Multi-granularity Locking in Hierarchies with Synergistic Hierarchical and Fine-Grained Locks

K. Ganesh, Saurabh Kalikar(✉), and Rupesh Nasre

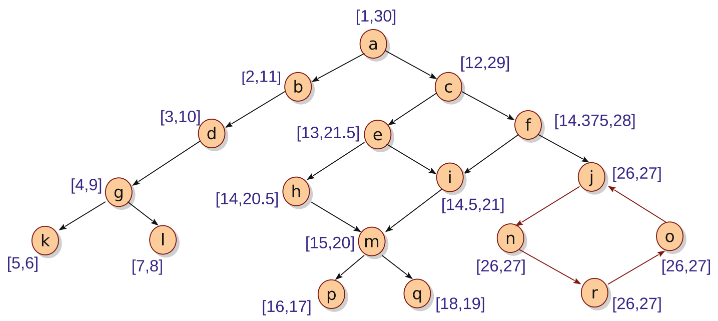
CSE, IIT Madras, Chennai, India
{cs16m006,saurabhk,rupesh}@cse.iitm.ac.in

Abstract. We propose a new locking mechanism for hierarchies wherein the locking requests can be a combination of coarse and fine. Existing protocols such as multiple-granularity locking (MGL) are efficient when all the requests are of the same granularity. MGL is either too coarse or too fine-grained when multiple threads request for various parts of the hierarchy with differing granularity requirements. Simultaneous handling of hierarchical and fine-grained requests poses new challenges in checking for racy requests. We propose a novel indexing technique for hierarchies which uniquely identifies every node as an interval value and effectively captures hierarchical dependencies between nodes even when the hierarchy is a tree, DAG or a cycle. Our experiments with real-world XML hierarchies and synthetic benchmarks show that the proposed locking technique provides a higher degree of concurrency with minimal locking cost resulting in overall performance improvement.

1 Introduction

One of the main challenges in developing a multi-threaded parallel application is the design of an efficient synchronization mechanism for shared data structures. *Lock* constructs are widely used for thread synchronization. The nature of data structures and their associated operations necessitate the use of various locking protocols. In the context of shared data structures, *hierarchies* are special linked structures, where each child node denotes a specialization or a part of its parents. For instance, a node representing a *department* in an academic hierarchy is a part of its parent *institute*. Conversely, a node representing an *institute* contains all its *departments*. In the concurrent setting, operating on different nodes in such a hierarchy is achieved using traditional fine-grained locking, which maintains a lock with each node. While fine-grained locks ensure consistency of the data structure, it also poses scalability challenges in the presence of a large number of threads and unpredictable locking request pattern. For instance, in fine-grained locking, an operation such as *calculate GPA for all the students in department*

© Springer International Publishing AG, part of Springer Nature 2018
M. Aldinucci et al. (Eds.): Euro-Par 2018, LNCS 11014, pp. 546–559, 2018.
https://doi.org/10.1007/978-3-319-96983-1_39



	S	X	IS	IX
S	✓	✗	✓	✗
X	✗	✗	✗	✗
IS	✓	✗	✓	✓
IX	✗	✗	✓	✓

Fig. 1. (a) Example hierarchy along with its HiFi numbering (b) Compatibility matrix for intention locking protocol

CS needs to acquire a lock on *each student* separately. This is clearly inefficient and happens because locking cost is proportional to the number of students.

In this example, it seems logical to acquire a single lock on the department CS, and process all the student records. This is achieved using hierarchical locking or multi-granularity locking. Multi-granularity locking (MGL) protocols [7,9] ensure that if a node in a hierarchy (say, CS department) is locked, then every node reachable from the locked node (i.e., every student, faculty and staff member) is also implicitly locked.

On the other hand, there exist several operations that do not require hierarchical locks. For instance, a fine-grained operation such as *update class-room-count = 20 where department = CS* does not need the whole department (students, staff, faculty) to get locked. Existing approaches do not support co-existence of hierarchical and fine-grained locks. Thus, all the MGL locks are either purely hierarchical (e.g., even to update number of classrooms), or purely fine-grained (e.g., even to update GPA of all the students). The former is too conservative leading to reduced concurrency, while the latter is too precise leading to locking overheads.

To address these issues, this paper makes the following contributions:

1. We propose a novel indexing technique which allows quick checking of overlaps between two thread requests in a hierarchy. The indexing has useful properties which can be independently exploited for other applications.
2. We propose HiFi, a locking protocol that allows synergistic co-existence of fine-grained and hierarchical locks. The protocol crucially relies on the new indexing mechanism and offers more concurrency.
3. We illustrate that HiFi considerably improves the parallel performance of the underlying application. Using real-world XML hierarchies and synthetic datasets, we identify scenarios where HiFi is a better choice for locking.

