A Comparison of Latched vs. Latch Free Lock Managers

Sylvan Zheng, David Hatch, Sachith Gullapalli, Minh Tri Pham

ABSTRACT

As multi-core systems become increasingly more common and affordable, the focus within the DBMS community on scalable, highly concurrent systems is growing rapidly. Many popular database systems exhibit a throughput collapse as load and concurrency increase, even with low amounts of logical contention. Many attribute this performance hit to latch contention within the lock manager, and champion the use and implementation of a latch-free lock manager in place of the traditionally latched architecture.

However, we are doubtful that a latch free solution is both practical and necessary. Many of the studied latched systems in these comparisons suffer from poor design, exposing the system to cache coherency problems in addition to latch contention issues. There is no reason why a well-designed latch based lock manager should suffer from the same issues.

We design and implement a lock manager with fine-grained latching, reducing latch contention and cache line bouncing issues. We also implement a latch free lock manager as described by Jung et. al[1], as well as the latched lock manager described in their experiments. We show that under all practical scenarios, the performance of the latched lock manager exceeds or matches that of the latch-free lock manager.

1 INTRODUCTION

Present hardware utilized by high end database systems can feature over 100 cores and many terabytes of main memory. It is thus critical that database systems can utilize such a large amount of cores efficiently.

As noted by many others in the DBMS literature, this is not a trivial problem. Higher concurrent DBMS workers increase contention for shared objects within the database system. The lock manager is particularly prone to such contention, as every transaction processed by the DBMS must pass through the lock manager.

The lock manager's data structures are traditionally protected by a global latch or mutex, which is requested by every transaction that needs to acquire or release locks. The throughput of the system is therefore bottlenecked by contention for this global latch. Even if there is no logical contention, multiple DBMS workers are unable to access the lock manager simultaneously.

[1] H Jung, H Han, A Fekete, G Heiser, H Yeom, *A Scalable Lock Manager for Multicores*, 2013

The bottleneck of global latch contention is not the worst problem, however. Even more concerning is the high occurrence of cache line bouncing. A single core, releasing the global latch, thus invalidates the copy of the latch held in every other core's cache lines. This can waste hundreds of CPU cycles fetching from main memory and can cause a significant *drop* in throughput as the number of concurrent transactions increases.

Many have suggested the use of *latch free* data structures and algorithms to implement the lock manager, thus eliminating latch contention and cache line bouncing. However, these algorithms, are extremely difficult to reason about, implement, and suffer from portability issues due to their dependence on processor-specific atomic memory instructions. Most importantly however, it is unclear that they are actually any better performing than a well designed latched system (We describe such a system in section 3). The existing literature only compares these latch free systems to pre-existing, poor implementations of latched lock managers that only use a single global latch. [2][3]

2 BACKGROUND

Databases are expected to provide the application level programmer with certain guarantees, among them that of transaction atomicity, consistency, isolation, and durability (ACID). This frees the programmer from managing hardware failure and concurrency issues, allowing multiple concurrent database clients. The usual approach to achieve serializability is to implement a lock manager in the lower level of the database. Transactions that wish to read or write a record in the database must first obtain a logical lock protecting that record; and while that transaction holds the lock, other transactions' access to the record is restricted. A more detailed treatment of the concurrency protocol employed (two phase locking) is presented by Gray and Reuter [4].

A lock manager is typically implemented using a hash table, with each bucket mapping to a subset of database record keys. The contents of each bucket is a linked list of lock requests, each lock request describing its request type (shared vs exclusive), its state (active vs waiting), and the calling transaction. Latched systems can freely mutate this list structure, adding new lock requests and removing old ones because of the mutual exclusion guaranteed by the lock manager's global mutex. A latch free system must be much more careful in its treatment of the linked list, since at any given time another worker may be altering the same list.

3 A BETTER LATCHED LOCK MANAGER

The key observation we make in our design of a latched lock manager is the fact that latch contention and cache coherency issues described by proponents of latch free systems comes not from the fact that the lock manager is latched necessarily, but that it uses a *global* latch. A single

[2] Jung et. al., *A Scalable Lock Manager for Multicores*, 2013

[3] Horikawa, *Latch-free data structures for DBMS: design, implementation, and evaluation, 2013*

[4] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

global latch obviously introduces high levels of contention and cache line bouncing, but there is no need for the lock manager to be so coarsely protected.

Our system uses a *fine-grained latching* model to guarantee mutual exclusion during lock request list manipulation. We latch on the granularity of buckets on the hash table, which immediately alleviates many of the symptoms experienced by a global latch system; if transaction A needs to lock on hash bucket 1, only other transactions which need to lock on the same hash bucket will induce latch contention. The granularity of latching can also be easily modulated by adjusting the hash table load factor.

On a practical level, we simply use two hash tables; one for the linked lists of lock requests, and another simply of mutexes. These mutexes are granted before lock acquisition and release, and released as soon as the acquire or release operation has completed.

