Toggle: Contention-Aware Task Scheduler for Concurrent Hierarchical Operations

Saurabh Kalikar(✉) and Rupesh Nasre

CSE, IIT Madras, Chennai, India
{saurabhk,rupesh}@cse.iitm.ac.in

Abstract. Rooted hierarchies are efficiently operated on using hierarchical tasks. Effective synchronization for hierarchies therefore demands hierarchical locks. State-of-the-art approaches for hierarchical locking are unaware of how tasks are scheduled. We propose a lock-contention aware task scheduler which considers the locking request while assigning tasks to threads. We present the design and implementation of Toggle, which exploits nested intervals and work-stealing to maximize throughput. Using widely used STMBench7 benchmark, a real-world XML hierarchy, and a state-of-the-art hierarchical locking protocol, we illustrate that Toggle considerably improves the overall application throughput.

1 Introduction

Managing concurrent data structures efficiently is challenging as well as error-prone. Due to *irregular* memory accesses, the access pattern cannot be precisely captured at compile time. Such a data-driven behavior necessitates runtime thread-coordination. For synchronizing across threads, logical locks continue to be prevalent for concurrent data structures.

We work with hierarchies; a hierarchy is a rooted directed graph wherein nodes at a level control or contain all the reachable nodes at the levels below. Such hierarchies are quite useful in modeling relations such as manager-employee. Our motivation arises from the use of hierarchies to store data in relational databases. For instance, Oracle *database* is composed of several *tablespaces*, each of which contains several *datafiles*. Each datafile, in turn, may host several *tables*, and each table may contain multiple *rows* where data is stored [12]. Similarly, Sybase database uses the hierarchy of database, extents, tables, datapages and rows [14]. The hierarchy is layered according to the *containment* property. Thus, a table is completely contained into an extent. Concurrent updates to part of the database requires thread synchronization. For instance, for answering *range queries* [11], the concurrency mechanism in the database server acquires locks on multiple rows. Two transactions accessing overlapping ranges (e.g., rows 10..20 and rows 15..25) may be allowed concurrent execution if their accesses are *compatible* (both are reads, for instance).

Existing hierarchical locking protocols such as *intention locks* [3] exploit this containment property for detection of conflicting lock-requests.

© Springer Nature Switzerland AG 2019

R. Yahyapour (Ed.): Euro-Par 2019, LNCS 11725, pp. 142–155, 2019.
https://doi.org/10.1007/978-3-030-29400-7_11

However, due to the requirement of multiple traversals (from root to the nodes being locked), intention locks are expensive for non-tree structures (such as DAGs). The state-of-the-art approaches [6,8] improve upon intention locks by making use of *interval numbering*. These approaches assign numeric intervals to nodes while respecting their hierarchical placement. The intervals can be quickly consulted for locking requests, leading to fast conflict-detection and improved throughput.

Unfortunately, the state-of-the-art approaches do not coordinate with the task scheduler. Thus, the task scheduler does not know how threads process hierarchical lock requests, and the threads do not know how various tasks (involving lock requests) get assigned to them by the scheduler. Such a lack of coordination forces the task scheduler to use uniform schemes across threads – for instance, using round-robin scheduling for assigning tasks to threads. While round-robin mechanism is fair, it is oblivious to locking requests and works well only when the requests are spread uniformly across the hierarchy. In practice, however, some parts of the hierarchy are more frequently accessed while several parts witness infrequent accesses. This gives rise to skewed access pattern, which also evolves over time (that is, different data records get more frequently accessed over time). To cater to these changes, it is crucial to marry lock management with task scheduling. Thus, the worker threads can provide information on how the locking requests are spread in the hierarchy, and how loaded each thread is. On the other hand, the task scheduler can distribute tasks to threads based on the feedback received from the worker threads – to improve overall throughput.

Making the lock manager talk to the task scheduler is challenging and involves several design changes to a traditional scheduler. Our primary contribution in this work is to motivate the need for those design decisions and to illustrate how those can be efficiently implemented. In particular,

- We propose Toggle, a novel hashing-based task-scheduling policy for hierarchies. Built upon the interval numbering, such a policy allows the task scheduler to quickly assign a task to a thread.
- We design and implement a communication protocol for threads using a lightweight concurrent data structure.
- We illustrate the effectiveness of our proposal by incorporating the design into STMBench7. Our proposal improves the overall throughput by 22%.

