Compas'2023 : Parallélisme / Architecture/ Système
LISTIC/USMB - Annecy France, 4-7 juillet 2023

CALock : Multi-Granularity Locking in Directed Graphs

Ayush Pandey[1], Swan Dubois[1], Marc Shapiro[1,2], Julien Sopena[1]

[1] Sorbonne Université, CNRS, LIP6, F-75005 Paris, France [2] Inria

Résumé
With the increasing demand for parallel processing and shared data structures in modern computer systems, the need for effective synchronization mechanisms has become increasingly important. Concurrent accesses to shared data require synchronization, and locking is a widely used strategy in this context. Multi-granularity locking approaches have been proposed to allow threads to lock the whole sub-graph concerning a query in a single lock acquisition, particularly when the data structure is hierarchical. However, existing strategies often suffer from performance limitations.
To address this challenge, this paper proposes CALock, a new labelling strategy for multi-granularity locking that exploits the topology of a rooted directed graph to identify the finest lock grain. The CALock approach is evaluated experimentally, and the results demonstrate that it offers higher concurrency and throughput than similar approaches when the underlying graph is dynamic. In particular, CALock exhibits a 200% increase in throughput compared to similar locking approaches for workloads with contended access requests.

Mots-clés : Parallel Algorithms, Locking, Dynamic Hierarchies, Graphs

1. Introduction

Context Hierarchical data structures, such as lists, trees, and graphs *, etc.* are characterized by directed edges between vertices of the structure. These edges define relationships between the vertices and help navigate the hierarchy based on query parameters.
In the presence of concurrent reads and writes, an application needs to ensure that the data remains consistent. Although weaker synchronization methods exist for specific classes of data structures [2, 10, 12, 9, 6, 13] locking is the predominant technique for synchronization[8]. The performance of any locking approach is a measure of the concurrency allowed by the lock while ensuring data consistency. For example, locking the entire data structure ensures data consistency but does not allow parallel access. In practice, locking approaches need to balance the cost of locking, maintaining data consistency, and allowing concurrency.
State-of-the-Art *multi-granularity locking* (MGL) [4] is one such approach that aims to strike a balance between maintaining data consistency and allowing efficient parallel access to shared data. By allowing threads to lock only the necessary portions of the data structure, multi-granularity locking can reduce contention and improve performance. In MGL, acquiring a lock on a vertex v (called the *lock target*) also locks several other vertices implicitly based on the locking algorithm used. The portion of a graph that is considered locked when locking a single vertex is called the *grain* of the lock. Locks with small granularity are called *fine-grained* whereas locks with large granularity are called *coarse-grain*. Coarse grain locks have a low cost of

acquisition but restrict concurrency. Fine-grain locks have a high cost of acquisition but allow high concurrency.

With granular locking, two lock requests conflict with each other if their lock grains contain at-least one common vertex and the intended operations, for which the lock is required, conflict. A measure of the size of these grains is called the *granularity* of the lock [11].

Several MGL techniques have been proposed in the literature and they fall into two broad categories based on the number of locks acquired per query.

Multi-lock MGL approaches take several locks for one query, For example, intention locks [4]. These approaches suffer from deadlocks. Applications need thus to implement effective deadlock avoidance or detection and resolution mechanisms.

Single-lock MGL approaches take a single lock for each query by exploiting the topology of the graph and acquiring a single lock at the root of the lock grain, thereby locking the subgraph at this root. These approaches can be more efficient than multi-lock approaches because they do not suffer from deadlocks. However, they may restrict concurrency depending on the topology of the graph. Some MGL techniques used in practice are explained below. *Intention lock* [4] is a multi-lock MGL technique in which, the vertices along the paths of the graph leading to the lock target are marked with intention locks. Intention locks "tag" the vertices with a shared or exclusive mode which indicates that an actual lock is being acquired at a finer granularity along this path. Other threads use intention tags to check for conflicts. While placing tags along a path, if a thread encounters a tagged vertex with a conflicting mode, then it waits until the tag is cleared or is not in conflict anymore. Commercial databases like SQLServer [1] use intention locks to optimize synchronized access to data and indices.

DomLock [7] is a single-lock MGL technique suitable for rooted directed graphs that labels the vertices with intervals of the form $[min, max]$ assigned by a post-order traversal over the graph. Figure 1 provides an example of such labelling. A leaf has a unit interval, *e.g.* vertex E has the label $[2, 2]$. The interval for an internal vertex is the minimum of the min values and the maximum of the max values from the intervals of its children. For example, vertex C has four children D, E, F, and G so, its interval is $[1, 4]$. In this situation, C subsumes the intervals of the vertices D through J. Based on the traversal order, a vertex can sometimes subsume its siblings even though they do not belong to the same grain. For example, vertex F is not a descendant of vertex G but G still subsumes F because their intervals overlap. This is called a *false subsumption*.

