DomLock: A New Multi-Granularity Locking Technique for Hierarchies

Saurabh Kalikar
IIT Madras
saurabhk@cse.iitm.ac.in

Rupesh Nasre
IIT Madras
rupesh@cse.iitm.ac.in

Abstract

We present efficient locking mechanisms for hierarchical data structures. Several applications work on an abstract hierarchy of objects, and a parallel execution on this hierarchy necessitates synchronization across workers operating on different parts of the hierarchy. Existing synchronization mechanisms are either too coarse, too inefficient, or too *ad hoc*, resulting in reduced or unpredictable amount of concurrency. We propose a new locking approach based on the structural properties of the underlying hierarchy. We show that the developed techniques are efficient even when the hierarchy is an arbitrary graph, and are applicable even when the hierarchy involves mutation. Theoretically, we present our approach as a locking-cost-minimizing instance of a generic algebraic model of synchronization for hierarchical data structures. Using STM-Bench7, we illustrate considerable reduction in the locking cost, resulting in an average throughput improvement of 42%.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

Keywords hierarchical data structure, trees, graphs, object graphs, locking, synchronization, dominators

1. Introduction

Performance of concurrent programs is critically dependent on the effectiveness of their underlying data structures. Use of an appropriate data structure bears the potential to considerably improve the overall application performance, especially at a large scale. A key piece in this scalable performance puzzle is the synchronization mechanism used to coordinate between conflicting threads.

Hierarchical structures are used in several application domains such as relational databases and software engineering. A hierarchy is characterized by a *containment* relationship, in which the child nodes are contained within their parent nodes. When a hierarchy is rooted, all the nodes in the hierarchy are contained in the root. Since containment relation is asymmetric, the hierarchy is depicted using directed edges. Such hierarchies are used in answering range queries in structured databases (Lomet 1993; Lomet and Mokbel 2009) and predicate queries in semi-structured databases (Eswaran et al. 1976). As a simple example, an institute with various departments, their faculty and students represent a hierarchy rooted at the institute. Note that all the nodes in a hierarchy need not be of the same type. Further, a faculty member may be associated with multiple departments – making the hierarchy a directed acyclic graph. In general, a hierarchy may or may not have cyclic structures within them (depending upon whether containment is reflexive or not). It should also be noted that a binary search tree (BST) is not a hierarchy, nor is a road network, as there is no containment relationship between nodes. However, a `range query` on the BST may necessitate construction of a hierarchy (which is different from the original BST structure).

In the context of such hierarchical structures, traditionally, various kinds of locks have been used for synchronization in the database domain (Gray et al. 1988). While recent advances have developed lock-free mechanisms to manipulate shared data in special structures (Sundell and Tsigas 2003, 2005; Chatterjee et al. 2014; Natarajan et al. 2013), locking based synchronization continues to dominate the world of irregular data structures in general. Therefore, we focus on locking-based protocols.

Hierarchical data structures such as trees and graphs pose scalability challenges for both of the extreme locking techniques: coarse-grain and fine-grain. In case of coarse-grain locking, a single lock is maintained at the root which needs to be acquired for accessing any part of the hierarchy. Clearly, coarse-grain locking reduces concurrency. On the other hand, in case of fine-grain locking, each thread locks precisely the set of nodes of interest. On a positive side, fine-grain locking improves concurrency. However, fine-grain locking is unable to take advantage of the hierarchical structure. Therefore, in case of hierarchical structures, both the coarse-grain and the fine-grain locking mechanisms lead to suboptimal performance. To address these challenges posed by the hierarchical structures, *multi-granularity locks* (MGL) have been proposed. In MGL, the whole substructure rooted at a node in the hierarchy gets locked. This reduces the locking cost as a thread does not need to lock every node in the substructure (as in fine-grain locking). In addition, MGL also improves concurrency, since it does not lock any node outside the substructure (as in coarse-grain locking). Therefore, transactions that operate on non-overlapping parts of the data structure may execute concurrently. Coarse-grain and fine-grain locks may be viewed as special cases of multi-granularity locks.

An important performance criteria in implementing MGL is how to quickly identify overlapping parts of a hierarchy (two overlapping parts may not be locked by different threads in non-compatible modes). To address this, Gray et al. (Gray et al. 1975) proposed *intention locks* which is now the *de facto* method of implementing MGL. In this approach, while acquiring lock on a subgraph rooted at a node, all the ancestors of the node along the path from the root are specially locked, marking *intention*, and then the node of interest is locked in shared or exclusive mode. Intention locks are useful to identify the overlap between subgraphs operated by multiple threads: marking the path with intention enables other

PPoPP '16 March 12-16, 2016, Barcelona, Spain
Copyright © 2016 ACM 978-1-4503-4092-2/16/03. . . $15.00
DOI: http://dx.doi.org/10.1145/2851141.2851164

threads to check for compatible intention when they try to lock any node in the hierarchy (intention locks are described in Section 2).

Unfortunately, intention locks are not well-suited for large real-world data structures with several millions of nodes (more details in Section 2). The traversal cost is prohibitive for *tall* structures, and DAGs and cycles may necessitate locking multiple paths. In practice, intention locks are targeted for a restricted setting: threads locking a single node and the hierarchy being a tree. This poses limitations to the applicability of MGL. Further, there exist many applications wherein threads routinely lock multiple nodes. For the best performance, a multi-granularity locking technique must exploit opportunities while locking a group of nodes. Similarly, DAG structured hierarchies are quite common, and are crucial for data sharing due to huge volumes of storage, and intention locks pose severe challenges while dealing with DAGs (all the paths to a node from the root need to be locked!).

We propose a practical MGL method that solves these issues and enables efficient concurrency control for arbitrarily-shaped hierarchies. Our technique exploits structural properties of the hierarchical data structure while acquiring locks. It works seamlessly with trees, DAGs and cycles, and supports hierarchy mutation (insertion and deletion of edges). Further, the technique is an instantiation of a more general locking framework. Other instantiations bear the potential of offering varied scalability trade-offs. Note that our proposal is a new way of locking, and not a new type of lock.

In particular, this work makes the following contributions.

- We qualitatively and quantitatively uncover scalability issues with the existing standard mechanism to deal with concurrent hierarchical data structures. The existing mechanism is intolerably inefficient for large structures.
- We propose an algebraic model for locking based synchronization. We illustrate that various elements of this algebraic structure offer a trade-off between the degree of concurrency and the locking cost. Based on this theoretical foundation, various locking techniques such as coarse-grain, medium-grain and fine-grain can be easily specified.
- We focus on a special element of the proposed algebraic structure which offers the minimum locking cost and maximizes the degree of concurrency. This element bears special structural properties in the context of hierarchical data structures, and it corresponds to dominator-based locking protocol. Hence, we name our technique as DomLock.
- We evaluate DomLock against an existing intention-lock based mechanism and study their relative performance. The study identifies scenarios under which each of the techniques works well. For general graphs, DomLock is two orders of magnitude faster. We also replace the locking mechanism in STMBench7 framework to use DomLock and achieve significant (up to 57%, average 42%) performance improvements.[1]

The sequel is organized as follow. Section 2 presents existing locking approaches and issues therein. Section 3 presents DomLock that solves these issues. Section 4 presents our algebraic model for lock-based synchronization. Section 5 evaluates the effectiveness of DomLock. Section 6 compares and contrasts with relevant related work, and Section 7 concludes.

2. Background and Motivation

Relational databases often use hierarchies to store data. For instance, Oracle *database* is composed of several *tablespaces*, each of which contains several *datafiles*. Each datafile, in turn, may host several *tables*, and each table may contain multiple *rows* where data

[1] DomLock is available at `http://pace.cse.iitm.ac.in/tools.php`

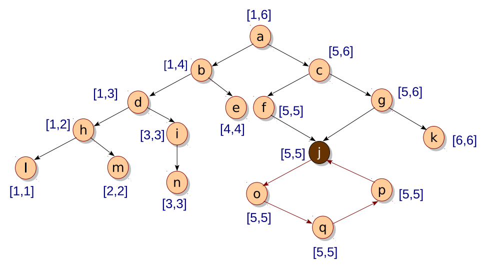


Figure 1. Running example: An arbitrary hierarchical data structure, along with each node's pre- and post-visit numbers

is stored (Oracle 2015). Similarly, Sybase database uses the hierarchy of database, extents, tables, datapages and rows (Sybase 2003). The hierarchy is layered according to the *containment* property. Thus, a table is completely contained into an extent. Concurrent updates to part of the database requires thread synchronization. For instance, for answering *range queries* (MSDN 2015), the concurrency mechanism in the database server acquires locks on multiple rows. Two transactions accessing overlapping ranges (e.g., rows 10..20 and rows 15..25) may be allowed concurrent execution if their accesses are *compatible* (both are reads, for instance).