4 LATCH FREE LOCK MANAGER

The latch free lock manager is implemented based on the description in Jung et. al[5].

In addition, we implement a performance improvement to the lock manager to avoid unnecessary list traversal when releasing locks. This optimization is applied to the latched managers and latch-free managers. It is particularly beneficial for the latch-free manager since latch-free lock lists can contain *OBSOLETE* nodes, making list traversal more expensive than in latched counterparts.

For each lock, we maintain an *outstanding count*, which tracks the number of transactions currently holding that lock. We must only grant new locks after a lock release when the *outstanding count* is zero. In our implementation, since granting a lock (setting the state to ACTIVE) and incrementing the *outstanding count* is not an atomic operation together, we must relax the condition, by granting new locks unless *outstanding count* is strictly positive.

5 EVALUATION

We measure the scalability of our fine-grained lock manager (Our Lock Manager) with the traditional globally-latched lock manager and the latch-free manager. We implemented the global lock manager and latch-free lock manager as described by Jung et. al.[6] and Our Lock Manager as described by Section 3. We measure the throughput of the three systems under varying levels of contention and varying multi-programming level. The purpose of these tests is to confirm that latch-free systems do not outperform well designed latch-based systems under high contention. We show that latch-free lock managers only outperform the poorly designed

[5] Jung et. al., *A Scalable Lock Manager for Multicores*, 2013

[6] Jung et. al., *A Scalable Lock Manager for Multicores*, 2013

globally-latched lock manager, which many papers unfairly compare their hyper-optimized latch-free implementations to.

We designed a custom testing framework that tests the scalability of the lock managers in an isolated fashion. On each test run, we generate a batch of 500,000 transactions, which the lock managers process. Each transactions contain a list of 20 different exclusive lock requests. These lock requests request randomly chosen keys from a set of keys of varying size. If a lock request is granted, the lock manager goes on to request the next lock in the list. If not, it spins on the lock state[7] until the lock is granted.

All tests are run on a Dell 64-core Linux machine, whose characteristics are listed in Table 1.

Table 1: Dell M915 Hardware Specifications

Component	*Specification*
Machine	Dell M915
Processors	64-Core AMD Opteron 6276
Clock Speed	2.30 GHz
RAM	512 GB

5.1 EXPERIMENTAL RESULTS

We evaluate the scalability of the three lock managers by measuring the transaction throughput per second of the three systems under varying levels of contention and varying multi-programming level. We change the size of the key set to vary the contention level and the number of threads to vary the multi-programming level. Our results confirm that latch-free systems do not dramatically outperform well designed latch-based systems.

5.1.1 CORE SCALABILITY

We evaluate the scalability of the lock managers with respect to the number of cores used. Figure 1 measures the throughput under low contention--500k 20-key transactions on 100k keys--and Figure 2 measures the throughput under high -- 500k 20-key transactions on 5k keys. Our purpose here is to compare the relative performance of the three systems under different multi-programming levels and different levels of contention. Under low contention, we can see that both the latched lock manager and the latch-free manager achieve maximum throughput at 64 threads for both tests. This makes sense, as we are running our tests on a 64-core machine.

[7] In future work we may investigate the effect of maintaining a transaction wait queue to maximize concurrency levels

Using any less than 1 thread per core, we are not taking full advantage of parallelization. Using more than 1 thread per core, we incur the large overhead of context switches while using the same amount of CPU time, which leads to a sharp decrease in performance. The key-lock manager slightly outperforms the latch-free manager under low contention until 64 threads, while with more threads the latch-free manager performs slightly better than the key-lock manager.

The high-contention results are more mixed. Again, the key-lock manager slightly outperforms the latch-free manager until we hit about 40 threads -- at this point, additional threads fail to increase the throughput of the key-lock manager (although performance does not significantly degrade while we keep the number of threads below the number of cores), while the latch-free manager continues to scale with the number of threads until every core is running a thread. This suggests that the latch-free algorithm is more parallelizable, which makes sense as only one thread can access a key at a time in the key-lock implementation. However, finer-grained locking might allow key-locking to perform better (currently, threads hold mutexes until the entire operation is complete -- we could probably release mutexes for some parts of the operation and relock them when necessary, potentially enabling more parallelism).

We can also see that the performance of the global lock manager degrades as we increase the number of threads. This makes sense, as locking the whole hash table on every lock request does not allow us to take advantage of parallelization in the lock manager. Furthermore, performance collapses due to cache line bouncing issues.

