Interval based Framework for Locking in Hierarchies

Saurabh Kalikar and Rupesh Nasre; IIT Madras, India. {saurabhk, rupesh}@cse.iitm.ac.in

1 HIERARCHIES AND HIERARCHICAL LOCKING

Hierarchies are special linked structures, where each child node denotes a specialization or a part of its parents. For instance, 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. *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 Figure 1, hierarchical locking of node G requires that no other thread currently holds locks on (i) G itself, (ii) its descendant nodes M and N, and (iii) its ancestors A and C.

However, as an implementation, hierarchical locking poses various challenges. First is the design of an efficient mechanism to quickly check these hierarchical overlaps. Clearly, the naïve mechanism of traversing through the descendants and ancestors for every lock request works, but is impractical. Second is the design of a lock manager for maintaining the granted and the waiting lock requests. Third, the hierarchical locking should be well suited for the structural modifications to the hierarchy. Traditionally, *intention locks* [1] have been used to restrict locks on a node if there is a lock acquired on at least one node in the sub-hierarchy. In the next section, we present our interval based hierarchical locking technique and show how it tackles all these challenges.

2 INTERVAL BASED LOCKING

Logical intervals. We use a bottom-up interval numbering scheme for hierarchies. In this scheme, each node is assigned an interval [low, high]. First, the leaf nodes are numbered with unique intervals $[i, i]$. Then, the intervals of the leaf nodes are propagated to their ancestor nodes level by level up to the root node. The intervals possess certain properties which are useful for hierarchical locking. (i) The intervals of two nodes overlap if and only if the two nodes have a common reachable descendant node. (ii) If the interval of a node n_1 completely subsumes that of n_2, then n_1 is a dominator of n_2 (that is, all the paths from root to n_2 pass via n_1). (iii) If two intervals do not overlap, their corresponding nodes can be simultaneously locked. Figure 1 shows the [low, high] intervals associated with each node. Since intervals of nodes D and E (that is, [1, 2] and [1, 4] respectively) overlap, it indicates that there exists a common descendant node (nodes H and I). Therefore, D and E cannot be locked simultaneously by two threads (otherwise, there can be a data-race at nodes H and I, leading to transactional inconsistency). Further, since the intervals of nodes D and G (that is, [1, 2] and [6, 7] respectively) do not overlap, they can be concurrently locked using a multi-granularity protocol (MGL). In the case of cycles, every pair of cycle nodes has an ancestor-descendant relation. Therefore, each cycle node receives the same interval value (which is logically equivalent to collapsing the cycle). In this way, the logical intervals allow us to quickly check the hierarchical overlaps.

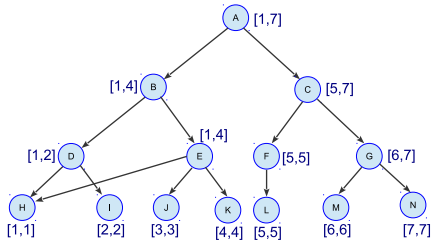

Figure 1: Example hierarchy

Lock Manager. The lock manager maintains locks in the form of numbered intervals. Unlike the conventional design of lock managers whihch is implemented as a hashed index of resources and waiting queues, we index our pool of locks according to thread-ids. Thus, every thread has a specific location where it inserts its lock entries. The lock acquisitions by multiple threads happen in parallel as follows. Initially, each thread requesting for locks receives a unique sequence number (such as timestamp of request), and its lock intervals are atomically inserted into respective positions. The lock intervals for each thread are maintained in sorted order to avoid deadlocks. After insertion, each thread independently checks whether there is any conflicting entry in the lock-pool having a smaller sequence number. If the thread does not find any such overlap, the lock on all the inserted intervals is granted. Otherwise, the thread needs to try again later. Ideally, the insertion of intervals and the overlap check should happen atomically to restrict any concurrent insertion from other threads. However, this demands heavy-weight reader-writer locks. We remove this synchronization bottleneck by careful use of sequence numbers and pointer manipulations, which allow us to check overlaps in lock-free manner.

