Benchmarking Concurrent Priority Queues: Performance of k-LSM and Related Data Structures

[Brief Announcement]

Jakob Gruber
TU Wien, Austria
gruber@par.tuwien.ac.at

Jesper Larsson Träff
TU Wien, Austria
traff@par.tuwien.ac.at

Martin Wimmer
Google
wimmerm@google.com

Keywords

Priority Queues, Concurrency, Relaxation, Benchmarking

ABSTRACT

A number of concurrent, relaxed priority queues have recently been proposed and implemented. Results are commonly reported for a *throughput benchmark* that uses a uniform distribution of keys drawn from a large integer range, and mostly for single systems. We have conducted more extensive benchmarking of three recent, relaxed priority queues on four different types of systems with different key ranges and distributions. While we can show superior throughput and scalability for our own k-LSM priority queue for the uniform key distribution, the picture changes drastically for other distributions, both with respect to achieved throughput and relative merit of the priority queues. The throughput benchmark alone is thus not sufficient to characterize the performance of concurrent priority queues. Our benchmark code and k-LSM priority queue are publicly available to foster future comparison.

1. CONCURRENT PRIORITY QUEUES

Due to the increasing number of processors in modern computer systems, there is significant interest in concurrent data structures with scalable performance that goes beyond a few dozen processor-cores. However, data structures (e.g., priority queues) with strict sequential semantics often present (inherent) bottlenecks (e.g., the `delete_min` operation) to scalability, which motivates weaker correctness conditions or data structures with relaxed semantics (e.g., one of the smallest k items for some k allowed to be deleted). Applications can often accomodate such relaxations, and in many such cases (discrete event simulation, shortest path algorithms, branch-and-bound), the priority queue is a key data structure.

Many lock-free designs have been based on Skiplists [5, 7, 8]. In contrast, the recently proposed, relaxed k-LSM priority queue [9] is based on a deterministic Log-Structured Merge-Tree (LSM), and combines an efficient thread-local variant for scalability with a shared, relaxed variant for semantic guarantees. The k-LSM priority queue is lock-free, linearizable, and provides configurable guarantees of `delete_min` returning one of the kP smallest items, where k is a configuration parameter and P the number of cores (threads). The SprayList [1] uses a lock-free Skiplist, and allows `delete_min` to remove a random element from the $O(P \log^3 P)$ items at the head of the list. MultiQueues [6] randomly spread both insertions and deletions over cP local priority queues, each protected by a lock, with tuning parameter c, but gives no obvious guarantees on the order of deleted elements.

2. A CONFIGURABLE BENCHMARK

Priority queue performance is often measured by counting the number of `insert` and `delete_min` operations that can be performed in a given amount of time, i.e., the *throughput*, which would ideally increase linearly with the number of threads. Like recent studies [1, 5, 7, 8, 9], we also measure throughput, but additionally we experiment with different *workloads*: (a) *uniform*, where each thread performs 50% insertions and 50% deletions, randomly chosen, (b) *split*, where half the threads perform only insertions, and the other half only deletions; and (integer) *key distributions*: (a) *uniform*, where keys are drawn uniformly at random from the range of 32-bit, 16-bit, or 8-bit integers, and (b) *ascending (descending)*, where keys are drawn from a 10-bit integer range which is shifted upwards (downwards) at each operation (plus/minus one).

Queues are prefilled with 10^6 elements with keys taken from the chosen distribution. This benchmark provides more scope for investigating locality (split workload), distribution and range sensitivity. The benchmark could be parameterized further [2] to provide for wider synthetic workloads, e.g., sorting as in [4]. To some extent, our ascending/descending distributions correspond to the *hold model* advocated in [3].

For relaxed priority queues, it is as important to characterize the deviation from strict priority queue behavior, also for verifying whether claimed relaxation bounds hold. We have implemented a rank error benchmark as in [6], where the rank of an item is its position within the priority queue as it is deleted.

3. EXPERIMENTAL RESULTS

We have benchmarked variants of the k-LSM priority queue [9] with different relaxation settings (`klsm128`, `klsm256`, `klsm4096`) against a Skiplist based queue [5] (`linden`), the

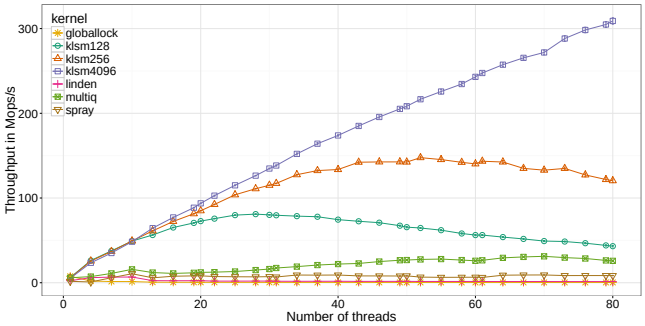


Figure 1: `mars`: Uniform workload, uniform keys.

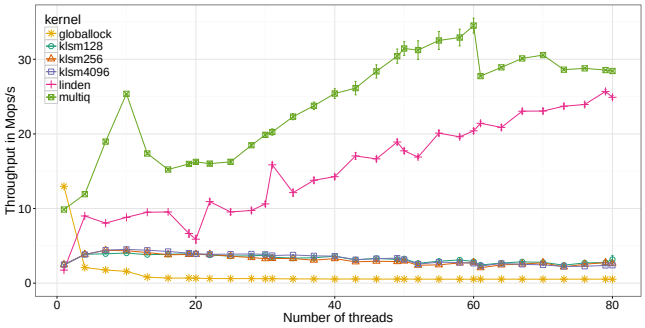


Figure 2: `mars`: Split workload, ascending keys.

MultiQueue [6] (`multiq`) and the SprayList [1] (`spray`). As a baseline we have used a sequential heap protected by a lock (`globallock`). The benchmarks ran on four different machines, but we give only results from an 80-core Intel Xeon E7-8850 2 GHz system (`mars`) here (without hyperthreading); see the appendix for full details and results on all machines. Each benchmark is executed 30 times, and we report on the mean values and confidence intervals. Our benchmark code can be found at https://github.com/klsmpq/klsm.

Figure 1 compares the seven priority queue variants under *uniform workload*, *uniform keys*. The k-LSM variant with $k = 4096$ exhibits superior scalability and throughput of more than 300 million operations per second (MOps/s), and vastly outperforms the other priority queues. Changing to a *split workload* and *ascending keys*, this picture changes dramatically, as shown in Figure 2 where the throughput drops by a factor of 10. Here `multiq` performs best, also in terms of scalability, surprisingly closely followed by `linden`. Restricting the key range likewise dramatically reduces the throughput, but the k-LSM performs better in this case, (Figure 3). In the latter two benchmark configurations, the SprayList code was not stable and it was not possible to gather results. Similar behavior and sensitivity can be observed for the other three machines. Hyperthreading only in rare cases leads to a throughput increase. Overall, `multiq` delivers the most consistent performance.

The rank error results in Table 1 for the uniform workload, uniform key situation show that `delete_min` for all queues return keys that are not far from the minimum, much better than the the worst-case analyses predict.

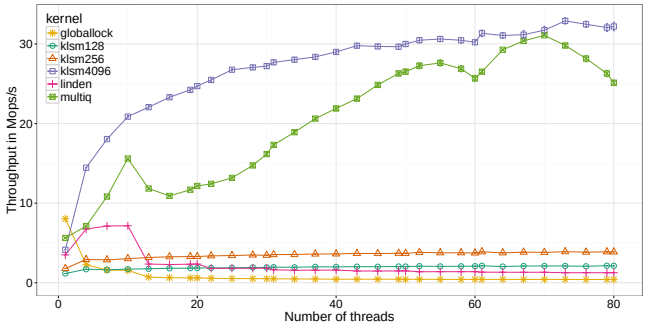


Figure 3: `mars`: Uniform workload, 8-bit restricted keys.

	20 threads		40 threads		80 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	32	29	57	48	298	288
klsm256	42	42	68	57	635	464
klsm4096	422	729	1124	1287	13469	13980
multiq	1163	3607	2296	7881	3753	12856

Table 1: `mars`: Rank error, uniform workload and keys.

References

[1] D. Alistarh et al. "The SprayList: a scalable relaxed priority queue". In: *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming*. 2015, pp. 11–20.

[2] V. Gramoli. "More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms". In: *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming*. 2015, pp. 1–10.

[3] D. W. Jones. "An Empirical Comparison of Priority-Queue and Event-Set Implementations". In: *Communications of the ACM* 29.4 (1986), pp. 300–311.

[4] D. H. Larkin, S. Sen, and R. E. Tarjan. "A Back-to-Basics Empirical Study of Priority Queues". In: *Proceedings of the 16th Workshop on Algorithm Engineering and Experiments*. 2014, pp. 61–72.

[5] J. Lindén and B. Jonsson. "A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention". In: *Principles of Distributed Systems*. Vol. 8304. Lecture Notes in Computer Science. 2013, pp. 206–220.

[6] H. Rihani, P. Sanders, and R. Dementiev. "Brief Announcement: MultiQueues: Simple Relaxed Concurrent Priority Queues". In: *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*. 2015, pp. 80–82.

[7] N. Shavit and I. Lotan. "Skiplist-Based Concurrent Priority Queues". In: *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS)*. 2000, pp. 263–268.

[8] H. Sundell and P. Tsigas. "Fast and lock-free concurrent priority queues for multi-thread systems". In: *Journal of Parallel and Distributed Computing* 65.5 (2005), pp. 609–627.

[9] M. Wimmer et al. "The Lock-free k-LSM Relaxed Priority Queue". In: *20th ACM Symposium on Principles and Practice of Parallel Programming*. 2015.

APPENDIX

This appendix contains the full set of experimental results and details on the experimental setup. We briefly recapitulate the main three priority queue implementations, and describe the benchmarks in more detail. The remainder are our current results from the four available machines.

