Thèse présentée pour l'obtention du grade de

DOCTEUR de SORBONNE UNIVERSITÉ

Spécialité

Ingénierie / Systèmes Informatiques

École doctorale

Informatique, Télécommunication et Électronique Paris (ED130)

CALock : Topological Multi-Granularity Locking for Hierarchical data

---

Ayush Pandey

Soutenue publiquement le : *21 Mars 2025*

Devant un jury composé de :

Jean-Michel DUPONT, Professeur, Sorbonne Université	*Rapporteur*
Alice DUPONT, Directrice de Recherche, CNRS	*Examinatrice*
Jean-Michel DUPONT, Professeur, Sorbonne Université	*Examinateur*
Pierre AAA, Maître de conférences, UPMC	*Examinateur*
Mesaac MAKPANGOU, Chargé de Recherche [HDR], INRIA	*Directeur de thèse*
Marc SHAPIRO, Directeur de Recherche Émérite, INRIA, Sorbonne Université, LIP6	*Directeur de thèse*
Julien SOPENA, Maître de Conférences, Sorbonne Université, LIP6	*Encadrant*
Swan DUBOIS, Maître de Conférences, Sorbonne Université, LIP6	*Encadrant*

Abstract

Hierarchies serve as fundamental structures across various disciplines, modeling hierarchical relationships in computer science, biology, social networks, and logistics. However, the dynamic and concurrent updates in real-world systems necessitate synchronization techniques for maintaining data consistency despite concurrent access. This paper explores a novel approach called CALock to synchronize operations on hierarchies by utilizing a labeling scheme that facilitates multi-granularity locking.

Our approach addresses both concurrent data reads and writes as well as structural modifications. CALock exploits the hierarchical topology via a new labeling scheme to identify the common ancestors of vertices. This enables a thread to identify an appropriate lock granule for its lock request. Leveraging variable lock granularities optimizes operations across the hierarchy while ensuring consistency and performance.

We provide a detailed discussion of the CALock labeling and the locking algorithm, prove its properties, and evaluate it experimentally. On static hierarchies CALock remains competitive with previous labeling schemes and has better concurrency and throughput when structural modifications change the hierarchy. In particular, CALock improves throughput by 4.5×, and response time by 1.5× for workloads that contain structural modifications.

Keywords: Multi-granularity locking, Hierarchical data, Graphs, Locking, Synchronization, Graph topology, Ancestors.

Résumé

Les hiérarchies servent de structures fondamentales dans diverses disciplines, modélisant les relations hiérarchiques en informatique, en biologie, dans les réseaux sociaux et en logistique. Cependant, les mises à jour dynamiques et simultanées dans les systèmes du monde réel nécessitent des techniques de synchronisation pour maintenir la cohérence des données malgré l'accès simultané. Cet article explore une nouvelle approche appelée CALock pour synchroniser les opérations sur les hiérarchies en utilisant un schéma d'étiquetage qui facilite le verrouillage multi-granularité.

Notre approche concerne à la fois les lectures et écritures concurrentes de données et les modifications structurelles. CALock exploite la topologie hiérarchique par le biais d'un nouveau schéma d'étiquetage permettant d'identifier les ancêtres communs des sommets. Cela permet à un thread d'identifier un granule de verrouillage approprié pour sa demande de verrouillage. L'utilisation de granularités de verrouillage variables optimise les opérations à travers la hiérarchie tout en garantissant la cohérence et les performances.

Nous présentons une discussion détaillée de l'étiquetage CALock et de l'algorithme de verrouillage, nous prouvons ses propriétés et nous l'évaluons de manière expérimentale. Sur des hiérarchies statiques, CALock reste compétitif par rapport aux schémas d'étiquetage précédents et offre une meilleure simultanéité et un meilleur débit lorsque des modifications structurelles changent la hiérarchie. En particulier, CALock améliore le débit de 4,5$\times$ et le temps de réponse de 1,5$\times$ pour les charges de travail qui contiennent des modifications structurelles.

Mots-clés: Verrouillage multi-granularité, Données hiérarchiques, Graphes, Verrouillage, Synchronisation, Topologie des graphes, Ancêtres.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

In the realm of database systems, ensuring consistency and integrity during concurrent access to data is a critical challenge. As systems grow in scale and complexity, the management of data concurrency becomes increasingly significant, particularly in environments where multiple transactions access shared data simultaneously. A key approach to addressing this challenge is the use of locking mechanisms to control access to data, with multi-granularity locking (MGL) being one such technique. In traditional database systems, the concept of multi-granularity locking has been well-researched for table spaces, particularly in relation to tree-structured data. However, as data models evolve to include complex hierarchies and highly connected structures such as graphs and networks, a more nuanced understanding of MGL is required to ensure efficient and scalable concurrency control.

Hierarchical data models are pervasive across various domains, ranging from file systems and organizational charts to ontologies and XML databases. Such models inherently capture relationships between entities, where the data is organized at different levels of abstraction. In these settings, multiple users or processes may attempt to access different parts of the hierarchy concurrently, leading to potential conflicts and the risk of data inconsistency. At the heart of this issue lies the challenge of efficiently managing locks across different levels of the hierarchy, while minimizing overhead and ensuring that transactions can proceed in parallel whenever possible.

Multi-granularity locking addresses this challenge by allowing locks to be set at different levels of granularity—ranging from coarse-grained locks at higher levels of the hierarchy to fine-grained locks at lower levels. This approach provides flexibility, enabling transactions to lock larger portions of the data when broad access is required, or smaller portions when fine-grained control is needed. However, implementing multi-granularity locking in systems with complex hierarchies introduces significant challenges, especially when balancing performance with correctness and fairness. The performance of the system is contingent on minimizing lock contention and deadlocks while ensuring that locks at different granularities can coexist without leading to unnecessary delays or resource wastage

1.1 Problem Statement

While multi-granularity locking is widely applied in simple hierarchical data models, its adaptation to more complex, irregular and deeply nested hierarchies remains under-explored. Traditional approaches often fail to scale efficiently in scenarios where hierarchies grow in size and depth, or where the hierarchical relationships become less rigidly defined, such as in graph like data models. Additionally, the interaction between locks at different levels of the hierarchy can lead to intricate synchronization issues, particularly in highly parallel and distributed environments where latency and partitioning are significant factors. Therefore, there is a need for new approaches that extend existing multi-granularity locking frameworks to support these complex hierarchical structures, while maintaining high levels of performance and ensuring data consistency.

1.2 Contributions

In this work, we present CALock, a novel multi-granularity locking protocol designed to address the challenges of managing concurrency in complex hierarchical data models. CALock extends the traditional multi-granularity locking framework to support hierarchies with arbitrary levels of depth and complexity, enabling efficient and scalable concurrency control. We use a topological partial ordering of the vertices of a hierarchy to define the lock levels and grains, allowing transactions to acquire locks at different levels of granularity based on their access patterns. This topological ordering is determined via a labelling scheme that assigns unique labels to each vertex in the hierarchy which are used at runtime to determine the appropriate lock grain for a lock request.

Existing MGL techniques like intention locks [Gra+75] use the topology of the hierarchy to detect lock conflicts but require multiple traversals from the root of the graph to the vertices being locked. This makes them expensive for non-tree-like hierarchies like DAGs. Label based techniques [KN16; AN22; KN19; AN24] use an ordering of vertices to assign labels that eliminate the need for traversals. However these have poor performance over dynamic graphs due to the overhead of relabelling.

CALock's labelling efficiently computes the closest common ancestor of a set of vertices in a rooted directed graph. By choosing the closest common ancestor of a set of lock targets as their guard, CALock minimizes grain size. In using paths,

by avoiding a fixed ordering of vertices, CALock circumvents expensive relabelling while providing the same locking guarantees as other MGL techniques. The label of a vertex in the graph is a set of its common ancestors, computed recursively via breadth-first traversal.

1.3 Thesis Structure

The remainder of this thesis is organized as follows:

Chapter 2 presents an overview of concurrency control, multi-granularity locking and the challenges associated with complex hierarchical data models.

Chapter 3 reviews existing research in the field of multi-granularity locking, highlighting the limitations of current techniques.

Chapter 4 provides a theoretical framework for CALock which includes the concept of lowest common ancestors in graphs, and the problem formulation for optimal locking grain selection and the proof of optimality of CALock labelling.

Chapter 5 introduces CALock, our novel multi-granularity locking protocol, and describes its design and implementation. We also discuss the properties of CALock like safety, liveness and fairness.

Chapter 7 presents an experimental evaluation of CALock, comparing its performance against state-of-the-art techniques in a variety of scenarios using both, micro-benchmarks and macro-benchmarks.

Finally, Chapter 8 concludes the thesis, summarizing our contributions and outlining future research directions in the field of multi-granularity locking for complex hierarchical data models.

2 Background

2.1 Concurrency Control

Concurrency control is a critical aspect of systems where multiple processes or threads access shared resources simultaneously, such as data structures, database tables, system resources etc. In general, concurrency control mechanisms utilize some form of synchronization to ensure safety and consistency of shared resources. According to Raynal [Ray13], synchronization is a set of rules and mechanisms that allows the specification and implementation of sequencing properties on statements issued by the processes so that all the executions of a multi-process program are correct.

The form of synchronization studied in our work is *competition synchronization* where multiple threads compete for access to shared resources. Locking is one of the primary mechanisms used to ensure data integrity and consistency in such environments. For data-structures like lists, trees or graphs, locking involves restricting access to a data structure by multiple threads or processes to prevent conflicts and ensure that only one thread can modify the data at a time. Here's a high-level overview of how locking works and its role in concurrency control:

2.1.1 Reader-Writer Locks

Reader-writer locks, also called as shared-exclusive locks, are a type of lock that allows multiple threads to read a resource simultaneously but restrict write access to one thread at a time. This is useful when read operations are more frequent than write operations since a multiple readers can access the resource concurrently.

```
1    ReaderWriterLock rwlock;
2    int counter = 0;
3
4    void reader(int iterations) {
5        for (int i = 0; i < iterations; ++i) {
6            rwlock.reader_lock();
7            // Read operation (e.g., print counter)
8            std::cout << "Read counter: " << counter << std::endl;
```

```
            rwlock.reader_unlock();
        }
    }

    void writer(int iterations) {
        for (int i = 0; i < iterations; ++i) {
            rwlock.writer_lock();
            ++counter;
            rwlock.writer_unlock();
        }
    }
```

Read-write locks are more complex than simpler mutex locks but can improve performance in scenarios where read operations are more frequent than write operations. However, they can introduce additional complexity and potential issues like writer starvation where writers are blocked indefinitely by readers in the absence of proper fairness mechanisms.

2.1.2 Role of Locking in Concurrency Control

Preventing Race Conditions Race conditions occur when multiple threads access and modify shared data concurrently, leading to unpredictable results. Locks ensure that only one thread can modify the data at a time, preventing race conditions.

Ensuring Data Consistency Locks help maintain data consistency by ensuring that operations on shared data are atomic (indivisible). This means that a series of operations can be completed without interruption, ensuring the data remains in a consistent state.

Deadlock Prevention Proper use of locks can help prevent deadlocks, where two or more threads are waiting indefinitely for each other to release locks. Techniques like lock ordering and timeout mechanisms can be used to avoid deadlocks.

2.2 Hierarchical data structures

Hierarchical data models, rooted in the early days of computer science, have played a critical role in the representation and storage of structured information. The origins of hierarchical data can be traced back to the 1960s with the development of

the Information Management System (IMS) by IBM [Bla98], which was one of the earliest database management systems designed for hierarchical data organization. IMS was created to manage the complex manufacturing data for the Apollo space program, establishing a clear, parent-child relationship between data entities, a key feature of hierarchical models [Dat00]. Over the decades, hierarchical data structures have remained relevant, particularly in areas requiring nested or tree-like relationships, such as XML data management, file system hierarchies, and organizational charts [ABS99]. The importance of hierarchical models has been magnified in modern big data contexts, where they are often combined with other models (like graph databases) to handle the challenges posed by the complexity and interconnectedness of data. For example, hierarchical data models have been instrumental in NoSQL databases like Apache HBase, where efficient storage and quick retrieval of large-scale, semi-structured data are essential [Geo11]. The structured nature of hierarchical models is well-suited for applications such as social networks, content management, and cloud-based storage systems, where data relationships are crucial. Thus, hierarchical data remains foundational to modern computing, providing a clear and efficient way to organize and retrieve complex, nested information.

Formally, a hierarchy is defined as follows:

Definition 1. *A hierarchy is a directed graph H=(V,E) where*

- *V is a finite set of vertices*
- *$E \subset V \times V$ is a set of directed edges where each edge (u,v) represents a parent-child relationship between vertices u (the parent) and v (the child).*

2.3 Locking in Hierarchical Data Structures

Locking in hierarchical data structures is a crucial mechanism for ensuring data consistency and integrity, especially in concurrent environments where multiple processes or threads may attempt to access or modify the data simultaneously. Hierarchical data structures often require specialized locking techniques to efficiently manage concurrent access due to their inherently nested and dependent relationships between nodes. The goal of locking in these structures is to prevent race conditions, where two or more operations interfere with each other, potentially leading to incorrect data state.