In DomLock, the grain for a lock request is rooted at the deepest vertex with the smallest interval that subsumes the intervals of vertices in the lock request. For example, to lock H and J, the grain has the range $[3, 4]$. G is the deepest vertex that subsumes $[3, 4]$ so a lock is requested on G, see Figure 1.

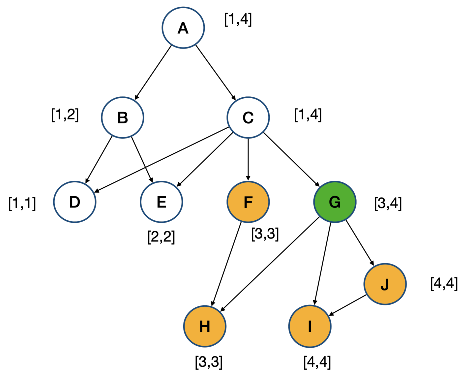


FIGURE 1 – DomLock labelling. If a lock is placed on G, all the vertices in orange are also implicitly locked.

Limitations MGL techniques have some limitations, With intention locks, the first limitation is the fact that deadlocks may occur so additional mechanisms are required to avoid or detect and resolve them. Second, intention locks may have a high latency for a lock request because of the traversals required to place intention tags along the paths. These traversals are especially expensive for graphs with a large number of edges. With DomLock, the first limitation is the fact that false subsumptions restrict concurrency. For example, in Figure 1 a lock on I and J prevents another thread from locking F even when these queries can run concurrently. Second, the cost of maintenance of the labels in the presence of structural modifications is high. Indeed, the re-computing of the intervals may be expensive since a traversal of the complete graph might be necessary.

Contribution In this paper, our contribution is a new single-lock MGL strategy that significantly improves the state of the art. This strategy, called *Common-Ancestor lock* or *CALock* for short, is based on a labelling scheme that efficiently computes the "closest" common ancestor of a set of vertices. Performance improvement *w.r.t.* DomLock comes from (i) a reduction of false subsumptions and (ii) a top-down relabelling in case of structural modifications.

Roadmap The remainder of this paper is structured as follows. Section 2 presents our labelling algorithm. Section 3 describes our locking algorithm. Section 4 summarizes our experimental evaluation of CALock using STMBench7. Finally, Section 5 concludes the paper.

2. CALock Labelling Strategy

To make understanding easier, consider a rooted DAG. Then, the main idea of CALock is the following; to protect a set of vertices, lock their *lowest common single ancestor*, or *LCSA*, which is the deepest vertex that lies on every path from the root to the vertices in the set [3]. For example, in Figure 2, lock A for a request over E and I. We can generalize this scheme to any rooted directed graph by reducing any strongly connected component to a single vertex. We explain now, a labelling function for graphs such that the labels of a set of vertices are sufficient to compute their LCSA without having to traverse the graph.

Labelling Let $G = (V, E)$ be a directed graph. V is the set of vertices of the graph connected by directed edges in the set E such that $E \subseteq V \times V$. A vertex $r \in V$ is the root of G. We label each vertex v using the following recursive function L_v :

$$L_v = \{v\} \cup \{\cap_{u \in parents(v)} L_u\} \tag{1}$$

We prove in Appendix A that the fix-point L_G for the label of a vertex in G satisfies the following properties : (i) For a vertex v, the label only contains the single ancestors of v in G. (ii) The deepest element in the label of a vertex v is the LSA of v in G. (iii) For a set of vertices S, the labels of vertices in S guarantee that the LCSA of vertices of S is the deepest vertex in $\cap_{v \in S} L_v$.

We compute L_G using algorithm 1. Processing starts at the root of the graph. The label for the root contains only its own identifier. Then, the function `BFLabel()` is called for every inner vertex of the graph. The label of an inner vertex is a set union of its own identifier and the set intersection of the labels of its parents (line 16). This recursion terminates when the label of a vertex reaches a fix-point (line 18) indicating that all paths to the vertex have been explored, even in the presence of strongly connected components.

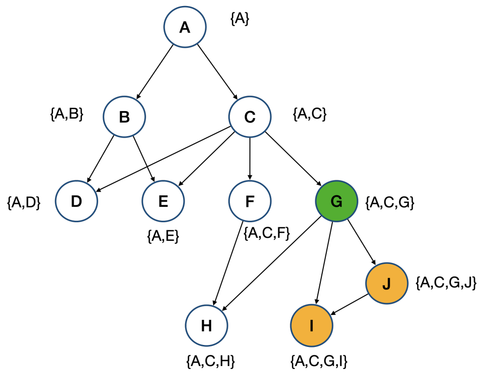


FIGURE 2 – CALock labelling. If a lock is placed on G, only the vertices in orange are also implicitly locked.

