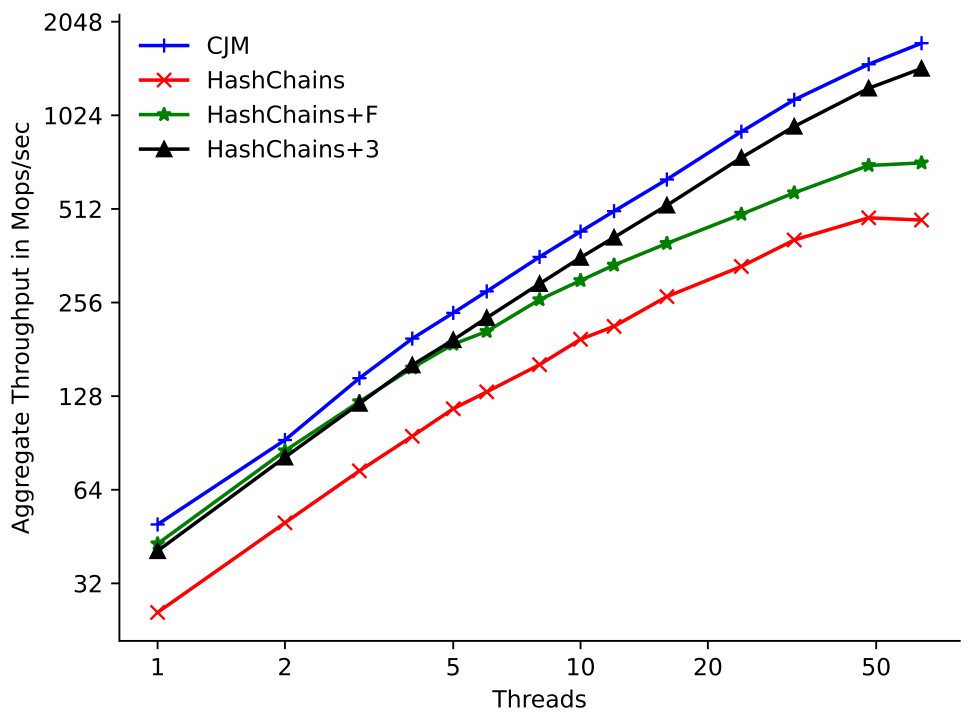

Figure 1 Uncontended locking : NL=64 and NA=1

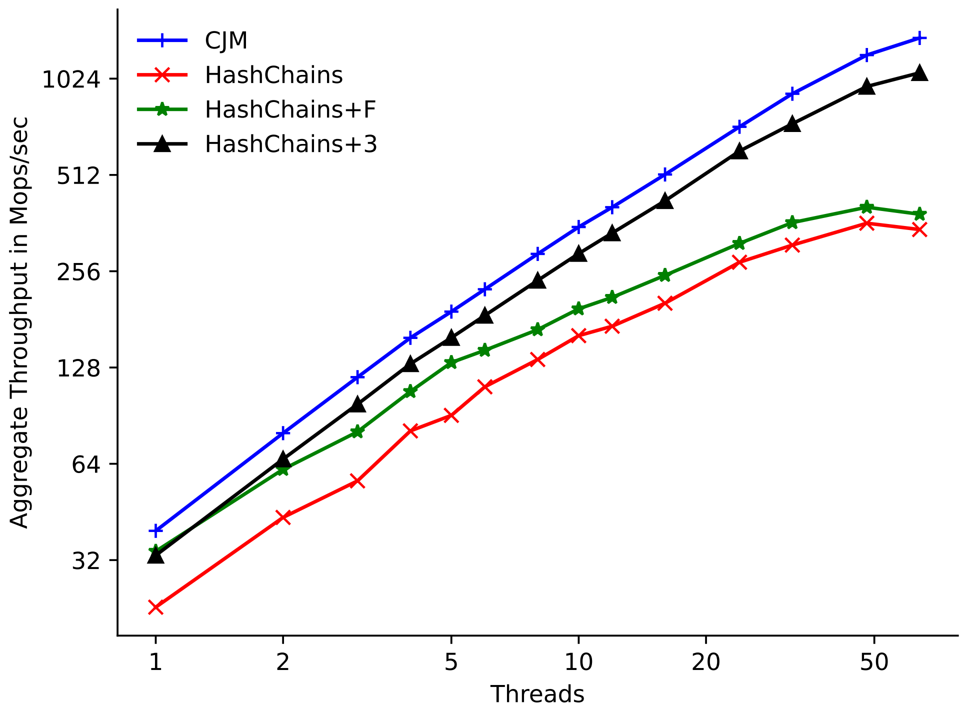

Figure 2 Uncontended locking : NL=64 and NA=10

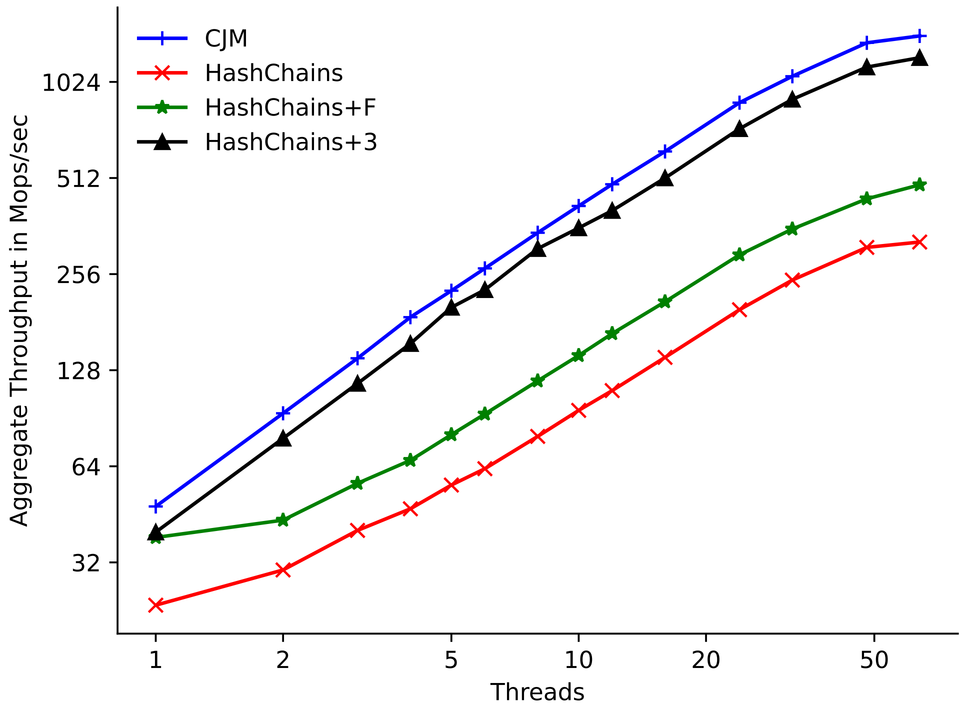

Figure 3 Uncontended locking : NL=4096 and NA=1

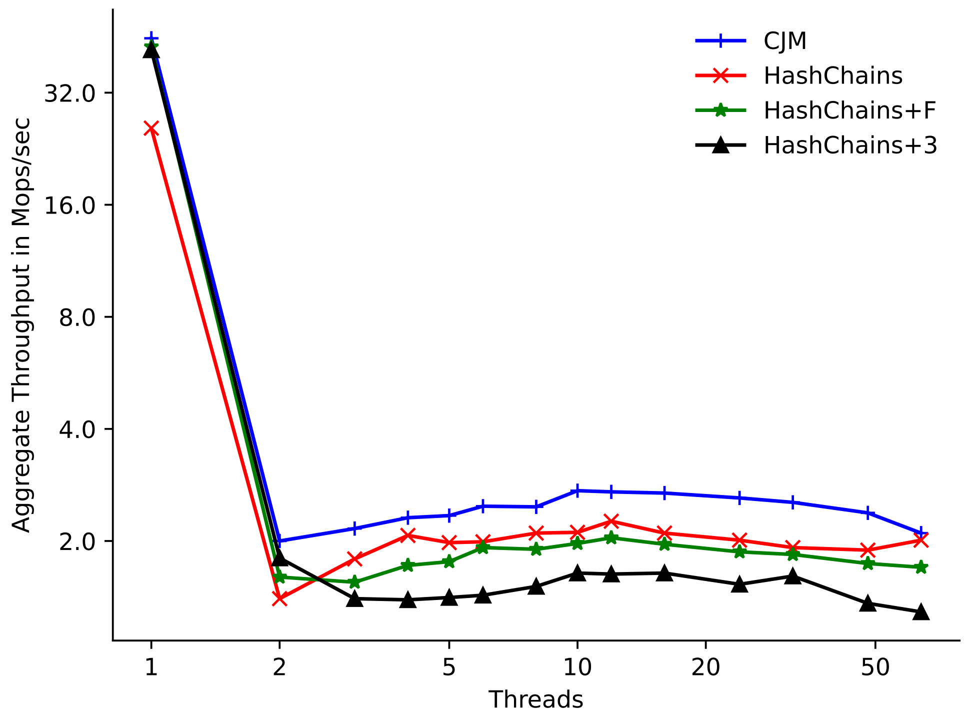

Figure 4 Contended locking : 1 lock with CSL=0 and NCSL=0

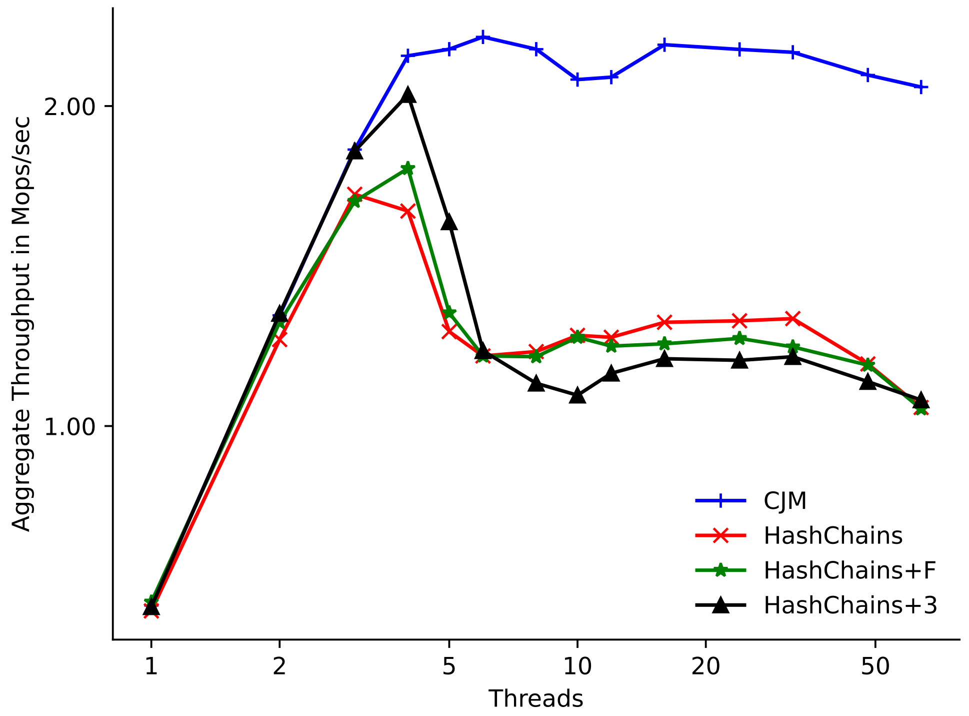

Figure 5 Contended locking : 1 lock with CSL=1 and NCSL=200

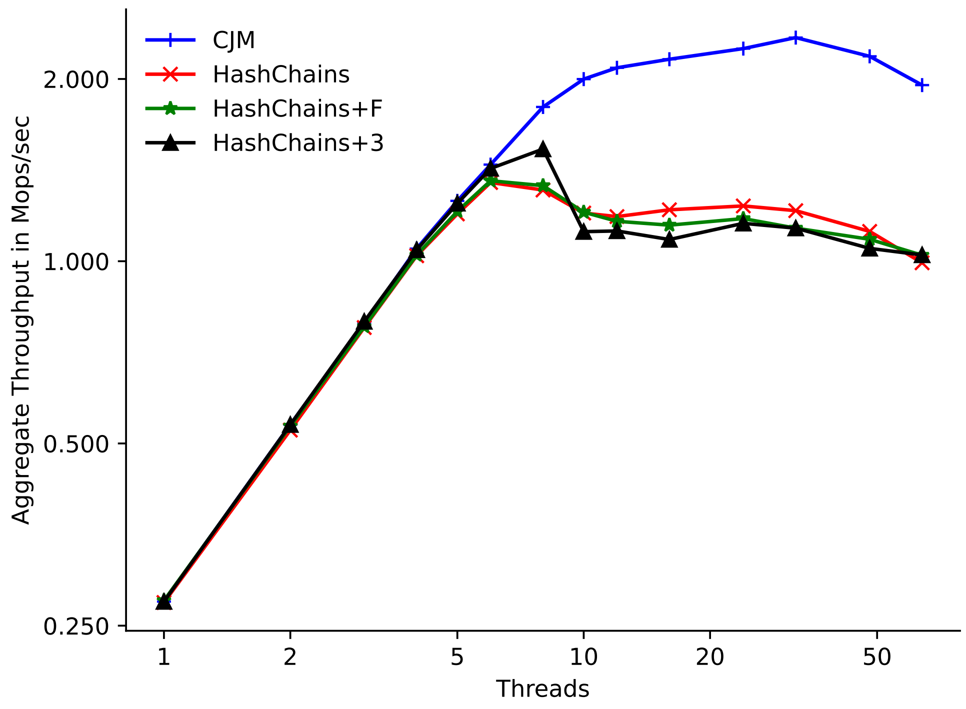

Figure 6 Contended locking : 1 lock with CSL=1 and NCSL=500

Figure 1, 2 and 2 reflect uncontended locking performance collected with BU. As we drive up the number of threads and locks, performance fades for `hashChains` and `hashChains+F`, relative to `CJM` and `hashChains+3`, as collisions increase in the shared hash table. In addition, in Figure 1, at one thread we see that `HashChains` provides relatively poor performance because of the latency required to acquire and release the bucket locks. The same behavior can be seen at one thread in Figure 4. As noted above, under BU there is no true monitor contention.