2.3.1 Classical locking approaches in hierarchical data

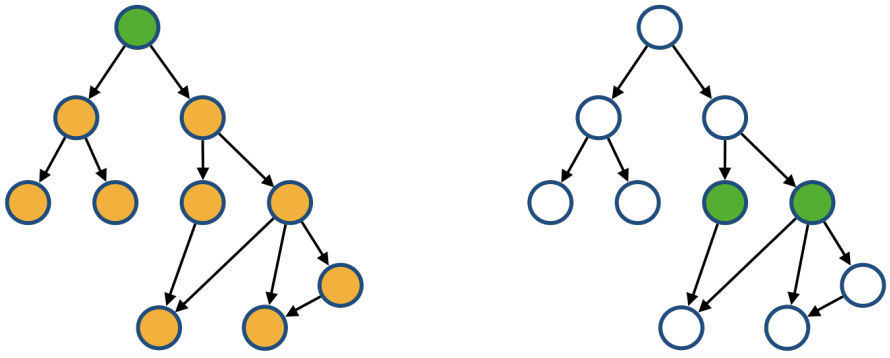
A common approach is lock coupling [BS77], also known as hand-over-hand locking, where a thread locks a vertex before traversing to one of its children in the hierarchy, unlocking the parent once the child is locked. This allows for fine-grained synchronization, which improves concurrency by permitting multiple threads to work on different parts of the hierarchy simultaneously. However, Leis et al. [LHN19] show that lock coupling can lead to suboptimal locking performance on modern hardware.

An alternative to lock coupling is a B-Tree variant called B-Link tree [LY81] in which, a thread holds a single lock at time with optimistic reachability checks. However, for in-memory workloads, B-Link trees suffer from the same performance issues as lock coupling.

Classical locking techniques for hierarchical data structures are often designed with a synchronization hypothesis. Their design limits their use in general purpose applications where the topology of the hierarchy is not known a priori.

2.3.2 Fixed Grain locking

As an alternative of the classical approaches, reader-writer locks are appropriated for hierarchical data by associating a lock with a predefined granularity. We refer to this as *fixed grain locking*. Fixed grain locking has two extremes: We can either have a single lock for the entire hierarchy (coarse grain locking) or a lock for each vertex (fine grain locking).



(a) Coarse grain lock in a hierarchy. **(b)** Fine grain locks in a hierarchy.

Fig. 2.1: Fixed grain locking in hierarchical data structures (lock guard in green and the corresponding grain in yellow).

Coarse Grain Locking An oversimplified approach to correctness is to guard the root of the hierarchy such that any thread accessing the hierarchy must first acquire the root lock. This approach is called *coarse grain locking* and is shown in Figure 2.1a. A thread that wishes to access any vertex in the hierarchy must first acquire the root lock and consequently block all other writers until it releases this lock. Coarse grain locking is simple to implement and has extremely low locking overhead but suffers from the highest possible contention as all threads must acquire the same lock to access any part of the hierarchy.

Fine Grain Locking Another well known approach to locking is when every target vertex is its own guard i.e. every vertex has its own lock. We call this *fine grain locking*. As shown in Figure 2.1b, the vertices are locked individually and a thread that needs to access multiple vertices must acquire multiple locks. Fine grain locking has the advantage of reducing contention as threads can access disjoint parts of the hierarchy concurrently. However, the overhead of acquiring multiple locks makes this approach less efficient.

2.3.3 Multi-Granularity Locking

An alternative approach to locking in hierarchical data structures is multiple granularity locking, which adapts the granularity of locks based on the structure of the hierarchy. According to Gray et al. [Gra+75], hierarchical or multi-granularity locking involves locking a vertex in a hierarchy such that the lock is sufficient to protect the vertex and its descendants i.e. a thread that requests a write (exclusive) lock on a vertex implicitly exclusively locks all its descendants when its request is granted.

A vertex which a thread intends to access is called the *lock target* of the thread. MGL methods generally associate a lock with each vertex in the hierarchy which we call the *guard* of the vertex. As such, when a thread wishes to access a target, it must acquire an appropriate lock on the guard of the the target vertex.

The approaches shown in Figures 2.1b and 2.1a have their own trade-offs. Coarse grain locking causes unnecessary contention by blocking threads that access disjoint parts of the hierarchy. Fine grain locking, on the other hand, introduces significant overhead due to the need to acquire multiple locks for accessing multiple vertices with the additional requirement of deadlock detection and prevention.

Multi-granularity locking (MGL) [Gra+75] techniques find a balance between the two extremes by identifying a theoretically optimal lock guard for a given target based on the topology of the hierarchy. Since grain size (the *granularity*) depends on the topology of the graph and on the lock request, different lock requests have different granularities, hence the name.

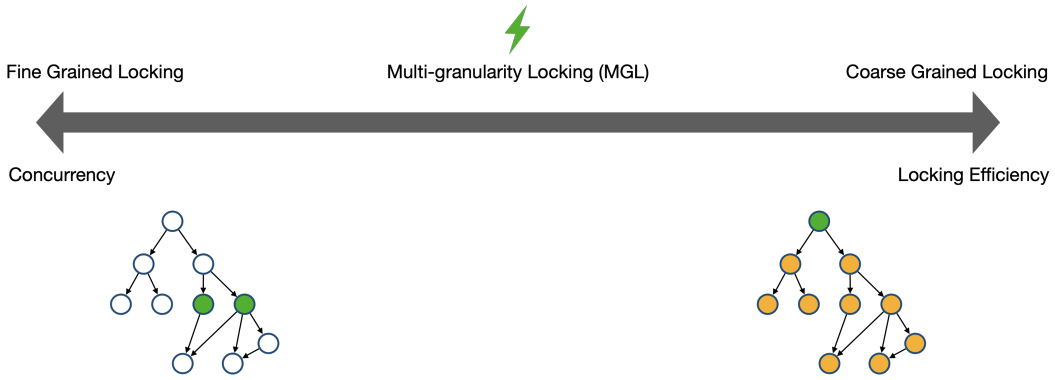


Fig. 2.2: Multi-Granularity locking provides a balance between Fine grain and Coarse grain locking.

A classical example of MGL is *Intention locks* [RS77] which are often used in database indices to optimize hierarchical access [Sql].

Another dimension of complexity with hierarchical data is the mutation of the hierarchy itself which causes topological changes. We refer to to such mutations as *structural modifications*.

Although there already exist effective MGL algorithms for graphs that do not undergo structural modifications, this is not the case for graphs whose structure can mutate. We discuss this in Chapter 3. A structural modification operation adds or removes vertices and/or edges and may conflict with ordinary operations, or with another structural modification. Such operations affect the identification of lock granules as their size and shape now depend on the new topology of the graph.

2.4 Context

This work on locking focusses on a multi-core system where threads concurrently access a shared graph. Such use-cases, prevalent as graph processing applications, provide the possibility to use a single centralized scheduler which decides on the ordering of the lock requests. In distributed deployments like HPC clusters where the communication can be guaranteed to be quasi-instantaneous, locking can still be

useful. Geo-distributed graphs and synchronization are out of the scope of this work. As such, locking in any form is not an efficient solution for use-cases where the graph is geo-distributed and the communication latency (ergo the synchronization delay) is high.

The requirement for any locking technique over graphs is to protect the vertices and edges of the graph against concurrent write access. Writes can modify the data on the vertices or add/remove edges from the graph. A application that uses CALock can have multiple threads that concurrently access the graph. Each thread can request a lock on a set of vertices(lock targets). The locking algorithm then grants the lock request by locking a single vertex(the guard), that protects the set of vertices. Once a thread identifies a guard and issues a lock request, the guard cannot to be changed i.e. the guard is fixed for the duration of the lock. This is to ensure that the lock grain remains consistent for the duration of the lock. If a conflict is detected between two lock requests, one of the threads is blocked until it is notified to resume i.e. no busy waiting.

In our design, we assume that the threads are not malicious and do not try to circumvent the locking and conflict detection mechanism

3 Related Work

Multi-granularity locking is a well-known solution to the problem of hierarchical locking in database systems. With MGL, locking a vertex in a hierarchy in a specific mode implicitly locks all its descendants in the same mode. This is a powerful concept that allows for efficient locking of hierarchical data structures without the need to lock large portions of the hierarchy. While powerful, implementing an MGL protocol that successfully achieves this goal is non-trivial. The primary challenge is an efficient computation of the ancestor-descendant relationships in the hierarchy. Along with this, there are other requirements that contribute to the performance trade-offs of the locking protocol.

These requirements can be broadly classified into 4 categories:

R1 Identifying a lock guard for every vertex of a hierarchy.

R2 Finding an appropriate, optimal lock guard for a request.

R3 Detecting conflicts between locks i.e. testing ancestor-descendant relationship.

R4 Housekeeping the metadata required to implement the locking protocol.

The different locking approaches discussed in this chapter use different mechanisms to address these requirements with each approach having its own set of trade-offs. It would be appropriate to claim that no approach is all encompassing and the choice of the locking approach depends on the specific requirements of the application and the hierarchy being used. However, a few locking protocols are more generally suited over others.

3.1 Fixed grain locks

Fixed grain locks are the simplest form of locking for uses where the hierarchy is known a priori and the semantic classification of the vertices is fixed. Fixed grain locking associates a fixed guard for a set of vertices in the hierarchy. The two extremes of locking discussed in Chapter 2 are variants of fixed grain locking. Another approach to fixed grain locking is to associate a lock guard per level of the

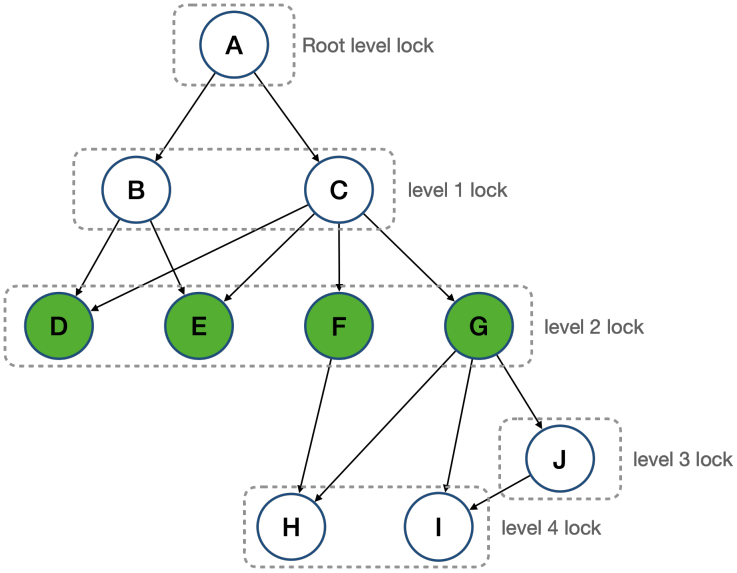


Fig. 3.1: Hierarchical locks with fixed grains per level and a lock on level 2

hierarchy. As shown in Figure 3.1, the hierarchy is divided into levels and a lock guard is associated with each level. In this approach, locking a vertex locks all the other vertices in the same level. For example, in Figure 3.1, locking vertex D locks vertices E, F and G as well since they are in level 2.

Fixed grain locks only fulfil requirement **R1** since the lock guard is fixed for a set of vertices. However, the lock guard is not optimal for any request that does not contain all vertices in a level. For example, consider two threads T_1 and T_2 that want to lock vertices D and E respectively. Since the lock guard for level 2 is the same, T_1 and T_2 will conflict with each other. However, D and E are not related by an ancestor-descendant relationship and hence should not conflict.

In using fixed grain locks, the lock guard is not optimal. Depending on the lock grains chosen, the ancestor-descendant relationship may not be correctly identified. This leads to unnecessary conflicts and can lead to performance degradation.

3.2 Intention Lock

Intention locking is one of the first approaches to hierarchical locking that attempts to address the requirements **R1** and **R3** with efficiency. **gray1993granularity** introduced three new lock modes: IS (Intention Shared), IX (Intention Exclusive) and SIX (Shared Intention Exclusive) which signify the intention of a transaction to acquire a

Mode	NL	IS	IX	S	SIX	X
NL	Y	Y	Y	Y	Y	Y
IS	Y	Y	Y	Y	Y	N
IX	Y	Y	Y	N	N	N
S	Y	Y	N	Y	N	N
SIX	Y	Y	N	N	N	N
X	Y	N	N	N	N	N

Tab. 3.1: Compatibility matrix for intention lock modes. **NL**: NoLock, **IS**: Intention Shared, **IX**: Intention Exclusive, **S**: Shared, **SIX**: Shared Intention Exclusive, **X**: Exclusive

shared or exclusive lock on a descendant of a vertex locked under IS, IX or SIX mode respectively. Table 3.1 shows the compatibility matrix for the intention locks.

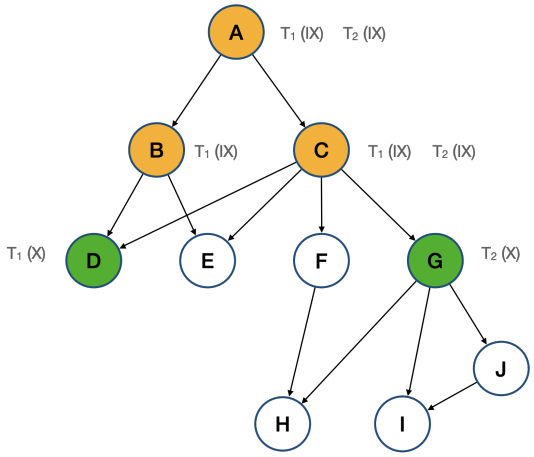


Fig. 3.2: Multi granularity locking via Intention locks.
T_1 takes an exclusive lock on vertex D and T_2 takes an exclusive lock on vertex G

Consider the example in figure 3.2 where a transaction T_1 takes an exclusive lock on vertex D and another thread, T_2 takes an exclusive lock on vertex G. Since D and G are not related by an ancestor-descendant relationship, they should not conflict. To achieve this with intention locks, both threads start from the root i.e A and place intention tags on the vertices from the root to their lock targets i.e D and G respectively. Let's assume that T_1 is faster than T_2 and manages to acquire its locks first. T_1 takes IX on vertices A, B and C; in this order, and then takes an exclusive lock on D.