Mutex or mutual exclusion locks (Herlihy and Shavit 2008) ensure that the protected data structure is accessed by at most one thread at a time. To allow concurrent reads, reader-writer locks (Courtois et al. 1971; Mellor-Crummey and Scott 1991) have been proposed. Sequence locks (Lameter 2005) improve upon reader-writer locks by allowing read-only threads to *pass-through* without using any atomic primitives. For read-mostly scenarios, sequence locks can offer significant benefits, due to which they are used in the Linux kernel.

For the sake of exposition, we consider mutual exclusion locks in the discussion, but our technique is oblivious to any specific type of lock. An easy way of achieving safe synchronization is to use a single coarse-grain lock for the whole database. Programmers often use coarse-grain locks to protect shared data because of the simplicity of using such locks. Automated tools such as parallelizing compilers often use coarse-grain synchronization as it is easy to argue about correctness of the transformed code. Especially in the context of concurrent databases, a transaction's fine-grained access to a large fraction of table rows may be satisfied by acquiring a coarse lock on the table.

However, as is well known, a coarse-grain lock adversely affects the degree of concurrency. Thus, even if two threads access disjoint parts of a data structure (e.g., nodes m and n in Figure 1), they won't be allowed to do so because of one-point synchronization of coarse-grain locking at the root level. This limits parallelism.

Fine-grain locking at the node level (row level in our database example) addresses this issue. In fine-grain locking, each node is associated with a separate lock. Therefore, different threads can work concurrently on different nodes by acquiring the corresponding locks. For instance, using fine-grain locks, two threads can concurrently access node m and node n separately in Figure 1.

Fine grain locking clearly offers more concurrency, but there are several issues with fine-grain locking. It is a folk-lore that the reasoning about fine-grain locks is considerably more complex compared to that for coarse-grain locking. Arguing about correctness of operations gets involved when the operation is split across multiple finer-granularity sub-operations (Hawkins et al. 2012). Incorrect use of fine-grain locks may lead to undesirable effects such as deadlocks. Furthermore, fine-grain locking increases the number of logical locks maintained in the system. This can be prohibitive for a huge data structure such as a social network containing millions of nodes[2].

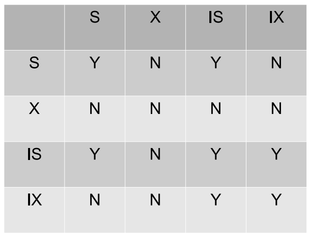
A critical issue with fine-grained locking occurs due to the requirement of a hierarchical lock which a fine-grained lock is unsuited for. Several operations on a hierarchy are defined at the granularity of a hierarchical group, rather than individual nodes. For instance, updating tallied records for each faculty member of a department can be performed by locking only the department object, instead of each faculty member's object as in the fine-grain case. This requires the department lock to be hierarchical in semantics, which covers the whole subgraph beneath it.

EXAMPLE 1 (Hierarchical locking). *Consider a scenario where thread T_1 is operating on the sub-tree rooted at node d in Figure 1 and thread T_2 is operating on the sub-tree rooted at node m. If thread T_2 acquires an exclusive (write) lock on node m and T_1 acquires a shared (read-only) lock on d, then according to fine-grain locking semantics, nodes d and m are different data nodes and both can be operated on concurrently. However, according to the hierarchical locking semantics, T_1 is locking the whole hierarchy below d and no other thread (in this case T_2) would be able to acquire lock on any node of the hierarchy. Similarly, if T_2 acquires a lock on m first then the hierarchical locking protocol would restrict T_1 from locking node d.*

A naïve way to enforce hierarchical locking semantics is to find intersection of the two subgraphs of interest, and allow simultaneous locking only if the intersection is empty. Such an approach is clearly not exciting in practice as it involves costly subgraph traversal, which diminishes the benefits of hierarchical locks (recall that we use hierarchical locks to avoid locking each subgraph node). Alternatively, there exist mechanisms which work with restricted structures. For instance, Golan-Gueta et al. (Golan-Gueta et al. 2011) propose automatic fine-grain locking only for trees, while Chaudhri and Hadzilacos (Chaudhri and Hadzilacos 1995) propose locking for trees and DAGs. These approaches work well when the *selectivity factor* of queries (fraction of the rows selected) is very small. Otherwise, these pose performance issues as they involve costly sub-graph traversal for finding intersection of sub-graphs. This traversal cost could be prohibitive in case of tall structures (for instance, Microsoft SQL Server maintains 11-level storage hierarchy in its database engine (MSDN 2015)). To avoid sub-graph traversal for computing intersection, intention locks have been proposed, which work for general graphs.

2.1 Intention Locking (IL)

In this section, we discuss an existing locking technique based on Intention Locks (IL), also known as Multi-Granularity Locking protocol (Gray et al. 1975; Liu and Zhang 2014). IL has four different modes of locking: *Shared* (S), *Exclusive* (X), *Intention Shared* (IS), and *Intention Exclusive* (IX). Each of these modes of locking follows hierarchical locking semantics: when a node is locked, it protects the whole sub-graph rooted at that node. S mode locks the node in a shared manner, that is, multiple threads can access the sub-graph in a read-only manner. X mode locks a node in an exclusive manner and restricts access from all the other threads. IS and IX play special roles to mitigate the traversal cost. When a node is locked in IS mode, it indicates that, there is at least one node in its sub-graph which is currently been locked in S mode but none of the subgraph nodes is locked in an X mode. Similarly, when a node is locked in IX mode, it indicates that, there is at least one node in its sub-graph which is currently locked in X mode and the same sub-graph may contain some node which is locked in an S mode. Using these four locking modes, IL applies the following locking rules:



	S	X	IS	IX
S	Y	N	Y	N
X	N	N	N	N
IS	Y	N	Y	Y
IX	N	N	Y	Y

Figure 2. Compatibility Matrix for Intention Locks

R1: If a thread tries to lock a node in X mode, then the thread should should lock the nodes belonging to the reference paths starting from root to that node in IX mode.

R2: If a thread tries to lock a node in S mode, then the thread should lock the nodes belonging to the reference paths starting from root to that node in IS mode.

R3: Two or more threads can acquire locks on the same node only if the lock modes are compatible. Figure 2 shows the lock compatibility matrix (Gray et al. 1975; Liu and Zhang 2014).

R4: When a new locking request comes at an already locked node, then locking modes can be upgraded to a more restrictive compatible mode. IX $>$ IS and S $>$ IS, where operator $>$ indicates *more restrictive* relationship among the modes.

Let us see how IL helps enforce hierarchical locking in Example 1. Recall that T_1 tries to lock node d in X mode and T_2 wants to lock node m in S mode. According to R1, T_1 needs to acquire nodes a and b in IX mode: IX (a), IX (b), X (d). Similarly, as per R2, T_2 needs to acquire nodes a, b, d, h in IS mode: IS (a), IS (b), IS (d), IS (h), S (m). According to R3, nodes a, b, d could be locked by both T_1 and T_2 as they are compatible as shown in Figure 2. Figure 2 shows that IX and IS are compatible with each other but both the threads can not acquire X and IS locks on node d simultaneously. Hence, IL restricts concurrent accesses by $T1$ and $T2$, and protects the parallel accesses to hierarchical data structures.

2.2 Issues with IL

Intention locks are often used in relational databases in implementing concurrency control. However, there are several issues with using IL. First, IL is a traversal-based protocol, that is, each locking request has to traverse the complete reference path starting from $root$ of the data structure. Even if we have direct reference pointers to individual nodes, still IL can not avoid path traversal as it needs to lock *intention* along the path. Second, according to rule R1 in IL, before locking any node, a thread has to lock *all* possible reference paths between $root$ and the node. For instance, in Figure 1, to lock node j, it is required that both the paths $a \to c \to f \to j$ and $a \to c \to g \to j$ should be intention-locked. In case of a tree data structure, there is a unique reference path to each node; however, in case of DAGs and cycles, there could be multiple (potentially infinite) reference paths to a node. Hence discovering all possible paths and locking all of them in intention mode introduces extra locking overhead and leads to increase in the locking cost (Hawkins et al. 2012). Third, existing intention lock based implementations handle locking of multiple nodes inefficiently. For instance, if a thread tries to lock nodes j and k from Figure 1 in the same transaction, the underlying locking implementation traverses the path $a \to c \to g$ twice. Prima facie, the traversal appears redundant, but to avoid the extra traversals, we require a formal argument about correct-

[2] While number of locks is limited only by the amount of RAM or virtual memory in modern Linux distributions, more number of locks considerably degrades locking performance due to their maintenance

ness and progress. Finally, in case of multi-node locking, the actual node-locking cost (ignoring the traversal cost) is proportional to the number of nodes being locked. It would be desirable to reduce the locking cost as much as possible (as we discuss in the next Section 3, DomLock has a constant node-locking cost). These issues make IL undesirable for achieving efficient synchronization in concurrent programs. As we also show in Section 5, the traversal cost of IL is prohibitively high for large general graphs.