A. CONCURRENT, RELAXED PRIORITY QUEUES

The priority queues considered here support two operations on key-value pairs, namely:

- `insert`, which inserts a key-value pair into the priority queue, and
- `delete_min`, which removes a key-value pair with a smallest key and copies the corresponding value into a given location.

Concurrent priority queues so far neither support operations on specific key-value pairs, like for instance `decrease_key` as needed for asymptotically efficient single-source shortest path algorithms, nor operations on the queues as a whole like `meld`.

A *linearizable*, *strict* priority queue imposes a real time order on priority queue operations, and in such an order each `delete_min` must return a least key-value pair. Relaxed consistency conditions like *quasi-linearizability* allow some non-determinism by accepting runs that are some bounded distance away from a strict, linear order [1] as correct. Recently, an even weaker conditions called *local linearizability* was proposed [8], but this may be too weak to be useful for priority queues. On the other hand, relaxed priority queue semantics relax the sequential priority queue semantics to allow that one of the k smallest keys is returned, for some value k, at the `delete_min` operation [25, 26]. Such relaxed priority queues are considered here. Presumably, the better and the more precisely k can be controlled, the better for the applications.

B. THE K-LSM PRIORITY QUEUE

The k-LSM [7, 26] is a lock-free, linearizable, relaxed priority queue consisting of a global component called the Shared LSM (SLSM), and a thread-local component called the Distributed LSM (DLSM). As their names imply, both the SLSM and DLSM are based on the LSM (Log-Structured Merge-Tree) [18] data structure. The LSM was first introduced to the database community in 1996 and later reinvented independently by Wimmer driven by the requirements of relaxed, concurrent priority queues [25]. Both the SLSM and the DLSM may be used as standalone priority queues, but have complementary advantages and disadvantages which can be balanced against each other by their composition.

The LSM consists of a logarithmic number of sorted arrays (called blocks) storing key-value containers (items). Blocks have capacities $C = 2^i$ and capacities within the LSM are distinct. A block with capacity C must contain more than $\frac{C}{2}$ and at most C items. Insertions initially add a new singleton block to the LSM, and then merge blocks with identical capacities until all block capacities within the LSM are once again distinct. Deletions simply return the smallest of all blocks' minimal item. It is easy to see that both insertions and deletions can be supported in $O(\log n)$ operations where n is the number of items in the LSM.

The DLSM is a distributed data structure containing a single thread-local LSM per thread. Operations on the DLSM are essentially embarassingly parallel, since inter-thread communication occurs only when a deletion finds the local LSM empty, and then attempts to copy another thread's items. Items returned by `delete_min` are guaranteed to be minimal on the current thread.

The SLSM consists of a single global, centralized LSM, together with a corresponding range of items called the pivot range. The SLSM's pivot range depicts a subset of the $k+1$ smallest items (where k is the relaxation parameter). Deletions randomly choose an item from this range, and thus are allowed to skip at most k items.

Finally, the k-LSM itself is a very simple data structure: it contains a DLSM, limited to a maximum capacity of k per thread; and a SLSM with a pivot range containing at most $k+1$ of its smallest items. Items are initially inserted into the local DLSM. When its capacity overflows, its largest block is batch-inserted into the SLSM. Deletions simply peek at both the DLSM and SLSM, and return the smaller item. Since deletions from the DLSM skip at most $k(P-1)$ items (where P is the number of threads) and deletions from the SLSM skip at most k items, k-LSM deletions skip a maximum of kP items in total.

We implemented the k-LSM using the C++11 memory model. A memory model determines the order in which changes to memory locations by one thread become visible to other threads; for instance, usage of the the `std::atomic` type together with its `load()` and `store()` operations ensures portable multithreaded behavior across different architectures. It is possible to vary the strictness of provided guarantees between sequential consistency (on the strict end) and relaxed behavior (guaranteeing only atomicity).

In our implementation, we extensively use the previously mentioned `std::atomic` type together with its `load`, `store`, `fetch_add`, and `compare_exchange_strong` operations. When possible, we explicitly use relaxed memory ordering as it is the potentially most efficient (and weakest) of all memory ordering types, requiring only atomicity.

Our implementation, consisting of a standalone k-LSM as well as our parameterizable benchmark, is publicly available at https://github.com/klsmpq/klsm, and desribed in detail in [7].

C. ALGORITHMS

Our benchmarks compare the following algorithms:

Globallock (`globallock`). A simple, standardized sequential priority queue implementation protected by a global lock is used to establish a baseline for acceptable performance. We use the simple priority queue implementation (`std::priority_queue`) provided by the C++ Standard Library (STL) [11].

Linden (`linden`). The Lindén and Jonsson priority queue [15] is currently one of the most efficient Skiplist-based designs, improving upon the performance of previous similar data structures [9, 22, 24] by up to a factor of two. It is lock-free and linearizable, but has strict semantics, i.e., deletions must return the minimal item in some real-time order.

SprayList (`spray`). This relaxed priority queue is based on the lock-free Fraser Skiplist [5]. Deletions use a random-walk method in order to return one of the $O(P \log^3 P)$ small-

est items, where P is the number of threads [2].

MultiQueue (`multiq`). The MultiQueue is a recent design by Rihani, Sanders, and Dementiev [20] and consists of cP arbitrary priority queues, where c is a tuning parameter (set to 4 in our benchmarks) and P is the number of threads; our benchmark again uses the simple sequential priority queue (`std::priority_queue`) provided by the STL [11], each protected by a lock. Items are inserted into a random priority queue, while deletions return the minimal item of two randomly selected queues. So far, no complete analysis of its semantic bounds exists.

k-LSM (`klsm128`, `klsm256`, `klsm4096`). We evaluate several instantiations of the k-LSM with varying degrees of relaxation, ranging from medium ($k \in \{128, 256\}$) to high relaxation ($k = 4096$). Results for low relaxation ($k = 16$) are not shown since its behavior closely mimics the Lindén and Jonsson priority queue.

Unfortunately, we were not able to measure every algorithm on each machine. The `linden` and `spray` priority queues require libraries not present on `ceres` and `pluto`. The SprayList implementation also turned out to be unstable in our experiments, crashing under most circumstances outside the uniform workload, uniform key distribution benchmark.

D. OTHER PRIORITY QUEUES

The Hunt et al. priority queue [10] is an early concurrent design. It is based on a Heap structure and attempts to minimize lock contention between threads by a) adding per-node locks, b) spreading subsequent insertions through a bit-reversal technique, and c) letting insertions traverse bottom-up in order to minimize conflicts with top-down deletions. It has been shown to perform well compared to other efforts of the time; however, it is easily outperformed by more modern designs.

Shavit and Lotan were the first to propose the use of Skiplists for priority queues [15]. Their initial locking implementation [22] builds on Pugh's concurrent Skiplist [19], which uses one lock per node per level. Herlihy and Shavit [9] later described and implemented a lock-free, quiescently consistent version of this idea in Java.

Sundell and Tsigas invented the first lock-free concurrent priority queue in 2003 [24]. Benchmarks show their queue performing noticeably better than both locking queues by Shavit and Lotan and Hunt et al., and slightly better than a priority queue consisting of a Skiplist protected by a single global lock.

Mounds [16, 17] is a recent concurrent priority queue design based on a tree of sorted lists. Liu and Spear provide two variants of their data structure; one of them is lock-based, while the other is lock-free and relies on the Double-Compare-And-Swap (DCAS) instruction, which is not available natively on most current processors.

One of the latest strict priority queues of interest, called the Chunk-Based Priority Queue (CBPQ), was presented recently in the dissertation of Braginsky [3]. It is primarily based on two main ideas: the chunk linked list [4] replaces Skiplists and heaps as the backing data structure, and use of the more efficient Fetch-And-Add (FAA) instruction is preferred over the Compare-And-Swap (CAS) instruction. Benchmarks compare the CBPQ against the Lindén and Jonsson queue and lock-free as well as lock-based versions of the Mound priority queue [16] for different workloads. The CBPQ clearly outperforms the other queues in mixed workloads (50% insertions, 50% insertions) and deletion workloads, and exhibits similar behavior as the Lindén and Jonsson queue in insertion workloads, where Mounds are dominant.

E. MACHINES

The benchmarks were executed on four machines:

- `mars`, an 80-core (8x10 cores) Intel Xeon E7-8850 at 2 GHz with 1 TB of RAM main memory, and 32 KB L1, 256 KB L2, 24 MB L3 cache, respectively. `mars` has 2-way hardware hyperthreading.
- `saturn`, a 48-core machine with 4 AMD Opteron 6168 processors with 12 cores each, clocked at 1.9 GHz, and 125 GB RAM main memory, and 64 KB of L1, 512 KB of L2, and 5 MB of L3 cache, respectively. The AMD processor does not support hyperthreading.
- `ceres`, a 64-core SPARCv9-based machine with 4 processors of 16 cores each. Cores are clocked at 3.6 GHz and have 8-way hardware hyperthreading. Main memory is 1 TB RAM, and cache is 16 KB L1, 128 KB L2, and 8 MB L3 , respectively.
- `pluto`, a 61-core Intel Xeon Phi processor clocked at 1.2 GHz with 4-way hardware hyperthreading. Main memory is 15 GB RAM, and cache 32 KB L1, 512 KB L2, respectively.

All applications are compiled using `gcc`, when possible: version `5.2.1` on `mars` and `saturn`, and version `4.8.2` on `ceres`. We use optimization level of `-O3` and enable link-time optimizations using `-flto`. Cross-compilation for the Intel Xeon Phi on `pluto` is done using Intel's `icc 14.0.2`. No further optimizations were performed, in particular vectorization was entirely delegated to the compiler, which probably leaves the Xeon Phi `pluto` at a disadvantage. On the other hand, all implementations are treated similarly.

F. BENCHMARKS

Our performance benchmarks are (currently) based on *throughput*, i.e., how many operations (insertions and deletions combined) complete within a certain timeframe. We prefill priority queues with 10^6 elements prior the benchmark, and then measure throughput for 10 seconds, finally reporting on the number of operations performed per second. This metric and a roughly similar setup is used in much recent work [2, 15, 22, 24, 26]. Alternatively, a number of queue operations could be prescribed, and the time (latency) for this number and mix of operations measured.