Figures 4, 5 and 6 report the aggregate throughput of contended locking operations with BC. Figure 4 shows extreme contention, with empty critical and non-critical sections, on a single "hot" lock. Interestingly, in Figure 4, we see that the efforts in `HashChains+3` and `HashChains+F` to improve latency, but also act against our interests and impinge on scalability. Figures 5 and 6 also have a single hot lock, but use a very short critical section with longer non-critical sections, reflecting more likely real-world scenarios. CJM provides reasonable scaling – in keeping with the existing implementation – while performance collapses quickly with the other locks, which exhibit retrograde scaling.

3 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.

It is worth comparing this idea the 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. Related ideas can be found in IBM's J9 *lock nursery* [1] `https://blog.openj9.org/2019/04/02/lock-nursery/`

3. It's worth noting that instead of CJM, we could continue to use the existing synchronization subsystem – with all its 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.
4. Both contended[2] and uncontended performance are critical quality-of-implementation metrics for the design of a synchronization subsystem. Previously, in the era of *bus locking*, before cache locking, techniques such as *biased locking*[12] were used to improve the latency of uncontended operations. Thankfully, changes in processor architecture have obviated the need for biased locking, and it's encumbent 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.
5. 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 bucket locks.
6. Interactions with projects *loom* and *graal* have not been considered.

7. 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 [3].
8. We assume a 64-bit JVM with a 64-bit header word.

References

1 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.

2 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.

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

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

5 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.

6 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.

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

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

9 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.

10 Dave Dice and Alex Kogan. Hemlock : Compact and scalable mutual exclusion. 2021. arXiv:2102.03863.

11 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.

12 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.

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

14 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.

15 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.

[3] A large fraction of current malloc allocators get this precisely wrong, placing the header word on the 1st word of a cache line, even though we expect the header word to be accessed infrequently relative to the words in the allocated block.