3. DomLock: Dominator-based Locking

In this section, we present DomLock, a new technique for locking in hierarchical data structures. We first present the structural relationships between nodes and their ancestors. Then, we discuss how DomLock uses such relationships for hierarchical locking. Next, we explain how two subgraphs can be quickly checked for an overlap, which is required for multi-granularity locking. Finally, we discuss about the trade-offs between the locking cost and the degree of concurrency that DomLock offers.

3.1 Dominators

Given a directed graph rooted at node $root$, we define a dominator as follows.

DEFINITION 1. *Dominator: A node* d *is a dominator of a node* n *if all the paths from* $root$ *to* n *pass through* d.

DEFINITION 2. *Immediate Dominator: A dominator* d *is an immediate dominator of a node* n *if there exists no other dominator for* n *on the paths between* d *and* n *and* $d \neq n$.

We denote dominator relationship using a *doms* relation, e.g., *root doms n*. The relation *doms* is a reflexive and transitive relation, and can be generalized to a set of nodes. Thus, a node d dominates a set of nodes S such that for each $n \in S$, all the paths from $root$ to n pass through d. In our running example of Figure 1, the root node a is the dominator of all the nodes in the data structure (including itself). Nodes b and c are the dominators of the left and the right sub-structures of node a respectively. Further, both a and c dominate f, but node c is the immediate dominator of node f.

When the shape of a data structure is a tree, for any node in the graph, each node in the reference path (including itself) becomes a dominator of that node. In Figure 1, for node m, nodes a, b, d, h, m are the dominator nodes. Further, for trees, each node's parent is its immediate dominator, and for a set of nodes, their *least common ancestor* becomes the immediate dominator. In case of a DAG, where a node may have more than one reference paths, the node that belongs to all the reference paths becomes the dominator. Figure 1 shows that *c doms j*. For DAGs, the least common ancestor may not be the immediate dominator. When a set of links forms a cycle, there are potentially infinite paths to each node. In our implementation, the *entry node* of the cycle (as defined by the depth-first traversal) is considered the dominator of all the cycle-nodes. For instance, in Figure 1, node j is the entry node which is also the dominator for nodes j, o, p, q.

3.2 Dominator-based Hierarchical Locking

Dominators provide an effective way of locking in hierarchical structures. Since, by definition, a dominator is present on all the paths from root node to a node, say n, locking the dominator is sufficient to lock all the paths to n and its descendants. Similarly, for a set of nodes S, their collective dominator is sufficient to lock all the paths to all the nodes in S. By paying a small preprocessing cost in finding dominator, and by dynamically maintaining them on graph mutation, we can considerably reduce the locking cost.

An important consideration that decides the locking cost is the number of nodes locked during a transaction. As discussed in Section 2, the cost of intention locks is especially high for non-tree data structures due to multiple reference paths. Therefore, we design DomLock to reduce the number of nodes locked as well as to reduce the graph traversal cost. However, choosing an arbitrary dominator may hamper concurrency. Therefore, we choose the immediate dominator for locking. Overall, our technique is based on finding immediate dominators for a set of nodes and locking the dominator instead of individual nodes in the set (detailed algorithm is presented later in this section).

For instance, consider a thread that wants to access nodes l, m and n in Figure 1. Instead of locking all three nodes individually, DomLock locks a common dominator node d for l, m and n.

Overall per-thread processing is as follows.

```
1: Procedure LockAll():
2:   $S \leftarrow$ GenerateRequests()
3:   $dom$ = FindDominator($S$)
4:   if !pool.IsOverlap($dom$)
5:       pool.insert($dom$)
6:   else
7:       ... // retry checking overlap
8:   ...   // Critical Section
9:   pool.remove($dom$)
```

`pool` is a concurrent data structure keeping track of all the locking requests. The procedure `IsOverlap(`dom`)` in Line 4 checks if the sub-hierarchy rooted at dom overlaps with any existing locked sub-hierarchies in `pool`. If it does not, then the current locking request does not conflict with any of the existing locked sub-hierarchies, and is inserted into `pool` (Line 5). Otherwise, the thread continues to retry locking (Line 7).

DomLock reduces the traversal cost as well as the locking cost (single node). An effect of locking a single node is that there is no possibility of a deadlock across transactions. However, for efficiency, it should be possible to quickly check if two subgraphs overlap (intention was used in IL along each path). We use logical intervals for efficient checking of hierarchy overlap across transactions.

3.3 Logical intervals

Let G = (V, E) be the data structure representing the underlying hierarchy, such that V is the set of vertices and E $\subseteq$ V $\times$ V is the set of directed edges. We define a function $\mathcal{L} : V \rightarrow \mathbb{N} \times \mathbb{N}$, such that for any element v $\in$ V, $\mathcal{L}$(v) is a pair , say [x, y] with x $\leq$ y, that denotes the logical interval associated with v. For a given data structure, we make sure that the following invariant property holds.

PROPERTY 1. *(Subsumption Property) For each element v $\in$ V, the logical interval $\mathcal{L}$(v) subsumes (or super-ranges) $\mathcal{L}$(v') for all the descendant elements v'$\in$ V reachable from v. Thus, if $\mathcal{L}$(v) = $[x_1, y_1]$ and $\mathcal{L}$(v') = $[x_2, y_2]$, then $x_1 \leq x_2$ and $y_1 \geq y_2$.*

Our running example from Figure 1 shows logical intervals for each node. As an example of Property 1, in Figure 1, the root node a[1, 6] subsumes intervals of all the nodes in the graph.

Computation of Logical Intervals. Logical intervals can be computed by a modified DFS traversal (similar to pre- and post-visit DFS numbers). This is presented in Algorithm 1 which assigns an interval value to each node. Each interval is represented as a pair (low, $high$). The algorithm ensures that all the assigned intervals strictly follow the invariant property (the correct intervals are maintained even on graph mutation). While performing DFS from $root$, we maintain the following three flags with each node. When DFS visits a node, it takes different actions depending upon the flag values. We present those below.

- *Active*: The *Active* flag of a node v is *true*, if DFS has visited node v and there exists at least one descendant node of v which is not yet visited by the DFS.
- *Explored*: A node is *Explored*, if all of its descendants are visited by the DFS.
- *ParentUpdated*: The *ParentUpdated* flag of a node is set to *true*, if the node has updated the interval values of its immediate parents at least once. This flag is used in cyclic graphs.

We use a counter variable *count*, to assign new interval values to the nodes. The flags mentioned above are initialized to *false* and *count* is initialized to 0. When we visit a node, if the *active* flag is *false*, we set it *true* and recursively call Algorithm 1 for its children. If a leaf node is encountered, counter gets incremented and the new interval value is assigned to that leaf node (Line 5 of Algorithm 1). For instance, in Figure 1, node l is a leaf node and is assigned interval [1, 1]; it is also marked as *explored*. The interval value of the explored node is propagated backwards to all of its parent nodes. Algorithm 2 updates the interval of a parent node based on that of its child so that the subsumption property (Property 1) is maintained. This is repeated for all the parents of the original leaf node. At the end, the flag *active* is reset to *false* and DFS backtracks. Note that, a node can receive interval information multiple times from different child nodes. The interval width of a parent node keeps on expanding as it receives interval updates from child nodes (details are in Algorithm 2). For example, in Figure 1, node h has two child nodes, l and m. Before visiting node m, node h receives interval value as [1, 1] and after visiting node m, node h receives interval [2, 2]. Hence, the new interval of h expands to [1, 2] which maintains the invariant property. If a node is marked *explored*, the procedure does not revisit the sub-graph rooted from that *explored* node.

Cyclic Structures. If the traversal visits an *active* node, then it means the graph is forming a cycle at that node (for example node j in Figure 1). We consider such a node as an entry node for the cycle and treat that node similar to a leaf node (Line 5 of Algorithm 1). We give new interval to the entry node of the cycle by incrementing the count value. Note that each node forming a cycle subsumes every other node in the cycle and hence all the nodes get the same interval value. Figure 1 shows nodes (j, o, q, p) form a cycle and all have the interval [5, 5]. In case of cycles, the interval of an *explored* node may get updated even after it has sent updates to its parents. The flag *ParentUpdated* ensures that if the interval of any node is getting updated by any of its child nodes and it is marked as *ParentUpdated*, then such a node should again send the updates to its parent nodes. In Algorithm 2, lines 14 - 17 shows the recursive update of parent nodes.