The behavior of our throughput benchmark is controlled by the two parameters *workload* and *key distribution*. The workload may be

- *uniform*, meaning that each thread executes a roughly equal amount of insertions and deletions,
- *split*, meaning that half the threads insert, while the other half delete, or
- *alternating*, in which each thread strictly alternates between insertions and deletions.

The key distribution controls how keys are generated for inserted key-value pairs, and my may be either

- *uniform* with keys chosen uniformly at random from some range of integers (we have used 32-, 16-, and 8-bit ranges), or
- *ascending* or *descending*, meaning that a uniformly chosen key from a 10-bit integer range ascends or descends over time by adding or subtracting the chosen key to the operation number.

We would like to supply a parameterized benchmark similar to the Synchrobench framework of Gramoli [6] with the following orthogonal parameters:

- *Key type*: integer, floating point, possibly complex type from some ordered set (here, we have experimented with integers only).
- *Key base range*, which is the range from which the random component of the next key is chosen (here, we have used 32-, 16-, 10- and 8-bit ranges).
- *Key distribution*, the distribution of keys within their base range (here, we have used only uniform distributions, but others, e.g., as in [12], are also possible).
- *Key dependency* switch (none, ascending, descending), which determines whether the next key for a thread is dependent on the key of the last deleted element by the thread. A dependent key is formed by adding or subtracting the randomly generated base key to the key of the last deleted item (we have experimented with dependent keys where the next key is formed by adding to or subtracting from the operation number).
- *Operation distribution*: insertions and deletions are chosen randomly with a prescribed probability of an operation being an insert (we have experimented with 50% insertions so that the queues remain in a steady state).
- Alternatively, an *operation batch size* can be set to alternate between batches of insertions and deletions (we have experimented with strictly alternating insertions and deletions).
- *Workload* determines the fraction of threads that perform insertions and the fraction of threads that perform deletions (we have experimented with uniform and split workloads, where in the latter half the threads perform the insertions and the other half the deletions).
- *Prefill* determines the number of items put in the queue before the time measurement starts; prefilling is done according to the workload and key distribution.
- *Throughput/latency* switch, where for throughput a duration (time limit) is specified and for latency the total number of operations.
- *Repetition count* and other statistic requirements.

For instance, giving an operation batch size of one with an insert following delete with dependent keys under a specific distribution would correspond to the *hold model* proposed in [12] and used in early studies of concurrent priority queues [13, 21]. Choosing large batches would correspond to the sorting benchmark used in [14].

In addition, as in [20] we also evaluated the semantic quality produced by `multiq` and the k-LSM with several different relaxations by measuring the rank of items (i.e., their position within the priority queue) returned by `delete_min`. The quality benchmark initially records all inserted and deleted items together with their timestamp in a log; this log is then used to reconstruct a global, linear sequence of all operations. A specialized sequential priority queue is then used to replay this sequence and efficiently determine the rank of all deleted items. Our quality benchmark is pessimistic, i.e., it may return artificially inflated ranks when items with duplicate keys are encountered.

G. FURTHER EXPERIMENTAL RESULTS

Each benchmark is executed 30 times, and we report on the mean values and confidence intervals.

Figure 4 shows throughput results for `mars`. Up to 80 threads, each thread is pinned to a separate physical core, while hyperthreading is used at higher thread counts.

Under uniform workload and uniform key distribution, the k-LSM has very high throughput, reaching over 300 MOps/s at 80 cores with a relaxation factor of $k = 4096$. For the remaining data structures, `multiq` shows the best performance, reaching around 40 MOps/s at 160 threads (a decrease over the k-LSM by around a factor 7.5). The SprayList reaches a maximum of around 11 MOps/s at 140 threads, while the Lindén and Jonsson queue and `globallock` peak at around 6.7 MOps/s (10 threads) and 7.5 MOps/s (1 thread), respectively.

Varying the key distribution has a strong influence on the behavior on the k-LSM. Ascending keys (Figure 4b) result in a significant performance drop for all k-LSM variants, all of which behave similarly: throughput oscillates between around 5 and 15 MOps/s until 80 threads, and then slowly increases to a local maximum at around 140 threads. On the other hand, descending keys (Figure 4c) cause a performance increase for the `klsm4096`, which reaches a new maximum of around 400 MOps/s. Behavior of `multiq`, `linden`, and `globallock` remain more or less stable in both cases.

Under split workloads (Figures 4d through 4f), the k-LSM's throughput is very low and never exceeds the throughput of our sequential baseline `globallock` at a single thread. Interestingly, the Lindén and Jonsson priority queue has drastically improved scalability when using a combination of split workload and ascending key distribution. We assume that this is due to improved cache-locality: inserting threads access only the tail end of the list, with deleting threads accessing only the list head. `linden` was unstable at higher thread counts under split workload and descending keys, and we omit these results.

A key domain restricted to 8-bit integers (Figure 4g) results in many duplicate key values within the priority queue. This also causes decreased throughput of the k-LSM: medium relaxations do not appear to scale at all, while the `klsm4096` does seem to scale well — but only to a maximum throughput of just over 30 MOps/s. The larger 16-bit domain of Figure 4h produces very similar results to the uniform key

benchmark with a 32-bit range.

Hyperthreading is beneficial in only a few cases. For instance, `multiq` makes further modest gains beyond 80 threads with uniform workloads (e.g., Figures 4b and 4g). However, in general, most algorithms do not appear to benefit from hyperthreading.

Table 2 contains our quality results for `mars`, showing both the rank mean and its standard deviation for 20, 40, and 80 threads. In general, the k-LSM produces an average quality significantly better than its theoretic upper bound of a rank of $kP + 1$. For example, the `klsm128` has an average rank of 32 under the uniform workload, uniform key benchmark at 20 threads (Table 2a), compared to the maximally allowed rank error of 2561. Relaxation of `multiq` appears to be somewhat comparable to `klsm4096`, and it seems to grow linearly with the thread count. The uniform 8-bit restricted key benchmark (Table 2g) has artificially inflated ranks due to the way our quality benchmark handles key duplicates.

Figure 5 shows results for our AMD Opteron machine called `saturn`. Here, MultiQueue has fairly disappointing throughput, barely achieving the sequential performance of `globallock`, and only substantially exceeding it under split workload and ascending key distribution (Figure 5e). Surprisingly, with keys restricted to the 8-bit range (Figure 5g), the `linden` queue has a higher throughput than all other data structures. Quality trends in Table 3 are consistent with those on `mars`.

Figure 6 and Table 4 display throughput and quality results for `ceres`. On this machine, we display results for up to 4-way hyperthreading. As previously on `mars`, throughput of tested algorithms does not benefit from hyperthreading in general. Only `multiq` appears to consistently gain further (small) increases at thread counts over 64. Split workload combined with uniform key distribution (Figure 6d) causes a local maximum of around 30 MOps/s for the `klsm4096`.

Figure 7 shows throughput results for our Xeon Phi machine `pluto` (there is no corresponding quality table, since we did not run our quality benchmark on this machine). On `pluto`, the k-LSM does not match the trend for very high throughput as previously exhibited for the uniform workload and uniform key benchmark (Figure 7a). While scalability is decent up to the physical core count of 61, its absolute performance is exceeded by the `multiq` at higher thread counts. Only in descending key distribution (Figure 7c) does the k-LSM reach previous heights. Note that this is also the only benchmark in which throughput on `pluto` exceeds roughly 15 MOps/s. Unfortunately, the k-LSM was unstable at higher thread counts under split workloads and we omit all data where it is not completely reliable. Again, hyperthreading results in modest gains for `multiq`, and in stagnant performance for all other data structures in the best case.

Finally, Figures 8 and 9 show results for alternating workloads on all of our machines. Although the alternating workload appears to be similar to uniform workloads (both perform 50% insertions and 50% deletions, and are distinguished only by the fact that operations are strictly alternating in the alternating workload), there are significant differences in the resulting throughput. Uniform keys on `mars` (Figure 8a) show increases for the k-LSM in both throughput (to almost 400 MOps/s) and scalability, with all k-LSM variants ($k \in \{128, 256, 4096\}$) scaling almost equally well until 80 threads. Likewise, descending keys (Figure 8c) sees all k-LSM variants reaching a new throughput peak of around 600 MOps/s. Behavior on `saturn` is similar, in which uniform and descending keys show improved throughput and scalability for the k-LSM, while results for ascending keys remain unchanged from the uniform workload benchmark. On `ceres`, only scalability seems to improve while throughput is again roughly unchanged compared to uniform workload. Finally, on `pluto`, the k-LSM surprisingly does not perform well in any case, not even under descending keys (which led to good results when using uniform workload). However, `multiq` throughput increases by almost a factor of 8, reaching over 80 MOps/s in all cases.

In general, the k-LSM priority queue seems superior to the other priority queues in specific scenarios: in uniform workload combined with uniformly chosen 32- or 16-bit keys, and with descending key distribution, throughput is almost 10 times that of other priority queues. However, in most other benchmarks its performance is disappointing. This appears to be due to the differing loads placed on its component data structures: whenever the extremely scalable DLSM is highly utilized, throughput increases; and when the load shifts towards the SLSM, throughput drops. The fact that the k-LSM is composed of two priority queue designs seems to cause it to be highly sensitive towards changes in its environment.

The MultiQueue does not seem to have the same potential for raw performance as the k-LSM at its peak. However, in the majority of cases it still outperforms all other tested priority queues by a good margin. And most significantly, its behavior is extremely stable across all of our benchmark types. Quality results show that relaxation of the MultiQueue is fairly high, but it appears to grow linearly with the thread count.

The `linden` queue generally only scales as long as participating processors are located on the same physical socket; however, a split workload combined with ascending key generation is the exception to this rule, in which the `linden` queue is often able to scale well until the maximal thread count.