2 Background and Motivation

In this section, we provide a brief background on a *de facto* hierarchical locking technique and highlight its limitations. To motivate HiFi, we use the example hierarchy from Fig. 1 which contains 18 nodes spread across six levels. The hierarchy is carefully crafted to contain paths, tree-like substructure, as well as DAGs and a cycle. Most real-world hierarchies we have come across are acyclic.

2.1 Hierarchical Locking

Hierarchical locking is a way to lock a node in a hierarchy which implicitly locks its descendants. This is useful because the whole sub-hierarchy rooted at a node can be protected using a single lock. A node could be hierarchically locked only if (i) it is not locked by any other thread, and (ii) none of its descendants are currently locked by any other thread, and (iii) none of its ancestors are currently locked by any other thread. For instance, in Fig. 1, hierarchical locking of node g requires that no other thread currently holds locks on (i) g itself, (ii) its descendant nodes k and l, and (iii) its ancestors a, b, and d. Clearly, the naïve mechanism of traversing through the descendants and ancestors for *every lock request* works, but is impractical. In practice, database management systems such as MySQL use an optimized traversal technique called *intention locking.*

Intention Locks. Traditionally, hierarchical locking is implemented using *intention locks* [7]. Unlike traversing sub-hierarchies, intention locking technique *marks* all the ancestors of the targeted node before acquiring a hierarchical lock. These markers serve as indicators to other concurrent threads, that there exists a node along this path on which a lock has been acquired. These markers are nothing but the intention locking modes, i.e., IS and IX of conventional *shared* (S) and *exclusive* (X) locks respectively. Thus, before locking any node in S or X mode, a thread has to lock all the ancestors (i.e., all the reference paths from root) in IS or IX modes respectively. For example, before locking node g in X mode, a thread must lock its ancestor nodes a, b and d in IX mode. Possible locking modes and their inter-operability are shown in the compatibility matrix 1. Intention locks is an effective way to achieve hierarchical locking for tree structures, as each tree node has a single reference path from the root. However, in case of directed graph structures (such as DAGs), a node may have multiple reference paths from the root necessitating intention locking across *each* reference path. For instance, for locking node i, we need to traverse the hierarchy to mark intention along each path a-c-e and a-c-f. For a large real-world hierarchy, such a traversal is costly, and increases the thread waiting time. In summary, intention locks are ill-suited for complex hierarchical structures.

Motivating Example. Consider a thread T_1 currently holding a hierarchical lock on node g for *exclusive access* and another thread T_2 which wants to perform a simple fine-grained update operation on node d. Intuitively, such a concurrent operation seems plausible. However, intention locking (IL) protocol does not support it, as MGLs support only hierarchical locking. Thus, locking node d using IL protocol also locks g. IL can be extended to support extra modes for fine-grained (shared and exclusive) locking. Thus, in this extension, a thread would also mark along the path whether it wants to lock the target node in fine-grained mode or hierarchical mode. Unfortunately, such an extension takes away the very benefits of fine-grained locking – although the update is local (fine-grained), the thread needs to traverse the hierarchy of ancestors. This poses several scalability challenges when both hierarchical and fine-grained locking modes are desired. To address this, we need a mechanism that (i) supports efficient co-existence of fine-grained and hierarchical locks, and (ii) allows quick checking of overlap between two lock requests (e.g., whether a node is already locked in a fine-grained manner within a sub-hierarchy which is to be locked in hierarchical mode).

In this paper, we propose a new locking protocol which supports efficient handling of fine-grained locks in presence of hierarchical (MGL) locks.

3 Our Proposal: HiFi

We design a new protocol that allows maximum concurrency; that is, if two lock requests do not overlap, they can be executed in parallel (assuming the availability of enough computing resources). Central to our method is a new interval numbering technique which converts the structural overlap between sub-hierarchies to interval overlap between numbers, which allows fast overlap check, retaining the benefits of fine-grained locking.

Overview. Our proposed numbering is shown for the example hierarchy in Fig. 1. Each node is assigned an interval [low, high] where low and high are floating-point numbers. The numbering of nodes in a hierarchy follows the following invariants:

$\mathcal{I}1$ The interval of each node is unique if the node is not part of a cycle. All the nodes in a cycle have the same interval.
$\mathcal{I}2$ The interval of every ancestor *strictly subsumes* the interval of each of its (transitive) descendants. Interval [a, b] *strictly subsumes* interval [c, d] iff $a < c$ and $b > d$.
$\mathcal{I}3$ Intervals of two nodes partially overlap if they have a common descendant.