When T_2 tries to acquire a lock on G, it places IX on vertices A, B and C;in this order, as well, before trying to acquire a lock on G. When T_2 tries to acquire a lock on A, it detects a pre-existing IX lock. Since two IX locks are compatible, as seen

in Table 3.1, T_2 can acquire the IX lock on A. Similarly, T_2 can acquire the IX lock on C and then proceed to acquire the exclusive lock on G.

The process of acquiring locks in this manner guarantees that the ancestor-descendant relationship is correctly identified. However, it involves multiple traversals from the root to the lock target. In the example in Figure 3.2, T_1 performs the following two traversals:

- $A \rightarrow B \rightarrow D$
- $A \rightarrow C \rightarrow D$

With trees, every vertex is reachable from the root via a unique path. This, however, is not the case with general graphs. In general graphs, as the number of paths from the root to a vertex increases, the number of traversals required to acquire a lock on that vertex also increases. This leads to a significant performance degradation.

So, while intention locks fulfil requirements $R1$ and $R2$, requirements $R3$ incurs a significant performance penalty.

3.3 DomLock

Dominator based locking (DomLock) [KN16] uses dominators to identify the ancestor descendant relationship in a hierarchy. DomLock finds a partial ordering of vertices in the hierarchy based on the ancestor-descendant relationship between them. To this end, DomLock uses the concept of dominators and immediate dominators.

Definition 2 (Dominator)**.** *A vertex d is a dominator of another vertex v if all the paths from root to v pass through d.*

Definition 3 (Immediate Dominator)**.** *Immediate Dominator: A dominator d is an immediate dominator of another vertex v if there exists no other dominator for v on the paths between d and v.*

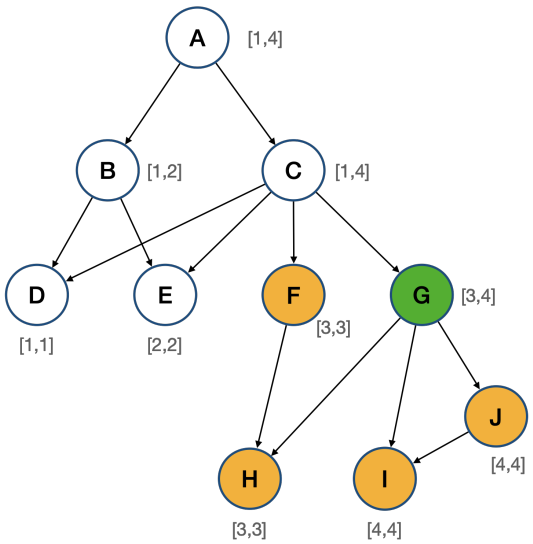


Fig. 3.3: Hierarchy labelled with DomLock intervals and DomLock on G (lock guard) with the grain of the grain of the lock (yellow).

3.3.1 Labelling through numeric intervals

In order to identify the dominators of a vertex more efficiently, DomLock uses a labelling scheme that assigns a pair of integers to each vertex. This pair is called the *interval* of the vertex. The interval of a vertex v is denoted as $[lv, rv]$. The interval of a vertex v is such that $lv \leq lu$ and $rv \geq ru$ for all vertices u that are descendants of v.

These intervals are computed by performing a depth-first traversal of the hierarchy. Consider the example in figure 3.3. The leaves of the hierarchy are labelled with unit intervals. For example, vertex D is labelled with the interval $[1, 1]$ since it is the first vertex in the DFS traversal. For an internal vertex, the interval is computed via the intervals of its children. The lv value of an internal vertex v is the minimum of the l values of its children. The rv value of an internal vertex v is the maximum of the r values of its children. For example, in figure 3.3, vertex G has three children H, I and J. The interval of G is $[3, 4]$ since the minimum l value of its children is 3 and the maximum r value of its children is 4. The computation of intervals is done in a bottom-up fashion via a post-order traversal of the hierarchy.

3.3.2 Lock Grain identification

The intervals of the vertices are used to check subsumption. A vertex u subsumes another vertex v when their intervals overlap i.e. $lv \leq ru \wedge lu \leq rv$. The lock

grain of a guard u is the set of vertices that have an overlapping interval with the interval of u. So, in figure 3.3, G subsumes F, H, I and J since the interval of G overlaps with the the intervals of F, H, I and J which are $[3,3]$, $[3,3]$, $[4,4]$ and $[4,4]$ respectively.

Herein lies the first major drawback of DomLock. *False subsumptions* occur when the intervals of two vertices overlap but they are not related by an ancestor-descendant relationship. For example, in figure 3.3, G is the dominator of F but F is not a descendant of G. Sometimes, due to false subsumptions disjoint subgraphs are locked together. This leads to spurious lock conflicts and consequently, performance degradation.

3.3.3 Label recomputation

Another drawback of DomLock is that it does not support dynamic hierarchies. The intervals of the vertices are computed once via a post-order traversal. When a vertex is added or removed from the hierarchy, this computation has to be redone.

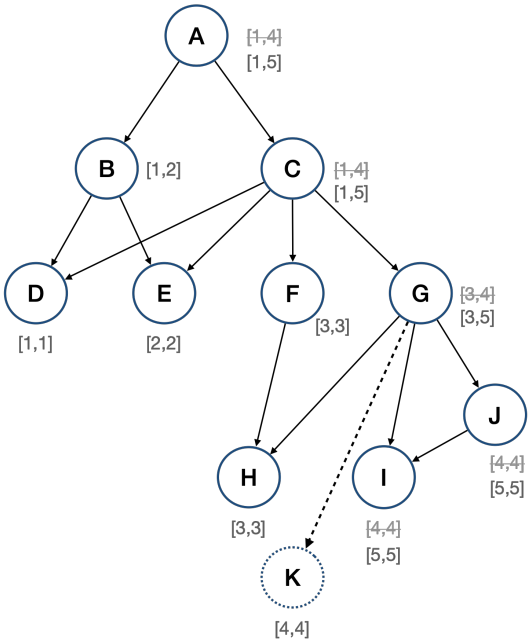


Fig. 3.4: DomLock interval recomputation for a vertex insertion

For example, consider the hierarchy in figure 3.4. If a vertex K is added as a child of G after H, K gets the interval $[4,4]$. I and J get the interval $[5,5]$. Following this, the interval of G is recomputed as $[3,5]$. This recomputation has to be done for all the ancestors of G as well, which includes the root of the hierarchy. In order

to prevent concurrent reads from interfering with the recomputation, a lock has to be placed on the root of the hierarchy. This lock is held until the recomputation is complete. In dynamic hierarchies, such modifications can lead to a significant performance penalty due to the lack of parallelism in the relabelling.

While DomLock addresses requirements $R1$ and $R2$, false subsumptions and the lack of support for dynamic hierarchies incur significant penalties against requirement $R3$ and $R4$ respectively.

3.4 Multi Interval DomLock (MID)

Multi Interval DomLock [AN22] is a successor to DomLock which uses a pair of intervals on each vertex to identify the dominator. MID maintains in addition to the DomLock intervals, another interval computed by a reverse post-order traversal of the hierarchy called the *'DFS-on-image'*. Figure 3.5 shows the MID intervals for a hierarchy.

3.4.1 Labeling through a pair of numeric intervals

Each vertex is labelled with two intervals: the DomLock interval and the MID interval. The only difference between the two intervals is the order of traversal of vertices. DomLock intervals being post-order and MID intervals being reverse post-order. Like DomLock, the interval of a leaf is a unit.

For example, in figure 3.5 , vertex H is labelled with a second interval $[2, 2]$ in addition to the DomLock interval $[3, 3]$ since it is the second vertex in the reverse post-order traversal. Again, like DomLock, for an internal vertex, the interval is computed via the intervals of its children. For both intervals, the $l(v)$ value of an internal vertex v is the minimum of the l values of its children. The $r(v)$ value of an internal vertex v is the maximum of the r values of its children. For example, in figure 3.5, vertex G has three children H, I and J. The intervals of G are $[3, 4]$ and $[1, 2]$ for DomLock and MID respectively since the minimum l value of its children is 3 and 1 and the maximum r value of its children is 4 and 2 respectively.

3.4.2 Lock Grain identification

The two pairs of intervals are used in an attempt to reduce *false subsumptions* where a vertex subsumes its siblings. When testing subsumption in MID, the two intervals are tested for overlap. A vertex v with intervals $[vl_1, vr_1]$ and $[vl_2, vr_2]$, subsumes another vertex u with intervals $[ul_1, ur_1]$ and $[ul_2, ur_2]$ iff $vl_1 \leq ur_1 \wedge ul_1 \leq vr_1 \wedge vl_2 \leq ur_2 \wedge ul_2 \leq vr_2$.

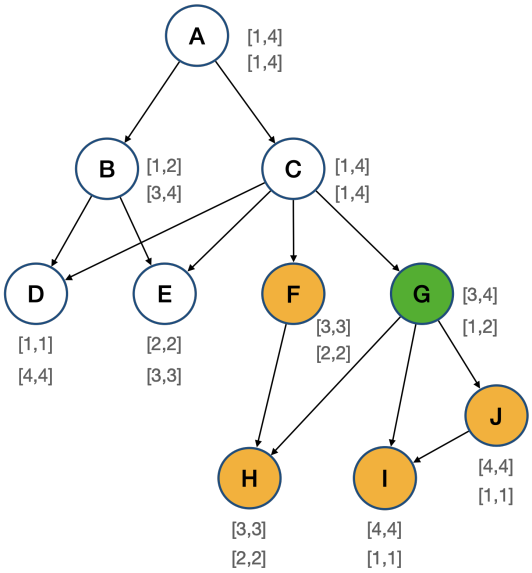


Fig. 3.5: MID labels with lock on guard G with the grain of the lock (yellow)

For example in figure 3.5, G subsumes F, H, I and J since both intervals of G overlap with the respective intervals of F, H, I and J. However, we still encounter a false subsumption. G subsumes F but F is not a descendant of G. Like DomLock, MID also suffers from poor performance due to such spurious conflicts.

3.4.3 Label recomputation

MID encounters double the penalty for recomputing vertex labels in dynamic hierarchies when a structural modification occurs. In DomLock, a single post-order-traversal is enough to compute the intervals so a structural modification leads to the recomputation of intervals in a single traversal. In MID, two traversals are required to compute the intervals. As a consequence, the intervals of all vertices is often recomputed.

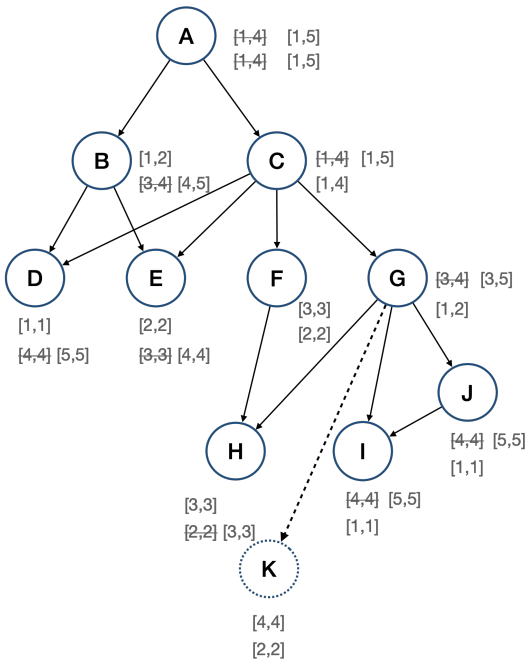


Fig. 3.6: MID interval recomputation for a vertex insertion

For example, when a vertex K is inserted as a child of G as shown in figure 3.6, At least one interval for every vertex is recomputed. These intervals are then propagated to the root of the hierarchy. In order to prevent concurrent reads from interfering with the recomputation, a lock has to be placed on the root of the hierarchy. This lock is held until the recomputation is complete. This lock is essentially a mutex, restricting concurrency.

MID tries to eliminate false subsumptions present in DomLock but does not do so successfully. The performance penalty for recomputation in dynamic hierarchies is doubled in MID. As such, MID fulfils requirements $R1$ and $R2$ but incurs even more penalties against requirements $R3$ and $R4$ than DomLock.

3.5 Flexible granularity Locking (FlexiGran)

FlexiGran [AN24] aims to enable the existence of MGL and fixed grain locks on the same hierarchy. The fixed grain locks used by flexigran are fine grained i.e. every vertex is its own guard. FlexiGran uses the intervals of DomLock as vertex labels and uses vertex depth to determine ancestor-descendent relationships between two identical intervals.

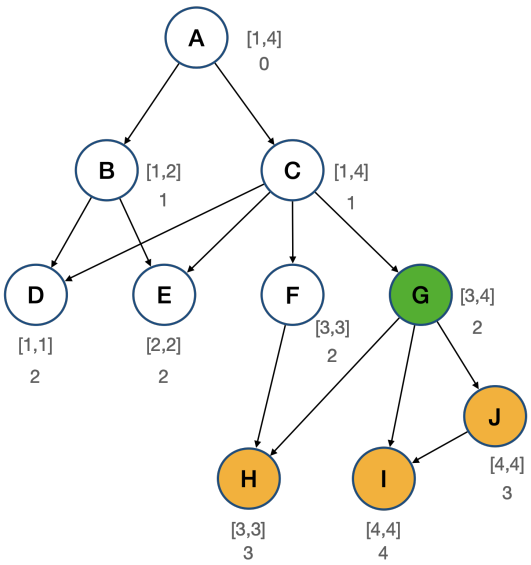


Fig. 3.7: FlexiGran labels and vertex depth with lock on guard G with the grain of the lock (yellow)

3.5.1 Labelling through numeric intervals

In FlexiGran, a post-order depth first traversal is performed to compute the labels of vertices. Like DomLock, the leaves of the hierarchy are labelled with unit intervals and internal vertices are labelled with intervals computed from the intervals of their children. In addition to these intervals, FlexiGran also computes the depth of each vertex in the hierarchy. The depth of a vertex is the length of the shortest-longest path from the root to the vertex. Figure 3.7 shows the intervals and vertex depths of a hierarchy labelled with FlexiGran.