Figure 1

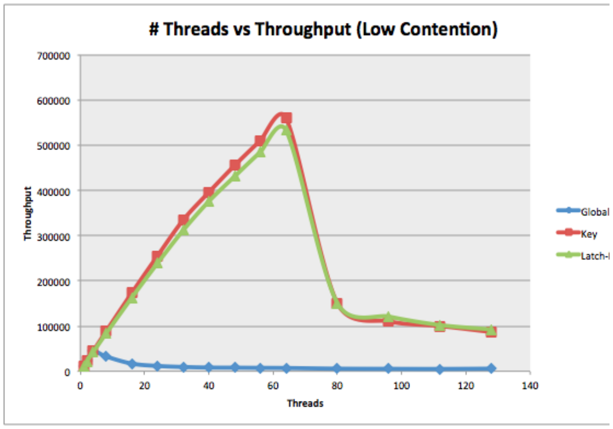


Figure 2

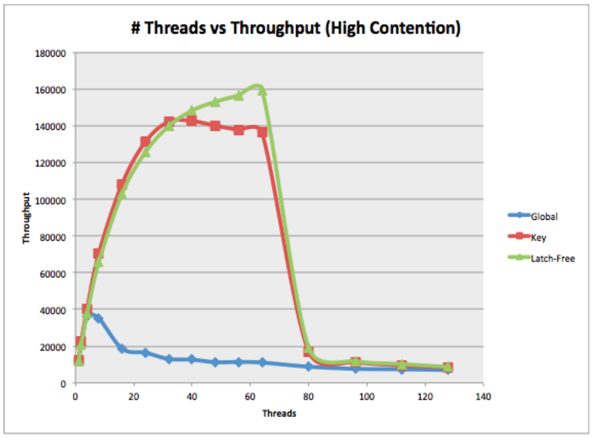
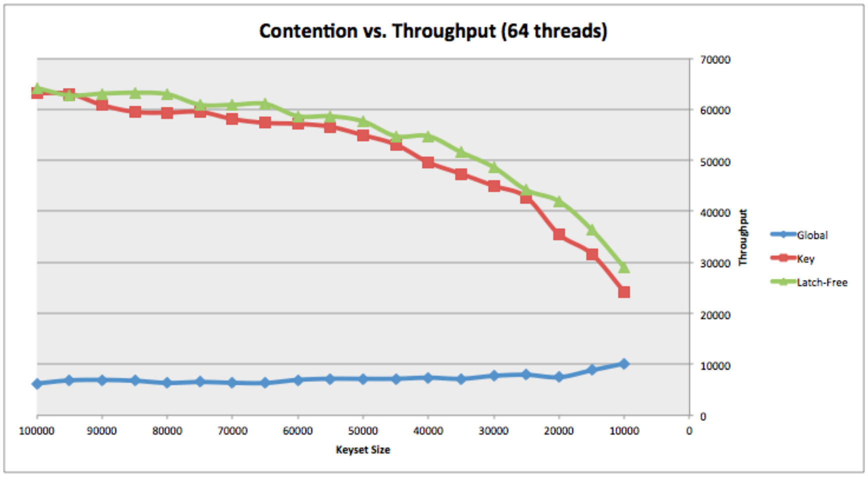


Figure 3 compares the throughputs under different contention factors while holding the number of threads constant at 64. We can see that the transaction throughput per second decreases for all 3 systems as we increase the contention factor, i.e. decrease the size of the hot set. This corresponds with our intuitions--the throughput is lower as higher contention factor generates more conflicts in the hash table data structure.

Figure 3



CONCLUSION

Contrary to the popular belief, we found that latch-free lock managers do not necessarily outperform latched-based lock managers. Most popular research compares a hyper-optimized latch-free lock manager to a naively designed globally-latched lock manager, which we believe is an unfair comparison. To address this problem, we implemented an improved latch-based lock manager, which utilizes more fine-grained locking. We also implemented a latch-free lock manager according to the description of Jung et. al. and the naive globally-latched lock manager. Our experiments show that our latch-based system achieves far more comparable transaction throughput to a system based on Jung et. al.'s latch-free system under relevant multi-programming levels, although a latch-free lock manager may still enable higher levels of parallelism under high contention levels. More work needs to be done to see if even finer-grained locking mitigates the scaling issue experienced by our key-lock manager under high contention.

In this paper, we tried to make the algorithms used by the latched and latch-free lock managers as similar as we could, in order to make our comparison as fair as possible, but the authors suspect that this "fairness" may be artificial. There were a number of simple optimizations in the latched manager we refrained from implementing because we couldn't make them latch-free. Working in a latch-free environment prevents the programmer from making a number of simplifying assumptions; it's likely that the performance of an optimal latched lock manager is significantly better than the performance of an optimal latch-free lock manager because the optimizations that can be made in the latter case are a small subset of the former.

Another potential weak point of our approach is the simulation-based approach. We are not using full DBMSs and workloads. Instead, we are merely testing the lock managers in isolation and generating random workloads. Further work investigating the effects of transaction duration time, request time, and mixed contention levels will allow for confirmation that latch-free algorithms do not perform better than fine-grained locking in a commercial DBMS system, making them not worth the added implementation complexity.

ACKNOWLEDGEMENTS

We would like to thank Jose Faleiro and Daniel Abadi for guiding us throughout the research process and giving us valuable comments and feedback.