Annales Univ. Sci. Budapest., Sect. Comp. 36 (2012) 63–74

PERFORMANCE ANALYSIS OF MULTI-THREADED LOCKING IN BUCKET HASH TABLES

Ákos Dudás, Sándor Kolumbán and Sándor Juhász

(Budapest, Hungary)

Communicated by László Lakatos

(Received November 30, 2011; accepted January 19, 2012)

Abstract. Data structures, such as hash tables, are often accessed from within critical sections in multi-threaded environments in order to preserve data integrity. The extent of mutual exclusion greatly affects the performance by limiting the level of achievable parallelism. Increasing the resolution of locking allows higher throughput at the cost of increased memory use with all its side effects. Highly concurrent bucket hash tables use fine-grained locking where each lock protects one or a few buckets. In this paper we analyze the behavior or large hash tables with respect to their performance when the granularity of locking changes. We offer a simplified queuing model for estimating the number of locks that the hash table should use for maximum efficiency. The model and its suggestions are evaluated under real-life circumstances by testing a concurrent custom bucket hash table.

1. Introduction

Hash tables used in multi-threaded environments are considered as shared resources where special care is required in order to preserve data integrity when modifying data. The traditional solution for avoiding race conditions is the use of mutual exclusion: parts of the data structure are protected by critical sections, which guarantees at all times that at most one thread is allowed to access (read or write) the protected area.

Critical sections are basic building blocks of algorithms, hence most modern hardware architectures provide support for atomic operations for locking

Key words and phrases: Hashing, bucket hash table, multi-threading, locking.
1998 CR Categories and Descriptors: D.1.3, D.4.8.

with the use of compare-and-swap and/or bit-test-and-set operations. These operations guarantee that no interruption occurs during the action.

When it comes to performance the most important question is the placement of the critical section(s). The most trivial solution is the use of a single lock that protects the entire data structure (such as the whole hash table in our case). Such solution is easy to implement but does not make full use of the multi-threaded potential as the lock practically serializes the accesses to the protected data. If the structure can be partitioned into disjoint blocks and it is guaranteed at all times that each thread can complete its operation by only accessing data within a single block finer grained locking can provide considerably better performance than a single critical section. Protecting each of these blocks with a dedicated lock allows parallel access to the different blocks of the original data structure, while it still protects the integrity of the data.

Fine-granularity also has its disadvantages. If the data structure is large in size (e.g. hash tables designed to contain tens of millions of items) the cardinality of the locks needed would be in the range of thousands or millions. Most runtime environments support only a finite number of locks [6]; plus the management cost and the allocated memory would make such a large number of locks impractical (see Section 3.1 for a detailed explanation).

In this paper we investigate how the change of the granularity of locking in a large hash table affects the performance. Given the number of locks and the number of concurrent threads we estimate the number of clashes that occur when the threads try to acquire the same locks. We argue that this is the most important factor, which determines the performance of the parallel data structure. This number will help in estimating the optimal number of locks that the hash table should use.

The rest of the paper is organized as follows. Section 2 cites related literature including a short introduction to hash tables and parallel implementations. Section 3 argues for fine-grained locking in bucket hash tables and presents the queuing model which describes such a hash table. The model is verified under real-life circumstances by implementing and testing a custom bucket hash table. In Section 4 the time a thread needs to wait before it can successfully acquire a lock is characterized with a simplified model and using that a criteria for the optimal number of locks is given. We conclude in Section 5 with a summary or our findings.

2. Related works

Hash tables [7] store and retrieve items identified with a unique key. A hash table is basically a fixed-size reference table, where the position of an

item is determined by a hash function. If the position calculated by the hash function is already occupied, a collision occurs. This collision can be resolved by reserving external storage space container for the colliding items (e.g. a linked list); this is the approach of bucket hash tables. The collection of items that map to the same slot of the table is called a bucket. In this paper we are only concerned with bucket hash tables, other collision handling solutions are not discussed.

Most operations that access the hash table use only a single bucket, hence access to different buckets seldom cause concurrency problems. Protecting the buckets with individual locks provides high level of parallelism, but comes with the cost of managing hundreds, thousands, or millions of locks. In order to avoid it the hash table can be divided into blocks containing one of more buckets (with blocks covering the same amount of data) and each of the blocks is protected by a lock. This allows tuning locking resolution depending on the size of the blocks.