DomLock *using Dominators and Logical Intervals.* DomLock exploits the following property for efficient lock management.

PROPERTY 2. *Node d is a dominator of a set S of nodes in a hierarchy, iff the logical interval of d subsumes (super-ranges) the intervals of all the nodes in S.*

As an example, in Figure 1, nodes $a[1, 6]$, $b[1, 4]$, $d[1, 3]$, $h[1, 2]$ super-range the intervals of $l[1, 1]$ and $m[2, 2]$. Therefore, nodes a, b, d and h are the dominators of l and m. Further, h is the *immediate common dominator* of l and m. In DomLock, we consider only such immediate common dominators for dominator locking[3]. In case of trees, dominator of a set of nodes and their least common ancestor (LCA) refer to the same node. However in DAGs, the two may be different. For instance, in Figure 1 node c is the immediate dominator, as well as the LCA of nodes f, g, and j. However, for nodes j and k, their LCA is g, but their immediate dominator is c. When a thread wants to lock multiple nodes, we find an immediate common dominator using the logical intervals.

Procedure for finding dominators is presented in recursive Algorithm 3. The procedure accepts two arguments, the root of the hierarchy and the set of nodes to be locked. A preprocessing step computes the intervals for each node in the hierarchy. The interval for the input set are computed using those of the individual nodes in the set (Lines 1–2). The procedure then traverses over the *root*'s subgraph to find out the node which *covers* the interval for the set. Such a last node (visited in the end on a path starting from $root$) is

[3] In our implementation, we optimize the processing using logical intervals to find a node which may be below the immediate dominator. This improves concurrency. For instance, to lock nodes j [5, 5] and k [6, 6], it suffices to lock g [5, 6] rather than their immediate dominator c [5, 6].

Algorithm 1 ModifiedDFS($root$) for computing logical intervals

```
Require: root node $root$
Ensure: each node receives its own logical Interval
 1: if $root == null$ then
 2:     return
 3: end if
 4: if $Explored[root] == false$ then
 5:     if $root$ is a leaf node or $Active[root]$ is true then
 6:         count++
 7:         $Interval[root].low \leftarrow$ count
 8:         $Interval[root].high \leftarrow$ count
 9:     else
10:         $Active[root] \leftarrow$ true
11:         for all $v \in$ descendants($root$) do
12:             call ModifiedDFS($v$)
13:         end for
14:     end if
15:     $Explored[root] \leftarrow$ true
16:     $Active[root] \leftarrow$ false
17: end if
18: for all $w \in$ parents($root$) do
19:     call UpdateParent($w, root$)
20: end for
21: ParentUpdated[root] $\leftarrow true$
22: return
```

Algorithm 2 UpdateParent($parent$, $node$)

```
Require: parent node $parent$, current node $node$
Ensure: $node$ updates intervals of all the parents
 1: if $parent == null$ then
 2:     return $true$
 3: end if
 4: if $Interval[parent]$ = defaultInterval then
 5:     $Interval[parent] \leftarrow Interval[node]$
 6: else
 7:     if $Interval[parent].low > Interval[node].low$ then
 8:         $Interval[parent].low \leftarrow Interval[node].low$
 9:     end if
10:     if $Interval[parent].high < Interval[node].high$ then
11:         $Interval[parent].high \leftarrow Interval[node].high$
12:     end if
13: end if
14: if $ParentUpdated[parent] == true$ then
15:     for all $w \in$ parents($parent$) do
16:         call UpdateParent($w, parent$)
17:     end for
18: end if
```

Algorithm 3 FindDominator($root$, S)

```
Require: root node $root$, set of nodes to be locked $S$
Ensure: immediate dominator for nodes in $S$
 1: $S.low$ = min($y.low$) where $y \in S$
 2: $S.high$ = max($y.high$) where $y \in S$
 3: $ptr \leftarrow root$
 4: for all $x \in$ ProperDescendants($ptr$) do
 5:     if $x.low \leq S.low$ and $x.high \geq S.high$ then
 6:         $ptr \leftarrow x$; break;
 7:     end if
 8: end for
 9: if $Interval_{ptr} \neq Interval_{root}$ then
10:     return FindDominator($ptr$, $S$)
11: else
12:     return $root$
13: end if
```

safe to lock the input set of nodes as well as improves concurrency. The intervals, defined by Algorithm 1, guarantee that a dominator always super-ranges its dominees. The super-range checking is performed as shown in lines 5 – 7 of Algorithm 3.

Algorithms. Let S denote the set of nodes to be locked before starting the transaction. $Interval_S.low$ and $Interval_S.high$ are the minimum low number and maximum $high$ number among all the nodes in set S respectively. The dominator-finding procedure starts the traversal of the data structure from the $root$ node by initializing ptr to $root$ in Line 3 of Algorithm 3. The *for* loop at Line 4 iterates over all the descendants of the current node ptr, and in Line 5, it checks whether any of the child nodes dominates all the nodes in set S . If any child node ptr dominates set S, then Algorithm 3 recursively gets called for ptr. The recursive call is needed to find the immediate common dominator for set S so that we can reach as close as possible to the nodes in set S. If none of the child nodes dominates the set then Algorithm 3 returns current node ptr as the dominator, which in the extreme case would be the root node.

As discussed earlier, we need to maintain hierarchical semantics of locking while dealing with hierarchical data structures. Similar to intention locks, DomLock has to ensure that none of the ancestors and descendants is locked in a non-compatible mode. IL acquires intention locks for all the ancestors belonging to all the reference paths before locking a node. In contrast, DomLock keeps the information of locked nodes, along with their access types (read or write), in a separate concurrent request pool. This concurrent pool keeps track of the intervals of all the dominators currently locked by all the threads.

DomLock *Checks.* Before locking any dominator d, we need to ensure that no other node $n \in L$ is an ancestor or a descendant of d, where L is the set of currently locked nodes. When a thread wants to lock d, it traverses L and checks the interval of d against intervals of all the nodes $n \in L$. For a given pair of nodes (u, v), intervals provide an $\mathcal{O}(1)$ solution to detect conflicts for dataraces.

- If the interval of u subsumes the interval of v or vice versa, then u and v have ancestor-descendant relationship.
- If intervals of u and v are overlapping but not subsume, then the two nodes must have common descendant nodes. The intervals of these descendant nodes subsume under the intersection of the intervals of u and v.
- If intervals of u and v are disjoint then they do not have any hierarchical dependency. According to the semantics, only disjoint intervals are allowed to be locked concurrently.

In practice, only a few nodes out of the whole hierarchy are locked at any time. Hence, the lookup operation in the interval pool adds negligible overheads to locking and, in fact, proves useful in quick identification of conflicting locks.

One may wonder that the concurrent pool may become a bottleneck in the presence of a large number of threads. To improve concurrency, we use reader-writer locks (provided by *pthreads* library). This allows checking of overlapping sub-hierarchies to be performed concurrently by multiple threads, and reduces contention. It is only the insertions into the pool that require exclusive access (writer lock). In our experiments up to 32 threads, we have not found the pool to be a bottleneck.

3.4 Structural Updates

DomLock supports structural updates to the underlying hierarchy which do not change its root. Link updates, i.e., insertion or deletion of an edge between any two nodes, poses more challenges in maintaining hierarchical semantics and the subsumption property. While inserting an edge $u \rightarrow v$, we first find a set S of only those ancestors of node u whose interval values need to be changed because of the insert operation. DomLock requests an exclusive lock on the dominator of S and executes insertion process followed by update to the interval values of the ancestor nodes $n \in S$. Once we acquire the lock on the dominator of S, it ensures that no other thread is concurrently operating on any of the ancestors. For instance, addition of edge $m \rightarrow n$ in Figure 1 populates the set as $S = \{m, h\}$. Immediate dominator of S is node h, hence DomLock acquires an exclusive lock on h and updates the intervals of the m and h to [2, 3] and [1, 3] respectively. In case of deletion of an edge $u \rightarrow v$, we acquire an exclusive lock only on node u and delete the edge. Unlike insertion, deletion of an edge need not necessarily trigger update to parent's intervals, because intervals of parents follow the subsumption property even if a child link is deleted. For instance, deletion of edge $g \rightarrow j$ in Figure 1 need not necessarily update g's interval to [6, 6] for correctness, although doing that would improve concurrency.