References

[1] Y. Afek, G. Korland, and E. Yanovsky. "Quasi-linearizability: Relaxed consistency for improved concurrency". In: *Principles of Distributed Systems*. Vol. 6490. Lecture Notes in Computer Science. Springer, 2010, pp. 395–410.

[2] D. Alistarh et al. "The SprayList: a scalable relaxed priority queue". In: *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming*. 2015, pp. 11–20.

[3] A. Braginsky. "Multi-Threaded Coordination Methods for Constructing Non-blocking Data Structures". PhD thesis. 2015.

[4] A. Braginsky and E. Petrank. "Locality-conscious lock-free linked lists". In: *Distributed Computing and Networking*. Vol. 6522. Lecture Notes in Computer Science. Springer, 2011, pp. 107–118.

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

[6] V. Gramoli. "More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms". In: *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming*. 2015, pp. 1–10.

[7] J. Gruber. "KLSM: A Relaxed Lock-Free Priority Queue". MA thesis. Vienna University of Technology (TU Wien), Jan. 2016.

[8] A. Haas et al. "Local Linearizability". In: *CoRR* abs/1502.07118 (2015). URL: http://arxiv.org/abs/1502.07118.

[9] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[10] G. C. Hunt et al. "An efficient algorithm for concurrent priority queue heaps". In: *Information Processing Letters* 60.3 (1996), pp. 151–157.

[11] ISO/IEC. *ISO International Standard ISO/IEC 14882:2014(E) – Programming Language C++*. Geneva, Switzerland, 2014.

[12] D. W. Jones. "An Empirical Comparison of Priority-Queue and Event-Set Implementations". In: *Communications of the ACM* 29.4 (1986), pp. 300–311.

[13] D. W. Jones. "Concurrent Operations on Priority Queues". In: *Communications of the ACM* 32.1 (1989), pp. 132–137.

[14] D. H. Larkin, S. Sen, and R. E. Tarjan. "A Back-to-Basics Empirical Study of Priority Queues". In: *Proceedings of the 16th Workshop on Algorithm Engineering and Experiments*. 2014, pp. 61–72.

[15] J. Lindén and B. Jonsson. "A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention". In: *Principles of Distributed Systems*. Vol. 8304. Lecture Notes in Computer Science. 2013, pp. 206–220.

[16] Y. Liu and M. F. Spear. "A lock-free, array-based priority queue". In: *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming*. 2012, pp. 323–324.

[17] Y. Liu and M. F. Spear. "Mounds: Array-Based Concurrent Priority Queues". In: *41st International Conference on Parallel Processing*. 2012, pp. 1–10.

[18] P. O'Neil et al. "The Log-Structured Merge-Tree (LSM-tree)". In: *Acta Informatica* 33.4 (1996), pp. 351–385.

[19] W. Pugh. *Concurrent maintenance of skip lists*. Tech. rep. 1998.

[20] H. Rihani, P. Sanders, and R. Dementiev. "Brief Announcement: MultiQueues: Simple Relaxed Concurrent Priority Queues". In: *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*. 2015, pp. 80–82.

[21] R. Rönngren and R. Ayani. "A Comparative Study of Parallel and Sequential Priority Queue Algorithms". In: *ACM Transactions on Modeling and Simulation* 7.2 (1997).

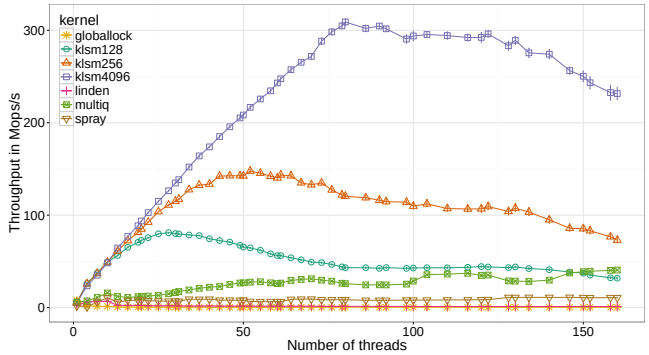
[22] N. Shavit and I. Lotan. "Skiplist-Based Concurrent Priority Queues". In: *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS)*. 2000, pp. 263–268.

[23] H. Sundell and P. Tsigas. "Fast and lock-free concurrent priority queues for multi-thread systems". In: *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*. IEEE. 2003, 11–pp.

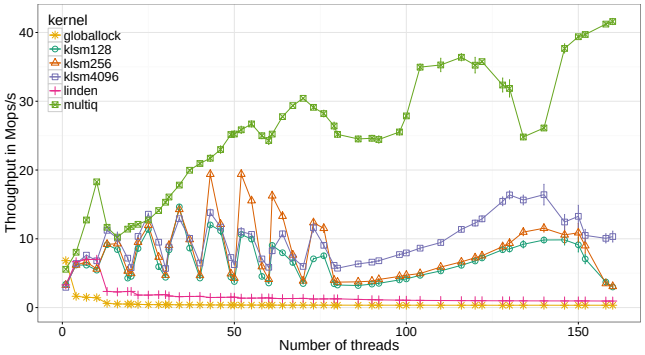
[24] H. Sundell and P. Tsigas. "Fast and lock-free concurrent priority queues for multi-thread systems". In: *Journal of Parallel and Distributed Computing* 65.5 (2005), pp. 609–627.

[25] M. Wimmer. "Variations on Task Scheduling for Shared Memory Systems". PhD thesis. Vienna University of Technology (TU Wien), June 2014.

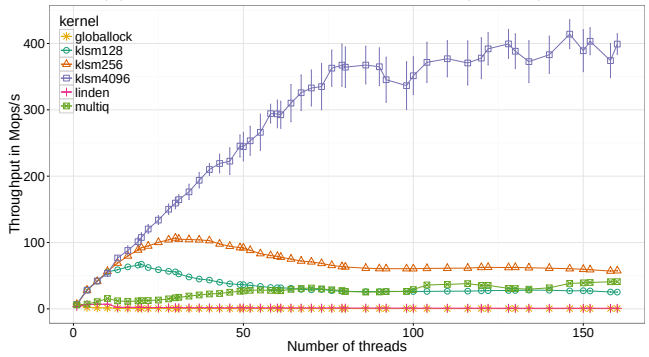
[26] M. Wimmer et al. "The Lock-free k-LSM Relaxed Priority Queue". In: *20th ACM Symposium on Principles and Practice of Parallel Programming*. 2015.



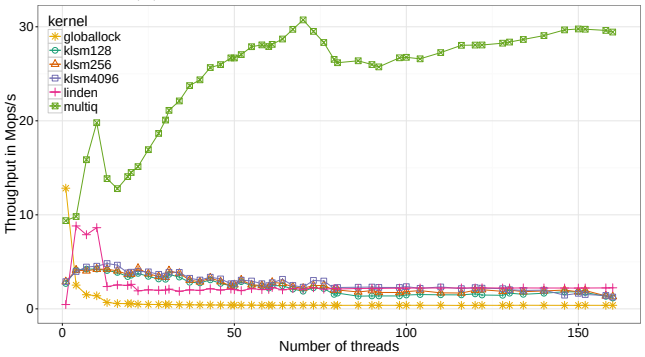
(a) Uniform workload, uniform keys (32 bits).



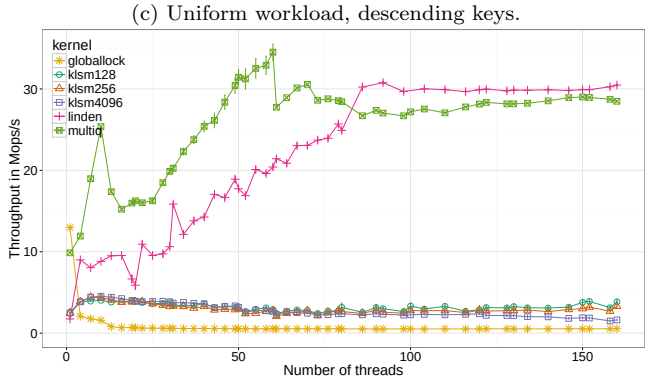
(b) Uniform workload, ascending keys.



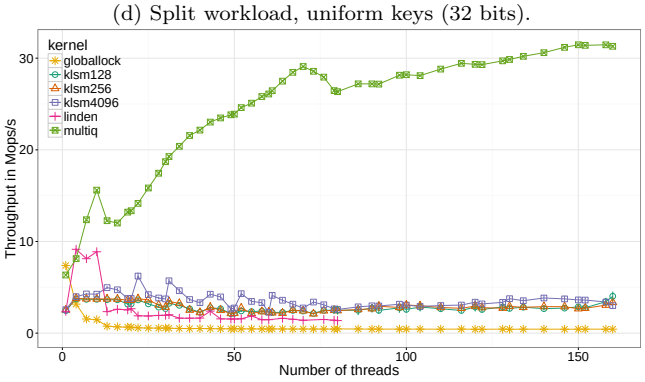
(c) Uniform workload, descending keys.



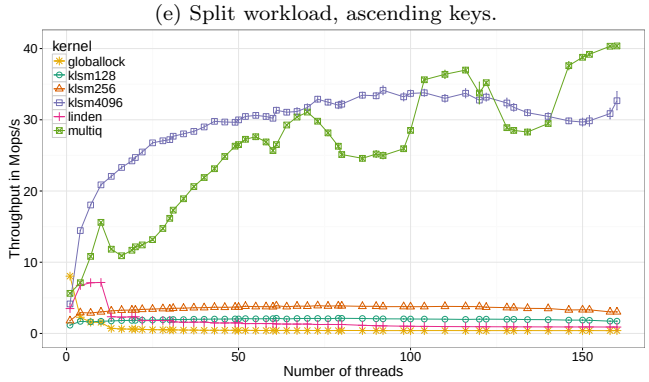
(d) Split workload, uniform keys (32 bits).