Larson et al. in [9] use two lock levels: a global table lock and a separate lightweight lock (a flag) for each bucket. The global table lock is held just as long as the appropriate bucket's lock is acquired. They describe that for fast access spin locks are used instead of Windows critical sections. It was shown by Michael [11] that in case of non-extensible hash tables simple reader-writer locks can provide a good performance in shared memory multiprocessor systems. More efficient implementation like [10] use a more sophisticated locking scheme with a smaller number of higher level locks (allocated for hash table blocks including multiple buckets) allowing concurrent searching and resizing of the table. More complicated locking scenarios are also available, such as a hierarchical lock system with dynamic lock granularity escalation, as presented by Klots and Bamford [6].

For the sake of completeness it must be mentioned that there are lock-free [3] hash tables too (Lanin and Sasha [8], Greenwald [2], Michael [11]). These lock-free solutions do not use mutual exclusion but present additional overhead or require primitives not supported by current CPUs. These lock-free methods promise great performance advantages compared to ones with locks, but at the same time they are more complex and their correctness needs verification [4].

3. Queuing model for locking regions in the table

This section presents some arguments for locking regions of buckets instead of individual buckets. We derive a queuing model for the locks in the hash table and calculate the probability of threads having to wait when acquiring the locks.

3.1. Locking blocks of buckets

We argue that it is unnecessary to lock individual buckets in a hash table. The level of parallelism the hash table supports is directly related to the time the threads spend waiting for entering a critical section. This is, of course, effected by the number of locks (more precisely by the probability of a thread picking a particular lock from the set of locks); but also depends on the frequency that the locks are accessed. This is determined by the time spent within and outside the critical section when there is only a single thread in the system. These properties can be best captured by a queuing model for the individual locks. If the model accurately describes the behavior of the locks, we can calculate the probability of a particular thread having to wait before it can enter a critical section protected by a lock. This probability gets smaller as we increase the number of locks, but at the same time management costs (memory overhead, cache miss rate) grows. The hash table enables maximal throughput if the right balance is found.

Considering the real-life implementation of locks there are two issues that support our effort for using as few locks as possible while still supporting good parallelism. The first is the cost of a lock. A lock, regardless of its actual representation, consumes memory. Using too many locks in order to increase performance can have the opposite effect as the locks take away precious space in the CPU caches that would otherwise be occupied by valuable data. The second issue is also relevant in terms of the caches of the CPU. Whenever a lock is acquired or released a flag of some kind needs to be altered in the memory and this change must be reflected in all cores of the CPU. This means that changing a lock implicitly means purging the cache line the lock is placed inside from all levels of caches of the CPU. This can attribute to increased amount of cache misses if the locks are placed alongside valuable data (this effect is called false sharing).

3.2. The queuing model

In a multi-threaded system multiple threads are trying to access a hash table, which is partitioned into smaller sections (blocks of buckets) and each of these blocks is guarded with a lock for mutual exclusion. The number of threads is denoted by N while L denotes the number of locks guarding the hash table ($1 \leq L \leq$number of buckets).

When a thread is trying to acquire a lock, which is already held by another thread, then the execution of the acquiring thread is stopped until it can access the lock. When multiple threads are waiting for a the same lock they are constantly checking if the lock has been released, and when it becomes free the first thread realizing this will get access to it (spin waiting).

Assumptions:

1. When a thread selects a lock to acquire this can be regarded as uniform selection meaning that every lock is selected independently from every other random phenomena in the system with probability $1/L$. This can be guaranteed by a good hash function that maps a particular request to a bucket/lock, supposing that items are chosen from the universe uniformly.

2. The time spent in the critical section is exponentially distributed with parameter μ. This means that the expected time span a thread spends in a critical section is $1/\mu$. This is common for all threads.

3. The time between leaving the critical section and accessing a new lock is also exponentially distributed with parameter γ.

With these notations the system under consideration can be seen as the following queuing system.