2 Background and Motivation

Hierarchy is a special linked data-structure where each child node exhibits a containment relationship with its parents. For instance, in a hierarchy of employees within an organization, an edge from a project-manager to its team-member indicates a containment relationship. Hierarchical structures are often operated on using hierarchical operations which work on sub-hierarchies. For instance, an operation "bulk updates to *a particular department* in an organization" accesses the sub-hierarchy rooted at the department. Traditional fine-grained locking necessitates locking each node in this sub-hierarchy that gets accessed by the

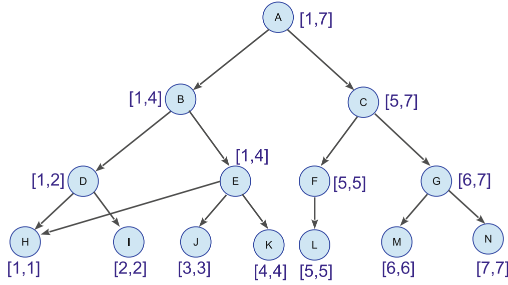


Fig. 1. Sample hierarchy. Numbers at nodes represent subsuming intervals.

bulk operation. As the hierarchy size grows, fine-grained locking is prohibitive from efficiency perspective. Therefore, hierarchical or multi-granularity locking (MGL) have been proposed. MGL at a node semantically allows the whole sub-hierarchy rooted at that node to be locked. In the extreme case, locking the root of the hierarchy locks the whole hierarchy. For instance, Fig. 1 represents a sample hierarchy wherein locking the node E locks nodes E, H, J and K. MGL is an efficient way to ensure race-free access for hierarchical operations. Various approaches towards implementing multiple-granularity locking have been proposed in the literature [2,3,6,8].

Existing MGL techniques lack coordination between thread scheduling and lock management. For instance, consider two threads operating on nodes B and D in Fig. 1. Due to MGL protocol, they both semantically try to lock the common node H. If the accesses are non-compatible (e.g., both are writes) then only one thread should be able to succeed in acquiring the lock, while the other must wait. As an aftereffect, the two operations get executed sequentially. If this is known to the scheduler, it may decide to assign the two operations to the same thread, reducing the contention and improving the throughput. In absence of such a knowledge, the scheduler may make suboptimal decisions while assigning tasks to threads. In fact, as we illustrate in our experimental evaluation, round-robin scheduling, which is fair and achieves good load balance, does not perform very well for hierarchical operations. This is because the load balance is achieved with respect to the *number of tasks* executed by a thread, rather than the *amount of work* done, which is dependent on both the sub-hierarchy size as well as the amount of contention.

3 Toggle: Contention-Aware Scheduler

In this section, we describe the design of our task scheduler. A *task*, denoted as $X : (Op, L)$ which consists of a set of operations (Op) to be performed atomically on the shared hierarchy and a set of lock objects L to be acquired at the beginning of the task. The execution of tasks follows standard 2-phase locking protocol in which all the required locks are acquired at the beginning and released at the end of the operation. Every hierarchical lock object $l_i \in L$ can be represented

by its reachable leaf level locks. For instance, in Fig. 1, node B represents a set of nodes H, I, J and K. We define a set of leaf level locks for set L as,

$$Leaf(L) = \bigcup_{l_i \in L} \{x \mid x \textit{is leaf node and reachable from } l_i \}$$

Any pair of tasks say, $X_1 : (Op_1, L_1)$ and $X_2 : (Op_2, L_2)$, are classified into two types: *independent* tasks and *conflicting* tasks. Two tasks are independent iff $Leaf(L_1) \cap Leaf(L_2) = \phi$, i.e., they do not access any common node in the hierarchy; otherwise they conflict. Scheduling two conflicting tasks to different threads may degrade performance due to the overhead of lock contention.

3.1 Representing Hierarchy as a System of Nested Intervals

Traversing lock hierarchy to compute the set of leaf level locks using reachability information is costly. To avoid such traversals, the hierarchies are pre-processed to compute and store the reachability information. One of the techniques for encoding reachability is using nested intervals where each node in the hierarchy keeps track of their leaf level descendents as an interval [6,7]. For example, in Fig. 1, nodes H, I, J, K are reachable from B. Initially each leaf node is assigned with a unique interval range H: [1, 1], I: [2, 2] and so on. Each internal node maintains its interval range such that the range subsumes the interval range of every reachable leaf node (e.g. D: [1, 2]). We build upon these intervals to compute leaf lock set L for quick classification and scheduling of tasks.