3.5 Concurrency versus Cost of Locking

DomLock finds the dominator node for a set of requested nodes to be locked and locks it. There exist situations when it is possible that the dominator ends up locking extra nodes than the ones requested. For instance, in Figure 1, if a thread wants to lock nodes m and n then node d is the immediate common dominator. Locking d also locks node l which is not required. This is an example of the trade-off between the locking cost and concurrency. In the worst case, if requested nodes are far apart such that the immediate common dominator is too close to the $root$ node, it adversely affects the degree of concurrency. A group of nodes may have multiple dominators, but to improve parallelism, it is best to choose a dominator as far away from the root as possible. This, in general, is an immediate dominator, which is closest to the nodes of interest. Since there exist multiple dominators for a given set of nodes, there are multiple ways of locking in a hierarchical data structure. Different dominators offer locking cost versus concurrency trade-off, and we model it more formally in the next section.

4. Algebraic Model

In this section, we introduce a model for locking-based synchronization, and show that DomLock is a special case of this model.

We define the effectiveness of each case in terms of soundness and precision. A hierarchical locking protocol is *sound* if it locks a superset of the requested nodes. A coarse-grain locking protocol is trivially sound. To control the amount of locking, we need to define its precision. A hierarchical locking protocol is *precise* if it does not lock any extra node over the requested set of nodes. Fine-grained locking is trivially precise. A precise protocol may lock a subset of the nodes of interest. An extreme case would be to not lock any node – but this would be unsound. In an ideal scenario, we need a locking protocol that is sound as well as precise, that is, it locks only the nodes of interest and nothing more. We focus on sound locking protocol, by trading off minimal precision for efficiency.

4.1 Degree of Concurrency versus Locking Cost

An extreme way to lock a subgraph is to use a fine-grained mechanism which locks each node in the subgraph. The other extreme is DomLock which locks the least common ancestor of the roots of the subgraphs of interest. A clear advantage of a coarse-grained strategy is reduced locking cost, as the subgraphs need not be traversed and only one object is locked.

Between these two extremes lie several options which provide us with an opportunity to trade-off the amount of concurrency for the locking cost. In order to be *precise*, none of these options should lock any path outside that implied by the original locking request. Similarly, in order to be *sound*, each option must lock all the objects of interest (implicitly or explicitly). For instance, consider our running example from Figure 1 wherein suppose that a (hierarchical) locking request involves nodes h and i. To ensure soundness, subgraph nodes rooted at h and i must be locked. This indicates that at least the three paths involving the two nodes must be locked: (i) $a \rightarrow b \rightarrow d \rightarrow h \rightarrow l$, (ii) $a \rightarrow b \rightarrow d \rightarrow h \rightarrow m$, and (iii) $a \rightarrow b \rightarrow d \rightarrow i \rightarrow n$. Thus, locking the dominator d would be sound, whereas, locking node l would be unsound. For precision, none of the paths that exclude nodes h and i should be locked. Locking the dominator node d in this case ensures precision (as it locks only the above three paths and no more), whereas, locking node b is imprecise, as it also locks the path $a \rightarrow b \rightarrow e$, which does not involve any of the requested nodes h and i.

For ensuring correct locking semantics in our applications, we focus on sound options[4]. Two locking parameters decide application performance: (i) degree of concurrency, and (ii) locking cost.

Depending upon the number of extra nodes locked, a locking option may correspondingly reduce the concurrency of operation. In other words, better precision implies more concurrency. In the context of hierarchical locking, we define concurrency as follows.

$$concurrency = \frac{\text{number of paths to be locked}}{\text{number of paths actually locked}} \tag{1}$$

Higher the denominator, higher is the imprecision and lower is the concurrency. Concurrency is unity when only the requested set of paths is locked, and in general, varies between 0 and 1.

Apart from the amount of concurrency, the second parameter that affects application performance is the locking cost. For our purpose, we define locking cost in terms of the number of nodes physically locked. For instance, for a request to lock nodes h and i in Figure 1, fine-grained locking has a cost of 2 units, whereas the coarsest locking (at the root node a) would incur a cost of 1 unit. Another sound option of locking leaf nodes l, m, n has a cost of 3 units, while DomLock always incurs a cost of 1 unit in locking dominator d. *Our goal is to choose a locking option in order to maximize the amount of path-concurrency and minimize the locking cost.*

4.2 Algebraic Structure

We represent each of the above locking options as an element, and define a relation $\leq$ which compares the amount of node concurrency as well as the locking cost across two options. $o_i \leq o_j$ implies that $concurrency_{node}(o_i) <= concurrency_{node}(o_j)$ and $nlocked(o_i) >= nlocked(o_j)$. Clearly, $\leq$ is reflexive (each option $o_i \leq o_i$), and transitive ($o_i \leq o_j \wedge o_j \leq o_k \implies o_i \leq o_k$).

[4] Unsound locking may be useful in speculative parallelization.

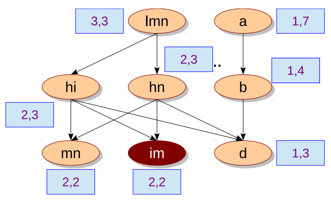


Figure 3. Lock options for the running example from Figure 1. The two numbers at each element denote the locking cost and the concurrency cost incurred by locking the nodes represented by that element in the original data structure.

Figure 3 shows part of the lock options relevant to this example. Each node in the graph represents a locking option, and each directed edge represents a relatively better option (defined by $\leq$ relation). Nodes at the bottom of the graph (such as mn, im, and d) represent options where the locking cost as well as the concurrency cost are low. Thus, option im has a locking cost of 2 units, and its concurrency cost is also 2 units. Similarly, option b has a locking cost of 1 unit, but its concurrency cost is 4 (as it locks four paths).

This algebraic structure allows us to model *useful* locking alternatives. For instance, all the ancestors of the options-of-interest are strictly worse, and can be ignored. Thus, overall, we have the bottom-most three options for locking nodes i and m. We call this set of options as *prime*. Fourth, DomLock option always appears at the bottom of this hierarchy for a given locking request. In other words, DomLock option does not have any outgoing edge from it. Therefore, ***DomLock always results in a prime option***. This indicates that no other option can be strictly better than DomLock for any locking request under our algebraic model.

The options-graph formulation allows an application to choose various locking options depending upon its needs of the degree of concurrency and the locking cost. o_{DomLock} is a special element of this structure which maximizes concurrency amongst all the options with the minimum locking cost (of unity).

5. Experimental Evaluation

All our experiments are carried out on an Intel Xeon E5-2650 v2 machine with 32 cores clocked at 2.6 GHz having 100 GB RAM running CentOS 6.5 and 2.6.32-431 kernel. We use three implementations for evaluating the efficacy of DomLock. First is the STMBench7 benchmark suite (Guerraoui et al. 2007) also used in the evaluation by Liu et al. (Liu and Zhang 2014). STMBench7 is a framework to check the performance of different software transactional memory implementations. Existing framework implements two forms of locking: coarse-grain and medium-grain. Second is the DomLock'ed version of STMBench7 – we replace the locking mechanism in it to use DomLock. Third implementation, which stress-tests DomLock measures the effect of various parameters (such as the number of nodes to be locked, critical section size, etc.) on its performance. It creates a hierarchy containing DAGs as well as cycles, and implements locking using intention locks as well as DomLock. Since IL works relatively better with trees (single path to each node), this implementation also has the provision of generating a random binary tree. We present performance numbers on both the binary tree and the general graph. Below, we provide more details of our implementation.

STMBench7. In order to check the effectiveness of different kinds of synchronization mechanisms, STMBench7 applies locking to an application derived from the OO7 benchmark (Carey et al. 1993). The application forms a complex data structure which consists of different levels of logical hierarchies among the data structure nodes. The root of the data structure is called *module* which forms the top level in the hierarchy. A *module* object has two child nodes: *manual* and *assemblies*. Object of type *manual* does not have any child node. Object of type *assemblies* is considered as a design root of the whole data structure. It forms a tree of the internal nodes of type *complex assemblies* where each node has three child nodes (the number of children can be controlled by changing a value in STMBench's configuration file). There are six levels of complex assemblies and the leaf nodes of this tree are called *base assemblies*. Each of the base assemblies contains a set of nodes of type *composite parts*. Each composite part has a link to one *document* type object and an arbitrary graph of *atomic parts* connected via *connection* as an edge between *atomic parts*. One composite part may be a part of two different base assemblies, thus having multiple parents in the hierarchy. Therefore, the composite part forms a DAG. The connections across atomic parts are randomly chosen, and the graph is built to contain at least one cycle.