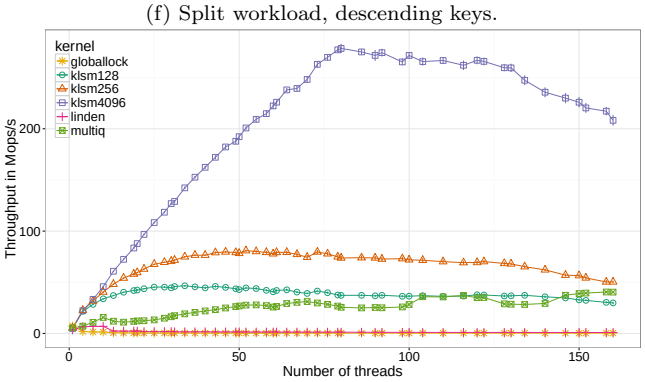
(e) Split workload, ascending keys.



(f) Split workload, descending keys.



(g) Uniform workload, uniform keys (8 bits).



(h) Uniform workload, uniform keys (16 bits).

Figure 4: Throughput on `mars`.

	20 threads		40 threads		80 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	32	29	57	48	298	288
klsm256	42	42	68	57	635	464
klsm4096	422	729	1124	1287	13469	13980
multiq	1163	3607	2296	7881	3753	12856

(a) Uniform workload, uniform key.

	20 threads		40 threads		80 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	21	18	22	19	26	22
klsm256	38	33	38	33	42	37
klsm4096	499	469	479	451	496	467
multiq	101	120	202	239	451	566

(b) Uniform workload, ascending keys.

	20 threads		40 threads		80 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	241	175	490	340	880	649
klsm256	472	341	942	667	1765	1234
klsm4096	2261	2601	2954	2728	3913	3709
multiq	329	1708	674	3641	1277	5985

(c) Uniform workload, descending keys.

	20 threads		40 threads		80 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	129	70	294	176	916	520
klsm256	217	133	557	340	1762	1174
klsm4096	4497	2495	11999	9176	41466	25387
multiq	198	617	506	1530	2528	7492

(d) Split workload, uniform keys (32 bits).

	20 threads		40 threads		80 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	20	16	27	62	44	152
klsm256	34	29	44	97	78	290
klsm4096	439	400	894	2772	1530	4027
multiq	163	634	514	1308	1031	2107

(e) Split workload, ascending keys.

	20 threads		40 threads		80 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	561	203	1138	340	2251	702
klsm256	1098	426	2385	541	4555	1286
klsm4096	13226	7884	34502	12221	56529	29092
multiq	1252	6376	6504	24567	1453	5442

(f) Split workload, descending keys.

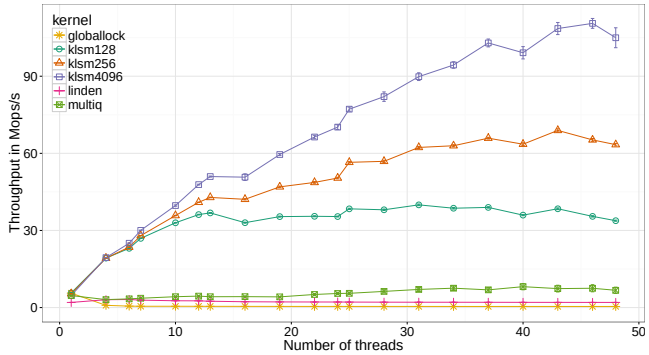
	20 threads		40 threads		80 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	992	1252	1006	1192	1059	1111
klsm256	1001	1384	1022	1399	1174	1306
klsm4096	1091	2274	1320	2480	12654	13036
multiq	1675	4263	2620	8243	3812	12166

(g) Uniform workload, uniform keys (8 bits).

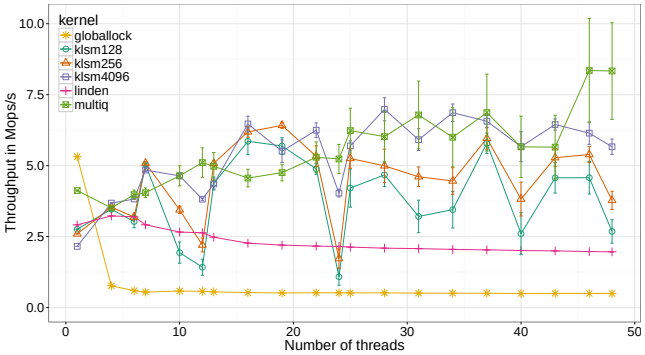
	20 threads		40 threads		80 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	36	34	58	49	267	165
klsm256	43	44	64	56	551	392
klsm4096	268	470	481	854	14060	14217
multiq	1173	3579	2398	8092	3744	12367

(h) Uniform workload, uniform keys (16 bits).

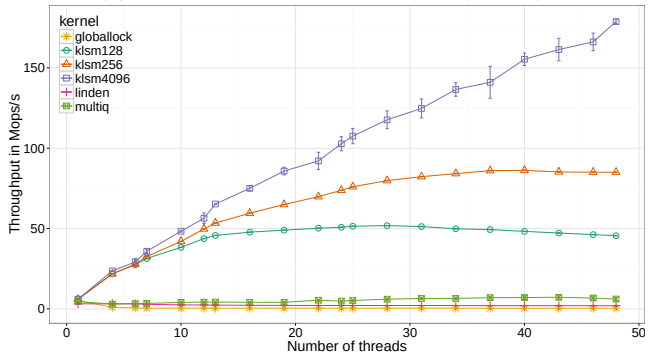
Table 2: Rank error on `mars`.



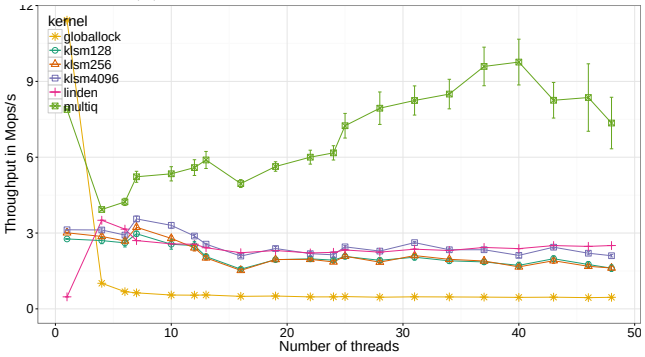
(a) Uniform workload, uniform keys (32 bits).



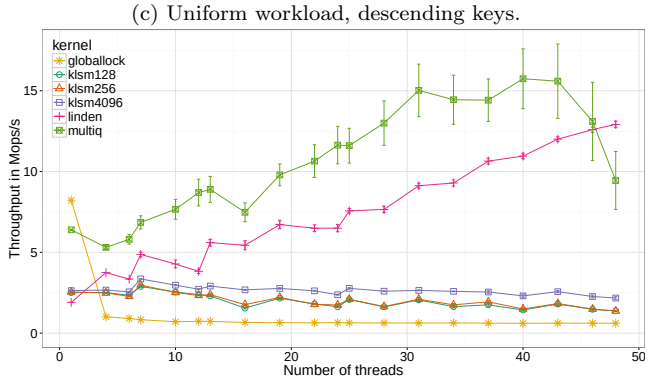
(b) Uniform workload, ascending keys.



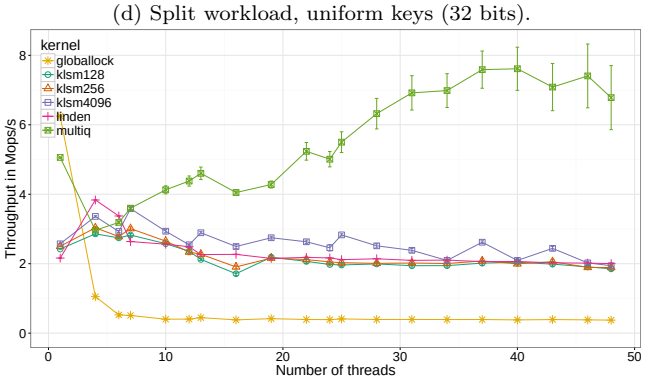
(c) Uniform workload, descending keys.



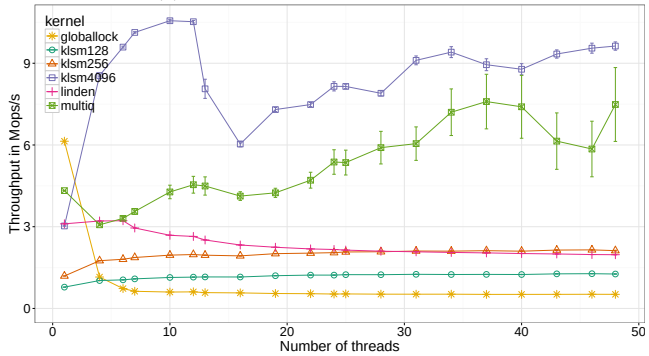
(d) Split workload, uniform keys (32 bits).



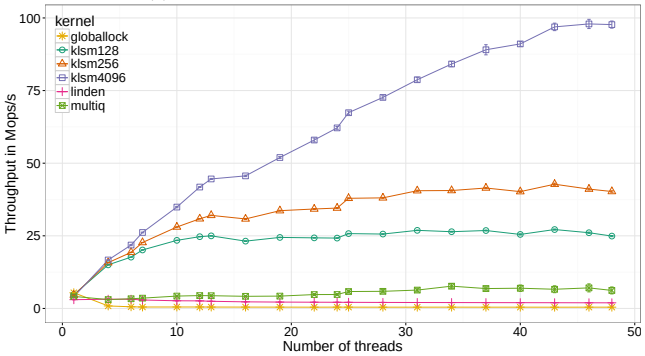
(e) Split workload, ascending keys.



(f) Split workload, descending keys.



(g) Uniform workload, uniform keys (8 bits).



(h) Uniform workload, uniform keys (16 bits).

Figure 5: Throughput on `saturn`.

	12 threads		24 threads		48 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	21	19	32	27	74	77
klsm256	31	32	43	39	120	136
klsm4096	310	452	2412	2096	3319	3006
multiq	277	625	899	2499	2298	6544

(a) Uniform workload, uniform keys (32 bits).