3.2 Concurrent Data Structure for Task Classification and Scheduling

A global task pool maintains a list of pending tasks to be executed. A worker thread picks one task at a time from the global task pool and executes it on the physical core. In absence of a proper scheduler, the allocation of a task to a thread happens arbitrarily depending on the order in which worker threads extract the next available tasks. Therefore, multiple threads can get blocked simultaneously based on the locking requests.

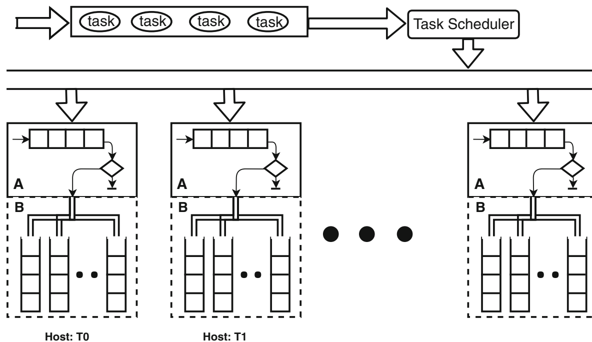


Fig. 2. Scheduler design: A and B represent outer and inner buckets resp.

Our proposed scheduler assigns a task to appropriate worker thread to avoid lock contention. Figure 2 shows the data structure used for task organization. The data structure contains a list of hash buckets. The size of the list is equal to the number of worker threads in the system. Every leaf node in the hierarchy is hashed to a specific bucket in the list. Consider there are 100 leaf node in the hierarchy indexed from 0 to 99 (during pre-processing) and there are 5 hash buckets in the list. Therefore, each bucket represents a range of 20 hierarchical nodes ([0–19], [20–39], ...). The scheduler, after extracting a task from the common task pool, analyzes the task by its set of lock objects. The scheduler inserts the task into one of the buckets as follows: if all the leaf level lock objects of the task fall into the range of a single bucket then insert the task in that bucket. Otherwise (request spans multiple buckets), insert the task into the bucket having the smallest bucket id. For instance, the hierarchy in Fig. 1 has 7 leaf nodes and say there are 2 hash buckets in the list, i.e., bucket 0 with range [0, 3] and bucket 1 with range [4, 6]. A task with lock set containing node D falls into bucket 0 as the leaf level locks H [1, 1] and I [2, 2], both fall into the range of bucket 0. On the other hand, if a locking request contains nodes J and G, then the leaf locks span both the buckets. In this case, the task gets inserted into the bucket with the least bucket id, i.e., bucket 0 (to ensure deterministic behavior). In the case of a task spanning multiple buckets, we maintain a bit mask of size equal to the total number of buckets with a bit set for each overlapping bucket. As the probability of two tasks within a bucket being conflicting is high, we assign one host thread to each bucket and mark it as primary responsible thread for executing the tasks. This imposes a sequential ordering among the tasks within a bucket which helps us minimizing the unnecessary lock contention.

3.3 Modified Locking Policy

Tasks are spread across buckets, and each bucket is served by a thread. The invariant the scheduler tries to maintain is that tasks in different buckets are independent. This invariant can allow a thread to operate on the nodes within its bucket without taking locks, since the conflicting requests are now spaced temporally. Unfortunately, this is violated by the locking requests that span buckets. While executing a multi-bucket task, the worker thread acquires locks (using bitmask) on all the overlapping buckets. This ensures safety of the concurrent operation across buckets.