3. CALock Locking Strategy

General idea In CALock, a request that manipulates a set S of vertices, issues a lock request for the LSCA of the vertices in S. This LSCA x is computed by taking a set intersection of the labels

Algorithm 1 Procedure for assigning labelling the graph

```
 1: procedure ASSIGNLABELS(v)
 2:     queue.PUSH(v)
 3:     while queue.HASNEXT( ) do
 4:         $v \leftarrow$ queue.NEXT( )
 5:         BFLABELA(v, queue)
 6:     end while
 7: end procedure

 8: procedure BFLABELA(v, queue)
 9:     C ←CHILDREN(v)
10:     if $parents(v) = \emptyset$ then
11:         $v.label \leftarrow \{v\}$
12:         queue.PUSH( C )
13:         return
14:     end if
15:     P ←PARENTS(v)
16:     tempLabel ←INTERSECTION(P.labels)
17:     tempLabel.APPEND(v)
18:     if $v.label \neq tempLabel$ then
19:         $v.label \leftarrow tempLabel$
20:         queue.PUSH( C )
21:     end if
22: end procedure
```

of vertices in S. The LSCA is the deepest element in this set intersection. A lock is requested on x, the root of the lock grain. The lock request is checked for conflicts with currently held locks and blocked if a conflict is detected otherwise, the thread proceeds to its critical section. The lock requests, pending or granted, are added to a lock pool.

Lock pool The lock pool is an array containing one entry per thread. Initially and when the thread is not holding a lock, its entry is `NULL`. Whenever a thread requests a lock, it atomically inserts its request into the lock pool at the corresponding index. This request contains the *Thread ID*, *Grain LSCA*, *Sequence number*, *Waiting condition*, *Lock mode*, *Grain LSCA label*. Algorithm 2 shows the flow of lock acquisition in the lock pool.

The sequence number is assigned to the thread before the request is added to the pool. It is used to order the requests by their arrival order. The waiting condition, which is an atomic boolean variable, is set to `true` after the sequence number is assigned indicating that the thread is holding the lock. Another thread with conflicting request blocks and waits for this boolean value to change to `false`.

Assignment of a sequence number, setting the wait condition, and addition of the lock request to the pool is done under a mutex to prevent the race where a new lock request with a conflicting operation and grain is granted before the conflict could be detected.

After the lock request is added to the pool, the thread checks for conflicts with other locks currently held or requested. If the following conditions simultaneously hold, the thread is blocked : (i) the lock request has a mode conflict with another lock request; (ii) the lock grains of the requests overlap (that is, either the lock request is for a vertex that is part of a locked grain or the lock request is for a lock grain, a part of which is already locked); and (iii) the sequence number of the request is greater than the conflicting request. Table 1 sums up the lock compatibility.

	$rl_i(x)$	$wl_i(x)$	$rl_i(y)$	$wl_i(y)$
$rl_j(x)$	✓	✗	✓	✤
$wl_j(x)$	✗	✗	✤	✤
$rl_j(y)$	✓	✤	✓	✗
$wl_j(y)$	✤	✤	✗	✗

TABLE 1 – Lock compatibilities between shared (rl) and exclusive (wl) locks requested by threads i and j on vertices x and y.✓(compatible); ✗(incompatible); ✤ (compatible iff the grains do not overlap)

Structural modifications CALock can be utilized for static graphs whose structure does not change and for dynamic graphs that change at runtime. A structural modification request leads to a partial relabelling of the graph for the vertices whose ancestors change as a consequence of the structural modification.

To add a new vertex v, no lock is required and $L_v = \{v\}$. To delete a vertex v (resp. to add or remove an edge between vertices u and v) a lock is acquired on the LSCA of the parents of

Algorithm 2 Lock acquisition request in the lock pool

```
1:  GSeq : Global sequence number
2:  Mutex : Mutex guarding the pool
3:  procedure LOCK(lockObject, threadID)
4:      LOCK(Mutex)
5:      lockObject.Seq ← Gseq + +
6:      lockObject.condition.TESTANDSET(true)
7:      Pool[threadId] ← lockObject
8:      UNLOCK(Mutex)
9:      for all lock ∈ Pool do
10:         if lock ≠ NULL
              ∧(lockObject.HASRWCONFLICT(lock))
              ∧(lock.grainRoot ∈ lockObject.ancestors
              ∨lockObject.grainRoot ∈ lock.ancestors)
              ∧(lockObject.Seq > lock.Seq) then
11:             thread.WAIT(lock.condition)
12:         end if
13:     end for
14:     return true
15: end procedure

16: procedure UNLOCK(lock)
17:     lockPool.REMOVE(lock)
18:     lock.condition.CLEAR( )
19:     lock.condition.NOTIFY_ALL( )
20: end procedure
```