	12 threads		24 threads		48 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	21	17	21	18	23	20
klsm256	38	33	38	33	39	35
klsm4096	515	484	486	466	748	1130
multiq	60	71	121	143	243	287

(b) Uniform workload, ascending keys.

	12 threads		24 threads		48 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	143	113	284	213	582	447
klsm256	262	215	532	419	1092	797
klsm4096	1406	1697	2720	3141	3948	4034
multiq	193	964	381	1853	754	3418

(c) Uniform workload, descending keys.

	12 threads		24 threads		48 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	61	31	219	118	257	140
klsm256	198	114	220	113	950	524
klsm4096	1982	1218	2832	1523	5482	3825
multiq	81	140	172	332	548	1476

(d) Split workload, uniform keys (32 bits).

	12 threads		24 threads		48 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	19	15	19	16	20	17
klsm256	34	29	34	29	35	30
klsm4096	446	411	422	403	405	376
multiq	62	93	118	150	219	267

(e) Split workload, ascending keys.

	12 threads		24 threads		48 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	192	159	431	284	835	786
klsm256	362	290	759	620	1559	1564
klsm4096	6471	4024	12923	8545	24879	17970
multiq	337	1780	329	1557	372	1115

(f) Split workload, descending keys.

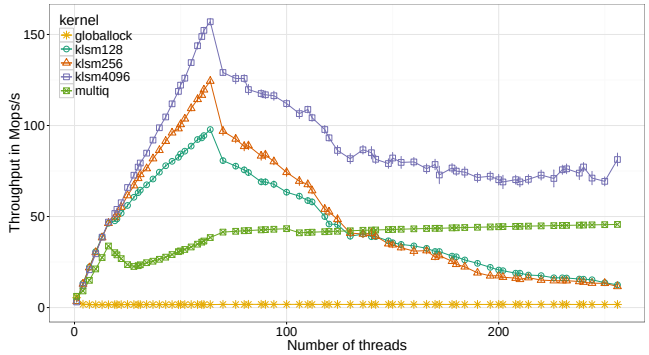
	12 threads		24 threads		48 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	986	1129	992	1119	1015	1108
klsm256	989	1253	1002	1241	1039	1234
klsm4096	1233	1858	1398	1973	2537	3540
multiq	1160	2039	1478	3094	2669	7097

(g) Uniform workload, uniform keys (8 bits).

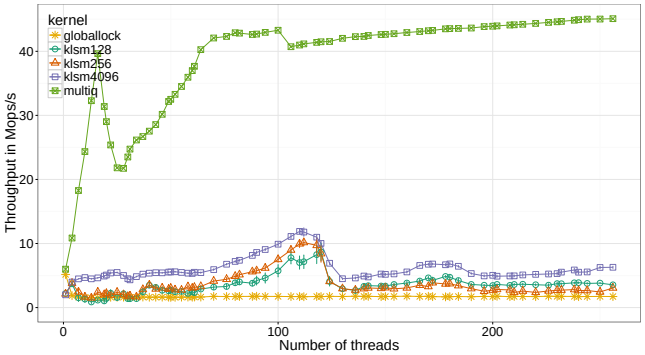
	12 threads		24 threads		48 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	26	24	37	32	82	76
klsm256	39	43	52	54	156	194
klsm4096	355	553	698	1097	2793	3934
multiq	342	843	898	2478	2314	6898

(h) Uniform workload, uniform keys (16 bits).

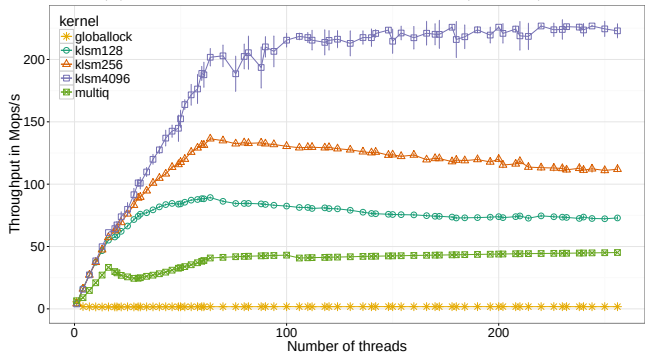
Table 3: Rank error on `saturn`.



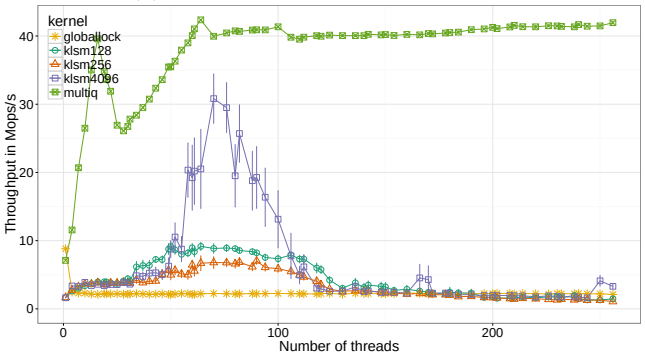
(a) Uniform workload, uniform keys (32 bits).



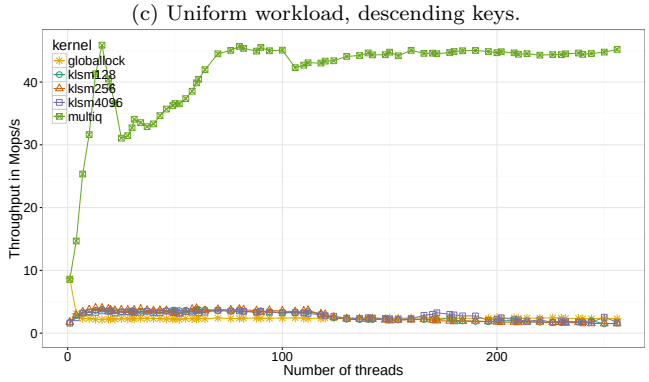
(b) Uniform workload, ascending keys.



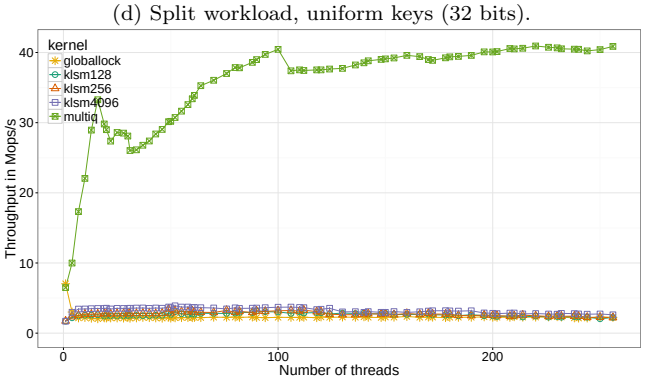
(c) Uniform workload, descending keys.



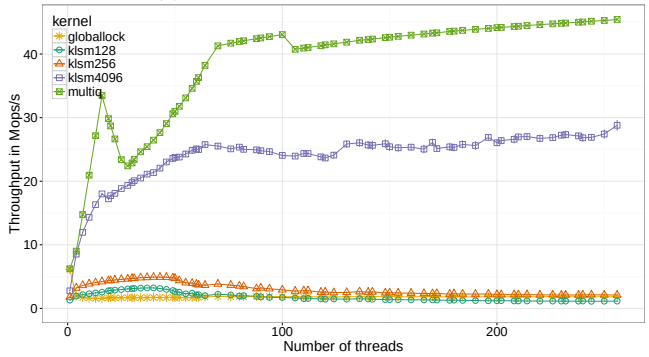
(d) Split workload, uniform keys (32 bits).



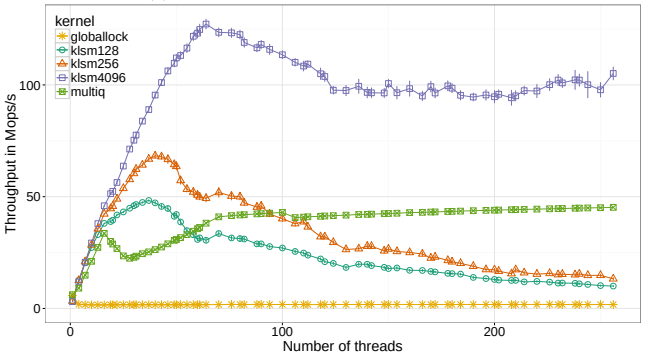
(e) Split workload, ascending keys.



(f) Split workload, descending keys.



(g) Uniform workload, uniform keys (8 bits).



(h) Uniform workload, uniform keys (16 bits).

Figure 6: Throughput on `ceres`.

	16 threads		32 threads		64 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	30	26	58	49	147	130
klsm256	43	44	79	72	231	231
klsm4096	289	488	594	1032	5831	7604
multiq	1258	4657	1995	6603	3315	11720

(a) Uniform workload, uniform keys (32 bits).

	16 threads		32 threads		64 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	20	17	23	19	25	22
klsm256	37	32	39	33	43	37
klsm4096	513	477	493	460	528	493
multiq	81	95	163	192	500	669

(b) Uniform workload, ascending keys.

	16 threads		32 threads		64 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	215	153	427	310	1064	848
klsm256	400	292	796	556	2057	1562
klsm4096	1097	961	4080	3970	7742	6931
multiq	295	1912	535	2766	1085	5348

(c) Uniform workload, descending keys.

	16 threads		32 threads		64 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	126	76	250	144	871	548
klsm256	218	127	486	287	1723	1027
klsm4096	4231	2614	13006	7627	16781	13269
multiq	376	1195	491	1290	969	2417

(d) Split workload, uniform keys (32 bits).

	16 threads		32 threads		64 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	19	15	21	18	25	22
klsm256	34	28	36	30	41	35
klsm4096	435	400	453	411	465	417
multiq	2342	10594	479	1160	5077	8644

(e) Split workload, ascending keys.

	16 threads		32 threads		64 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	381	195	696	376	1115	626
klsm256	779	389	1009	919	1584	1211
klsm4096	10584	6209	14480	11416	27109	22042
multiq	3690	18381	257	702	1363	4570

(f) Split workload, descending keys.