3.4 Nested Bucketing

The approach discussed so far works well when the incoming tasks are uniformly distributed across the hierarchy, wherein each worker thread receives almost the same amount of work. However, for skewed distribution, the tasks are non-uniformly distributed across buckets, leaving some threads idle. In an extreme case, all the tasks may get scheduled to a single thread leading to completely sequential execution. Two tasks scheduled to a single bucket are not always conflicting and they can be run in parallel. To mitigate such effects, Toggle

Algorithm 1. Toggle Protocol

```
   Input: Bucket B
1  count ← 0
2  while there are tasks to execute do
3      if B.OuterBucket is not empty then
4          Task t ← ExtractTaskFromOuterBucket()
5          if t spans single inner bucket then
6              call InnerBucket.insert(t)
7              if InnerBucket.size() ≥ Threshold then
8                  call SwitchToInnerBucket()
9          else
               /* Task spans multiple inner buckets */
10             call Execute(t)
11             count ← count + 1
12             if count ≥ DelayThreshold then
13                 call SwitchToInnerBucket()
14                 count ← 0
15     else
16         if InnerBucket is not empty then call SwitchToInnerBucket()
17         else call StealRemoteTask()
18
```

divides each bucket into two sub-parts: outer bucket A and inner bucket B, as shown in Fig. 2. Algorithm 1 presents the detailed protocol to be followed by each worker thread. While executing the tasks from the outer part of bucket, the host thread checks whether the task can be moved into one of the inner buckets (according to the hash range of inner buckets – line 12). If yes, the host thread removes the task from the outer bucket, and schedules its execution from the inner bucket. After every few cycles of tasks (line 7), the host toggles the mode of execution from outer to inner buckets. An invariant strictly enforced is that the inner bucket tasks are single-bucket tasks. Therefore, two tasks from two inner-buckets never overlap. Once the host thread changes its execution mode to inner bucket (line 8, 13, 16), it also allows other non-host threads to steal tasks from the inner bucket. Note that, no two threads execute tasks from the same inner bucket. This achieves good load balancing in the case of skewed distributions, and no thread remains idle for long.

3.5 Task Stealing

Worker threads can execute tasks from the inner buckets of remote threads if the worker thread is idle. In Algorithm 1, each worker thread starts stealing (line 17) if both outer and inner buckets are empty. While stealing, the worker thread iterates over remote buckets and checks whether any of the host threads

are operating on inner bucket. If yes, it coordinates with the host thread to finish its task from inner bucket by picking one inner bucket at a time. However, while stealing, we need a proper thread synchronization for checking the status of the host thread and picking inner buckets.

3.6 Thread Communication

Every host thread maintains a *flag* indicating one of the execution states, namely, outer or inner. The worker, by default, executes tasks from the outer bucket. According to the protocol mentioned in Algorithm 1, whenever the host changes its state from the outer-bucket to the inner-bucket by reseting the flag, it broadcasts the message to all the threads. Any remote thread trying to steal the work, has to wait till the host thread toggles its state to inner bucket and is disallowed from stealing from the inner bucket when the host thread is operating on the outer bucket. Maintaining this protocol ensures that there are no conflicts between the remote thread and the host thread. Once the first condition of the toggle state is satisfied, remote threads are allowed to pick any one of the inner buckets and start executing the tasks one-by-one. However, as multiple threads steal in parallel, Toggle enforces (using atomic instructions) that one bucket gets picked by maximum one thread.

Algorithm 2 presents our lock-free implementation of communication mechanism and the task execution at the inner buckets. Variables `ToggleFlag` and `Atomic_Counter` are associated with each hash bucket, and are shared across threads. Host thread initializes these variables while switching the execution mode (lines 2, 3). All the remote threads check the state of these variables at the entry point and enter only when the state of the host thread has changed to inner and there is at least one inner bucket available for stealing (line 8). Instead of using heavyweight locks, we use relatively lightweight memory-fence instructions and atomic increment instructions at lines 4, 5, 7. The calls to `memory_fence(release)` and `memory_fence(acquire)` ensure that the values of the atomic counter and the ToggleFlag are written/read directly to/from the main memory (and not from the caches with stale data). Remote as well as host threads atomically and repetitively pick one inner bucket at a time (by incrementing the Atomic_Counter) till all the inner buckets become empty. Finally, the host thread resets the shared variables and returns to the execution of the outer bucket.