Stress-Test Implementation. To study different aspects of synchronization over the data structure, we implemented a concurrent binary tree and a DAG. The binary tree nodes have an integer type data field as well as left and right pointers to the left sub-tree and the right-subtree respectively. The directed graph is generated for 1 million nodes and edges are connected by choosing two nodes randomly. The directed graph nodes have lists of incoming and outgoing edges. Using pthread library, we create multiple threads, which operate concurrently on the data structure. Each created thread calls a $ParallelTask$. Each $ParallelTask$ chooses random nodes, on which the thread should acquire locks before entering into critical section. The command-line parameter *distribution*, indicates the skewness in the access pattern of a thread. If the value of the parameter *distribution* is 2, then there are 2 disjoint partitions of nodes available and the randomly chosen set of nodes is restricted to a partition. According to the semantics, each thread has to lock all the chosen nodes before entering the critical section. We implement the *Intention Locking* policy and compare its performance with DomLock under different values of *distribution*, *number of nodes*, *critical section size* and *number of structural updates*.

5.1 Overall Performance

Figure 4 shows the throughput (number of operations per second) of STMBench7 synchronized using DomLock and the existing STMBench7 locking techniques. Baseline throughput is the maximum achievable throughput of the system which does not use any synchronization. This version ignores race conditions, and thus leads to a possibly incorrect execution. We use this version to study how far DomLock's performance is from a theoretical maximum. The performance of *coarse grain locking*, *medium grain locking* and DomLock varies according to the number of threads. Except for a single thread, DomLock shows performance improvement over *coarse grain locking* and *medium grain locking*. The reason behind the performance improvement is, DomLock provides more concurrency than that of existing locking mechanisms. In STMBench7, the *medium grain locking* uses one lock per type of object. In case of atomic parts, single lock protects all the atomic parts (for big size benchmark, STMBench7 has 100,000 atomic parts). Though two operations are operating on two different sets of atomic parts, medium and coarse grain locks in STMBench7 forbid their concurrent execution. DomLock can support such operations to run concurrently because two disjoint sets of atomic parts lock separate dominator nodes. As we achieve more degree of concurrency, more number of operations can proceed in parallel. However, in any of the locking policies, the throughput does not scale according to the number of threads. This happens because the operations

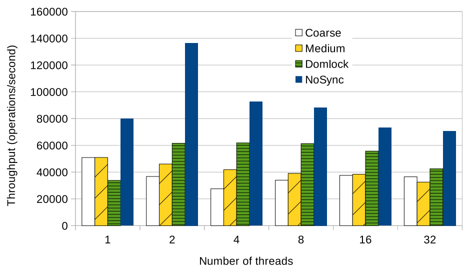


Figure 4. Throughput comparison between DomLock and locking techniques in STMBench7. *NoSync* is a theoretical max. performance obtained by removing all the synchronization calls.

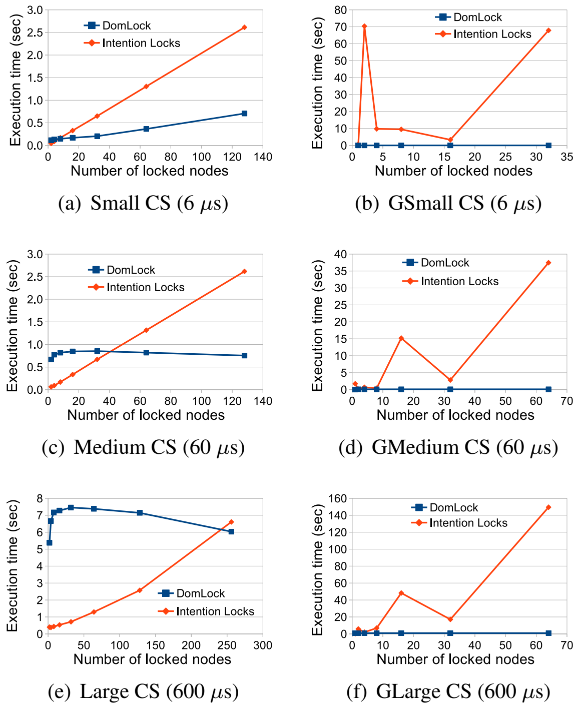
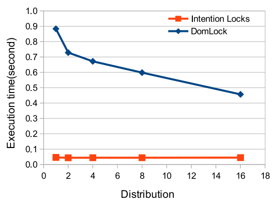


Figure 5. Effect of number of locked nodes (CS=critical section). Left and right plots refer to binary tree and general graph resp.

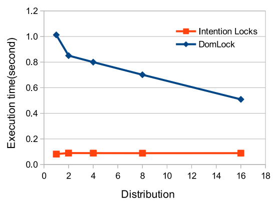
on the data structure are not only dependent on the data *within* the data structure but also *outside* it. For example, the operations in STMBench7 use the map, *atomic_part_int_index*, while operating on atomic parts, and this creates another bottleneck with more number of threads. Overall, DomLock is on an average, 56% faster than coarse, and 42% faster than medium.

5.2 Effect of Number of Nodes (L_{nodes})

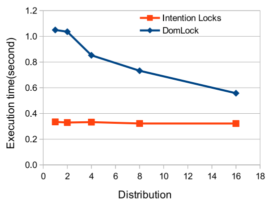
We use our stress-test implementation to evaluate the effect of various parameters on DomLock's performance. Figure 5 describes the effect of the number of nodes in the data structure to be locked. We assign equal amount of work to each thread and measure the overall execution time for all the threads. Synchronization using *Intention Lock* locks each requested node separately along all the paths whereas DomLock locks the common dominator node of all the requested nodes. As we can see in Figure 5, *Intention Lock* takes less time when operating on fewer nodes. While working on large number of nodes, DomLock outperforms IL. In IL, the cost of lock-



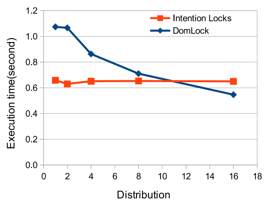
(a) 2 nodes



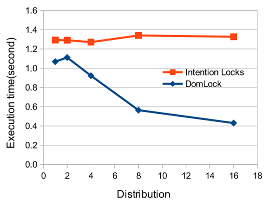
(b) 4 nodes



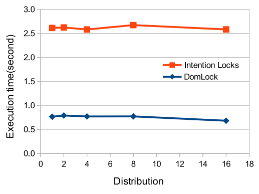
(c) 16 nodes



(d) 32 nodes



(e) 64 nodes



(f) 128 nodes

Figure 6. Effect of distribution (skewness of nodes). The configuration uses medium critical section size (60 μs) on a binary tree.

ing grows linearly with L_{nodes} in binary tree because each node is separately locked. Unlike IL, DomLock is not sensitive to the L_{nodes} and takes almost constant execution time for different value of L_{nodes}. This shows the robustness of DomLock. The intersection point of two graphs in Figure 5 gives the threshold value of L_{nodes} for a particular application beyond which DomLock takes much lesser time than IL. For the number of nodes lesser than the threshold value DomLock takes more time because it introduces extra overhead of finding dominator node and it may also reduce some concurrency if the height of the dominator is more than the average height of nodes. This overhead gets amortized by the performance benefit for larger number of nodes.

Another issue with multi-node locking in IL is that the nodes need to be locked in specific order across threads to avoid deadlocks. Following this discipline is often costly as it requires frequent sorting or maintaining a priority queue. Note that since DomLock finally locks a single node, no such issues arise.

5.3 Effect of Distributions

We now present the effect of distribution on performance for a fixed value of the number of requested nodes. Figure 6 depicts the variation in execution time with respect to different distributions. To vary the distribution, we restrict the thread to access the node from a particular subset of nodes. X-axis gives the number of available concurrent subsets of nodes in the system. For example, the value 1 on x-axis shows that there is only one set available which means any thread can access any node in the data structure. This simulates a random distribution of the data structure nodes. If the value on x-axis is 2 then there are two separate subsets available. Each thread operates on any one of the two subsets and can access nodes from that particular set. As we go on increasing the value on x-axis, the distribution becomes more and more skewed. We observe from Figure 6, for random distribution (distribution = 1) DomLock takes more time than that of a skewed distribution. The execution time of DomLock gradually decreases with increase in skewness of distribution. The reason behind this behavior is that when we set a random distribution, the two arbitrarily chosen nodes may have their common dominator node close to the root and hence it reduces the degree of concurrency. As we introduce skewness, the number of extra nodes we lock while locking the dominator is less, which provides more concurrency than a random distribution. Several real world applications exhibit skewed access pattern, i.e., some nodes in the graph are more frequently accessed than many other. In such situations DomLock is a promising locking protocol.