This shortest-longest path is useful for depth determination in the presence of connected components like cycles. In a cycle, the path is recursive and based on the method of computation, the longest path can be infinite. The shortest-longest path breaks this recursion to a limit. This limit is the length of a path from the root to a vertex which contains the target vertex at most twice, as is the case with cycles. In a connected component, all the vertices have the same depth.

3.5.2 Lock Grain identification

FlexiGran uses the depth information in addition to the intervals to determine the ancestor-descendant relationship between two vertices. Since both fine grained and MGL locks can exist in flexigran, the process of testing overlaps involves multiple

tests. Table 3.2 shows the compatibility matrix for FlexiGran locks. When checking if two hierarchical locks are compatible, the intervals of their guards are tested for overlap. If the intervals are identical then the depth is used to break the tie and determine the ancestor-descendant relationship.

	Lock On Ancestor				Lock On Descendent			
Lock Mode	**FR**	**FW**	**HR**	**HW**	**FR**	**FW**	**HR**	**HW**
FR	Y	Y	Y	N	Y	Y	Y	Y
FW	Y	Y	N	N	Y	Y	Y	Y
HR	Y	Y	Y	N	Y	N	Y	N
HW	Y	Y	N	N	N	N	N	N

Tab. 3.2: Flexigran compatibility matrix showing the protocol for the co-existence of hierarchical and fine-grained locks in a system. **F**: Fine-grained, **H**: Hierarchical, **R**:Read, **W**: Write

When a hierarchical lock is tested for compatibility with a fine grained lock, then a traversal is performed to test the reachability of the fine grained lock guard from the hierarchical lock guard. If the fine grained lock guard is reachable from the hierarchical lock guard or vice-versa, then the two locks are incompatible. For example, in figure 3.7, a hierarchical lock is acquired on G. This lock guards vertices H, I and J. Unlike DomLock and MID, FlexiGran does not subsume F since F is not a descendant of G since their depths are the same. In this situation, if a fine grained lock is requested on F, it would be compatible with the hierarchical lock on G since there is no path from G to F or vice-versa.

3.5.3 Label recomputation

Like DomLock, structural modifications in FlexiGran require recomputation of the intervals of the vertices. A single depth first traversal is enough to recompute the intervals and depths of vertices. Since the root might be involved in the structural modification, a lock is placed on the root to prevent concurrent reads from interfering with the recomputation. This restricts concurrency and leads to a degradation in performance.

As such, FlexiGran fulfils requirements $R1$ and $R2$ and incurs a significant performance penalty against requirement $R3$ due to the expensive compatibility checks required to detect conflicts between MGL and fine grained locks. FlexiGran also incurs a performance penalty against requirement $R4$ due to the non-parallel recomputation of intervals in dynamic hierarchies.

3.6 Requirements and Trade-offs

The different MGL techniques discussed in this chapter have tradeoffs. Recall the four primary requirements identified for MGL:

R1 Identifying a lock guard for every vertex of a hierarchy.

R2 Finding an appropriate, optimal lock guard for a request.

R3 Detecting conflicts between locks i.e. testing ancestor-descendant relationship.

R4 Housekeeping the metadata required to implement the locking protocol.

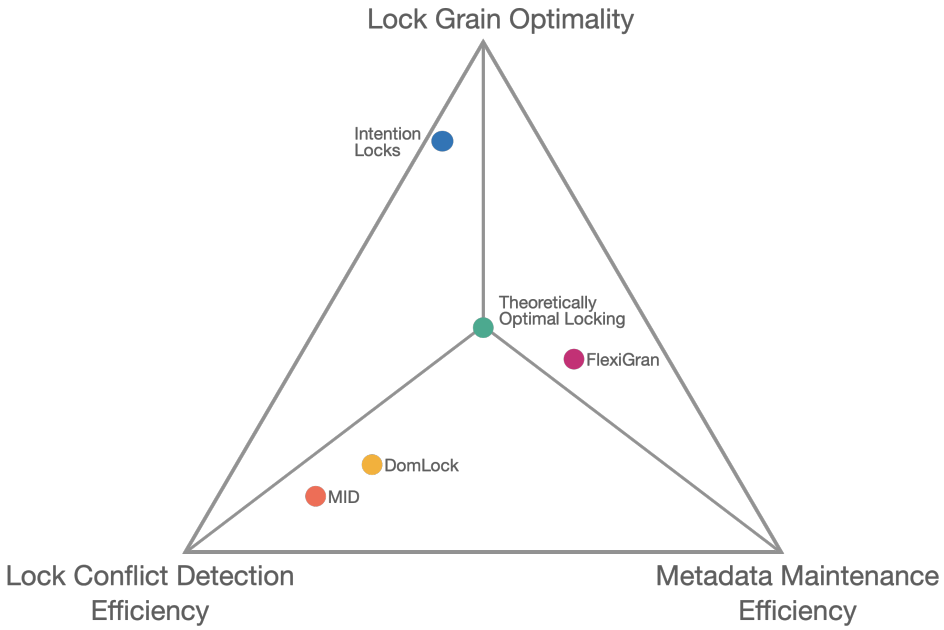


Fig. 3.8: Trade-offs in MGL techniques

Fixed Grain locks are inefficient and only fulfil requirement **R1**. MGL based on traversals like Intention Lock, fulfils requirements **R1** and **R2** but incurs a significant performance penalty against requirement **R3** due to the sheer number of traversals required.

Label based techniques like DomLock, MID also fulfil requirements **R1** and **R2** but incur a performance penalty against requirement **R3** due to false subsumptions. In addition, due to the metadata required to implement the locking protocol, these techniques incur a performance penalty against requirement **R4** as well.

FlexiGran that implements MGL and fine grained locks suffer from poor performance due to the expensive compatibility checks and label recomputation. FlexiGran fulfils requirements **R1** and **R2** only.

Figure 3.8 shows the trade-offs in the different MGL techniques. Intention locks offer the most optimal lock grains at the cost of lock conflict detection efficiency. DomLock and MID offer efficient conflict detection at the cost of metadata management and do not have optimal lock grains. FlexiGran offers better lock grains but incurs a performance penalty due to the expensive lock conflict detection. None of the techniques balances all the requirements to come close to an optimal MGL protocol.

4 Theoretical Framework

In this chapter, we present the formal definitions and theoretical underpinnings of CALock. We begin by introducing the basic concepts of graphs, paths, and vertex relationships. Next we discuss the bounds of commonality between vertices in a graph and how they can be used to determine the optimal locking grain. Finally, we formulate the optimal locking grain problem mathematically and prove its correctness.

4.1 Graphs, Paths and Vertex Relationships

Let $G = (V, E)$ be a directed graph. V is its set of vertices, connected by directed edges in the set $E \subseteq V \times V$. A rooted graph has a distinguished vertex $r \in V$ such that all other vertices are reachable from r.

A pair of vertices (u, v) can be connected by a sequence of edges, called the path between u and v, noted p. The set of vertices of p is noted $\mathcal{V}(p)$. The length l_p of this path is the size of $\mathcal{V}(p)$ i.e. $l_p = |\mathcal{V}(p)|$.

In the general case, several such paths may exist between a pair of vertices. The set of paths between u and v is noted $\mathcal{P}_{(u,v)}$. The depth $\delta(v)$ of a vertex v is the length of the shortest path in the set $\mathcal{P}_{(r,v)}$.

Definition 4 (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.*

Definition 5 (Guarding Ancestor)**.** *A guarding ancestor of a vertex u is a vertex $v \in G$ that lies on all paths from r to u. The set of guarding ancestors of u is noted L_u.*

Definition 6 (Lowest guarding Ancestor)**.** *The lowest guarding ancestor $LGA(u)$ of a vertex u is a guarding ancestor of u with the maximum depth.*

Definition 7 (Common Ancestor)**.** *A common ancestor (CA) of two vertices u and v is a vertex c that is an ancestor of both u and v.*

Definition 8 (Lowest Common Ancestor)**.** *The lowest common ancestor $LCA(u, v)$ of two vertices u and v is a common ancestor of u and v with the maximum depth.*

The Lowest Guarding Common Ancestor $LGCA(Q)$ of a set of vertices Q is the deepest vertex that is both a guarding ancestor and a common ancestor of all vertices in Q. Fischer et. al [FH10] derive the following relationships between the LGA and the LGCA of vertices in a rooted DAG.

Lemma 1.

$$\textit{For a vertex } v \in G : v \neq r, LGA_G(v) = LGCA_G(parents_G(v))$$

Definition 9.

$$LGCA_G(w_1, ..., w_k) = LGCA_G(w_1, LGCA_G(w_2, ..., w_k))$$

The $LGCA$ of a set of vertices is the intersection of the sets of guarding ancestors of those vertices. We have the following theorem:

Theorem 1.

$$LGCA_G(v, w) \textit{ is the vertex of the maximum depth in } L_v \cap L_w$$

Proof. Let $l = LGCA_G(v, w)$ be the LGCA of $\{v, w\}$, 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 all paths from l to v and l to w in G. Since l' lies on the paths in the sets $\mathcal{P}_{(l,v)}$ and $\mathcal{P}_{(l,w)}$, inductively, l' also lies on all the paths from the root (r) of the graph to the vertices v and w. If l' lies on these paths then it is a guarding ancestor of v and w and the following holds true: $(l' \in L_v) \wedge (l' \in L_w) \equiv l' \in L_v \cap L_w$.

Since l' lies on the paths to v and w after l, by the definition of depth $\delta(v)$ of a vertex, $\delta(l') > \delta(l)$.

Now we have two conclusions, $l' \in L_v \cap L_w$ and $\delta(l') > \delta(l)$.

Since l' is deeper than l, it should be the deepest member of $L_v \cap L_w$. Which means, l' is the LGCA of v and w. But this contradicts our original assumption that l is the LGCA of v and w. This means our assumptions on l' contradict the definition $l = LGCA_G(v, w)$ and l is the deepest element in the set $L_v \cap L_w$. □

Tab. 4.1: Terms used in the definitions

Term	Meaning
G	Graph
V	Vertices of G
E	Edges of G
r	Root of G
u, v, w	Vertices
$\delta(v)$	Depth of vertex v
$GA(u)$	Guarding ancestors of u
$LGA(u)$	Lowest guarding ancestor of u
Q	Set of vertices
$CA(Q)$	Common ancestors of Q
$LCA(u, v)$	Lowest common ancestor of u and v
$LGCA(Q)$	Lowest guarding common ancestor of Q
L_u	Label of vertex u

4.2 CALock labelling scheme

CALock label for a vertex is the set of its guarding ancestors. To ease the implementation, we derive a recursive function from the definitions in Section **??** that can be used to compute these labels. An implementation of this function is shown in Algorithm 1

4.2.1 Recursive labelling function

Theorem 1 is defined for rooted DAGs. We combine it with the definitions and lemmas from Section **??** to derive a recursive function that we use to label the vertices in a graph. To this end, Lemma 1 can be rewritten using Definition 9 as follows:

$$LGA_G(v) = LGCA_G(p_1, LGCA_G(p_2, ..., p_k)) \\ \text{where } p_1, p_2, ...p_k \text{ are the parents of } v \tag{4.1}$$

On combining equations 4.1 and theorem 1; $LGA_G(v)$ is the deepest vertex 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 sets of guarding ancestors of parents of v, are enough to compute the lowest guarding ancestor of v. We use this property to recursively compute the labels of the vertices in a graph using the following function.

$$L_v = \begin{cases} \{v\} & \text{v is the root} \\ \{v\} \cup \{\cap_{u \in parents(v)} L_u\} & \text{otherwise} \end{cases} \tag{4.2}$$

4.2.2 Labelling graphs recursively

Labelling acyclic graphs

Labelling starts at the root of the graph when the function `AssignLabel(r)` is called. It uses Relation 4.2 implemented in lines 12 - 17 to assign the labels. The root vertex does not have parents, so the label of the root is the set containing the ID of the root itself (lines 8 - 11). For example, in figure 5.1 vertex A has the label $\{A\}$.

Then, the children of the vertex v are explored via a breadth-first traversal over the graph. Using the labels of the parents, the label of the child is computed via Relation 4.2. For example, in figure 5.1, the label of vertex B is $\{A, B\}$ since it has only one parent. Vertex E has two parents which have the labels $\{A, B\}$ and $\{A, C\}$ respectively. Their set intersection is $\{A\}$. Using Relation 4.2, the label of E is $\{A, E\}$.The graph is explored and labelled until the recursion reaches a fix-point and terminates (line 15).

Labelling strongly connected components

Theorem 1 assumes that the graph does not contain strongly connected components. In real-life applications, maintaining this constraint is difficult. There are strategies to find strongly connected components in a directed graph [Sha81; Tar72; CM96] that can be contracted by vertex contraction [Gra+08]. We, however, maintain that finding and eliminating strongly connected components is not required to identify lock grains using CALock when the labels are assigned using Relation 4.2.

Assume that $N \subseteq V$ is a set of vertices of a graph that are strongly connected. Since every vertex in N is reachable from every other vertex in N, the path set $\mathcal{P}_{(u,v)}$ for any pair of vertices $u, v \in N$ contains the same vertices in different orders and hence, every vertex has the same set of guarding ancestors. The function `BFLabel` recurses over vertices in N until all the paths to a vertex are explored and its label does not

change anymore (line 15). When `BFLabel` terminates, all vertices in N have the same label.