For instance, in Fig. 1, each node has a unique interval value, except for the cycle nodes j, n, r, o, which all have the same interval [26, 27], validating Invariant $\mathcal{I}1$. Also, interval of node e, which is [13, 21.5], subsumes those of its descendants h, i, m, p, q, validating Invariant $\mathcal{I}2$. The root subsumes the intervals of all other nodes in the hierarchy (except when the root itself is part of a cycle). Similarly, each leaf node has a non-overlapping interval with other leaves. We can observe that nodes h and f have overlapping intervals that do not subsume one another. The overlap is justified by the common descendants m, p, q, thereby validating Invariant $\mathcal{I}3$. Note that although h and f do not subsume one another's intervals, they both individually subsume the intervals of m, p, and q, due to Invariant $\mathcal{I}2$.

Using such an interval numbering, HiFi can quickly check if two hierarchies overlap. Thus, the proposed numbering acts as an alternative to IL. In other words, if there are two *hierarchical* locking requests, HiFi can exploit the proposed numbering mechanism to identify if the two requests can be simultaneously satisfied. For instance, if thread T1 wants to lock the hierarchy rooted at

h, and thread T2 wants to lock the hierarchy rooted at j, then HiFi can check their intervals [14, 20.5] and [26, 27] which do not overlap, and permit access. In contrast, if thread T2 wants to lock the hierarchy rooted at node i, then HiFi can check the intervals [14, 20.5] and [14.5, 21] which overlap, and disallow one of the locking requests (the other thread needs to block or try again later). Note that, unlike in IL, HiFi protocol does not need to traverse the hierarchy to identify overlap. This considerably improves the performance of the underlying application – which usually has locking in its critical path. Once the hierarchy is numbered as a pre-processing step, all runtime locking requests can be quickly served.

3.1 Compatibility in HiFi

Assuming a numbering such as the previous subsection exists, we now describe how fine-grained and hierarchical locks can co-exist in HiFi. Note that the locking request could be shared or exclusive, and fine-grained or hierarchical.

Figure 2 shows the compatibility matrix for various locking scenarios in HiFi. Using this matrix, our locking methodology allows/disallows locking the input set of nodes in the given mode (S/X, fine/hierarchical). For instance, in our running example from Fig. 1, if thread T1 has locked descendant node g in f_x mode, and thread T2 requests node d in H_s mode, then the requests being incompatible according to the matrix, would be denied concurrent access. In contrast, if T1 has locked node g in H_x mode, and T2 requests node d in f_s mode, then the matrix deems these operations compatible with each other for concurrent execution. Compared to the original compatibility matrix from Fig. 1, clearly inter-operability of fine-grained and hierarchical locks allows more concurrent operations.

	Lock held by							
	Ancestor				Descendant			
Lock requested	f_s	f_x	H_s	H_x	f_s	f_x	H_s	H_x
f_s	✓	✓	✓	✗	✓	✓	✓	✓
f_x	✓	✓	✗	✗	✓	✓	✓	✓
H_s	✓	✓	✓	✗	✓	✗	✓	✗
H_x	✓	✓	✗	✗	✗	✗	✗	✗

Fig. 2. Lock compatibility matrix in HiFi (Legend: H = Hierarchical, f = Fine-grained, s = shared, x = exclusive)

3.2 Numbering Algorithm

We now describe our numbering algorithm in detail. We first attend to cyclic substructures in the hierarchy, if present. Unlike trees and DAGs, the nodes forming cycles in a hierarchy do not have well-defined ancestor-descendant relationship among themselves, that is, every node is an ancestor as well as a descendant of every other node. For instance, nodes j, n, r and o are part of a cycle and each of them is assigned the same interval [26, 27]. Therefore, it is logical to treat all the nodes forming a cycle as one entity, with a single interval value.

Our interval-numbering algorithm has two passes. In the first pass, we perform a conventional depth-first traversal (DFS) from the root node and track pre-visit and post-visit numbers of each node in the hierarchy. The advantage of DFS-based numbering is that the intervals obtained using the pre- and post-visit numbers [pre, post] satisfy *all* the three invariants when the underlying hierarchy is a tree. This can be validated from the left subtree of the root node a in our running example from Fig. 1. Such a numbering, however, does not satisfy the invariants in case of DAGs because DFS does not explore the already visited nodes. Due to this, the numbering may miss out on some descendants, essentially failing to satisfy invariants $\mathcal{I}2$ and $\mathcal{I}3$.