4 Experimental Evaluation

All our experiments are carried out on an Intel Xeon E5-2650 v2 machine with 32 cores clocked at 2.6 GHz having 100 GB RAM running CentOS 6.5 and 2.6.32-431 kernel. We implement our scheduler as part of STMBench7 [4], a widely used benchmark for the evaluation of synchronization mechanisms for hierarchical structures. STMBench7 represents an application operating on a complex real-world hierarchy consisting of various types of hierarchical nodes, such as *modules→assemblies→complex assemblies→composite-parts* and finally *atomic parts*

Algorithm 2. Thread communication in Toggle

```
   Input: Bucket B, Thread T
1  if T == B.HostThreadId then
2  |  B.Atomic_Counter ← 0
3  |  B.ToggleFlag ← inner
4  |  MyInnerBucketId ← atomic_increment(Atomic_Counter)
5  |  memory_fence(release)
6  else
7  |  memory_fence(acquire)
8  |  if B.ToggleFlag == outer OR B.Atomic_Counter > MaxInnerBucketId
   |  then
9  |  |  return
10 |  MyInnerBucketId ← atomic_increment(B.Atomic_Counter)
11 while MyInnerBucketId ≤ MaxInnerBucketId do
12 |  while MyInnerBucket is not empty do
13 |  |  Task t ← ExtractTask()
14 |  |  call Execute(t)
15 |  MyInnerBucketId ← atomic_increment(B.Atomic_Counter)
16 if T == B.HostThreadId then
17 |  wait till all remote threads finish their respective tasks
18 |  B.ToggleFlag ← outer
19 |  memory_fence(release)
```

at the leaf level. The operations accessing different parts of the hierarchy are defined to evaluate various synchronization aspects. We evaluate Toggle against four different locking mechanisms: two from STMBench7 (namely, coarse-grained and medium-grained), and two MGL-based techniques: DomLock [6] and NumLock [8].

We also use a real-world hierarchical dataset, *tree-bank* [15], (available in XML format) having 2.4 million nodes. For stress-testing our approach, we use a synthetically generated hierarchy having k-ary tree structure of 1 million nodes to study the effect of various parameters. A command-line argument *skewness* controls the probability of an operation accessing certain parts of the hierarchy.

4.1 STMBench7

Figure 3(a) shows the overall throughput achieved on STMBench7 by different locking mechanisms. In STMBench7, each parallel thread iteratively picks one operation at a time from a common pool. This mimics a fair round-robin task scheduling policy. Our scheduler operates on top of the worker threads and assigns tasks at runtime. In Fig. 3(a), x-axis shows the number of parallel threads and y-axis is the throughput (tasks/second) across all threads. We primarily compare with DomLock and NumLock, and extend them with our scheduler. Across all threads, Toggle achieves maximum throughput compared

to coarse-grained, medium-grained, DomLock, and NumLock. This is primarily due to reduced lock contention. NumLock is an improved version of DomLock with optimal locking choices. We plug Toggle with NumLock; Toggle improves overall throughput with NumLock as well. We observe that the STMBench7 operations access certain part of hierarchy frequently. Every operation accessing such a subset of nodes gets sequentialized irrespective of the thread performing the operation. The scheduler dynamically detects such sequential bottleneck among the tasks and assigns them to only few threads letting other threads remain idle. In our experiments, the maximum throughput is achieved with the configuration of only four parallel threads. Overall, Toggle achieves 44%, 22% and 10% more throughput compared to coarse-grained, medium-grained and DomLock respectively.

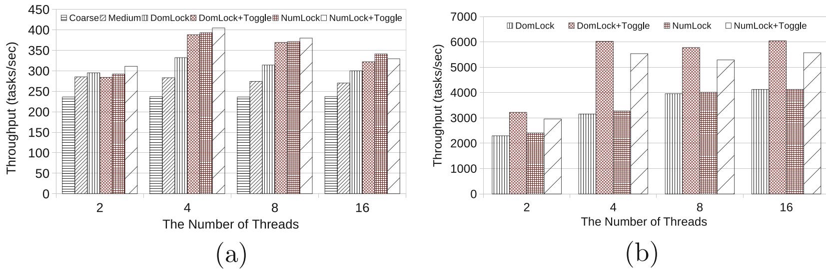


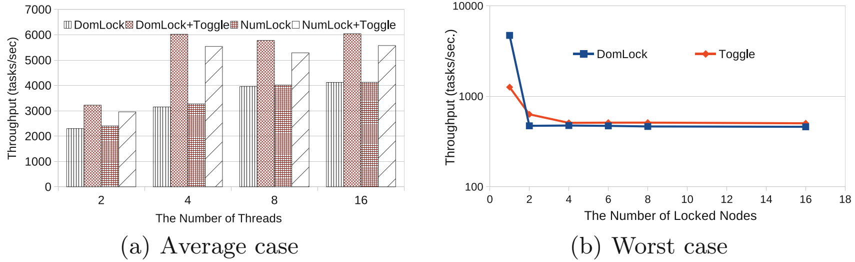
Fig. 3. (a) Comparison with STMBench7 (b) Effect on XML hierarchy