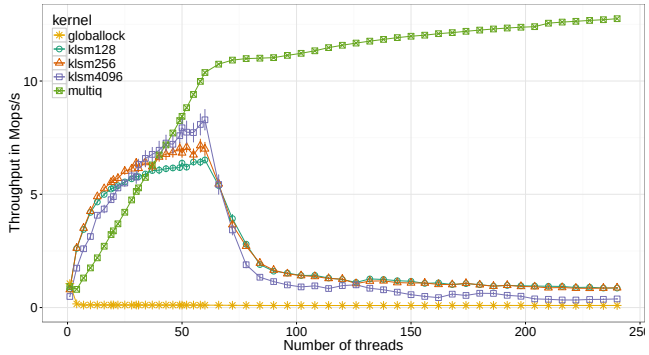
	16 threads		32 threads		64 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	997	1404	1006	1454	1108	1552
klsm256	1007	1579	1028	1610	1212	1721
klsm4096	1102	2241	1491	2693	8001	9726
multiq	1838	6381	2249	7262	3582	12676

(g) Uniform workload, uniform keys (8 bits).

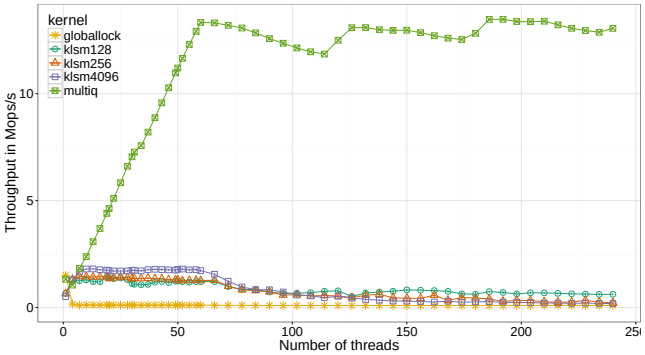
	16 threads		32 threads		64 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	33	32	63	55	142	119
klsm256	45	47	77	73	242	275
klsm4096	306	515	638	1201	4137	5347
multiq	1280	5041	1768	5602	3421	12338

(h) Uniform workload, uniform keys (16 bits).

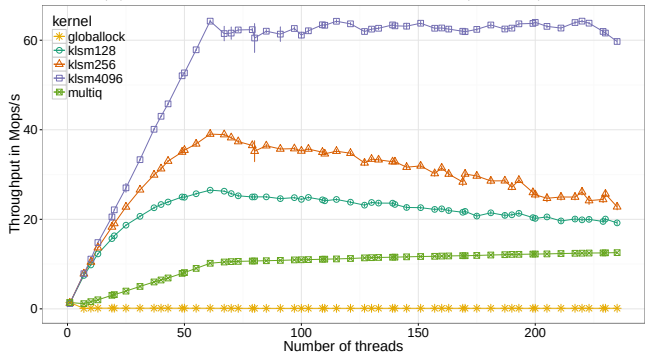
Table 4: Rank error on `ceres`.



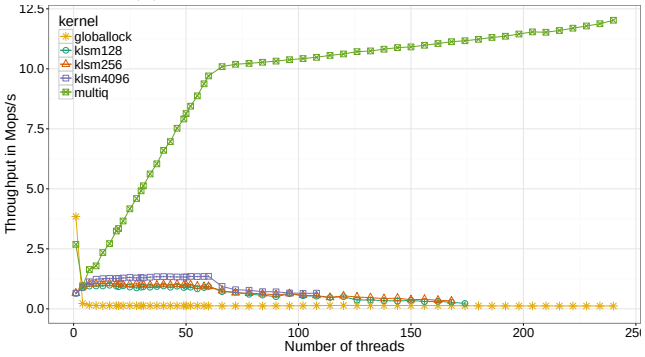
(a) Uniform workload, uniform keys (32 bits).



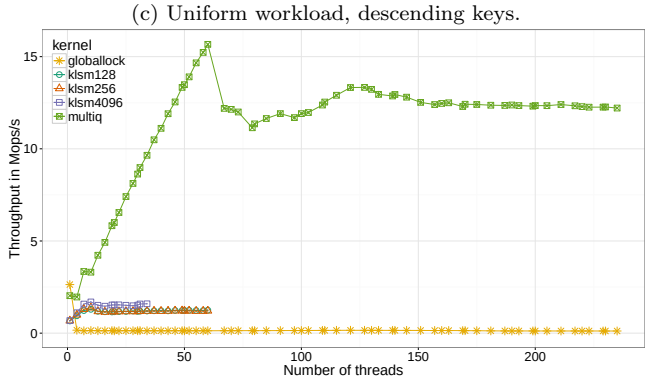
(b) Uniform workload, ascending keys.



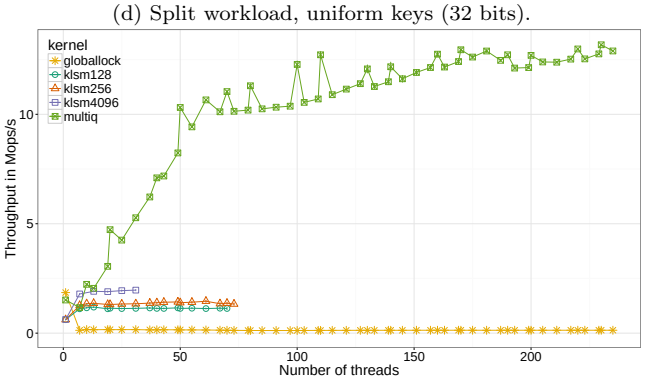
(c) Uniform workload, descending keys.



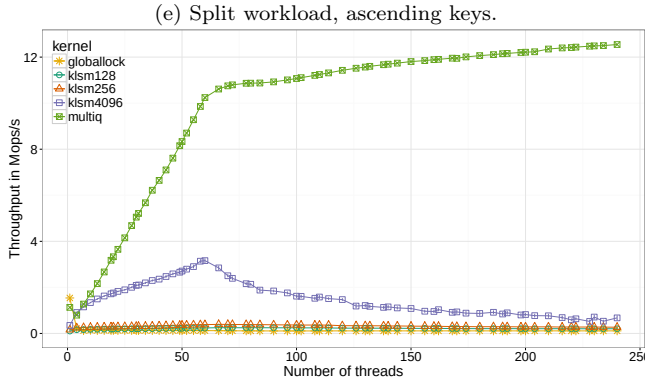
(d) Split workload, uniform keys (32 bits).



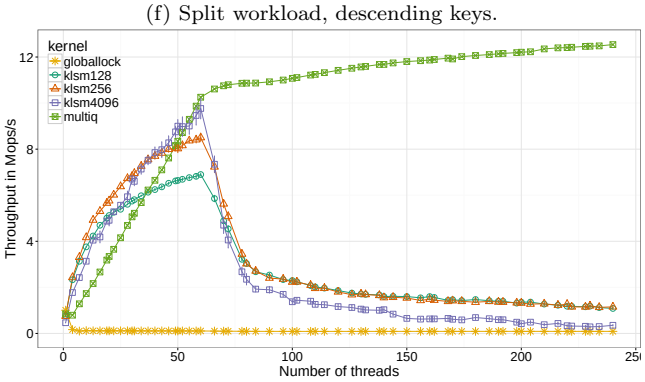
(e) Split workload, ascending keys.



(f) Split workload, descending keys.

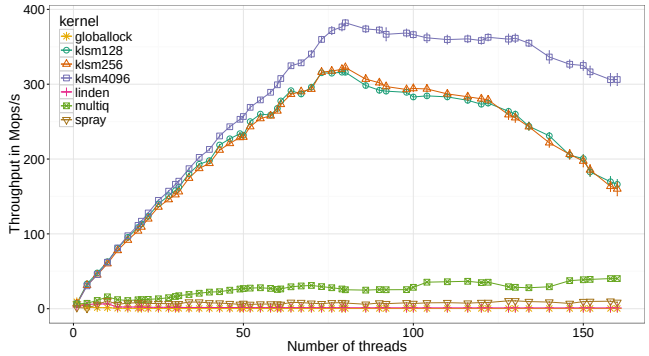


(g) Uniform workload, uniform keys (8 bits).

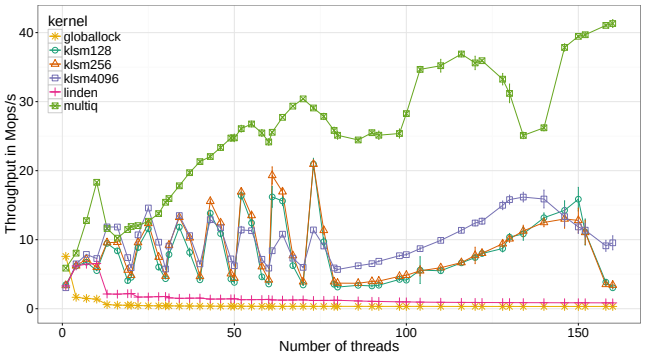


(h) Uniform workload, uniform keys (16 bits).

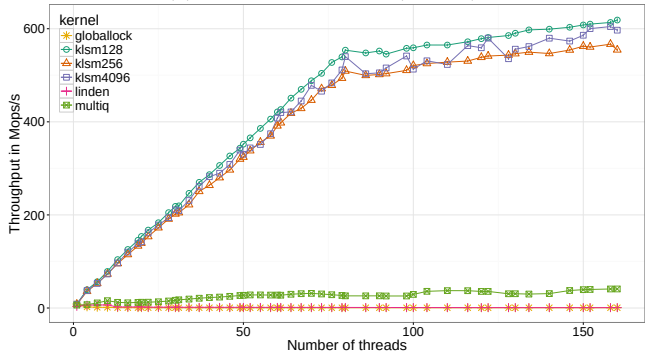
Figure 7: Throughput on `pluto`.