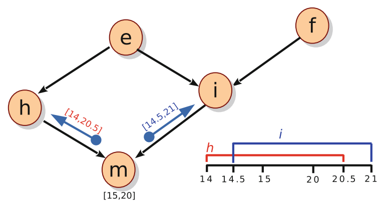
Meeting the invariants necessitates a bottom-up propagation of [pre, post] intervals from leaf nodes towards the root. Such a propagation re-adjusts the intervals of the ancestors to satisfy all the invariants.

Interval Propagation. Intervals are propagated in bottom-up fashion, therefore the intervals of leaf nodes remain unchanged to [pre, post] intervals according to DFS traversal. According to Invariant $\mathcal{I}2$, a node's interval must *strictly* subsume those of all its descendants. Therefore, the smallest *integer* interval which subsumes the interval of the child nodes is assigned to the parent. For instance, with k [5, 6] and l [7, 8] as children, g's interval becomes[4, 9]. Similarly, the intervals of nodes d, b and m become [3, 10], [2, 11] and [15, 20] respectively.

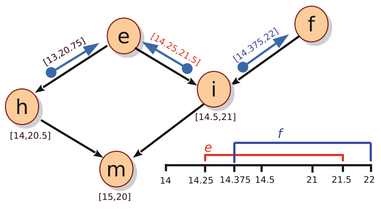
Such a mechanism works well when the sub-hierarchy is a tree. When a node has more than one parent (e.g. node m), the subhierarchy is no longer a tree and it poses interesting challenge to the numbering algorithm. The interval assigned to each parent should be such that (i) it strictly subsumes the children's interval, and (ii) the parent intervals must overlap with each other but not subsume one another. Thus, intervals of nodes h and i must overlap with each other, but not subsume each other; however, they both should subsume m's interval. To satisfy these conditions, we exploit the range of floating point numbers. For instance, we assign intervals [14, 20.5] to h and [14.5, 21] to i which both subsume m's interval [15, 20] and also have partial overlap with each other.

The propagation faces another difficulty due to strict subsumption. Since the intervals of the ancestors are always larger than those of the children, it can happen that an ancestor's interval becomes so big that it subsumes that of another of its sibling! For instance, when i propagates its interval [14.5, 21] to its parent node f, strict subsumption property can assign interval [13, 22] to f. While this does satisfy Invariant $\mathcal{I}2$ between i and f, it also falsely satisfies the invariant between h and f ([13, 22] subsumes [14, 20.5]). Note that f is not an ancestor of h, and hence their intervals should not subsume each other. This necessitates *limiting* the interval propagation to parents (as explained next).

Maintenance of Locks. To check if the new locking request overlaps with an existing one, HiFi needs to track the currently locked nodes (along with their type S/ X and fine-grained / hierarchical). We use the lock-pool imple-



(a) m sends updates



(b) i sends updates

Fig. 3. Example of interval propagation

Algorithm 1. Bottom-up traversal (invoked with ROOT as the parameter)

```
1 Function bottomUp(root):
2     if root.isLeafNode() then root.lowLimit = root.upLimit = $-1$ ;
3     else
4         forall the $c \in$ root.children do
5             if $c$.explored == $false$ then bottomUp($c$);
6         root.mergeIntervals()
7     sendPartitionLimits(root.lowLimit, root.upLimit, root)
8     root.explored = $true$
```

mentation from DomLock [9], which maintains per-thread lock information in a table. A thread needs to check the complete table before inserting its entry, which requires heavy synchronization. To counter this inefficiency, the lock-pool exploits sequence locks to make the reading synchronization-free.

3.3 Main Algorithm

We now present Algorithm 1 for interval numbering and explain it using our running example. After the first phase of DFS pre-post visit numbering for the leaf nodes, we call Algorithm 1 with the root of the hierarchy as a parameter. Algorithm recursively traverses down to the leaf nodes and starts back-propagation of intervals to respective parents (line 2). We use two flags lowLimit and upLimit to indicate the restrictions placed on the interval updates that a node can send to its parent(s). For instance, starting from the root node a in Fig. 1, the algorithm recursively descends to the leaf node k and assigns default limit of -1. Node k with interval [5, 6] invokes the method `sendPartitionLimits` to propagate the limits upward (line 7). The function `sendPartitionLimits` is presented in Algorithm 2. It partitions an interval into np (number of parents) partially overlapping intervals, each of which strictly subsumes the child's interval and is within the range of lowLimit and upLimit. Thus, Algorithm 2 ensures that the interval updates sent to the parents conform to the invariants $\mathcal{I}1 - \mathcal{I}3$.

For instance, When k invokes the method to partition and update its parents, Case 1 (line 2) of the method is invoked. It sends the intervals [4, 7] to g.