4.2 XML Hierarchy

We also evaluate our technique on *tree-bank*, a real-world hierarchical dataset available in XML format [15]. The hierarchy represents a non-uniform structure consisting of 2,437,666 nodes with maximum depth of 36 and average depth of 7.87 levels. We perform parallel tasks on the hierarchy where tasks access hierarchical nodes with equal probability. The tasks are diverse in nature: Some tasks exhibit skewed access whereas others are equally distributed. Tasks also vary with respect to the size of critical sections, i.e., some tasks spend less time in critical sections representing short transactions and while some are long transactions. We maintain the ratio of read:write operations as 7:3. Figure 3(b) shows the throughput gain of our scheduler with a fair round-robin scheduler. We observe that, our scheduler yields 16% more throughput and scales well with the number of threads whereas round-robin fails to scale due to lock-contention.

4.3 Effect of Skewness

We use our stress-test implementation to evaluate the effect of various parameters on the performance of Toggle. Figure 4 shows the overall throughput with 16 parallel threads with different levels of skewness. The x-axis shows the skewness index; *skewness = 1* indicates uniform random distribution. As we increase the



(a) Average case (b) Worst case

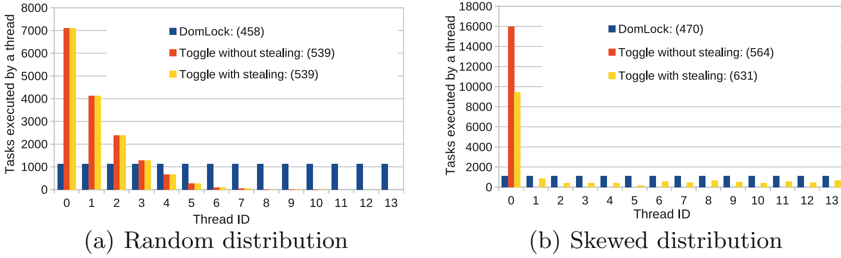
Fig. 4. Throughput improvement with Toggle

skewness index on x-axis, the access pattern becomes more localized to certain part of the hierarchy. For instance, skewness = 2 indicates two disjoint subsets of hierarchy nodes and any task can access nodes within only one subset. Figure 4(a) shows that the maximum throughput improvement is obtained with the most skewed and localized access pattern. Toggle achieves more throughput than round-robin scheduling because the conflicting tasks gets assigned to particular buckets according to the hash ranges of buckets.

It is crucial to handle the worst-case scenario for the task scheduling. As Toggle assigns tasks according to the bucket range and the host thread executes them sequentially, it is possible that every task falls into a single bucket leaving all other buckets empty. Note that, the tasks from one bucket may not conflict and may exhibit concurrency. We evaluate this scenario by forcing tasks to access multiple nodes from one particular range. In Fig. 4(b), x-axis shows the number of nodes a task is operating on and y-axis shows throughput (log-scale) for round-robin and Toggle. As we keep on increasing the value on x-axis, the probability of conflicts becomes very high. The throughput obtained using Toggle is consistently better than round-robin because of two reasons. First, even though the tasks get assigned to single bucket, the host thread allows other threads to steal tasks from the inner-buckets, utilizing available parallelism. Second, individual threads do not execute conflicting tasks, therefore they do not introduce extra lock contention. However, for tasks accessing only one node, (i.e., at x-axis value = 1), every task is guaranteed to be independent except the two tasks accessing exactly the same node. In this case, even though Toggle assigns all the tasks to a single thread and internally allows other threads to steal tasks, it fails to achieve better throughput because of synchronization cost involved in stealing. This is the only case where round-robin scheduling performs better than Toggle (although round-robin is also suboptimal for this scenario). Figure 4 shows the throughput with DomLock. The results with NumLock are omitted from Fig. 4(b) as both plots coincide due to the logscale.