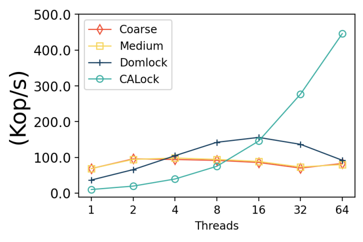
v (resp. the LSCA of u and v). Upon structural modifications, a relabelling may be necessary. Note that, by the very definition of our labels, this relabelling affects only vertices in the grain lock of the chosen LCSA.

4. Experimental evaluation

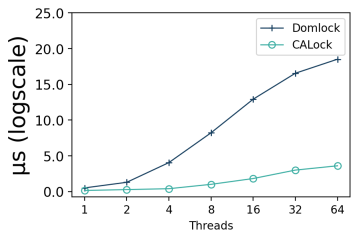
Experimental setup Our evaluation uses the same benchmark as DomLock [7]. We implemented CALock in STMBench7 [5] and ran our experiments on a machine with an AMD EPYC 7642 CPU with 48 Cores and a base clock of 2.3 GHz and 512 GB of RAM. The benchmark was deployed on a docker container with Ubuntu 20.04. To build the benchmark, GCC 12.1 and Cmake 3.22 is used. The compilation was done at the C++ 20 standard without any compiler optimization flags.

STMBench7 provides coarse-grain and medium-grain lock implementations. The coarse lock in STMBench7 is a reader-writer lock over the entire graph. Structural modification operations with coarse and medium locks take a write lock on the entire graph.

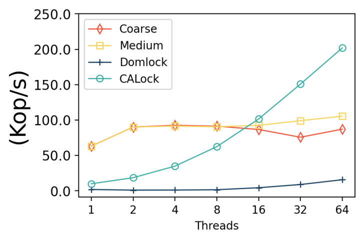
The provided implementation of DomLock uses busy-waiting. For a more fair comparison, we modified DomLock to use a condition variable, like CALock does.



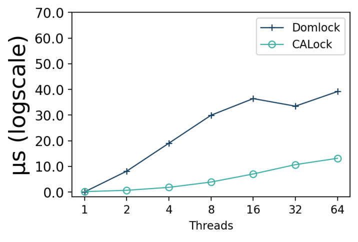
(a) R :90%,W :10%



(b) R :90%,W :10%



(c) R :90%,W :9.9%,SM :0.1%



(d) R :90%,W :9.9,SM :0.1%

FIGURE 3 – (a, c) Throughput ; (b, d) Response time

Overall performance Figure 3 shows the throughput with a read-dominated load. The charts in Figure 3 plot the number of concurrent threads on the x-axis and the throughput (op/s) or latency(µs) on the y-axis. Figure 3a shows reads and writes without structural modifications. Coarse and medium-grained locks perform better than both DomLock and CALock for up to 4 concurrent threads due to the overhead of computing lock grain in DomLock and CALock. Beyond 8 threads, both CALock and DomLock are better than coarse and medium-grained locks. CA-

Lock is 2x faster than DomLock for 32 threads and 4.5x faster than DomLock for 64 threads because with DomLock, due to false subsumptions, threads are blocked even when lock grains are disjoint.
Figures 3c show reads and writes that interleave with structural modifications. With as low as 0.1% of the operations being structural modifications, the performance of all locking strategies suffers. This is because structural modifications with coarse-grained, medium-grained locks and DomLock happen under mutexes. CALock is able to parallelize structural modifications and is 2x faster than coarse and medium-grained locks and about 8x faster than DomLock.
Response Time We measure the time spent by a thread waiting for locks. This is the time taken from when the thread issues a lock request until the lock is granted. Figure 3b shows the wait time per thread between DomLock ad CALock for static graphs. DomLock on average waits longer for a lock to be granted than CALock. The wait time increases with the number of concurrent threads because of an increase in the number of conflicts and overlapping grains. For 64 threads, CALock is 6x faster than DomLock. In dynamic graphs, DomLock waits longer as shown in figure 3d. This is because DomLock has to relabel the entire graph after every structural modification. Response times for CALock also increase but CALock still remains 4x faster than DomLock.