Therefore, Relation 4.2 is sufficient to label rooted directed graphs with strongly connected components. An implementation of recursive labelling is shown in Algorithm 1.

Relabelling graphs

The topology of a graph can change due to *structural modifications*. Then, the labels of the vertices need to be recomputed. The same function `AssignLabel(v)` can be used to relabel the graph. Relabelling begins at the vertex affected by the structural modification via the function `AssignLabel(v)`. In the same fashion as the initial labelling, the parents of the affected vertex are used to compute its label and then recursively of its children. Relabelling is terminated as soon as it reaches a fixpoint (line 15).

Algorithm 1 Labelling the graph

```
 1: procedure ASSIGNLABELS(v)
 2:     queue.PUSH(v)
 3:     while queue.HASNEXT( ) do
 4:         v ← queue.NEXT( )
 5:         BFLABELA(v, queue)

 6: procedure BFLABELA(v, queue)
 7:     C ←CHILDREN(v)
 8:     if parents(v) = ∅ then
 9:         v.label ← {v}
10:         queue.PUSH( C )
11:         return
12:     P ←PARENTS(v)
13:     tempLabel ←INTERSECTION(P.labels)
14:     tempLabel.APPEND(v)
15:     if v.label ≠ tempLabel then
16:         v.label ← tempLabel
17:         queue.PUSH( C )
```

5 CALock: A New Approach to Multi-Granularity Locking

In CALock, when a thread wants to lock a set of lock targets, it issues a lock request for the LGCA of targets, computed by taking a set intersection of their labels (see theorem 1). This LGCA is a vertex in the graph and is the *guard* for this lock request. A guard is the root of the smallest grain that contains the lock targets. For example, in Figure 5.1, to lock I and J, we find their LGCA which is G and place a lock on G (the lock guard).

As soon as the thread identifies the LGCA, it issues a lock request by adding its request to a pool used to identify conflicts (Section 5.1). The thread then proceeds to check for conflicts and blocks if one exists. In the absence of conflicts, the thread proceeds to its critical section.

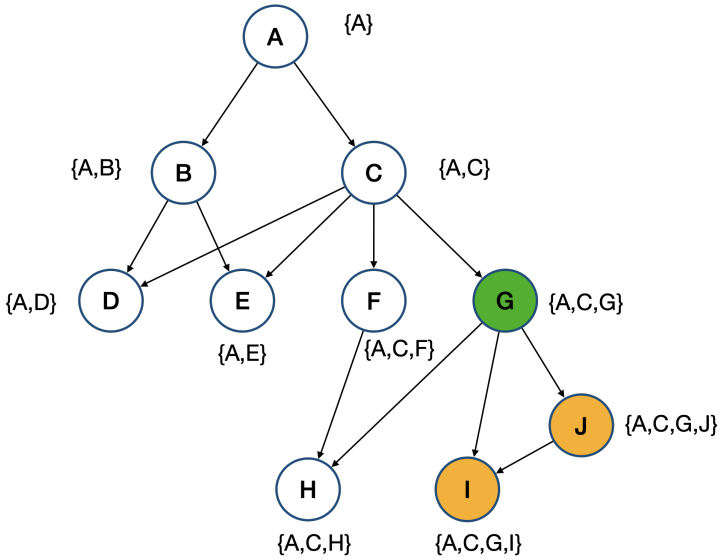
5.1 Identifying lock conflicts

In MGL on graphs, conflict detection is not as simple as testing for read/write conflicts. Since lock guards have a grain that they protect, it is necessary to ensure that the grain remains exclusive for a writer. With CALock, two conditions indicate a lock conflict. If both conditions hold for a pair of lock requests, they are in conflict.

- The lock requests have a *mode conflict* i.e. a read/write conflict.
- The lock requests have a *grain overlap* i.e. they are trying to lock overlapping grains.

Table 5.1 shows compatibility between lock modes.

In order to remain fair and prevent thread starvation, CALock uses sequence numbers to resolve conflicts. The *sequence numbers* are an indication of the order in which the locks are requested. In the presence of conflicts, threads are granted locks in a *First come first serve* order.



Lock on G: {A,C,G} with grain: {I, J}

Fig. 5.1: CALock labels

A thread that wants to lock a set of targets creates a lock request and adds it to a *lock pool* for conflict detection. The request contains the following data.

- **Thread ID**: of the thread requesting the lock
- **Guard ID**: of the vertex being locked in order to lock the set of targets.
- **Guard label**: Label of the lock Guard vertex
- **Sequence number**: allocated to the lock request when it is added to the lock pool.
- **Activity Indicator**: used by other threads to wait in the presence of conflicts.
- **Lock mode**: read or write

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`. Algorithm 2 shows the flow of lock acquisition and conflict detection in the lock pool. When a thread arrives in the lock pool with the information of the lock it wants to acquire, it is given a sequence number (line 6) and it sets its activity indicator to true (line 7) indicating that it is active and using the lock. With this additional information, the lock request is added to the pool (line 8). In order to conserve the first come first serve order and prevent race conditions where a later lock request is granted first before the thread with an earlier request could detect the conflict, the three steps involving the assignment of

Tab. 5.1: Lock compatibilities between read (rl) and write (wl) locks requested by threads $i \neq j$ on vertices x and y.

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

✓ compatible.
✤ compatible iff grain of x is disjoint from grain of y.

a sequence number, setting the activity indicator and addition to the lock pool are done atomically.

After the lock request is added to the pool, the thread checks for conflicts with other requests in the pool.

- **Mode conflict**: is checked by comparing the lock modes in the requests.
- **Grain overlap**: is checked using the guard ID and guard label of the requests. For two lock requests R_1 and R_2, if the guard ID of $R1$ is present in the guard label of R_2 or vice-versa, then the grains overlap.
- **Arrival order**: If there is a mode and grain conflict, the request with a lower sequence number is granted first.

A thread checks these conditions (line 11) by iterating over the pool (line 10). When requesting a lock, any thread T can be blocked for a maximum of n turns where n is the maximum number of threads. After n blocks, the sequence number of T will be the lowest among all requests in the lock pool and it will be allowed to proceed. This is essential to prevent thread starvation.

Figure 5.2 shows the snapshot of a lock pool in use by threads. This pool contains three active threads T_1, T_2 and T_7. T_7 arrives first and is assigned the sequence number 1. T_1 and T_2 arrive later and have sequence numbers 2 and 3 respectively. T_1 and T_2 conflict because they have overlapping grains. T_2 is requesting a lock on C which overlaps with the grain locked by T_1. This is detected by checking if C is present in the label of G i.e. $C \in \{A, C, G\}$. To resolve the conflict, T_1 is granted a lock since it arrived first and T_2 has to wait for T_1 to release the lock. T_7 does not have a conflict with any thread and acquires a lock on B.