4.4 Effect of Task Stealing

In this section, we compare the effect of scheduling in terms of resource utilization. As we discussed in the previous section, round-robin and Toggle achieve



(a) Random distribution (b) Skewed distribution

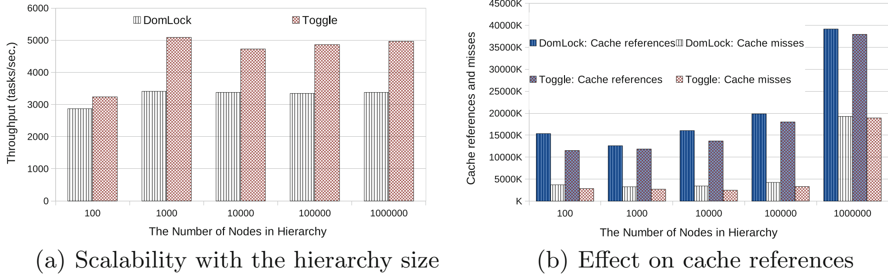
Fig. 5. Effect of task-stealing

similar throughput for random distribution. Figure 5(a) shows the work-load distribution across worker threads (threadIDs 0..13) for round-robin, Toggle without and with task stealing. Out of total 16 threads, remaining two threads are reserved: threadID 14 is a task-generator and threadID 15 is Toggle task-scheduler. x-axis represents different thread IDs and the y-axis represents the number of tasks executed by each thread. The obtained throughput with each scheduling policy is shown along with the legends. We observe that, in round-robin scheduling, every thread execute nearly equal number of tasks. However, for Toggle, threads with smaller IDs execute more tasks than the other. In fact, half of the threads remain idle. Despite this, the obtained throughput is better than round-robin. Note that the work-load distribution gradually decreases with the higher thread IDs. This happens because the tasks accessing nodes from a random distribution generally span multiple buckets, but our scheduler assigns such multi-bucket tasks to smaller thread IDs. Moreover, there is almost no scope for task stealing in the case of random distribution, as Toggle with and without stealing executes a similar number of tasks and achieves equal throughput.

Unlike this, for a skewed distribution, task stealing plays important role in performance improvement. Figure 5(b) shows tasks distribution across threads for the configuration of the worst-case scenario with every task accessing exactly 2 nodes. In absence of task stealing, each task gets assigned to a single bucket and each of them is executed sequentially, still achieving better throughput than round-robin scheduling. However, permitting other threads to steal tasks from remote buckets, we achieve further improvement in the overall throughput (shown with the legends). This shows the importance of task stealing in skewed distributions, which is prevalent in real-world scenarios. Our scheduler dynamically enables and disables task stealing based on the run-time load distribution.

4.5 Effect of Hierarchy Size

Figure 6(a) shows the throughput against different hierarchy sizes, from 100 to 1 million nodes. As we see, parallel operations on a smaller hierarchy are likely to get blocked. Toggle achieves throughput gain even in high-contention scenario. As we increase the hierarchy size, Toggle consistently outperforms DomLock.



(a) Scalability with the hierarchy size (b) Effect on cache references

Fig. 6. Effect of hierarchy size on throughput and cache misses

We also see the benefit of scheduling in terms of caching. A worker thread primarily executes operations within its bucket. Tasks within a bucket are mostly conflicting, i.e., access same nodes, therefore it avoids extra cache misses as the data would be available in processors private cache. As we increase the hierarchy size, the bucket ranges become wider. The tasks in such a wider bucket are less likely to be conflicting, therefore for every task, accessing new nodes leads to cache miss. In Fig. 6(b), this effect is visible for large sized hierarchies.

5 Related Work

Hierarchical locking is imposed in many ways. Traditionally, in database context, hierarchical locking is implemented as multiple granularity locks using intention locks [3] at ancestor nodes. However, in multi-threads applications, threads get contended while acquiring these intention locks. DomLock [6] solves this problem by dominator locking using nested intervals. NumLock [8] further improves by generating optimal hierarchical locking combination. Our task scheduler can be used with any of these hierarchical locking protocols. Similar to the idea of interval locking, key-range locking [9,10] is in databases locks a range of records satisfying certain predicates. In this case, key-range lock guards not only the keys present in the databases but also any phantom insertions. However, it is not a hierarchical locking technique. Although Toggle works with hierarchical locking, it can be tuned to work with key-range locking as well.

