Concurrent article repository through fine grained locks

Concurrency and Parallelism - Project

José, Carneiro
Universidade Nova de Lisboa - FCT/DI
2829-516 Caparica, Portugal
jm.carneiro@campus.fct.unl.pt

Ramalho, Ruben
Universidade Nova de Lisboa - FCT/DI
2829-516 Caparica, Portugal
r.ramalho@campus.fct.unl.pt

ABSTRACT

Concurrent computing is a form of computing in which several computations are executed during overlapping time periods concurrently, instead of sequentially. Which has the advantage of exploiting the increasing number of cpu cores that todays *modern machines* have at their disposal.

The design of concurrent-multi-threaded programs is not trivial tough, as there is a need to preserve consistency between shared resources while trying to minimize performance costs.

In this work we were given the task of improving an existing application, an article repository.

In our solution we propose an implementation that uses fine-grained locks on shared hash tables to protect accesses of threads to each bucket.

Keywords

Concurrency; ReadWriteLock; Mutual Exclusion; HashTable; Deadlock

1. INTRODUCTION

The provided/initial application uses three hashmaps to implement an in memory database for scientific articles, where articles are indexed by their associated title, list of keywords and list of authors. This initial implementation has concurrency issues; that is, it wasn't projected with the intent of allowing concurrent operations.

The goal of this project is to extend this program by implementing proper *concurrency control* such that concurrent operations preserve the consistency of the application; that is, if one article has already been inserted it must be accessible from every table and if it has been deleted it should not be up for retrieval, which relates to the problem of mutual exclusion. The mutual exclusion problem states that two basic requirements need to be satisfied:

Mutual exclusion.
Two processes can't be in their critical sections at the same time.

Deadlock-freedom.
If a process is trying to enter its critical section, then some process, not necessarily the same one, eventually enters its critical section.

2. APPROACH

Throughout this project we tried to develop multiple ideas for the aforementioned problem. In this section we try to present them and outline the limitations of each approach:

2.1 Coarse-grained locks

In our initial approach we made use of java's *Synchronized* keyword to make sure each insert and remove operations of the repository would be mutually exclusive.

While this solution solves the problem of the application getting into an inconsistent state it does so by completely blocking the access of other threads to the map. This policy leads to poor scalability and performance, as real concurrency is lost by locking all the tables per operation.

2.2 Fine-grained locks

To improve on this initial solution we removed the coarse grained locks and implemented a locking policy where individual hash table buckets are locked instead, in contrast with the previous policy: insertions and removals don't lock the whole table but instead only lock the bucket that is subject to change. Regarding the locking discipline we lock bucket's using *ReentrantReadWriteLock*,the motivation for this is that this lock discerns the types of accesses(read and write), it allows multiple reads, provided that no write operations are in course.

In our implementation we also use repository level locks to ensure consistency between the three shared data structures.

2.3 Very Fine-grained locks

We also considered the use of even more fine grained locks by locking at the node level, but we decided not pursue them since the added grain leads to poorer results than bucket level locking, as stated in Dudas et al [1].

3. VALIDATION

In order to validate the consistency of our system a validation procedure is executed at 5 sec intervals; this validation procedure retrieves all the existing articles in the article table and asserts that if an article exists then the corresponding keywords and authors should also exist in the auxiliary tables. If this invariant doesn't verify then the validation procedure is aborted and the evaluator is informed of the lack of consistency in the system.

Part of the objective for this project was the design of additional invariants for the system. We came up with an additional invariant that checks if the number of different authors references by articles is equal to the total number of

authors in the *byauthor* structure. With the base verification it could be the case that remove operations weren't being processed at all and the validation would pass. This same invariant also applies to the *bykeyword* structure.

Overall we have found the baseline code to be very comprehensive and quite sensible to the detection of incoherencies. Additional invariant and verifications could be added but most of them can't be readily implemented without major changes to the original program.

4. EVALUATION

In this section we will evaluate our approach under different parameterizations; that is, when allowed more computing resources and when subject to different environments, for example in situations where the write rate is high.