Algorithm 2 Lock acquisition request in the lock pool

```
1: GSeq : Global sequence number for lock requests
2: Mutex : Mutex used when adding requests to the lock pool
3: Condition : Atomic boolean value used by waiting threads when in conflict

4: procedure LOCK(req, threadID)
5:     LOCK(Mutex)
6:     req.Seq ← Gseq++
7:     req.condition.TESTANDSET(true)
8:     Pool[threadId] ← req
9:     UNLOCK(Mutex)
10:    for all lock ∈ Pool \ threadID do
11:        if lock ≠ NULL
12:            ∧(req.HASRWCONFLICT(lock))
13:            ∧(lock.guardID ∈ req.label ∨ req.guardID ∈ lock.label)
14:            ∧(req.Seq > lock.Seq) then
15:            thread.BLOCKANDWAIT(lock.condition)
16:    return true

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

T_1	T_2	T_3	T_4	T_5	T_6	T_7
G,read,2,{A,C,G}	C,write,3,{A,C}	NULL	NULL	NULL	NULL	B,read,1,{A,B}

Fig. 5.2: Lock pool containing details of locks held by threads

5.2 Structural modifications, locking and relabelling

CALock can be utilized for static graphs whose structure does not change and for dynamic graphs that change at runtime. When a structural modification occurs, the grain in which the modification takes place needs to be relabelled. Unlike DomLock, MID and FlexiGran in which the relabelling happens under a global mutex, relabelling in CALock is done under the same lock that is acquired to perform the structural modification. Here, we explain the locking and relabelling mechanism for dynamic graphs.

Vertex addition and deletion Adding a vertex to the graph does not change the observable topology because this new vertex is not connected to the graph and

hence it is also not reachable from the root. So, addition does not require locking or relabelling of any kind. Deleting a vertex that does not have any edges does not require synchronization either because a disconnected vertex is not reachable from the root of the graph.

However, to delete a vertex that is reachable from the root i.e. connected to the graph, a lock is required. To do this, a write (exclusive) lock is taken on the LGCA of the vertex that needs to be deleted. The LGCA guards all the paths reaching this vertex and prevents inconsistent access to it. Once the lock is acquired, the vertex is disconnected from the graph and then deleted.

After a vertex is deleted, relabelling might be necessary if the deleted vertex's descendants are still connected to the graph since the set of their ancestors has changed. Relabelling is done by recursively calling the function `BFLabel()` in Algorithm 1 on the children of the deleted vertex. The lock acquired to delete the vertex is released after the relabelling is complete.

Edge addition and deletion Adding and deleting edges to the graph changes its topology and also the paths to the vertices. Both operations are performed under a write (exclusive) lock. When adding or deleting an edge between a source vertex u and a target vertex v, a write (exclusive) lock is taken on the LGCA of u and v. Both operations change the ancestors of a vertex so relabelling is initiated at the target (v) of the affected edge using `BFLabel()` function in Algorithm 1.

When a vertex has only one incoming edge, deleting that edge disconnects it from the graph. In this case, relabelling can be omitted because the target vertex v has no parents and is also not reachable from the root of the graph.

5.3 Properties of CALock

The properties of any algorithm hold importance in guaranteeing behavioural stability. In this section, we show that CALock is safe, live and does not lead to thread starvation. However, it's crucial to emphasize that the discussion of these formal properties only holds true value when the implementation of the locking algorithm is correct. CALock requires the assignment of lock request sequence numbers, setting of the activity indicator and addition of the lock pool to happen atomically. In our implementation, we use a mutex to achieve this.

5.3.1 Safety

Property A thread holding a write lock on a guard has exclusive access to the corresponding grain. This means that when a thread requests a write lock, it is granted the lock only if no other thread can access this grain.

Discussion By contradiction, assume that two threads T_1 and T_2 are granted a write lock on the same vertex v. Then the lock pool would contain, at indices 1 and 2 respectively, two lock requests R_1 and R_2 with the same lock target v and lock mode wl. However, they would have different sequence numbers since sequence numbers are assigned atomically before the requests are added.

At the entry to the critical section, T_1 (resp. T_2) iterates over the pool to check for conflicts of three kinds (Mode conflict, grain conflict, sequence number conflict). Since the iteration is always done in the same order, T_1 would detect a mode and grain conflict with T_2 and block if its sequence number is higher and vice-versa. Hence, T_1 and T_2 are never, simultaneously granted a lock on v. Therefore, CALock is safe.

5.3.2 Liveness

Property When a thread requests a lock, it is guaranteed to be granted the lock eventually i.e. The lock acquisition algorithm does not block forever.

Discussion In a correct implementation of an application that uses CALock, when a thread leaves its critical section, it releases the lock it holds and this is guaranteed to happen i.e. threads do not sleep after lock acquisition. The lock pool prevents a thread from holding multiple locks. If a thread wishes to lock multiple vertices, it requests a lock on their LGCA. Since threads hold at most one lock at any given time, deadlocks never occur.

The absence of deadlocks combined with the guaranteed progress of threads ensures that CALock is live.

5.3.3 Fairness

Property When a thread requests a lock, it is granted the lock after a defined number of blocks. The lock acquisition algorithm does not lead to a starvation of threads.

Discussion CALock uses a FIFO mechanism to grant locks. When a thread is blocked, it is always blocked by a thread that has a lower sequence number. A thread can be blocked and bypassed by other threads in the presence of conflicts at most n times (n is the size of the lock pool). Since sequence numbers are assigned atomically and are monotonically increasing, a thread is granted its lock after at most n bypasses. After a thread is bypassed n times, the sequence number of that blocked thread would be the lowest and it would be granted the lock it requested. This ensures that no thread starves and CALock is fair.

5.4 Complexity analysis of CALock

The labeling algorithm involves two operations.

- *Traversal*: Recursive BFS over the graph starting from the root. The number of edges examined is dependent on the average degree of vertices. In the worst case, where the graph is complete, the degree of any vertex is the number of vertices in the graph.

- *Label computation*: For each vertex, the intersection of its parents' labels needs to be calculated. The number of elements in the label of a vertex is inversely proportional to the number of its parents. A vertex with more parents has more paths from the root and hence, a smaller label. We can approximate this in terms of the vertex degree as well.

For a graph $G = (V, E)$, let $v = |V|$ be the number of vertices, $e = |E|$ be the number of edges and d_{avg} be the average degree of a vertex. According to the Handshake lemma [Eul41], the average degree of a vertex is

$$d_{avg} = \frac{2 \times e}{v} \tag{5.1}$$

	Labelling	Relabelling	Lock Guard computation	Conflict detection
DomLock	$\theta(v^2 + \frac{v^2}{d_{avg}})$		$\theta(v^2 + \frac{v^2}{d_{avg}})$	$O(n)$
MID	$\theta(2v^2 + \frac{2v^2}{d_{avg}})$		$\theta(v^2 + \frac{v^2}{d_{avg}})$	$O(n)$
FlexiGran	$\theta(v^2 + \frac{v^2}{d_{avg}})$		$\theta(v^2 + \frac{v^2}{d_{avg}})$	$O(n)$
CALock	$\theta(v^2 + \frac{v^2}{d_{avg}})$		$\theta(\frac{v}{d_{avg}})$	$O(n)$

Tab. 5.2: Average case complexities of MGL techniques

	Labelling	Relabelling	Lock Guard computation	Conflict detection
DomLock	$\theta(v^2)$		$\theta(v^2)$	$\theta(n)$
MID	$\theta(2v^2)$		$\theta(v^2)$	$\theta(n)$
FlexiGran	$\theta(v^2)$		$\theta(v^2)$	$\theta(nv^2)$
CALock	$\theta(v^2)$		$\theta(1)$	$\theta(n)$

Tab. 5.3: Worst case complexities of MGL techniques

The complexity of breadth-first traversal is $\theta(v + v.d_{avg})$. For each vertex, the size of its label is inversely proportional to the number of parents it has and consequently, label computation is inversely proportional to the average degree i.e. $\theta(\frac{v}{d_{avg}})$.

The combined complexity of these operations for the initial labeling is $\theta(v^2 + \frac{v^2}{d_{avg}})$.

In the best case, the graph contains only one vertex. The best case complexity is $\Omega(1)$. In the worst case, the average degree of vertices is 1. Thus, the worst-case complexity is $O(v^2)$.

When a thread requests a lock, it finds the LGCA of the lock targets and issues a lock request. The LGCA is computed via a set intersection of the labels of the lock targets. If the lock request is for q lock targets and the average label size is $\theta(\frac{v}{d_{avg}})$, the complexity of finding the lock guard is $\theta(\frac{q.v}{d_{avg}})$. Since $q << v$ for real-world applications, the average case complexity can be reduced to $\theta(\frac{v}{d_{avg}})$. In the best case, the lock request is for a single vertex and the complexity is $\Omega(1)$. In the worst case, the lock request is for all vertices of the graph and the complexity is $O(\frac{v^2}{d_{avg}})$.

These labels are then used to detect conflicts. A thread checks conflicts with all other threads i.e. n times. For each conflict, a set membership test is performed. With an efficient set implementation, the membership can be tested in $O(1)$. The complexity of conflict detection is $\theta(n)$.

6 Implementation

details of the implementation about how the labelling was designed. Some optimizations about thread waiting and backoff.

Don't forget to put the source code link ;-)

7 Experimental Evaluation

With benchmark experiments, we evaluate the performance of CALock and compare it with the state of the art against several parameters. Our evaluation uses the same benchmark experiments as DomLock, MID and FlexiGran [KN16; AN22; AN24] i.e. STMBench7[GKV07].

We run our experiments on a machine with an AMD EPYC 7642 CPU, with 48 Cores, a base clock of 2.3 GHz and 512 GB of RAM. The benchmark is deployed on a Docker container under Ubuntu 20.04. To build the benchmark, GCC 12.1 and Cmake 3.22 are used. The compilation is done at the C++ 26 standard.

7.1 STMBench7

STMBench is a well known benchmark based on the classical OO7 object oriented benchmark suite [CDN93]. It is used to evaluate the performance of hierarchical algorithms and data structures [Pro+19; Val+16; Fel+16; CC16; KPR15; FB15; RC14; KN16; AN22; AN24; KN18; GKN18; LZ14]

The graph in STMBench7, as represented in Figure 7.1, consists of a *module* at the root level. This module consists of several levels of *complex assemblies*. Each of the deepest complex assemblies consists of a set of *base assemblies*. A *composite part* can be contained in several base assemblies. Each composite part contains a set of *atomic parts*. These atomic parts form a near-complete graph. The root of this graph is connected to a single composite part.

STMBench7 provides coarse-grain and medium-grain lock implementations. These are representatives of fixed granularity locking techniques, used in practice. The coarse lock is a reader-writer lock on the entire graph. Figure 7.1 shows the medium lock grains for a module. Each level of complex assemblies is guarded by a single lock. Deeper in the graph, the atomic parts are guarded by a lock of their own. Structural modification operations take a mutex on the entire graph and happen in isolation. STMBench uses the following operations to stress the locking algorithms.