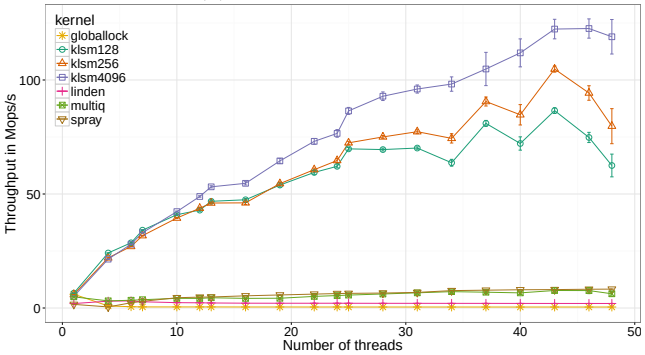
(a) **mars**, uniform keys (32 bits).



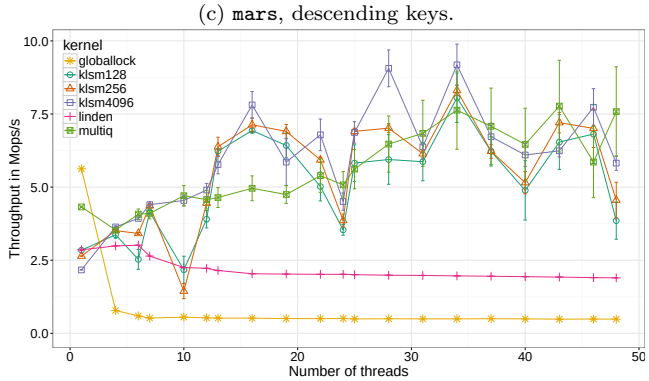
(b) **mars**, ascending keys.



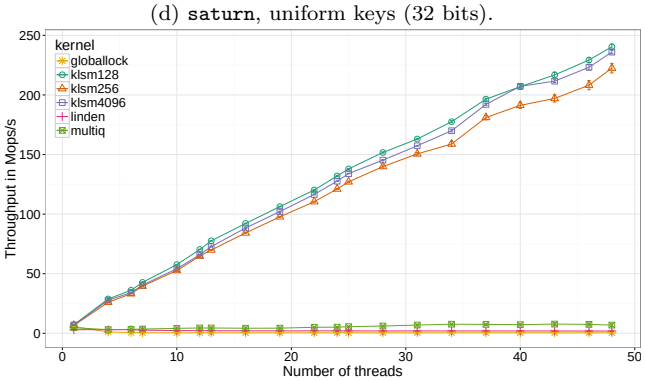
(c) **mars**, descending keys.



(d) **saturn**, uniform keys (32 bits).

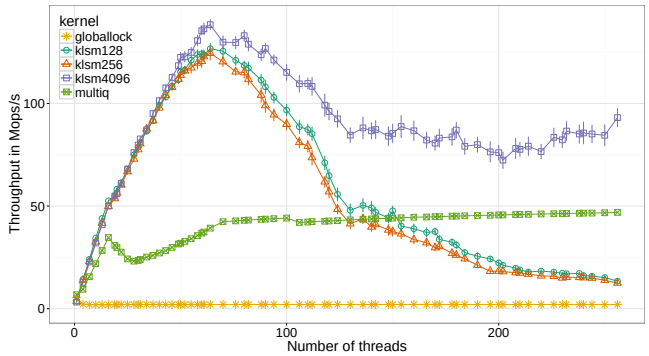


(e) **saturn**, ascending keys.

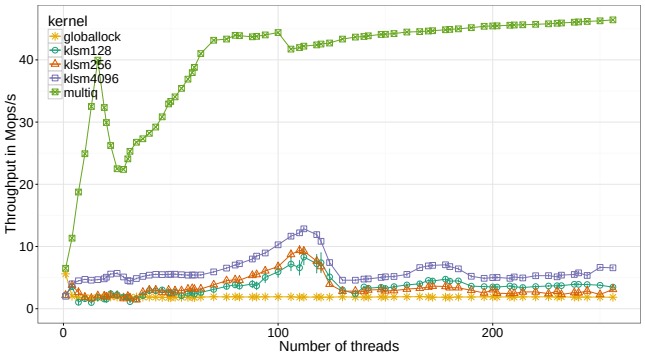


(f) **saturn**, descending keys.

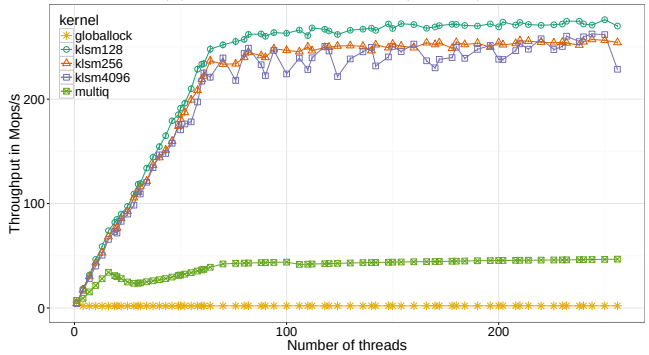
Figure 8: Throughput with alternating workload.



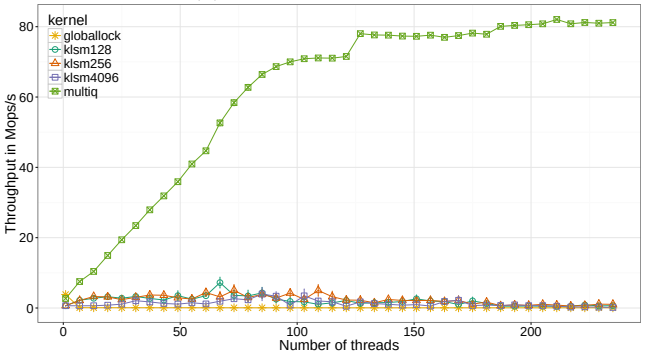
(a) `ceres`, uniform keys (32 bits).



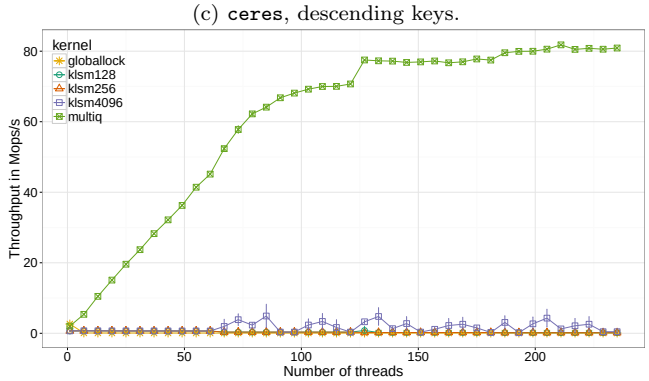
(b) `ceres`, ascending keys.



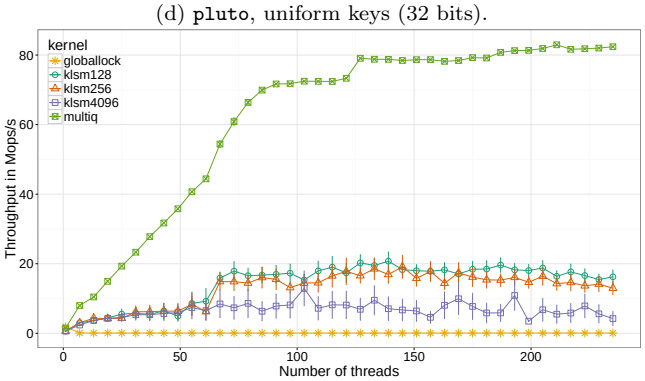
(c) `ceres`, descending keys.



(d) `pluto`, uniform keys (32 bits).



(e) `pluto`, ascending keys.



(f) `pluto`, descending keys.

Figure 9: Throughput with alternating workload.

	20 threads		40 threads		80 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	41	39	26	21	443	277
klsm256	25	30	43	50	1377	885
klsm4096	457	726	1628	1615	9486	13502
multiq	1084	3243	2422	8278	2890	7927

(a) `mars`, uniform keys (32 bits).

	20 threads		40 threads		80 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	21	18	22	19	26	22
klsm256	38	33	39	34	42	37
klsm4096	495	463	490	464	484	449
multiq	100	118	202	238	413	490

(b) `mars`, ascending keys.

	20 threads		40 threads		80 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	10	7	24	13	40	30
klsm256	11	7	23	13	41	28
klsm4096	8	6	20	15	41	36
multiq	334	1634	674	3640	1194	4935

(c) `mars`, descending keys.

	12 threads		24 threads		48 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	17	20	22	23	65	72
klsm256	31	39	34	39	115	165
klsm4096	379	599	396	591	915	900
multiq	340	839	815	2178	2216	6763

(d) `saturn`, uniform keys (32 bits).

	12 threads		24 threads		48 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	20	17	21	18	23	20
klsm256	37	32	38	33	40	35
klsm4096	509	478	492	467	849	1402
multiq	60	70	120	142	244	287

(e) `saturn`, ascending keys.

	12 threads		24 threads		48 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	6	5	11	8	24	16
klsm256	6	4	11	7	24	17
klsm4096	5	4	14	10	22	19
multiq	182	858	364	1612	766	3619

(f) `saturn`, descending keys.

	16 threads		32 threads		64 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	19	21	30	32	99	129
klsm256	30	36	50	71	151	195
klsm4096	355	576	958	1317	3602	6421
multiq	1268	4727	1944	6454	3433	12385

(g) `ceres`, uniform keys (32 bits).

	16 threads		32 threads		64 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	20	17	22	19	25	22
klsm256	36	31	39	33	42	37
klsm4096	521	485	491	461	532	491
multiq	80	95	163	192	1049	1993

(h) `ceres`, ascending keys.

	16 threads		32 threads		64 threads	
	Mean	St.D.	Mean	St.D.	Mean	St.D.
klsm128	10	6	16	11	33	23
klsm256	9	6	17	11	35	25
klsm4096	9	7	19	15	37	29
multiq	280	1706	551	3118	1071	5497

(i) `ceres`, descending keys.

Table 5: Rank error with alternating workload.