Similarly, g receives the interval [6, 9] from l, which is merged along with k's update to g (Algorithm 1, line 6), expanding g's interval to [4, 9]. Similarly, node m, as seen previously, has the default limits of -1 (line 2), and therefore splits the range [14, 21] among its parents. In this case, the offsets for the limit partitioning $\varDelta_l$ and $\varDelta_h$ are set to $\frac{1}{2}$ since m has two parents (line 5). As illustrated in Fig. 3a, the interval [14, 20.5] is updated to h while the interval [14.5, 21] is sent to i (lines 20–25). With the interval updates, m also sends the relevant upper and lower limits to its parents for interval expansion, to be used while they further propagate intervals towards the root of the hierarchy. In case of node i, it receives interval [14.5, 21] along with a lower limit of 14, and no upper limit from its child node m (Case 3, line 10). This indicates to i that the interval updates it sends to its parents must be bounded within the range [14.25, 22]. As illustrated in Fig. 3b, node i splits the interval [14.25, 22] into two equal partitions. Note that in this case $\varDelta_l$ and $\varDelta_h$ are $\frac{1}{4}$ and $\frac{1}{2}$ respectively (line 13). Therefore, via i nodes e and f's intervals are updated to [14.25, 21.5] and [14.375, 22] respectively. Node e also merges the interval updates from h and i and assigns the interval [13, 21.5] to itself. The backpropagation of intervals continues until the root receives all the updates.

4 Experimental Evaluation

All our experiments are carried out on an Intel Xeon E5-2640 v4 machine with 40 cores clocked at 2.40 GHz having 64 GB RAM running CentOS 7.4. To assess the effectiveness on real-world data, we use XML hierarchy from Treebank [16]. Further, to check scalability aspects and how HiFi works on various structures, we also use synthetically generated hierarchies. In our synthetic dataset, we generate k-ary trees with a million nodes, and arbitrary graphs with 0.1 million nodes. k-ary trees allow us to assess the effect of HiFi on *bushy* versus *skinny* structures by varying k, while graphs allow us to check for multiple path locking. Our test-driver creates multiple pthreads which operate concurrently on the underlying data structure. We note that the standard deviation in all our results is quite small (about 2%). We compare HiFi against state-of-the-art DomLock [9] and Intention Locking [7], under different values of critical section (CS) size, number of nodes locked, and the density of the hierarchy (k-ary trees). DomLock is an alternative to IL which locks the dominator of the requested nodes, and hence has a constant locking cost. Note that neither of the two protocols support co-existence of fine-grained and hierarchical locks.

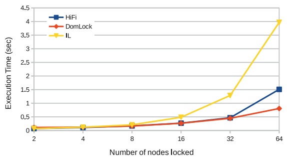
4.1 Effect of Number of Nodes

Figure 4 describes the effect of varying locking request size. Figures 4a–c capture the performance of concurrent requests with increasing number of nodes on binary trees, while Figs. 4d–f show the same for arbitrary graphs. IL acquires intention locks on nodes lying on all the paths that lead to the requested set of nodes, while DomLock acquires a single lock on their dominator. With increasing

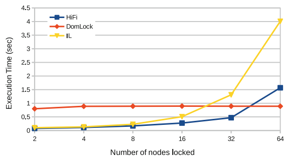
Algorithm 2. Procedure `sendPartitionLimits` to propagate interval limits

```
1  Function sendPartitionLimits(lowLimit, upLimit, root):
       // np is number of parents of the root node
2      if lowLimit == −1 AND upLimit == −1 then
           // Case 1: No Limits on interval expansion on either side
3          if np == 1 then
4              updateParent(lowLimit,upLimit,root.l −1,root.h +1)
5          $\Delta_l = \frac{1}{np}$; $\Delta_h = \frac{1}{np}$; l' = $root.l - 1$; h' = $root.h + 1$
6      else if lowLimit == −1 AND upLimit ≠ −1 then
           // Case 2: Limited interval expansion only on right side
7          if np == 1 then
8              updateParent(lowLimit,upLimit,root.l −1,$\frac{root.h+upLimit}{2}$)
9          $\Delta_l = \frac{1}{np}$; $\Delta_h = \frac{upLimit-root.h}{2np}$; l' = $root.l - 1$; h' = $\frac{root.h+upLimit}{2}$
10     else if lowLimit ≠ −1 AND upLimit == −1 then
           // Case 3: Limited interval expansion only on left side
11         if np == 1 then
12             updateParent(lowLimit,upLimit,$\frac{root.l+lowLimit}{2}$,root.h +1)
13         $\Delta_l = \frac{root.l-lowLimit}{2np}$; $\Delta_h = \frac{1}{np}$; l' = $\frac{root.l+lowLimit}{2}$; h' = $root.h + 1$
14     else if lowLimit ≠ −1 AND upLimit ≠ −1 then
           // Case 4: Limited interval expansion on both sides
15         if np == 1 then
16             updateParent(lowLimit,upLimit,$\frac{root.l+lowLimit}{2}$,$\frac{root.h+upLimit}{2}$)
17         $\Delta_l = \frac{root.l-lowLimit}{2np}$; $\Delta_h = \frac{upLimit-root.h}{2np}$;
18         l' = $\frac{root.l+lowLimit}{2}$; h' = $\frac{root.h+upLimit}{2}$
19     for i = 1 to np AND np ≠ 1 do
20         if i == 1 then
21             updateParent(lowLimit,root.$h$+2$\Delta_h$,l',root.$h$+$\Delta_h$)
22         else if  i == 1 to np −1 then
23             updateParent(l' +(i-2)$\Delta_l$,root.$h$+(i-1)$\Delta_h$,l' +(i-1)$\Delta_l$,root.$h$+i $\Delta_h$)
24         else if i == $np$ then
25             updateParent(l' +(np −2)$\Delta_l$,upLimit,l' +(np −1)$\Delta_l$,h')
```