- Reading an atomic part (Q1).

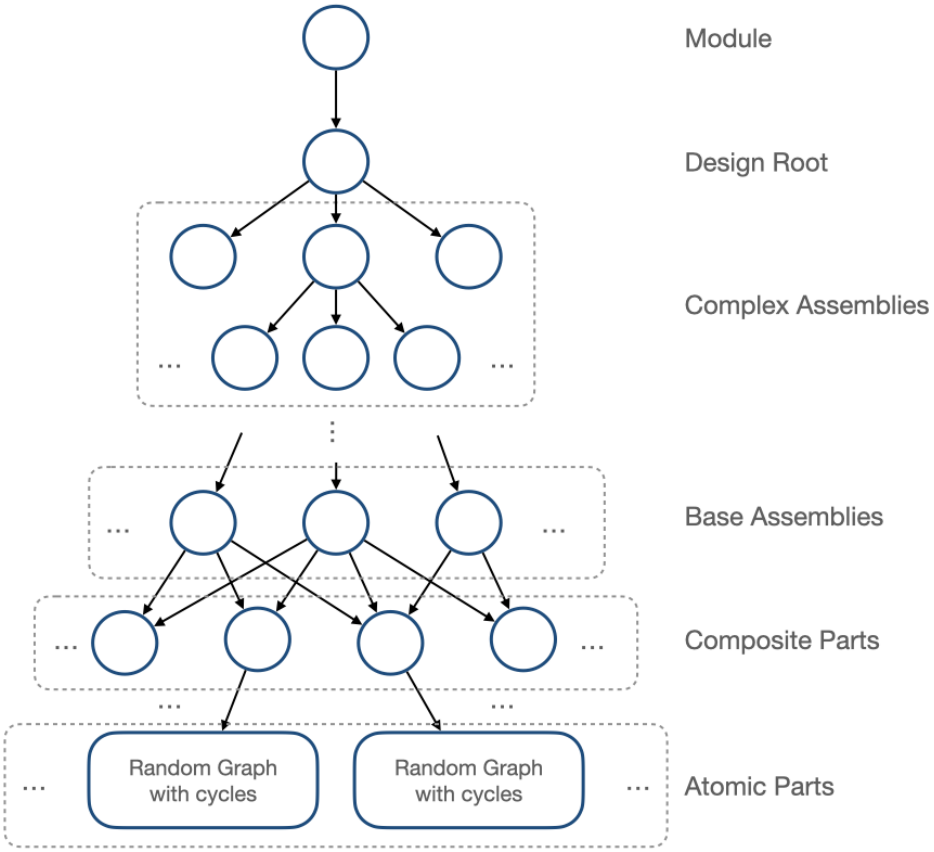


Fig. 7.1: Structure of a module in STMBench with lock boundaries

- Reading a set of atomic parts (Q2).
- Reading a complex assembly (OP1).
- Reading a set of complex assemblies (OP1).
- Reading a base assembly (OP2).
- Reading a set of base assemblies (OP2).
- Writing data to an atomic part (OP3).
- Writing data to a set of atomic parts (OP4).
- Deleting a composite part (SM1).
- Adding an edge between a base assembly and a composite part (SM2).

7.1.1 Synopsis

The research questions addressed by the benchmarks are the following:

§7.2 How long do different operation types take to complete under different locking strategies?

§7.3 What is the cost of maintaining metadata for CALock compared to DomLock, MID and FlexiGran?

§7.5 What are the relative sizes of lock grains between CALock, DomLock, MID?

§7.6.1 What is the performance of CALock compared to other lock strategies for locking on static graphs?

§7.6.2 What is the performance of CALock compared to other lock strategies for locking in dynamic graphs?

7.2 Per operation latency

Figure 7.2 shows the time to completion for different operations in STMBench7. For coarse and medium grained locks, reads are fast and finish relatively quickly since they occur under pthread read locks which do not require additional metadata. Write operations, due to their nature are slower since they require exclusive access to the graph. However, with additional performance metrics (see Sections 7.6.1, 7.6.2), we observe that fixed grain locks do not scale.

Intention locks often encounter deadlocks. This is because, unlike trees, the paths to a vertex are not unique. As such, a thread needs to intention lock all the vertices on the paths to a target vertex. Concurrent threads are not guaranteed to lock the all vertices in the same order and hence, deadlocks occur. When a deadlock is detected, the lock request is retried. If the deadlock persists, the lock request is rejected and counted as failed.

MGL techniques like DomLock, MID, FlexiGran and CALock are slower than coarse and medium grained locks. They are slow for reads and writes due to the presence of false subsumptions between grains causing spurious thread blocks. For structural modifications, DomLock, MID and FlexiGran spend time relabelling the hierarchy which is very expensive.

CALock is faster than DomLock, MID and FlexiGran for reads and writes since it avoids false subsumptions (see Section 7.5). For structual modifications, unlike DomLock, MID and FlexiGran, the relabelling can be parallelised if the lock grains do not overlap. When deleting vertices from the hierarchy (SM1), CALock does not require relabelling.

7.3 Metadata management: bulk labelling and relabelling

7.3.1 Bulk labelling

We measure the time it takes to assign the labels to a graph when it is first created. This simulates loading data into a database. Coarse-grain, medium-grain locks and intention locks do not need labelling. This time is measured for three different sizes of the STMBench7 graph. The results are shown in Figure 7.3. We observe

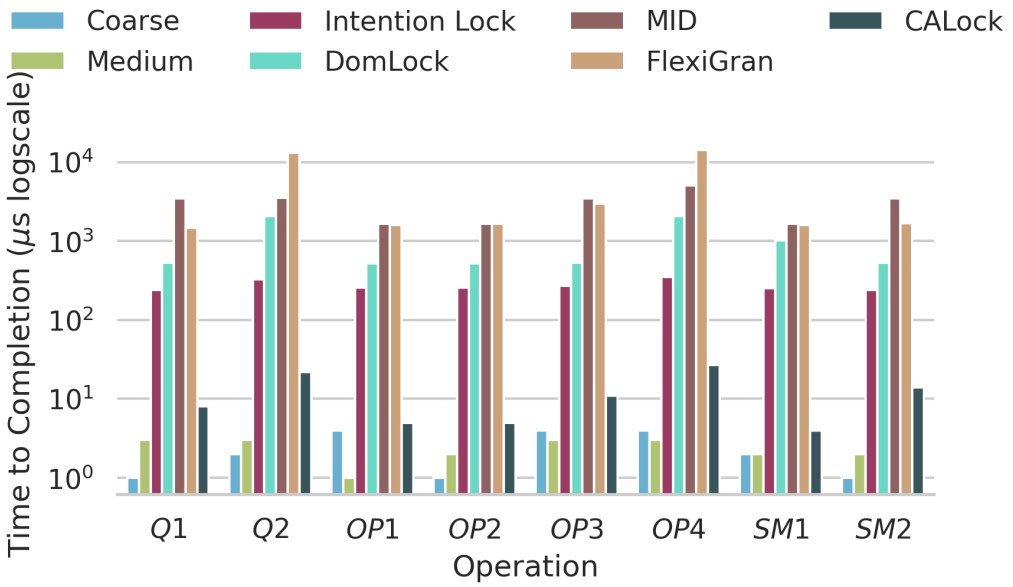


Fig. 7.2: Time to completion for different operations(lower is better). Deadlocks detected in Intentionlocks in grey.

that DomLock is the fastest at preprocessing the hierarchy since a single depth-first-traversal is sufficient to compute the intervals. MID is slower than DomLock because it needs to compute two pairs of intervals for each vertex. One pair the intervals is computed by a depth-first traversal and another by a depth-first traversal on the mirror image of the graph. FlexiGran is faster than DomLock but slower than MID because of the additional level information that needs to be computed per vertex.

CALock labels take the logest to preprocess since the labels are defined by a recursive breadth-first traversal with a fixpoint dependent on the number of paths to a vertex from the root.

7.3.2 Relabelling

Figure 7.4 shows the average time spent relabelling the graph per structural modification. Both Coarse, medium grained locks and intention locks do not have additional metadata and hence do not require relabelling. In CALock, relabelling is significantly faster than DomLock, MID and FlexiGran. This is because in DomLock, MID and FlexiGran a structural modification anywhere in the graph changes the depth-first traversal order of several vertices of the graph. A change in this traversal order requires re-computing a lot of intervals which propagate to the root of the hierarchy. Since the root is involved in the relabeling, intervals are computed under a mutex

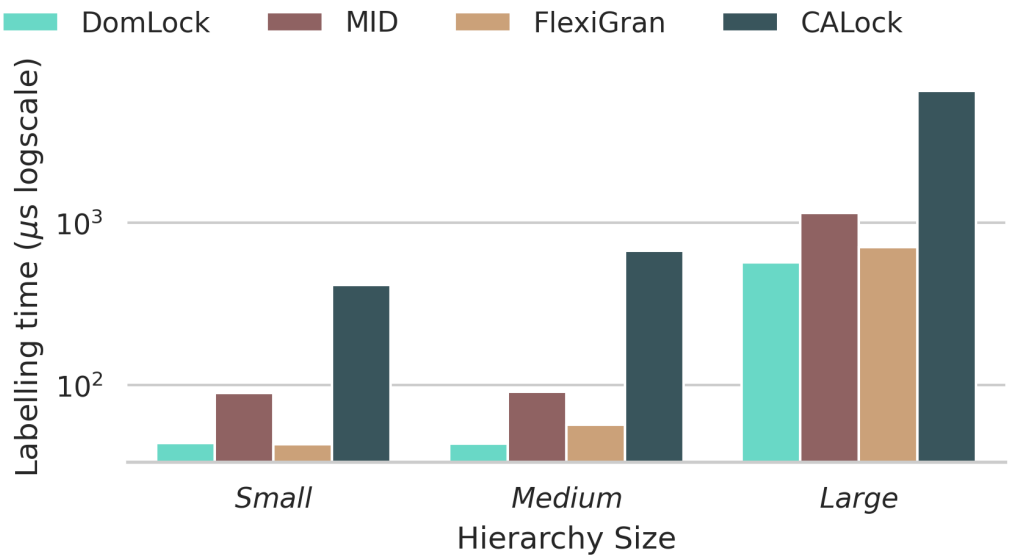


Fig. 7.3: Time to compute initial labels (lower is better)

on the graph which prevents parallel relabelling of disjoint subgraphs leading to poor performance of DomLock, MID and FlexiGran.

In contrast, CALock relabels only the affected subgraphs directly affected by a structural modification. CALock labels always popagate away from the root towards the leaves and are computed under the lock that is acquired to perform the structural modification. Thus, multiple grains can be locked, modified and relabelled in parallel. CALock is 2× faster at relabelling than DomLock, MID and FlexiGran.

7.4 Metadata size in memory

All MGL techniques utilise metadata to identify the lock grains. DomLock utilizes integer ranges as labels, which require less memory due to their compact representation. MID uses two pairs of integer ranges to represent the intervals of the vertices. FlexiGran uses the DomLock intervals along with an integer to store the level of a vertex. In contrast, CALock employs sets of vertex identifiers as labels, leading to a larger memory footprint. As shown in Figure 7.5, CALock's metadata consumes approximately 1.5× more memory than DomLock, MID and FlexiGran. In DomLock, MID and FlexiGran, regardless of the topology of the graph, the label at each vertex consists of integers. This has low memory requirement however, the information about the exact topology of the graph is lost, leading to false subsumptions. CALock

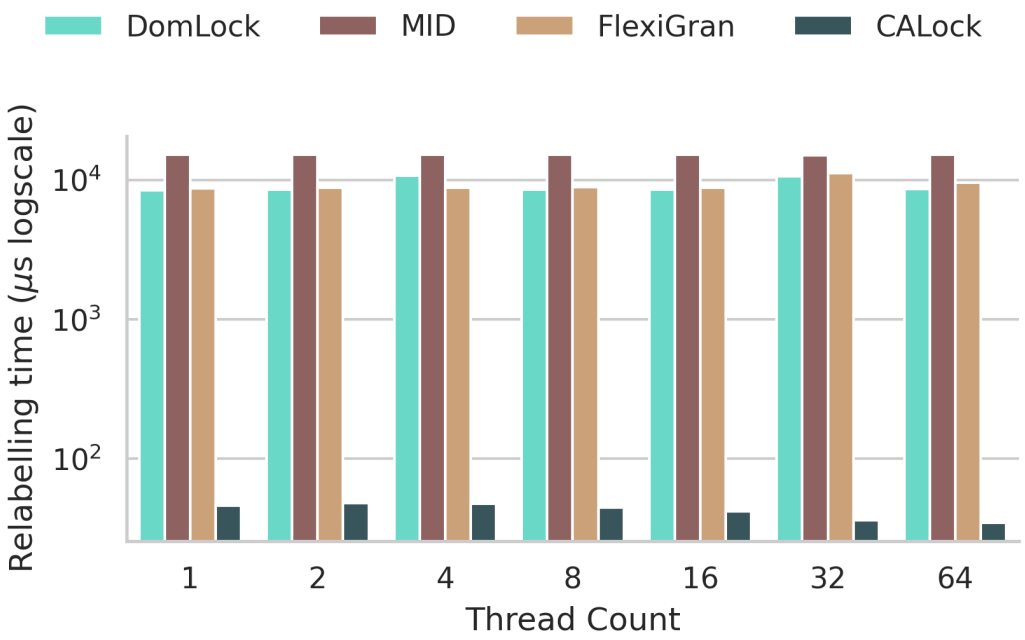


Fig. 7.4: Time spent relabelling the graph per modification operation (lower is better)

labels, being sets, contain more information allowing for smaller grain sizes and prevent the problem of false subsumptions.

7.5 Lock granularity and false subsumptions

Locking a vertex in MGL implicitly also locks all other vertices present in the lock grain. To compare the grain size for them, we measure the number of targets implicitly locked for a guard vertex in the STMBench7 graph. Figure 7.6 compares the granularity of DomLock, MID, FlexiGran and CALock.

Locking an atomic part has almost the same effect for all techniques with CALock being marginally better and reducing grain sizes. This is because, in STMBench7, the atomic parts are strongly connected and locking a single vertex often causes the whole graph of atomic parts to be locked. When locking higher up in the graph, the effects vary. Locking a complex assembly is more expensive with intervals from DomLock, MID and FlexiGran because complex assemblies often share composite parts leading to large grains due to false subsumptions. When locking base assemblies, with DomLock, MID and FlexiGran, multiple base assemblies are locked due to the one-to-many relationship between base assemblies and composite parts, again, owing to false subsumptions.

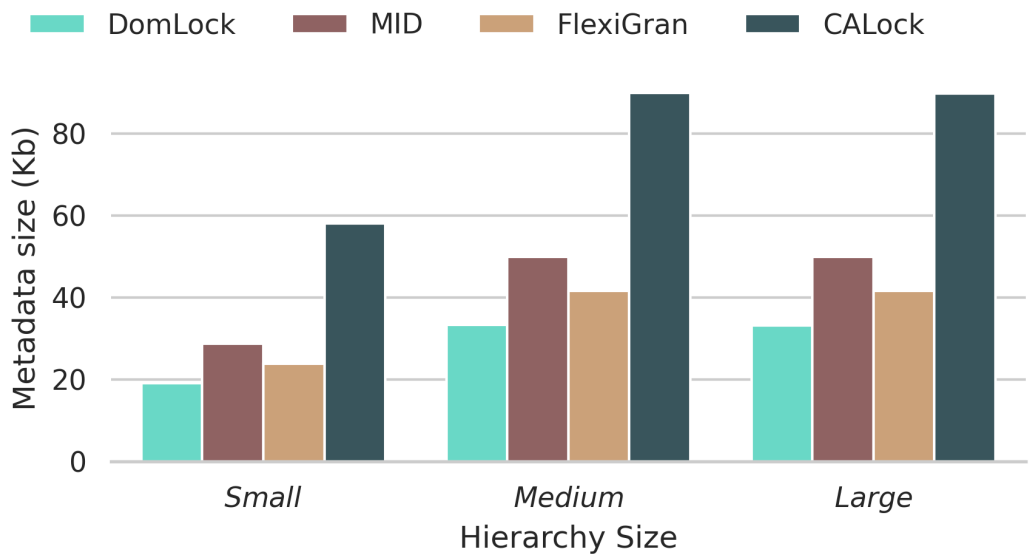


Fig. 7.5: Size of the metadata used for labelling

The granularity of the lock and its effect on subsumption is highly dependent on the topology of the graph. False subsumptions aggravate the problem of large grains and lead to more grain overlaps between threads. The grain sizes of CALock are always smaller other locking techniques.

7.6 Overall locking performance

Sections 7.2, 7.3, 7.4 and 7.5 study individual parameters in isolation. Now, we study all of them together and evaluate their affects on overall performance of the locking techniques. For FlexiGran, the percentage of fine grained locks is set to 50% as recommended by the authors in their paper [AN24]. In these set of benchmarks, we study both static and dynamic graphs in STMBench under the same workloads. Figures 7.7 and 7.8 show the throughput of different workloads on static graphs and dynamic graphs respectively. The charts in these figures are plotted with the number of concurrent threads on the x-axis and the throughput (op/s) or response time (ns) on the y-axis. Response time is measured from when the thread issues a lock request until the lock is granted.

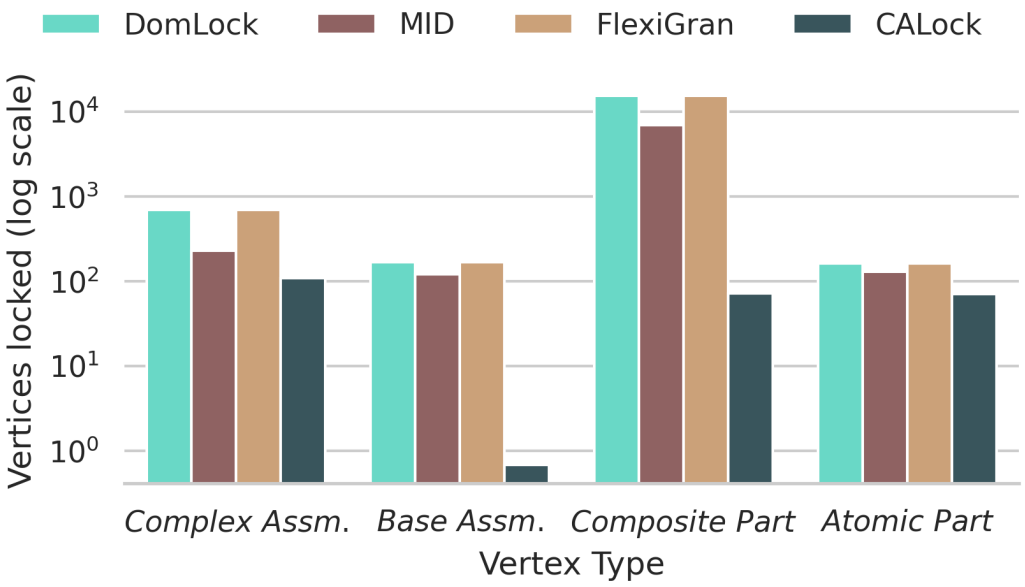


Fig. 7.6: Vertices locked per vertex type (lower is better)

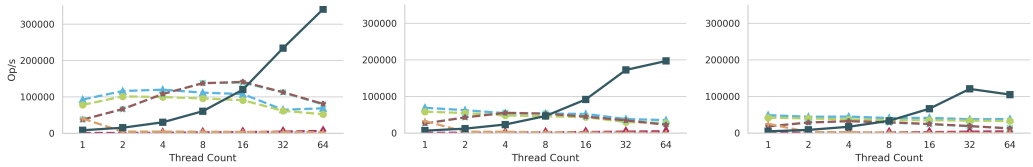
7.6.1 Static Graphs

Throughput figures 7.7a, 7.7b and 7.7c show that coarse-grain and medium-grain locks perform the best for up to 4 concurrent threads due to the additional computation of the lock grain required for DomLock, MID, FlexiGran and CALock. Beyond 4 threads, MGL techniques give better throughput. However, the performance of DomLock and MID stagnates at 8 threads.

The graph in STMBench is irregular and has multiple paths leading to a vertex. Due to this, the lock grain sizes are uneven and DomLock, MID and Flexigran intervals often lead to false subsumptions which block a thread even when there is no conflict (see Section 7.5). Intention locks often deadlock due to these irregular paths leading to poor performance(see Section7.2).

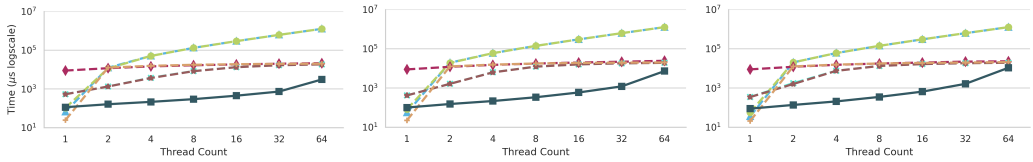
CALock successfully minimises the size of the lock grains and allows threads to lock disjoint grains in parallel achieving better scalability than DomLock, MID and FlexiGran. We observe that CALock is 2x faster than DomLock for 32 threads and 4.5x faster than DomLock for 64 threads.

Observe that as the ratio of writes in a workload increases, the maximum throughput of any locking mechanism decreases because of inherent contention due to conflicts between write lock requests.



(a) R:90%,W:10% (b) R:60%,W:40% (c) R:10%,W:90%

7.7a, 7.7b, 7.7c: Throughput (higher is better)



(d) R:90%,W:10% (e) R:60%,W:40% (f) R:10%,W:90%

7.7d, 7.7e, 7.7f: Average time to grant lock request (lower is better)

Fig. 7.7: Performance with different workload types on static graphs (R: reads, W: writes).

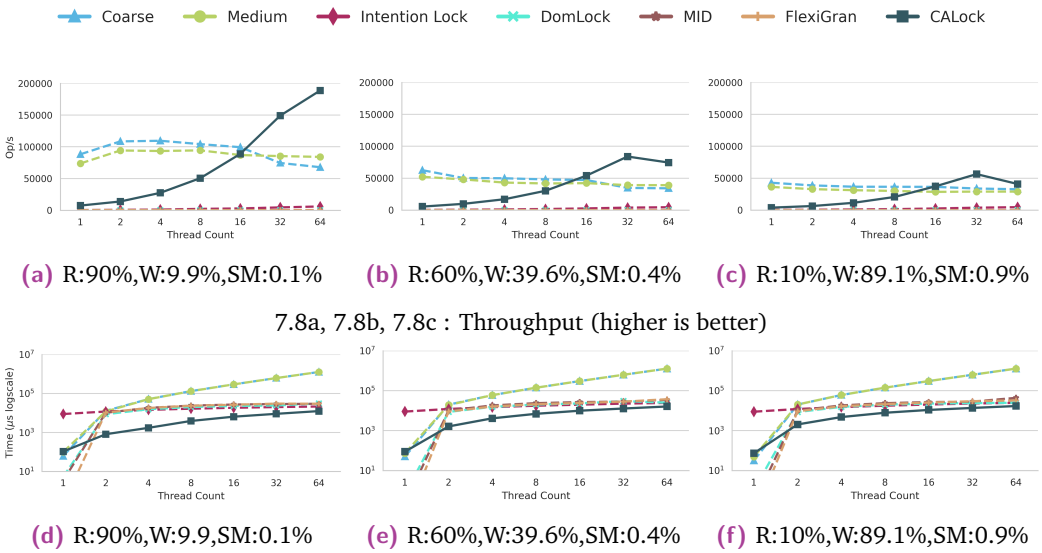
Response time figures 7.7d, 7.7e and 7.7f plot the wait time per thread between coarse grain locks, medium grain locks, Intention locks, DomLock, MID, FlexiGran and CALock. Due to the static nature of coarse and medium lock grains as the number of threads increases, the response time also increases due to the increase in conflicts. For a single thread, coarse and medium locks are the fastest but with any amount of parallelism, their performance suffers.

Between the MGL techniques, FlexiGran and Intention locks on average take the longest to grant a lock. With intention locks, this is because of the expensive traversals required to place intention tags on vertices. With FlexiGran, lock conflict detection is very expensive because of the coexistance of MGL and fine grained locks. DomLock and MID are faster than FlexiGran but remain significantly slower than CALock.

In interval based MGL techniques like DomLock, MID and FlexiGran, once a lock request identifies an interval it wishes to lock, the thread traverses the hierarchy to find the lock guard, which is a vertex with the requested interval (resp. interval pair for MID). This traversal is especially expensive when locking deeper in the hierarchy. CALock on the other hand uses a set intersection to find the LGCA which is the ID of the vertex that needs to be locked. In doing so, CALock avoids traversals alltogether, giving faster response times for lock requests.