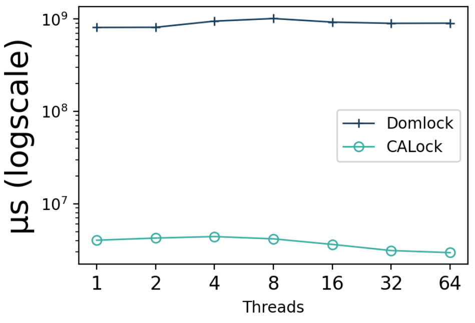


FIGURE 4 – Relabelling time per op. with 0.1% SM

Relabelling time for graphs We measure the cost of the graph following a structural modification. Figure 4 shows the average time spent relabelling the graph per structural modification for the same workload as Figure 3c. In DomLock a structural modification causes global relabelling via a post-order traversal of the graph which is expensive and blocks every other operation. With CALock however, a structural modification is followed by a relabelling of locked grain only. This allows disjoint lock grains to be processed in parallel.

5. Conclusion and Future work

In this paper, we present CALock, an improved single-lock multi-granularity locking strategy that exploits the single ancestors of vertices of a graph to identify the lock grain for a lock request. The claims are supported by the benchmarks done using STMBench7. While DomLock performs better when the graph is static and regular, CALock is better overall when the graph is irregular or dynamic. We also reduce the problem of false subsumptions that can happen with DomLock.
We plan to extend CALock in multiple dimensions. (i) To make CALock suitable for generic graphs, we need to get rid of the constraint the graph has a single root. (ii) In CALock, a thread can hold a single lock at a time; we need to loosen this constraint and allow a single thread to hold multiple locks. (iii) In the current implementation, the work set of a transaction needs to be known before taking a lock to identify the lock grain. We plan to implement a strategy that allows resizing the lock grains as the transaction progresses.

Bibliographie

1. Transaction locking and row versioning guide - sql server, Dec 2022.

2. Chatterjee (B.), Dang (N. N.) et Tsigas (P.). – Efficient lock-free binary search trees. *CoRR*, vol. abs/1404.3272, 2014.
3. Fischer (J.) et Huson (D. H.). – New common ancestor problems in trees and directed acyclic graphs. *Information Processing Letters*, vol. 110, n8-9, 2010, pp. 331–335.
4. Gray (J.), Lorie (R. A.), Putzolu (G. R.) et Traiger (I. L.). – Granularity of locks and degrees of consistency in a shared data base. – In Nijssen (G. M.) (édité par), *Modelling in Data Base Management Systems, Proceeding of the IFIP Working Conference on Modelling in Data Base Management Systems, Freudenstadt, Germany, January 5-8, 1976*, pp. 365–394. North-Holland, 1976.
5. Guerraoui (R.), Kapalka (M.) et Vitek (J.). – Stmbench7 : a benchmark for software transactional memory. – In Ferreira (P.), Gross (T. R.) et Veiga (L.) (édité par), *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23, 2007*, pp. 315–324. ACM, 2007.
6. Kaki (G.), Priya (S.), Sivaramakrishnan (K. C.) et Jagannathan (S.). – Mergeable replicated data types. *Proc. ACM Program. Lang.*, vol. 3, nOOPSLA, 2019, pp. 154 :1–154 :29.
7. Kalikar (S.) et Nasre (R.). – Domlock : A new multi-granularity locking technique for hierarchies. *ACM Transactions on Parallel Computing*, vol. 4, n2, 2017, pp. 7 :1–7 :29.
8. Kimura (H.), Graefe (G.) et Kuno (H. A.). – Efficient locking techniques for databases on modern hardware. – In Bordawekar (R.) et Lang (C. A.) (édité par), *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2012, Istanbul, Turkey, August 27, 2012*, pp. 1–12, 2012.
9. Letia (M.), Preguiça (N. M.) et Shapiro (M.). – Crdts : Consistency without concurrency control. *CoRR*, vol. abs/0907.0929, 2009.
10. Natarajan (A.), Savoie (L.) et Mittal (N.). – Concurrent wait-free red black trees. – In Higashino (T.), Katayama (Y.), Masuzawa (T.), Potop-Butucaru (M.) et Yamashita (M.) (édité par), *Stabilization, Safety, and Security of Distributed Systems - 15th International Symposium, SSS 2013, Osaka, Japan, November 13-16, 2013. Proceedings*, *Lecture Notes in Computer Science*, volume 8255, pp. 45–60. Springer, 2013.
11. Ries (D. R.) et Stonebraker (M.). – Effects of locking granularity in a database management system. *ACM Transactions on Database Systems*, vol. 2, n3, 1977, pp. 233–246.
12. Sundell (H.) et Tsigas (P.). – Fast and lock-free concurrent priority queues for multi-thread systems. *J. Parallel Distributed Comput.*, vol. 65, n5, 2005, pp. 609–627.
13. Valois (J. D.). – Lock-free linked lists using compare-and-swap. – In Anderson (J. H.) (édité par), *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, August 20-23, 1995*, pp. 214–222. ACM, 1995.