There are L parallel queuing systems with infinite buffer. New requests go into the queue of the selected lock and wait there until they are served. The serving discipline is random (SIRO). The phenomena, which makes the system hard to analyze is that the arrival intensity for any given queue depends on the length of other queues in the system as the number of threads that may want this particular lock is less than N (number of all available threads) due to some being occupied otherwise (in other critical sections of waiting for other locks).

The following simplifying assumption is used.

4. We will assume that there is an infinite number of threads in the system. Thus requests arrive to any given queue with intensity

$$\lambda = \frac{N\gamma}{L}.$$

This means that the total number of threads in a particular critical section and waiting for the lock is negligible compared to the total number of threads. Usually this is true for fewer number of threads as well, if the number of candidate critical sections is much higher than the number of threads and the average time spent in the critical section is small. This ensures that in the vast majority of the time all N threads can check in for a lock.

From assumptions 1-4 it follows that every single lock with its waiting queue is (in Kendall's notation) an $M/M/1/\infty/\infty/SIRO$ queue with arrival intensity

λ and service intensity μ. If the number of waiting threads for a given lock ℓ is denoted by X_ℓ then the queue length of a lock ℓ is given as

$$\mathbb{P}(X_\ell = k) = \left(1 - \frac{\lambda}{\mu}\right)\left(\frac{\lambda}{\mu}\right)^k = (1-\rho)\rho^k. \tag{3.1}$$

We would like to characterize the event when a thread acquires a lock but is blocked because it needs to wait for a critical section to become available. We call this **clashing**. Clashing occurs if a thread is trying to acquire a lock which is being used, meaning that its queue is not empty. The probability of such an event is calculated as follows

$$\mathbb{P}(\text{clash}) = \sum_{k=0}^{L} \mathbb{P}(\text{choose occupied lock}|k \text{ locks occupied})\mathbb{P}(k \text{ locks occupied}) =$$

$$= \sum_{k=0}^{L} \frac{k}{L}\binom{L}{k}\rho^k (1-\rho)^{L-k} = \frac{1}{L}\sum_{k=0}^{L} k\binom{L}{k}\rho^k(1-\rho)^{L-k} =$$

$$\overset{\text{expectation of a binomial}}{=} \frac{1}{L}L\rho = \rho = \frac{\lambda}{\mu}.$$

3.3. Evaluation of the model

In order to verify the model we implemented a custom bucket hash table and prepared it for parallel access by providing locking mechanisms on configurable blocks of buckets. We chose physical evaluation over simulation in order to capture the real-life circumstances in multi-threading.

The items that are inserted into and searched in the hash table (we measured these two operations) are distributed among the buckets with a good hash function (FNV [12]). The hash table is large in size: there are 8 million items in the table and as a total of 16 million operations are executed on the table. For the locks custom assembly-level locks were used with a cost of one bit per lock (in order to allow large amounts of them).

Our previous research [5] showed that the bucket hash table performs best under these circumstances when it has more buckets than items inserted into the table. Therefore 12 million buckets were allocated. The number of locks varied between 100 and 12 million in order to explore the whole range; the former resulted in 120 thousand buckets assigned to a lock while the latter one of course means a single bucket for each lock. We counted the number of clashes that occurred during the operations (the number of times threads failed to acquire a lock on their first try). The relative frequency of clashing is this number divided by the total number of operations. Because of the high number of operations this is a good estimate for the probability of clashing.

The hash table was implemented and carefully optimized in C++ compiled with Microsoft Visual Studio 2010 default release build optimizations. The physical hardware contained an Intel Core i7 CPU with 4 physical cores and HyperThreading technology resulting in 8 parallel execution units that allow $N = 8$ concurrent threads to execute without diminishing the performance of each other. The measurement result provided here reflects an average of 50 independent executions.

Figure 1 plots the estimated and the calculated probability for $\mu = \mu_1$ and $\gamma = \gamma_1$. The actual values for μ_1 and γ_1 were determined by measuring the time a thread spends within the critical section and outside the critical section in case of $L = 1$. These times determine the arrival intensity and the service intensity of the queues, hence their value relative to each other is important. By introducing additional work that the hash tables can perform outside the critical section a different type of workload is created and tested (see Figure 2). For example the additional workload includes calculating the hash function, which can be performed outside the critical section too, and is in fact computationally expensive.