4.1 Resource-wise

In order to evaluate the performance of our approaches, here measured through the proxy variable *number of operations*, we fixed the program with the default/proposed parametrization: *time*:30, *nkeys*:1000, *puts*:10, *removes*:10, *gets*:80,*nauthors*:100, *nkeywords*:100, *nfindlist*:5. And incrementally increased the number of threads for both approaches. We then proceeded to plot the number of operations for both approaches and draw conclusions.

Through this plot we hoped to evaluate the degree of concurrency allowed by both approaches, to no surprise we observed that our coarse grained approach ceases to benefit from extra computing resources very soon, the mean number of operations is steady beyond 4 threads.

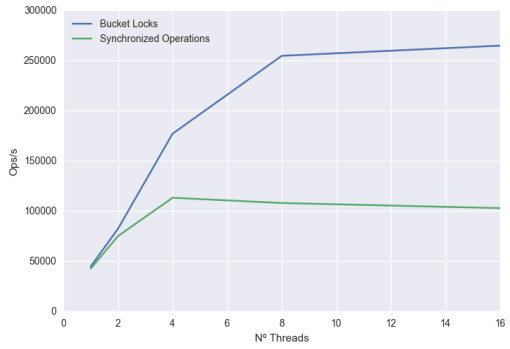

Figure 1: Number of operations as a function of the number of threads.

For our second approach we observe that the performance; that is, number of operations, steadily increases with the allocation of additional threads. To the point were we allocate more threads than the number of cores and this performance ceases to grow and reaches a point of equilibrium. From where it can only degrade.

4.2 Operation-wise

On a second evaluation approach we defaulted the number of threads to 8, as stated before this parametrization exhibited the best performance under our 8 core machines.

And proceeded to incrementally increase the write rate, which we define as the percentage of insert + remove operations. The write rate starts at 10%; that is, 5% inserts, 5% removes and 90% gets and linearly increases up to 10% gets, 45% writes, 45% removes.

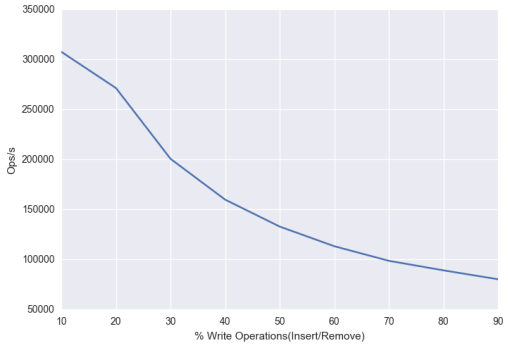

Figure 2: Number of operations as a function of the number of writes.

This experiment issues very interesting results, when the number of get operations is high operations are cheap and our system is more performant. But as the number of write operations increases, which we recall are reentrant, the overall performance of the system degrades. The reason is twofold: write operations are high priority which leads to a slow down in gets, and write operations have mutual exclusion requirement which overall leads to it being a more expensive operation.

Additional Parameters.

We are aware of the impact that the additional parameters could have on our program. *nkeys*, *nauthors* and *nkeywords* would affect the amount of collisions which would hurt performance in case of an increase and improve it otherwise. Thus we didn't explore these options any further.

5. CONCLUSION

In this project we implemented and analyzed synchronization methods for a concurrent article repository. We learnt that coarse-grained methods don't often translate into performant programs but if the granularity is too fine, the overhead and memory required might hurt performance as well. Thus a proper balance between the two is essential; in our solution we seek this balance through a bucket-wise locking policy, which we propose to be implemented with *ReadWriteLocks*. These units only lock concurrent reads in the presence of a write operation, which allows for a higher degree of concurrency under regularity assumptions.

We then proceeded to validate our solution and verify that our validation procedure finds no inconsistencies in our solution; that is, all invariants pass. Finally we evaluate our solution under a simulated data setup, and verify that our solution offers better scalability than a naive coarse grained solution.

6. REFERENCES

[1] Akos Dudas, Sandor Kolumban and Sandor Juhasz Performance analysis of Multi-Threaded locking in bucket hashtables