The overall wait time increases with the number of concurrent threads because of an increase in the number of conflicts due to overlapping grains and the expensive conflict detection. For 64 threads, CALock is 6× faster than DomLock in a read-dominated load and 1.5× faster in a write-dominated load.

7.6.2 Dynamic Graphs



(a) R:90%,W:9.9%,SM:0.1% **(b)** R:60%,W:39.6%,SM:0.4% **(c)** R:10%,W:89.1%,SM:0.9%

7.8a, 7.8b, 7.8c : Throughput (higher is better)

(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%

7.8d, 7.8e, 7.8f: Average time to grant lock request (lower is better)

Fig. 7.8: Performance with different workload types on dynamic graphs (R: reads, W: writes; SM: structural modifications).

With structural modifications, the topology of the graph changes, triggering relabelling for MGL locking techniques like DomLock, MID, FlexiGran and CALock. Structural modifications happen under a mutex for coarse grain locks, medium grain locks, DomLock, MID and FlexiGran. Throughput figures 7.8a, 7.8b and 7.8c show the performance of locking algorithms under reads and writes that interleave with structural modifications leading to relabelling of the graph. In these workloads, 10% of the total writes are structural modifications. In write-heavy workloads (Throughput figure 7.8c), structural modifications can be as high as 0.9%. While in read-heavy workloads (Throughput figure 7.8a), they are as low as 0.1%. Even under a small load of structural modifications, we observe that coarse locks, medium locks do not scale well as the number of threads increases. Intention locks are no different. While structural modifications do not require any additional care in intention locks, the cost of placing intention tags on all vertices is very high.

DomLock, MID and FlexiGran perform significantly worse compared to other algorithms because along with the lack of parallelism, an additional relabelling step is required for each structural modification and this relabelling happens under a mutex. This is shown in the curves in Throughput figures 7.8a, 7.8b and 7.8c for DomLock, MID and FlexiGran, which are relatively flat, indicating a lack of scalability. CALock can parallelize structural modifications and is 2x faster than coarse and medium-grained locks and about 8x faster than DomLock, MID and FlexiGran.

Response time for Intention locks is independent of the dynamicity of graph topology. However, response time for DomLock, MID and FlexiGran is longer in dynamic graphs compared to static graphs, as shown in Response time figures 7.8d, 7.8e and 7.8f. Response time for CALock also increases for dynamic graphs when compared to static graphs but CALock remains faster than all other lock techniques for even the most contended workload.

7.7 Summary of experimental results

7.7.1 Discussion

The initial labelling time is proportional to the size of the graph but in conjunction with other benchmark results, CALock is a better choice for runtime performance since the fast integer labels of DomLock, MID and FlexiGran require more expensive relabelling when the graph is modified by structural modification operations (see Section 7.3).

7.7.2 Summary

With the results from the experiments using STMBench7 and microbenchmarks, we can derive the following conclusions.

- Intention locks are the most expensive locking technique for any workload and are not suitable for irregular graphs.
- CALock is faster than DomLock, coarse-grain and medium-grain locks for any workload that involves more than 8 concurrent threads.
- In workloads with low contention on static graphs, CALock is at least 3× faster compared to any other locking technique.
- In workloads with low contention and dynamic graphs, CALock is 1.5× faster than coarse-grain and medium-grain locks and 8x faster than DomLock, MID and FlexiGran.
- In workloads with high contention on static graphs, CALock is 2× faster than coarse-grained and medium-grained locks and 4× faster than DomLock, MID and FlexiGran

- In workloads with high contention and dynamic graphs, CALock performs as good as coarse-grain and medium-grain locks and 4× faster than DomLock, MID and FlexiGran
- CALock is an appropriate locking technique if the graph is irregular and dynamic (which makes DomLock, MID and FlexiGran unsuitable).

8 Conclusion

8.1 Summary of Findings

Recap of the key findings and achievements of the research.

8.2 Contributions

Reiteration of the thesis contributions in the context of the broader field.

8.3 Future Work

Suggestions for further research and potential enhancements of CALock.

8.4 Final Remarks

Reflection on the research journey and its impact.

Bibliography

[ABS99] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999 (cit. on p. 13).

[AN24] Anju Mongandampulath Akathoott and Rupesh Nasre. "FlexiGran: Flexible Granularity Locking in Hierarchies". In: *Proceedings of the Euro-Par 2024 Conference*. Madrid, Spain: Springer LNCS, 2024, pp. 0–0 (cit. on pp. 8, 27, 49, 56).

[AN22] M. A. Anju and Rupesh Nasre. "Multi-Interval DomLock: Toward Improving Concurrency in Hierarchies". In: *ACM Trans. Parallel Comput.* 9.3 (2022), 12:1–12:27 (cit. on pp. 8, 25, 49).

[BS77] Rudolf Bayer and Mario Schkolnick. "Concurrency of Operations on B-Trees". In: *Acta Informatica* 9 (1977), pp. 1–21 (cit. on p. 14).

[Bla98] K. R. Blackman. "Technical note: IMS celebrates thirty years as an IBM product". In: *IBM Systems Journal* 37.4 (1998), pp. 596–603 (cit. on p. 13).

[CDN93] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. "The oo7 Benchmark". In: *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*. Ed. by Peter Buneman and Sushil Jajodia. ACM Press, 1993, pp. 12–21 (cit. on p. 49).

[CC16] Fernando Miguel Carvalho and João Cachopo. "Optimizing memory transactions for large-scale programs". In: *Journal of Parallel and Distributed Computing* 89 (Mar. 2016), pp. 13–24 (cit. on p. 49).

[CM96] Joseph Cheriyan and Kurt Mehlhorn. "Algorithms for Dense Graphs and Networks on the Random Access Computer". In: *Algorithmica* 15.6 (1996), pp. 521–549 (cit. on p. 36).

[Dat00] C. J. Date. *An introduction to database systems (7. ed.)* Addison-Wesley-Longman, 2000 (cit. on p. 13).

[Eul41] Leonhard Euler. "Solutio problematis ad geometriam situs pertinentis". In: *Commentarii academiae scientiarum Petropolitanae* (1741), pp. 128–140 (cit. on p. 45).

[Fel+16] Pascal Felber, Shady Issa, Alexander Matveev, and Paolo Romano. "Hardware read-write lock elision". en. In: *Proceedings of the Eleventh European Conference on Computer Systems*. London United Kingdom: ACM, Apr. 2016, pp. 1–15 (cit. on p. 49).

[FB15] Ricardo Filipe and João Barreto. "Nested Parallelism in Transactional Memory". en. In: *Transactional Memory. Foundations, Algorithms, Tools, and Applications: COST Action Euro-TM IC1001*. Ed. by Rachid Guerraoui and Paolo Romano. Cham: Springer International Publishing, 2015, pp. 192–209 (cit. on p. 49).

[FH10] Johannes Fischer and Daniel H. Huson. “New common ancestor problems in trees and directed acyclic graphs”. In: *Inf. Process. Lett.* 110.8-9 (2010), pp. 331–335 (cit. on p. 34).

[GKN18] K. Ganesh, Saurabh Kalikar, and Rupesh Nasre. “Multi-granularity Locking in Hierarchies with Synergistic Hierarchical and Fine-Grained Locks”. In: *Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings*. Ed. by Marco Aldinucci, Luca Padovani, and Massimo Torquati. Vol. 11014. Lecture Notes in Computer Science. Springer, 2018, pp. 546–559 (cit. on p. 49).

[Geo11] Lars George. *HBase - The Definitive Guide: Random Access to Your Planet-Size Data*. O’Reilly, 2011 (cit. on p. 13).

[Gra+08] Tracy Grauman, Stephen G. Hartke, Adam S. Jobson, et al. “The hub number of a graph”. In: *Inf. Process. Lett.* 108.4 (2008), pp. 226–228 (cit. on p. 36).

[Gra+75] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. “Granularity of Locks in a Large Shared Data Base”. In: *Proceedings of the International Conference on Very Large Data Bases, September 22-24, 1975, Framingham, Massachusetts, USA*. Ed. by Douglas S. Kerr. ACM, 1975, pp. 428–451 (cit. on pp. 8, 15, 16).

[GKV07] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. “STMBench7: a benchmark for software transactional memory”. In: *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23, 2007*. Ed. by Paulo Ferreira, Thomas R. Gross, and Luís Veiga. ACM, 2007, pp. 315–324 (cit. on p. 49).

[KN16] Saurabh Kalikar and Rupesh Nasre. “DomLock: a new multi-granularity locking technique for hierarchies”. In: *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*. Ed. by Rafael Asenjo and Tim Harris. ACM, 2016, 23:1–23:12 (cit. on pp. 8, 22, 49).

[KN18] Saurabh Kalikar and Rupesh Nasre. “NumLock: Towards Optimal Multi-Granularity Locking in Hierarchies”. In: *Proceedings of the 47th International Conference on Parallel Processing, ICPP 2018, Eugene, OR, USA, August 13-16, 2018*. ACM, 2018, 75:1–75:10 (cit. on p. 49).

[KN19] Saurabh Kalikar and Rupesh Nasre. “Toggle: Contention-Aware Task Scheduler for Concurrent Hierarchical Operations”. In: *Euro-Par 2019: Parallel Processing - 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26-30, 2019, Proceedings*. Ed. by Ramin Yahyapour. Vol. 11725. Lecture Notes in Computer Science. Springer, 2019, pp. 142–155 (cit. on p. 8).

[KPR15] Junwhan Kim, Roberto Palmieri, and Binoy Ravindran. “On Scheduling in Distributed Transactional Memory: Techniques and Tradeoffs”. en. In: *Handbook on Data Centers*. Ed. by Samee U. Khan and Albert Y. Zomaya. New York, NY: Springer, 2015, pp. 1267–1283 (cit. on p. 49).

[LY81] Philip L. Lehman and S. Bing Yao. “Efficient Locking for Concurrent Operations on B-Trees”. In: *ACM Trans. Database Syst.* 6.4 (1981), pp. 650–670 (cit. on p. 14).

[LHN19] Viktor Leis, Michael Haubenschild, and Thomas Neumann. “Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method”. In: *IEEE Data Eng. Bull.* 42.1 (2019), pp. 73–84 (cit. on p. 14).

[LZ14] Peng Liu and Charles Zhang. “Unleashing concurrency for irregular data structures”. In: *36th International Conference on Software Engineering, ICSE ’14, Hyderabad, India - May 31 - June 07, 2014*. Ed. by Pankaj Jalote, Lionel C. Briand, and André van der Hoek. ACM, 2014, pp. 480–490 (cit. on p. 49).

[Pro+19] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, et al. “Renaissance: benchmarking suite for parallel applications on the JVM”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. New York, NY, USA: Association for Computing Machinery, June 2019, pp. 31–47 (cit. on p. 49).

[Ray13] Michel Raynal. *Concurrent Programming - Algorithms, Principles, and Foundations*. Springer, 2013 (cit. on p. 11).

[RS77] Daniel R. Ries and Michael Stonebraker. “Effects of Locking Granularity in a Database Management System”. In: *ACM Trans. Database Syst.* 2.3 (1977), pp. 233–246 (cit. on p. 16).

[RC14] Hugo Rito and João Cachopo. “ProPS: A Progressively Pessimistic Scheduler for Software Transactional Memory”. en. In: *Euro-Par 2014 Parallel Processing*. Ed. by Fernando Silva, Inês Dutra, and Vítor Santos Costa. Cham: Springer International Publishing, 2014, pp. 150–161 (cit. on p. 49).

[Sha81] Micha Sharir. “A strong-connectivity algorithm and its applications in data flow analysis”. In: *Computers & Mathematics with Applications* 7.1 (1981), pp. 67–72 (cit. on p. 36).

[Tar72] Robert Endre Tarjan. “Depth-First Search and Linear Graph Algorithms”. In: *SIAM J. Comput.* 1.2 (1972), pp. 146–160 (cit. on p. 36).

[Sql] *Transaction locking and Row Versioning Guide - SQL Server*. 2022 (cit. on p. 16).

[Val+16] Tiago M. Vale, João A. Silva, Ricardo J. Dias, and João M. Lourenço. “Pot: Deterministic Transactional Execution”. en. In: *ACM Transactions on Architecture and Code Optimization* 13.4 (Dec. 2016), pp. 1–24 (cit. on p. 49).