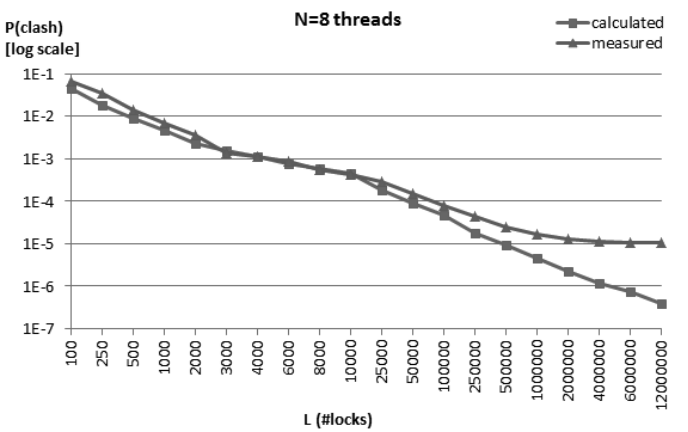

Figure 1. Measured and calculated probability of threads having to wait for a lock with various amounts of locks; $\mu = \mu_1$ and $\gamma = \gamma_1$

In both workloads the estimated and the calculated values align quite well; difference can only be observed for large values of L. These results verify that the model performs well for calculating the probability of clashing.

It must be noted that the scheduler of the operating system has not been included in the model. In practice it affects the overall performance of the system as if one thread holding a lock is paused by the scheduler all threads waiting for that particular lock are delayed too. The consideration of these effects, however, are beyond the scope of this paper.

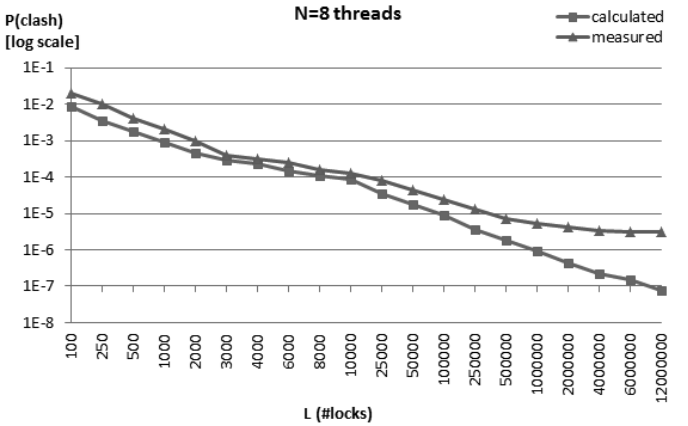

Figure 2. Measured and calculated probability of threads having to wait for a lock with various amounts of locks; $\mu = \mu_1$ and $\gamma = \gamma_1/5$

4. Waiting time for the locks

Having the probability of a thread needs to wait we would like to characterize the time spent without active work. We consider parallelism to be useful when threads perform active work; otherwise fewer number of threads would attribute to the same throughput.

4.1. Choosing lock granularity

When two threads are racing for the same shared resource the second thread attributes to better performance as long as there is some action it can perform while the first one is in the critical section protecting the shared resource. But if the time the second thread needs to wait for the lock is comparable to the time the two threads act in parallel there is no overall performance enhancement. Let us consider N parallel threads in this scenario. Whenever a new thread is added to the system it generates performance increase if the cumulated waiting of all the treads is less then the useful time of a single thread: $Nw < \frac{1}{\gamma} + \frac{1}{\mu}$.

Since the maximum number of threads that can execute in parallel without diminishing each others performance is determined by the system architecture, we want to choose the number of locks so that the threads are used to their full potential. Based on the previously presented heuristic argument let us consider a new lock to be useful when the increase in the value $\mathbb{P}\left(w < \left(\frac{1}{\gamma} + \frac{1}{\mu}\right)/N\right)$ is significant.