Task scheduling has been the topic of interest in many domains. Hugo Rito et al. proposed ProPS [13], a fine-grained pessimistic scheduling policy for STMs. ProPS also used STMBench7 for evaluating different STM implementations. We use it for evaluating Toggle over locking techniques. Several techniques have been proposed for scheduling *task DAGs* where nodes represent tasks and directed edges are precedence relation. Zhao et al. [5] proposed a policy for scheduling multiple task DAGs on heterogeneous systems. Cui et al. [1] present a lock-contention-aware scheduler at the kernel level. However, none of the schedulers addresses the challenges with lock-contention for hierarchical locks.

6 Conclusion

We presented Toggle, a contention-aware task scheduler for hierarchies. It coordinates with the lock manager for scheduling tasks to maximize throughput, using nested bucketing and work-stealing. Using large hierarchies and STMBench7 benchmarks, we illustrated the effectiveness of Toggle, which considerably improves the average throughput over DomLock.

7 Data Availability Statement and Acknowledgments

We thank all the reviewers whose comments improved the quality of the paper substantially. The artifacts of the work have been successfully evaluated and are available at: https://doi.org/10.6084/m9.figshare.8496464.

References

1. Cui, Y., Wang, Y., Chen, Y., Shi, Y.: Lock-contention-aware scheduler: a scalable and energy-efficient method for addressing scalability collapse on multicore systems. ACM Trans. Archit. Code Optim. **9**(4), 44:1–44:25 (2013). https://doi.org/10.1145/2400682.2400703
2. Ganesh, K., Kalikar, S., Nasre, R.: Multi-granularity locking in hierarchies with synergistic hierarchical and fine-grained locks. In: Aldinucci, M., Padovani, L., Torquati, M. (eds.) Euro-Par 2018. LNCS, vol. 11014, pp. 546–559. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96983-1_39
3. Gray, J.N., Lorie, R.A., Putzolu, G.R.: Granularity of locks in a shared data base. In: VLDB. pp. 428–451. ACM, New York (1975)
4. Guerraoui, R., Kapalka, M., Vitek, J.: STMBench7: a benchmark for software transactional memory. In: Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, pp. 315–324. EuroSys 2007, ACM, New York (2007). https://doi.org/10.1145/1272996.1273029
5. Zhao, H., Sakellariou, R.: Scheduling multiple dags onto heterogeneous systems. In: Proceedings 20th IEEE International Parallel Distributed Processing Symposium, p. 14. April 2006
6. Kalikar, S., Nasre, R.: Domlock: a new multi-granularity locking technique for hierarchies. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 23:1–23:12. PPoPP 2016, ACM, New York (2016). https://doi.org/10.1145/2851141.2851164
7. Kalikar, S., Nasre, R.: DomLock: a new multi-granularity locking technique for hierarchies. ACM Trans. Parallel Comput. **4**(2), 7:1–7:29 (2017)
8. Kalikar, S., Nasre, R.: NumLock: towards optimal multi-granularity locking in hierarchies. In: Proceedings of the 47th International Conference on Parallel Processing, pp. 75:1–75:10. ICPP 2018, ACM, New York (2018). https://doi.org/10.1145/3225058.3225141
9. Lomet, D., Mokbel, M.F.: Locking key ranges with unbundled transaction services. Proc. VLDB Endow. **2**(1), 265–276 (2009)
10. Lomet, D.B.: Key range locking strategies for improved concurrency. In: Proceedings of the 19th International Conference on Very Large Data Bases, pp. 655–664. VLDB 1993, Morgan Kaufmann Publishers Inc., San Francisco (1993)

11. MSDN: Sql server 2016 database engine (2015). https://msdn.microsoft.com/en-us/library/ms187875.aspx
12. Oracle: Oracle database 10g r2 (2015). http://docs.oracle.com/cd/B19306_01/index.htm
13. Rito, H., Cachopo, J.: ProPS: a progressively pessimistic scheduler for software transactional memory. In: Silva, F., Dutra, I., Santos Costa, V. (eds.) Euro-Par 2014. LNCS, vol. 8632, pp. 150–161. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09873-9_13
14. Sybase: Adaptive server enterprise: Performance tuning and locking (2003). http://infocenter.sybase.com/help/topic/com.sybase.help.ase_12.5.1/title.htm
15. Treebank: Xml data repository (2002). http://aiweb.cs.washington.edu/research/projects/xmltk/xmldata/www/repository.html