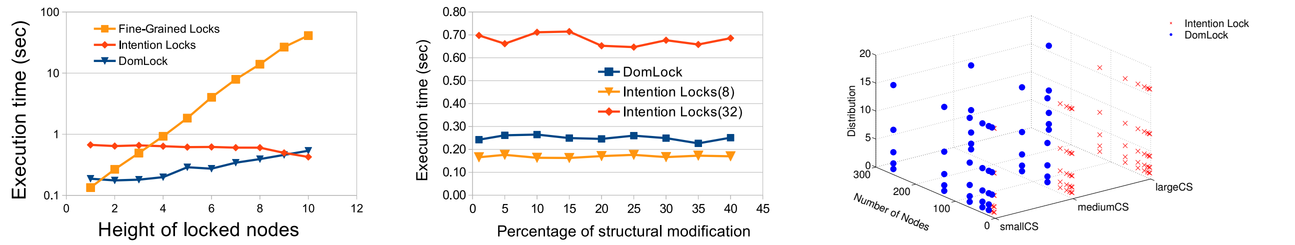
DomLock exhibits almost similar performance on changing L_{nodes} for a particular distribution. However, the performance of IL does not vary much across distributions. IL is less sensitive to distribution while operating on few L_{nodes}. As we increase L_{nodes}, IL also shows performance improvement according to the distribution. This happens as the skewness gives more concurrent subsets and hence the probability of conflict reduces.

5.4 Effect of Critical Section Size

The size of the critical section (CS) varies across real world applications. To assess the behavior of DomLock and IL for different critical section sizes, we execute the same program but for different critical section sizes. As we see in Figure 5, DomLock performs better than IL after the threshold value of L_{nodes}. Figures 5(a), 5(c), 5(e) depict the behavior for different sizes of critical section, i.e., smallCS (6μs), mediumCS (60μs), largeCS (600μs). For small critical section, the threshold value for the number of threads is approximately 10 (see Figure 5(a). This means that, with 10 nodes to be locked and having small critical section(CS) DomLock outperforms IL. If we increase CS size to mediumCS (60μs) and largeCS (600μs) then the threshold value of L_{nodes} also increases to 45 nodes and 260 nodes respectively. This shows that performance is affected by the critical section size. For small critical section, IL poses higher locking cost and if we keep on increasing the number of nodes to be locked, this cost goes on increasing. On the other hand, DomLock is not sensitive to L_{nodes} factor and poses less locking cost. DomLock may reduce concurrency in case of random distribution (see Section 5.3). To compensate the concurrency loss with larger CS, L_{nodes} should be higher. As can be observed from Figures 5(a), 5(c), 5(e), the threshold value for number of locked nodes goes on increasing with the critical section size. In case of a general graph data structure, the cost incurred by IL in locking multiple paths is considerably large and it almost always performs inferior to DomLock as shown in Figures 5(b), 5(d), 5(f).

5.5 Fine-grained Locking and Effect of Locking Height

Multi-granularity locking (MGL) can be implemented by locking nodes at different heights. In the degenerate case, locking the root node is coarse-grain locking, while locking the leaf nodes is most fine-grain. A fine-grain (hierarchical) locking of an interior node involves locking each node in its subgraph separately. In this section we compare the performance of fine-grained locking, intention locks and DomLock while varying the height of nodes to be locked. In Figure 7(a), x-axis shows the height of a node to be locked and y-axis is the execution time in seconds. Leaves are at height 0, their parents at height 1, and so on. Unlike Intention locks and DomLock, the cost of fine-grain locking increases exponentially as the locking height increases. Recall that fine-grain locking at a node needs to traverse complete sub-hierarchy beneath the node and lock each node separately. As the height increases, the size of the sub-



(a) Effect of locking height (note the logscale) (b) Effect of structural modifications (c) Overall performance distribution: DomLock vs. IL

Figure 7. Performance comparison of DomLock versus IL based on various parameters

hierarchies increases exponentially, proportionately increasing the locking cost. We observe that the execution time for IL is more than that of fine-grained locks nearer the leaves (up to height four). In case of IL, before locking the actual node, the locking protocol needs to lock all the ancestor nodes in intention mode which introduces overhead. The overhead reduces slowly as the locking height increases. DomLock avoids both: locking each node in the sub-hierarchy as well as traversing all the paths to a node. Therefore, it usually performs better than both IL and fine-grain locks up to a certain height. However, DomLock also reduces some concurrency as the locking height increases. This results in (slowly) increased execution time for DomLock. Nearer the root, the costs of DomLock and IL are nearly the same. From a theoretical perspective, coarse-grain locking (locking the root node) is an extreme case for both the IL and the DomLock where both of them perform constant amount of work. In contrast, fine-grain locking performs the maximum amount of work in this extreme case.

5.6 Effect of Structural Modifications

Structural modifications, such as edge-addition, are usually costly in data structures, compared to simpler read-update operations. However, this affects the critical section size; it does not affect the locking cost (assuming no specialized locks for read versus mutation, as in our setting). From the perspective of the application, it would still lock a set of nodes before performing structural modification. Thus, for both IL and DomLock, it is expected that the locking cost should be largely independent of the operations being performed. We study the effect of structural modifications in this section, which involve an edge-addition or an edge-deletion. In Figure 7(b), x-axis is the percentage of structural modification operations and y-axis is the execution time in seconds. The number of nodes to be locked is fixed as 32 nodes. Before adding or deleting an edge between two nodes, IL locks both the nodes in exclusive (X) mode and all the ancestors in intention-exclusive (IX) mode. IL treats both structural modification operations and regular write operations equally, and locks the two nodes involved in the mutation. The plot for IL (see Figure 7(b)) shows that IL is not sensitive to the number of structural modifications. However, in case of DomLock, there is additional bookkeeping involved in the presence of structural modifications. This is because change in the structure leads to change in the logical numbering of the nodes. Therefore, one would expect the cost of DomLock increase with the number of structural updates. Figure 7(b) indicates otherwise; it shows that even DomLock's cost is independent of the number of structural modifications. This happens because the amount of hierarchy-traversal required to *update* intervals (in presence of structural modifications) is the same as that required to *find* dominators (in absence of structural modifications). Thus, as in IL, we lock the dominator of the two nodes involved in a mutation in DomLock also (recall Algorithm 1). Because of the new changes in structure, we may have to update the intervals of their ancestors. However, the changes need to be propagated upwards only until the dominator. A node outside the sub-hierarchy rooted at their dominator would never be required to update its interval. By locking the dominator node, we ensure that all the interval-update operations are safe. In case of operations that do not involve structural updates, the same set of ancestor nodes until the dominator are traversed, leading to the same cost of locking.

5.7 Putting It All Together

Figure 7(c) shows a scatter-plot of relative performance of DomLock and IL for various configurations. The three axes represent critical section size, number of nodes to be locked, and the skew in the distribution. Each discrete point in the 3D space indicates a configuration of these three parameters. In this 3D space, for each configuration, we run the same code with intention locks and DomLock, and measure the throughput. When the throughput of DomLock is better for a configuration, we indicate it using a red circle, otherwise, we plot a blue diamond in the figure. We observe a clear separation of configurations for which IL or DomLock performs better. Specifically, the red dots, which represent DomLock, have higher density when we increase the value for the number of nodes to be locked. The density of red dots is thin for smaller value of L_{nodes} and larger critical section size. On the other hand, blue diamonds are gathered towards smaller value of L_{nodes} and higher value of CS size. We can see the cutting plane between red dots and blue diamonds. At one side this plane DomLock wins over IL. Hence, we can consider this cutting plane as an indicator to solve the puzzle of choosing one locking policy over another.

6. Related Work

DomLock enables practical usage of multi-granularity locking (MGL) in large and general hierarchies. The concept of MGL originated from the database community. MGL with intention locks is initially proposed by Gray (Gray et al. 1975) for shared databases. Other locking protocols such as *validation-based protocol*, *basic timestamp ordering protocol*, *multiversion timestamp protocol* are widely used in databases (Carey 1983). Compared to other hierarchical techniques, intention locks are widely used in databases.

Intention locks (IL) have now been adopted by domains outside relational databases. Liu et al. (Liu and Zhang 2014) presented the fine-grain locking for hierarchical data structures in parallel programs based on IL. They considered the fields of an object as the lowest level of granularity and applied fine-grain locking at the field level. In this technique, they create an abstract object graph for the parallel procedure and based on the node which is getting accessed, their method locks the root-to-node path according to the IL protocol. Providing fine-grained locking at the field level necessitates a separate lock for each field of each object. This increases the memory overhead of lock variables.

Cherem et al. (Cherem et al. 2008) adopted MGL to transform atomic sections in a concurrent program to the program that supports normal locking constructs. Their method refines the usage of coarse-grain and global locks to use finer-grain locks to reduce contention. Inferring fine-grain locks requires alias analysis information and their method locks the points-to set of each location to be locked using fine-grain locks. They use intention locks mainly to avoid deadlock when one atomic section wants to acquire multiple locks. In DomLock, we lock a single node (the immediate common dominator) on a request for multiple nodes, and do not encounter deadlocks.