A. Definitions and Proof

Let $G = (V, E)$ be a directed graph. V is the set of vertices of the graph connected by directed edges in the set E such that $E \subseteq V \times V$. A vertex $r \in V$ is the root of G.
A pair of vertices (u, v) is connected by a sequence of edges. This sequence p is called a *path* from u to v. The set of vertices on a path p is denoted by $\mathcal{V}(p)$. The length l_p of this path is the size of $\mathcal{V}(p)$ i.e. $l_p = |\mathcal{V}(p)|$.
Since G is a graph, multiple such paths are possible between a pair of vertices. The set of all the paths between (u, v) is denoted by $\mathcal{P}_{(u,v)}$. The depth $\delta(v)$ of a vertex v is the length of the shortest path in the set $\mathcal{P}_{(r,v)}$.

A.1. Ancestors and Descendants

Definition 1 (Ancestor and Descendent). *An ancestor of a vertex u is a vertex v in G that lies on a path from r to u. The vertex u is then called the descendant of vertex v.*

A.2. Single Ancestors

Definition 2 (Single Ancestor). *A single ancestor of a vertex u is a vertex v in G that lies on all paths from r to u.*

Definition 3 (Lowest Single Ancestor). *The lowest single ancestor (LSA) of vertex u is the single ancestor v with the maximum depth.*

Definition 4 (LSA-Tree). *The LSA-tree T_G of G is a tree structure that has vertices V, and its edges are defined such that the parent vertex of $v \neq r$ is LSA(v).*

A.3. Common Ancestors

When referring to a set of vertices, we define their common ancestor.

Definition 5 (Common Ancestor). *A common ancestor (CA) of two vertices u and v is a vertex c in G that is an ancestor of both u and v.*

Definition 6 (Lowest Common Ancestor). *The lowest common ancestor (LCA) of two vertices u and v is a vertex l in G that is a common ancestor of u and v with maximum depth.*

A.4. Lowest Single Common Ancestor

The Lowest single common ancestor (LSCA) is defined using the single ancestor and common ancestor definitions. Fisher and Huson [3] derive the following relationships for the LSA and the LSCA of vertices in a DAG.

Lemma 1. *Let G be a DAG, rooted at r, and T_G its corresponding LSA-tree. Further, let $v, w \in V$ be two arbitrary vertices in G. Then* $LSCA_G(v, w) = LCA_{T_G}(v, w)$

Lemma 2. *For a vertex* $v \neq r$, $LSA_G(v) = LSCA_G(parents_G(v))$

Definition 7. $LSCA_G(w1, ..., wk) = LSCA_G(w1, LSCA_G(w2, ..., wk))$.

Definition 8. $LCA_{T_G}(u_1, u_2, ..., u_n) = LCA_{T_G}(u_1, LCA_{T_G}(u_2, ..., u_n))$.

A.5. Computing labels of a graph

Since the LSA-tree T_G is well defined for a DAG G; The label for a vertex u, denoted by L_u, is an ordered set of vertices that lie on the path from the root of T_G to u. Since T_G is a tree, the path from the root to u in T_G is unique and so is the label L_u. In order to compute the labels of a vertex u in G we take the set intersection of the labels of u's ancestors in G. To this end, we have the following theorem :

Theorem 1. $LSCA(v, w)$ *is the vertex of the maximum depth in* $L_v \cap L_w$

Démonstration. Let $l = LCA_{T_G}(v, w)$ be the Lowest common ancestor of *v, w* in the LSA tree T_G. By lemma 1, we can say that $l = LSCA_G(v, w)$.
We need to show that l is the deepest vertex in $L_v \cap L_w$.
Let's assume that there is a vertex $l' \neq l$ that lies on the paths from l to both v and w in G. Since l' lies on the paths $l \to v$ and $l \to w$, the depth of l' must be greater than the depth of l.
The labels on v and w, which are L_v and L_w respectively, contain the single ancestors of v and w. Since l' lies on the paths $l \to v$ and $l \to w$, inductively, l' also lies on all the paths from the root of the graph r to the vertices v and w.
If l' lies on these paths then $l' \in L_v$ and $l' \in L_w$. This in turn implies that $l' \in L_v \cap L_w$.
Since l' is deeper than l, it should be the deepest member of $L_v \cap L_w$. But l' cannot be the deepest element in $L_v \cap L_w$ because then, l would not be the $LCA_{T_G}(v, w)$ which means that our assumptions on l' contradict the definition $l = LCA_{T_G}(v, w)$ and l is the deepest element in the set $L_v \cap L_w$. □