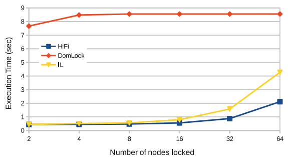
locking request size, the cost of marking intention across paths also increases, making IL a less favorable choice until a certain threshold, as shown in Figs. 4a–c. We do not depict IL's performance in case of graphs as it is far worse compared to that of DomLock and HiFi. In comparison, HiFi acquires exactly n_r locks, where n_r is the number of nodes requested. In the cases shown in Figs. 4a, b, d and e HiFi and DomLock perform almost similarly when the number of nodes requested is less than 32, irrespective of the size of the critical section (small and medium here). Beyond a threshold number, the cost of individual fine-grained operations increases, and it is expected that hierarchical locking performs better, which is evident from better performance of DomLock. In summary, HiFi is better suited for operations with fewer node requests.



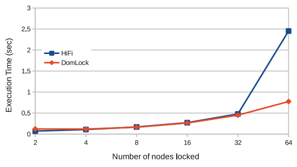
(a) Small CS, Tree



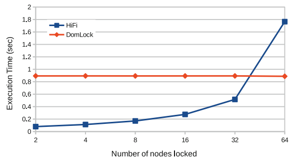
(b) Medium CS, Tree



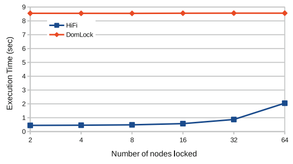
(c) Large CS, Tree



(d) Small CS, Graph



(e) Medium CS, Graph

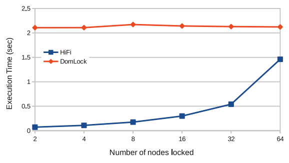


(f) Large CS, Graph

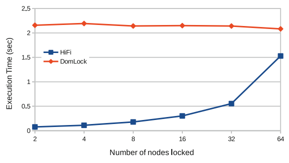
Fig. 4. Effect of the number of nodes locked, for varying critical section sizes (small CS = 6 µs, medium CS = 60 µs, large CS = 600 µs)

4.2 Effect of Critical Section Size

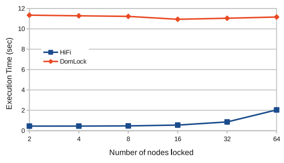
Critical section is critical in deciding the overall performance of an application. It can be observed from Fig. 4 that for large critical section (CS) size, the critical section quickly becomes the bottleneck compared to the rest of the processing. Thus, the importance of fine-grained locking is more imperative for large CS. This can be observed from Figs. 4c and f, wherein co-existence of fine-grained and hierarchical locks improves concurrency in case of HiFi. This suggests that HiFi is better suited for large critical sections. Figure 6 indicates that HiFi scales well with increasing the value of k in a k-ary tree.