Chaudhri et al. (Chaudhri and Hadzilacos 1995) proposed a locking policy for dynamic databases. Their methodology is based on the rule that each node can be locked by a transaction at most once. If the data structure contains any strongly connected component (SCC) then the transaction needs to lock all the entry points of the SCC. However, to guarantee safety, this policy needs to rely on a path-based traversal to ensure that all the ancestors are locked.

Hawkins et al. (Hawkins et al. 2012) formalized various aspects of lock placements including lock granularity and speculation. They presented a proof system for lock placements which forms a mapping between a lock and the data nodes guarded by that lock. Our DomLock technique also falls in such a category where one lock guards multiple nodes. Various locking options that our algebraic model generates provide locking at multiple granularity.

Several works (Cherem et al. 2008; Liu and Zhang 2014) on MGL have implemented their own library to support intention modes. Unlike MGL, DomLock does not need extra support and can be implemented relatively easily in standard concurrent libraries.

Dominators have been used in several data-flow analyses and in program analyses for software engineering. In the context of parallel programs, Liu et al. (Liu and Mellor-Crummey 2013) use common prefix property of calling contexts for efficient profiling of heap accesses. In their method, they identify adjacent memory locations for which duplicate calling context calculation can be avoided as an optimization, and then inserting a marker (called as *trampoline*) to find the dominator in the call stack for the two accesses. This method reduces the repeated unwindings of the call stack and is especially helpful when the heap allocations are frequent or the call-depth is high.

7. Conclusion

In this work we modeled an algebraic formulation of locking options which provides an intuitive way of trading off degree of concurrency against the locking cost. Exploiting the formulation, we presented DomLock, a new locking protocol based on the structural properties of the underlying data structure. We showed several configurations for which DomLock outperforms existing multi-granularity technique based on intention locks. Our experimental results using STMBench7 and a concurrent binary tree indicate interplay of parameters which affect the overall application performance. Specifically, we observed that the skewness in the distribution, the number of nodes to be locked and the critical section size play an important role in choosing a particular locking policy. Our formulation is versatile to allow one to choose a locking policy based on this trade-off. In future, we would like to use DomLock in auto-parallelization of codes.

Acknowledgments

We thank the reviewers and our shepherd Brandon Lucia whose comments improved the paper substantially. This work is supported by IIT Madras Grants CSE/13-14/812/NFIG/RUPS and CSE/13-14/636/NFSC/RUPS.

References

M. J. Carey. Granularity hierarchies in concurrency control. In Proceedings of the 2Nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, PODS '83, pages 156–165, New York, NY, USA, 1983. ACM. ISBN 0-89791-097-4. doi: 10.1145/588058.588079. URL `http://doi.acm.org/10.1145/588058.588079`.

M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 Benchmark. In Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, SIGMOD '93, pages 12–21, New York, NY, USA, 1993. ACM. ISBN 0-89791-592-5. doi: 10.1145/170035.170041. URL `http://doi.acm.org/10.1145/170035.170041`.

B. Chatterjee, N. Nguyen, and P. Tsigas. Efficient Lock-free Binary Search Trees. In Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing, PODC '14, pages 322–331, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2944-6. doi: 10.1145/2611462.2611500. URL `http://doi.acm.org/10.1145/2611462.2611500`.

V. K. Chaudhri and V. Hadzilacos. Safe locking policies for dynamic databases. In Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS '95, pages 233–244, New York, NY, USA, 1995. ACM. ISBN 0-89791-730-8. doi: 10.1145/212433.212464. URL `http://doi.acm.org/10.1145/212433.212464`.

S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '08, pages 304–315, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375619. URL `http://doi.acm.org/10.1145/1375581.1375619`.

P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent Control with "Readers" and "Writers". Commun. ACM, 14(10):667–668, Oct. 1971. ISSN 0001-0782. doi: 10.1145/362759.362813. URL `http://doi.acm.org/10.1145/362759.362813`.

K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. Commun. ACM, 19(11):624–633, Nov. 1976. ISSN 0001-0782. doi: 10.1145/360363.360369. URL `http://doi.acm.org/10.1145/360363.360369`.

G. Golan-Gueta, N. Bronson, A. Aiken, G. Ramalingam, M. Sagiv, and E. Yahav. Automatic Fine-grain Locking Using Shape Properties. In Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '11, pages 225–242, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0940-0. doi: 10.1145/2048066.2048086. URL `http://doi.acm.org/10.1145/2048066.2048086`.

J. N. Gray, R. A. Lorie, and G. R. Putzolu. Granularity of locks in a shared data base. In Proceedings of the 1st International Conference on Very Large Data Bases, VLDB '75, pages 428–451, New York, NY, USA, 1975. ACM. ISBN 978-1-4503-3920-9. doi: 10.1145/1282480.1282513. URL `http://doi.acm.org/10.1145/1282480.1282513`.

J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Data Base. In M. Stonebraker, editor, Readings in Database Systems, pages 94–121. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988. ISBN 0-934613-65-6. URL `http://dl.acm.org/citation.cfm?id=48751.48758`.

R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: A Benchmark for Software Transactional Memory. In Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, EuroSys '07, pages 315–324, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-636-3. doi: 10.1145/1272996.1273029. URL `http://doi.acm.org/10.1145/1272996.1273029`.

P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. Reasoning about lock placements. In Proceedings of the 21st European Conference on Programming Languages and Systems, ESOP'12, pages 336–356, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-28868-5.

doi: 10.1007/978-3-642-28869-2_17. URL http://dx.doi.org/10.1007/978-3-642-28869-2_17.

M. Herlihy and N. Shavit. The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. ISBN 0123705916, 9780123705914.

C. Lameter. Effective Synchronization on Linux/NUMA Systems, 2005. URL https://www.kernel.org/pub/linux/kernel/people/christoph/gelato/gelato2005-paper.pdf.

P. Liu and C. Zhang. Unleashing Concurrency for Irregular Data Structures. In Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pages 480–490, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568277. URL http://doi.acm.org/10.1145/2568225.2568277.

X. Liu and J. Mellor-Crummey. A Data-centric Profiler for Parallel Programs. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13, pages 28:1–28:12, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2378-9. doi: 10.1145/2503210.2503297. URL http://doi.acm.org/10.1145/2503210.2503297.

D. Lomet and M. F. Mokbel. Locking key ranges with unbundled transaction services. Proc. VLDB Endow., 2(1):265–276, Aug. 2009. ISSN 2150-8097. doi: 10.14778/1687627.1687658. URL http://dx.doi.org/10.14778/1687627.1687658.

D. B. Lomet. Key range locking strategies for improved concurrency. In Proceedings of the 19th International Conference on Very Large Data Bases, VLDB '93, pages 655–664, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc. ISBN 1-55860-152-X. URL http://dl.acm.org/citation.cfm?id=645919.672802.

J. M. Mellor-Crummey and M. L. Scott. Scalable Reader-writer Synchronization for Shared-memory Multiprocessors. In Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '91, pages 106–113, New York, NY, USA, 1991. ACM. ISBN 0-89791-390-6. doi: 10.1145/109625.109637. URL http://doi.acm.org/10.1145/109625.109637.

MSDN. Sql server 2016 database engine, 2015. URL https://msdn.microsoft.com/en-us/library/ms187875.aspx.

A. Natarajan, L. Savoie, and N. Mittal. Concurrent Wait-Free Red Black Trees. In T. Higashino, Y. Katayama, T. Masuzawa, M. Potop-Butucaru, and M. Yamashita, editors, Stabilization, Safety, and Security of Distributed Systems, volume 8255 of Lecture Notes in Computer Science, pages 45–60. Springer International Publishing, 2013. ISBN 978-3-319-03088-3. doi: 10.1007/978-3-319-03089-0_4. URL http://dx.doi.org/10.1007/978-3-319-03089-0_4.

Oracle. Oracle database 10g r2, 2015. URL http://docs.oracle.com/cd/B19306_01/index.htm.

H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. In Parallel and Distributed Processing Symposium, 2003. Proceedings. International, pages 11 pp.–, April 2003. doi: 10.1109/IPDPS.2003.1213189.

H. Sundell and P. Tsigas. Fast and Lock-free Concurrent Priority Queues for Multi-thread Systems. J. Parallel Distrib. Comput., 65(5):609–627, May 2005. ISSN 0743-7315. doi: 10.1016/j.jpdc.2004.12.005. URL http://dx.doi.org/10.1016/j.jpdc.2004.12.005.

Sybase. Adaptive server enterprise: Performance tuning and locking, 2003. URL http://infocenter.sybase.com/help/topic/com.sybase.help.ase_12.5.1/title.htm.