A.6. Labelling scheme for graphs without strongly connected components

We now use Theorem 1 and use the definitions and lemmas from Section A.4 to derive a recursive function that can be used to label a graph. Lemma 2 can be rewritten, using definition 7, as follows :

$$LSA_G(v) = LSCA_G(p_1, LSCA_G(p_2, ..., p_k)) \text{ where } p_1, p_2, ...p_k \text{ are the parents of } v \tag{2}$$

Definition 8 can be rewritten using Theorem 1 as follows :

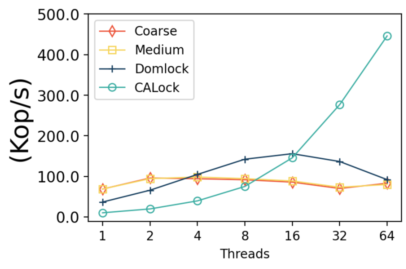
$$LCA_{T_G}(u_1, ..., u_n) = LSCA_G(u_1, ..., u_n) \text{ is the deepest element in } L_{u_1} \cap ... \cap L_{u_n} \tag{3}$$

Combining equations 2 and 3, we can say that $LSA_G(v)$ is the deepest element in $L_{p_1} \cap L_{p_2} \cap ... \cap L_{p_k}$ where $p_1, p_2, ...p_k$ are the parents of v. Therefore, the set of single ancestors of v is enough to compute the lowest single common ancestor of a set of nodes. The recursive function that we use on the vertices of the graph is :

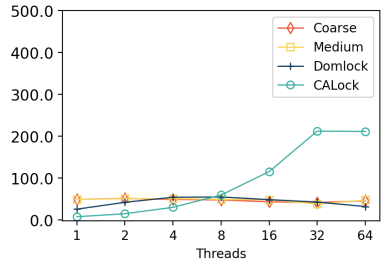
$$L_v = \{v\} \cup \{\cap_{u \in parents(v)} L_u\} \tag{4}$$

B. Detailed Benchmarks

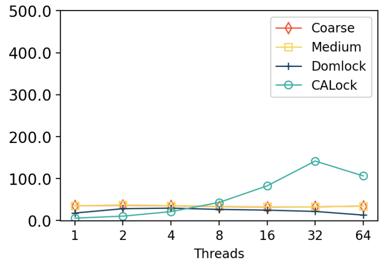
B.1. Throughput



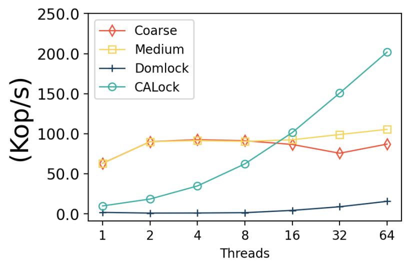
(a) R :90%,W :10%



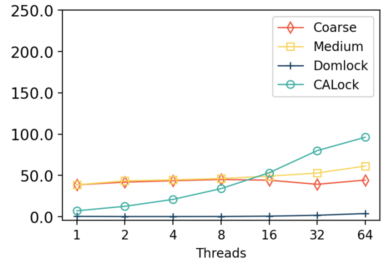
(b) R :60%,W :40%



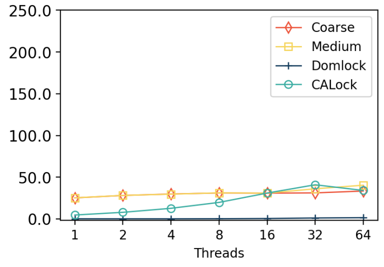
(c) R :10%,W :90%



(d) R :90%,W :9.9%,SM :0.1%



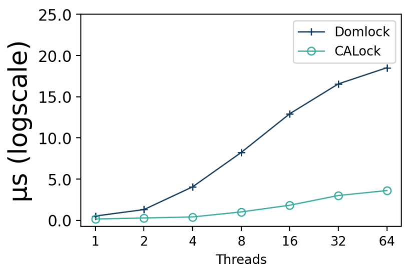
(e) R :60%,W :39.6%,SM :0.4%



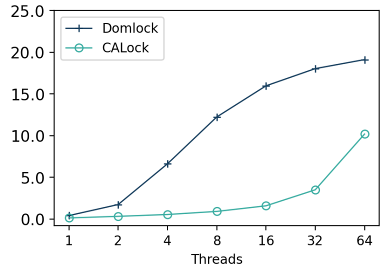
(f) R :10%,W :89.1%,SM :0.9%

FIGURE 5 – Performance with different workload types (higher is better)