(a) Small critical section

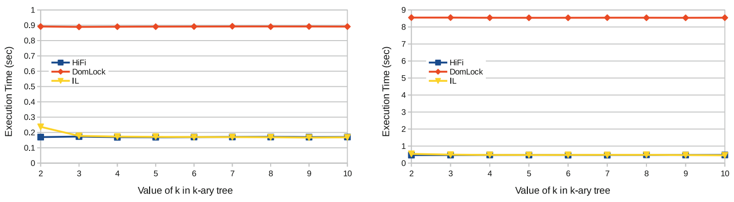


(b) Medium critical section



(c) Large critical section

Fig. 5. Effect of the number of nodes in XML hierarchy, with varying CS size



(a) 8 Nodes requested, Medium CS (b) 8 Nodes requested, Large CS

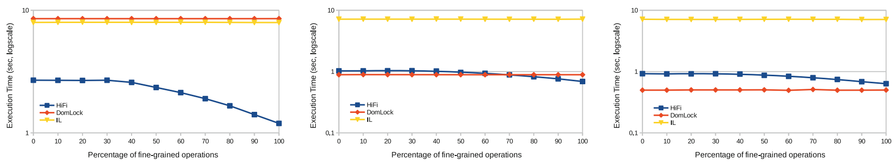
Fig. 6. Effect of varying k in k-ary trees

4.3 Effect on Real-World XML Hierarchy

XML, since its inception has been used to transfer and store information. Its hierarchical method of organizing data provides us with a real-world use case to verify the effectiveness of our locking protocol. We use a real dataset, Treebank hierarchy, publicly available in XML format [16] as the input hierarchy. The XML hierarchy contains 2,437,666 nodes, over 57% of which are leaf nodes. Over 37% of all the nodes are at a height of one from the leaf nodes, and the maximum out-degree among the non-root nodes is 51. This indicates that the hierarchy is quite bushy towards the bottom. Figure 5 shows the performance of HiFi against DomLock for varying critical section sizes. We observe that HiFi performs consistently better than DomLock. Due to the bushy nature of the hierarchy, the dominator of the requested nodes occurs closer to the root, reducing concurrency in case of DomLock. DomLock acquires a lock on the dominator, in this case the root or a node close to the root, thereby blocking concurrent access to the underlying hierarchy. HiFi, however, acquires locks only on the requested nodes, improving the concurrency. To summarize, HiFi is better suited when the concurrent data structure accessed is irregular and has a large fanout.

4.4 Effect of Variation in Fine-Grain Operations

We now study the behavior of HiFi, DomLock and IL for different percentages of hierarchical versus fine-grained operations. Figure 7 shows the effect for different critical section sizes. If all the operations are fine-grained, HiFi is expected



(a) Small critical section (b) Medium critical section (c) Large critical section

Fig. 7. Effect of varying the percentage of fine-grained locks

to perform better than IL and DomLock. On the other hand, if all the operations are hierarchical, then HiFi should ideally be comparable to DomLock, as both the approaches avoid traversal using intervals. However, a major difference between HiFi and DomLock is that DomLock acquires a lock on a single dominator node, while HiFi locks each interval separately. Therefore, for hierarchical-only operations, DomLock performs better than HiFi, as shown in Figs. 7a and b. Interestingly, however, for large critical section size (Fig. 7c), HiFi outperforms DomLock. This is primarily because of the imprecise nature of DomLock which internally restricts the degree of concurrency.

In a general case of mixed fine-grained and hierarchical operations, we observe that the performance of HiFi improves as we increase the percentage of fine-grained operations. IL and DomLock are unaffected by the percentage of fine-grained operations as all the operations are treated uniformly as hierarchical operations. We believe that HiFi would offer an attractive alternative for synchronization in hierarchies.

5 Related Work

The idea of MGL was introduced in database systems [7]. Ries and Stonebraker [14,15] report that there are cases where a coarse-grain-only approach may not be desired. In particular, if all the transactions requesting access to the database are randomly requesting small parts of it, then finer granularity is to be preferred. Their work also reported that transactions operating on more than one percent of the database must use few large locks rather than many locks of finer granularity. This indicates the need for co-existence of fine- and coarse-grained locks. Unrau et al. [1] describe a hybrid approach combining properties of both coarse and fine-grained for four types of access behaviors, namely, non-concurrent accesses, concurrent accesses to independent data structures, concurrent read-shared accesses, and concurrent write-shared accesses. Their method uses coarse-grained locks held for short duration to collectively lock multiple data structures, and fine-grained locks should the underlying data be held for longer duration. The method, however, locks one resource only in one type of mode and does not support co-existence. Golan-Gueta et al. [6] proposed automatic fine-grain locking for trees, while Chaudhri et al. [2] proposed locking for DAGs and trees. These methods perform sub-graph traversals to compute lock request intersections thereby giving rise to performance issues when large number of rows are queried. Liu and Zhang [10] presented fine-grain locking for hierarchies based on intention locks by applying fine-grain locks on fields of objects in the hierarchy. The hierarchy under access here is an abstract object graph which is statically constructed to approximate the runtime object graph. This method also suffers from the same issues as IL. Recent advances in automatic lock inferences [3,4,13] for parallel programs also adopted MGL for efficient lock placements. Cherem et al. [3] use static analysis for extracting points-to information of shared objects and apply MGL locking for avoiding deadlocks.