Because of spin waiting (and inherently the randomized service in the queuing model) it is not easy to calculate the cumulative distribution function

L	$\gamma = \gamma_1$	$\gamma = \gamma_1/5$
1	0,0000060	0,1564200
2	0,0001200	0,7654200
10	0,6352400	0,9677400
25	0,8647800	0,9869000
50	0,9331400	0,9936200
100	0,9659600	0,9971200
250	0,9864200	0,9985000
500	0,9934400	0,9992600
1000	0,9968600	0,9997200
2000	0,9982000	0,9998000
3000	0,9989000	0,9997400
4000	0,9990600	0,9999400
6000	0,9994600	0,9999600
8000	0,9994200	0,9999800
10000	0,9997200	0,9999400
25000	0,9999400	0,9999600

Table 1. The probability $\mathbb{P}\left(w < \left(\frac{1}{\gamma} + \frac{1}{\mu}\right)/N\right)$ for different parameter values

(CDF) of the waiting time. According to our knowledge, there is no closed form result for this question. The simplest description of the CDF we know is given in [1]. Flatto gave the CDF in terms of an integral, but that formula proved to be highly unstable for numerical evaluation. Instead of using the analytical expression, we chose to estimate the CDF empirically using simulations. $M/M/1/\infty/\infty/SIRO$ queues were simulated with the appropriate parameters. We simulated each setting for 50000 arrivals starting with an empty queue. From the estimated CDFs we compiled Table 1 containing the $\mathbb{P}\left(w < \left(\frac{1}{\gamma} + \frac{1}{\mu}\right)/N\right)$ values for different values of L and γ.

We choose the threshold probability level to be 0.95 and the number of locks at which this value of the CDF is reached is considered a good enough choice for the number of locks. It can be seen from Table 1 that in case of γ_1 this number is somewhere between 50 and 100, whereas for the second workload type γ_2, the number of locks is around 10.

4.2. Performance with various numbers of locks

Let us examine the actual performance (measured in terms of execution time) of the hash table. The number of locks varied between 1 (meaning a global table-level lock) and 12 million (one bucket per lock). Exploring the whole range helps us with viewing the number of locks and the attainable performance gain in context. See Figures 3 and 4 for the results.

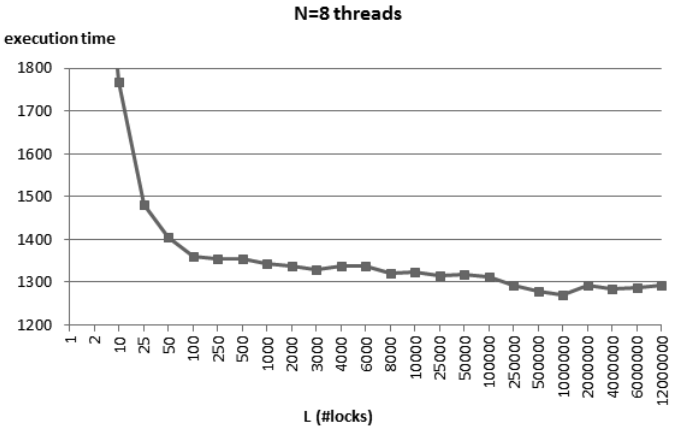

Figure 3. The execution time of completing 16 million insert/lookup operations in the hash table with 8 threads and various amounts of locks; $\mu = \mu_1$ and $\gamma = \gamma_1$.

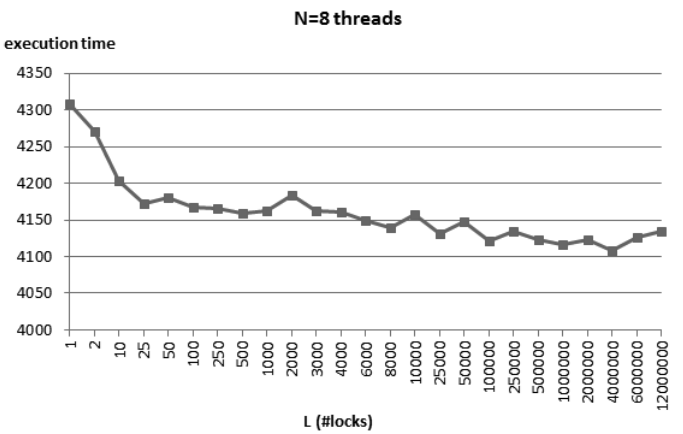

Figure 4. The execution time of completing 16 million insert/lookup operations in the hash table with 8 threads and various amounts of locks; $\mu = \mu_1$ and $\gamma = \gamma_1/5$.