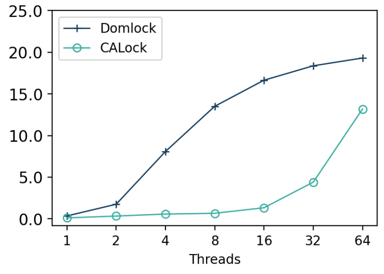
B.2. Latency



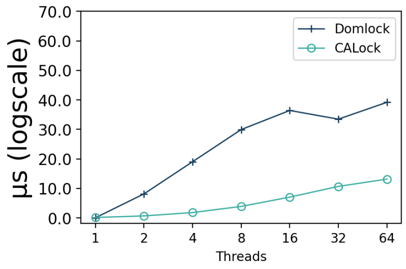
(a) R :90%,W :10%



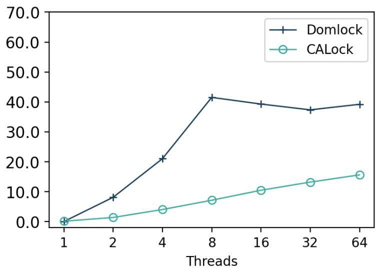
(b) R :60%,W :40%



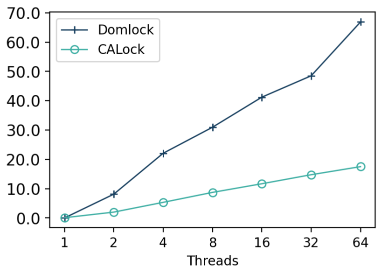
(c) R :10%,W :90%,SM :0%



(d) R :90%,W :9.9,SM :0.1%



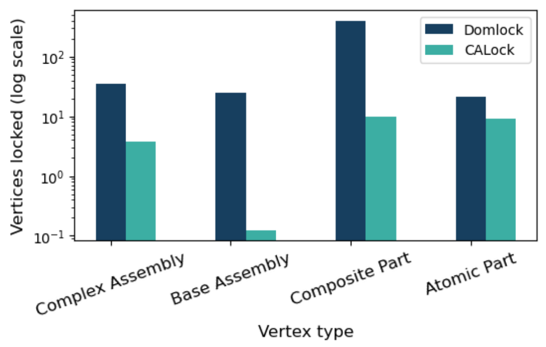
(e) R :60%,W :39.6%,SM :0.4%



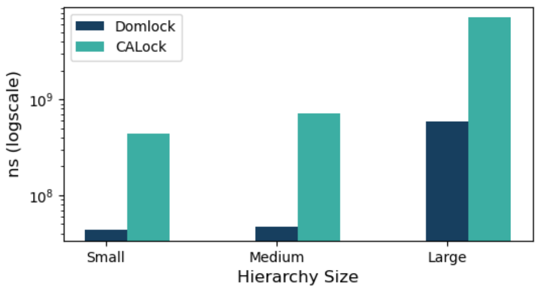
(f) R :10%,W :89.1%,SM :0.9%

FIGURE 6 – Wait time for locks (lower is better)

B.3. Labelling and Granularity



(a) Granularity



(b) Time to compute labels

C. STMBench Hierarchy structure

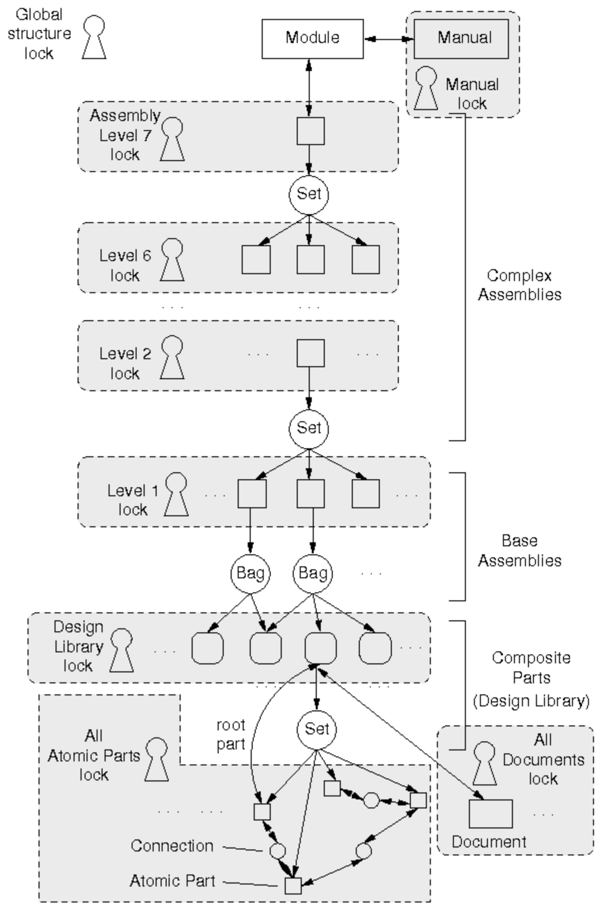


FIGURE 8 – Structure of a module in STMBench with lock boundaries