The idea of using logical intervals to capture structural subsumption property for hierarchical locking was originally proposed in DomLock [9]. However, the interval numbering in DomLock does not work with fine-grained locks, let alone together for fine-grained and hierarchical. HiFi proposes a new indexing mechanism to support this. The interval labels assigned unique intervals only to the leaf nodes in the hierarchy, otherwise leaving the internal nodes indistinguishable in case of chain like structures within the hierarchy.

The key-range locking [11,12] and predicate queries in semi-structured databases [5] also use locks as a range of keys in the databases community. In key-range locking, every lock protects the key value of a record as well as the keys which are absent during the transaction. The locks on absent keys restrict the insertion of any phantom record by other parallel transaction. However, the notion and the purpose of our interval locking is quite different from the key-range locking.

6 Conclusion

We proposed HiFi, a new locking protocol that allows simultaneous co-existence of fine-grained and hierarchical locks. The protocol devises a new indexing scheme for hierarchies, which ensures quick identification of concurrent, overlapping lock requests. We illustrated the effectiveness of our approach using real-world XML hierarchies, and the scalability using synthetic datasets of varying complexity. We believe HiFi would pave the way for newer locking protocols in future.

Data Availability Statement and Acknowledgments. We thank all the reviewers whose comments improved the quality of the paper substantially. The work is supported by IIT Madras Grants CSE/13- 14/812/NFIG/RUPS and CSE/13-14/636/NFSC/RUPS and the travel is supported by The Department of CSE, IIT Madras through Kris Gopalakrishnan Endowment – Student Travel Grant.

The datasets generated during and/or analyzed during the current study are available in the Figshare repository: https://doi.org/10.6084/m9.figshare.6390554 [8].

References

1. Unrau, R.C., Krieger, O., Gamsa, B., Stumm, M.: Experiences with locking in a NUMA multiprocessor operating system kernel, November 1994
2. Chaudhri, V.K., Hadzilacos, V.: Safe locking policies for dynamic databases. In: PODS, pp. 233–244. ACM, New York (1995)
3. Cherem, S., Chilimbi, T., Gulwani, S.: Inferring locks for atomic sections. In: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2008 (2008)
4. Emmi, M., Fischer, J.S., Jhala, R., Majumdar, R.: Lock allocation. In: POPL 2007, pp. 291–296. ACM (2007). https://doi.org/10.1145/1190216.1190260
5. Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L.: The notions of consistency and predicate locks in a database system. Commun. ACM **19**(11), 624–633 (1976). https://doi.org/10.1145/360363.360369

6. Golan-Gueta, G., Bronson, N., Aiken, A., Ramalingam, G., Sagiv, M., Yahav, E.: Automatic fine-grain locking using shape properties. In: OOPSLA, pp. 225–242. ACM, New York (2011)
7. Gray, J.N., Lorie, R.A., Putzolu, G.R.: Granularity of locks in a shared data base. In: VLDB, pp. 428–451. ACM, New York (1975)
8. Kalikar, S., Nasre, R.: Source code and experiment scripts for HiFi hierarchy locking technique: Euro-Par 2018 artifact (2018). https://doi.org/10.6084/m9.figshare.6390554
9. Kalikar, S., Nasre, R.: DomLock: a new multi-granularity locking technique for hierarchies. ACM Trans. Parallel Comput. **4**(2), 7:1–7:29 (2017)
10. Liu, P., Zhang, C.: Unleashing concurrency for irregular data structures. In: ICSE, pp. 480–490. ACM, New York (2014)
11. Lomet, D., Mokbel, M.F.: Locking key ranges with unbundled transaction services. Proc. VLDB Endow. **2**(1), 265–276 (2009)
12. Lomet, D.B.: Key range locking strategies for improved concurrency. In: Proceedings of the 19th International Conference on Very Large Data Bases, VLDB 1993, pp. 655–664. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1993)
13. McCloskey, B., Zhou, F., Gay, D., Brewer, E.: Autolocker: synchronization inference for atomic sections. In: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, pp. 346–358. ACM, New York (2006). https://doi.org/10.1145/1111037.1111068
14. Ries, D.R., Stonebraker, M.: Effects of locking granularity in a database management system. ACM Trans. Datab. Syst. **2**(3), 233–246 (1977)
15. Ries, D.R., Stonebraker, M.R.: Locking granularity revisited. ACM Trans. Datab. Syst. **4**(2), 210–227 (1979)
16. Treebank: Xml data repository (2002). http://aiweb.cs.washington.edu/research/projects/xmltk/xmldata/www/repository.html