In both cases there is a steady decline of execution time (meaning better performance) with the increase of the number of locks until about a 100 is reached. Further increasing the number of blocks provides minor improvement (about 1 to 4%). These results confirm that our proposed heuristics for estimating the number of locks required to provide good level of parallelism in the hash table is valid under real-life circumstances.

5. Conclusions

In this paper we analyzed the behavior of multi-thread capable bucket hash tables. Our goal was to determine the appropriate number of critical sections in the data structure. Using a global table-level lock has unfavorable consequences while using bucket level locks is undesired due to vast amount of locks that would be required.

Partitioning the buckets of the table into disjoint blocks and using a lock for each of these blocks is a good compromise, but leaves the question of choosing the block size open. We presented a queuing model that describes the behavior of the locks and the threads in the parallel hash table.

We proposed a heuristic argument which describes when is fine-grained locking beneficial for a data structure with respect to the bottleneck that comes with the mutual exclusion. This argument and the queuing model was used to estimate the order of magnitude of locks that provides reasonable performance at the lowest cost.

Both the stochastic model and the final results were verified by implementing the bucket hash table and examining its performance under real-life circumstances. When having as many threads as the environment can concurrently execute without performance penalty we presented that the most of the increase in the performance with the introduction of additional locks happens before and up to reaching about 100-500 locks; which is smaller than the number of buckets by three to four orders of magnitude.

References

[1] **Flatto, L.,** The waiting time distribution for the random order service M/M/1 queue, in *The Annals of Applied Probability*, **7(2)** 1997, 382–409.

[2] **Greenwald, M.,** *Non-blocking Synchronization and System Design*, Phd thesis, Stanford University, Aug. 1999.

[3] **Herlihy, M.,** Wait-free synchronization, in *ACM Transactions on Programming Languages and Systems*, **13(1)** (Jan. 1991), 124–149.

[4] **Herlihy, M.P. and J.M. Wing,** Linearizability: a correctness condition for concurrent objects, *ACM Transactions on Programming Languages and Systems*, **12(3)** (July 1990), 463–492.

[5] **Juhász, S. and Á. Dudás,** Optimising large hash tables for lookup performance, in: *Proceeding of the IADIS International Conference Informatics 2008*, Amsterdam, The Netherlands, 2008, pp. 107–114.

[6] **Klots, B. and R.J. Bamford,** Method and apparatus for dynamic lock granularity escalation and de-escalation in a computer system, US Patent 6144983, 1998.

[7] **Knuth, D.E.,** *The Art of Computer Programming, Vol 3*, Addison-Wesley, Nov. 1973.

[8] **Lanin, V. and D. Shasha,** Concurrent set manipulation without locking, in: *Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, New York, Mar. 1988, ACM Press, pp. 211–220.

[9] **Larson, P.-A., M.R. Krishnan and G.V. Reilly,** Scaleable hash table for shared-memory multiprocessor system, US Patent 6578131B1, Apr. 2003.

[10] **Lea, D.,** Hash table util.concurrent.ConcurrentHashMap, revision 1.3, in JSR-166, the proposed Java Concurrency Package, 2003.

[11] **Michael, M.M.,** High performance dynamic lock-free hash tables and list-based sets, in: *ACM Symposium on Parallel Algorithms and Architectures* (2002), pp. 73–82.

[12] **Noll, L.C.,** Fowler/Noll/Vo (FNV) Hash, `http://www.isthe.com/chongo/tech/comp/fnv/`

Á. Dudás, S. Kolumbán and S. Juhász
Budapest University of Technology and Economics
Department of Automation and Applied Informatics
H-1117 Budapest, Magyar Tudósok krt. 2 QB207
Hungary
`akos.dudas@aut.bme.hu`
`kolumban.sandor@aut.bme.hu`
`juhasz.sandor@aut.bme.hu`