Structural modifications. Our technique 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 interval subsumption property. While inserting an edge $u \rightarrow v$, we first find a set S of only those ancestors of node u (including u) whose interval values need to be changed because of the insert operation. Therefore, an exclusive lock has to be acquired on the dominator of S and executes insertion process followed by update to the interval values of the ancestor nodes $G \in S$. 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, the addition of edge $G \rightarrow L$ in Figure 1 populates the set as $S = \{G\}$. We acquire an exclusive lock on G and update the intervals of G to [5, 7]. In the 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,

	Locking options	Locking cost	Concurrency cost
1	{H, J, M, N}	4	0
2	{G, H, J}	3	0
3	{E, M, N}	3	1
4	{B, C}	2	3
5	{E, G}	2	1
6	{A}	1	3

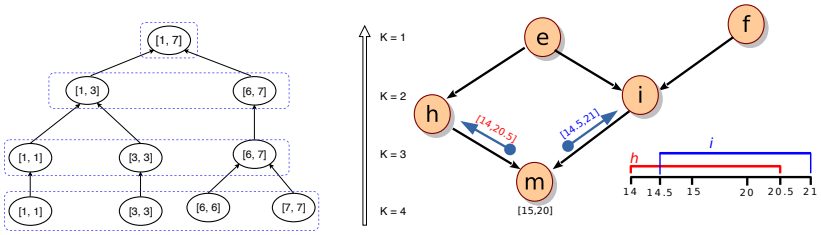

Figure 2: (a) All possible locking options (b) Pareto-optimal options (c) Hi-Fi interval propagation

the deletion of the edge $G \rightarrow M$ in Figure 1 need not necessarily update G's interval to [7, 7] for correctness, although doing that would improve concurrency.

3 SPECIAL CASES OF HIERARCHICAL LOCKS

DomLock [3]. The operations accessing more than one node in the hierarchy incur high locking overhead due to the number of overlap checks. DomLock suggests that if locking request consists of multiple nodes, then lock the immediate common dominator node of all the requested nodes. Consider a locking request with nodes L and N. DomLock finds their immediate common dominator C using intervals and the subsumption property. Actual locking happens by keeping track of the interval value of the dominator node in the lock pool, (which is C[5, 7]) and ensuring that none of the intervals of the locked nodes overlaps with [5, 7]. Therefore, DomLock has unit locking cost. Note that, locking the dominator locks an extra node M which introduces an extra concurrency cost (by losing concurrency). In many real-world cases, DomLock in spite of incurring concurrency cost outperforms the existing hierarchical locking technique *intention locks* [1] because of efficient hierarchical locking.

NumLock [4]. DomLock may incur high concurrency cost if the requested nodes are far apart in the hierarchy. For instance, the dominator of node H, N is the root node which locks whole hierarchy. Here, instead of the root, locking H and N separately could be a better option. NumLock systematically addresses this issue. There could be multiple ways to serve a locking request. Consider a locking request for nodes {H, J, M, N}. Figure 2(a) shows different locking options for the request. The number of such options could be exponentially large in number. We target the problem in two stages: 1. Generate a set of few pareto-optimal locking options (highlighted in Figure 2 (a)) 2. Apply a cost model to pick the best one among those. Stage 1 sorts the intervals of the nodes to be locked and gradually merges two intervals into one with least merging cost. In Figure 2(b) each level shows one pareto-optimal option. Stage 2 applies the cost model to compare these options. The cost model pinpoints the one with minimum overall execution time by considering various parameters such as, locking cost, size of critical section, number of parallel threads, and contention index.

Hi-Fi [2]. Simultaneous handling of hierarchical and fine-grained requests poses new challenges in checking for racy requests. We propose a new interval indexing technique for hierarchies which uniquely identifies every node as an interval value and effectively captures hierarchical dependences between nodes of the hierarchy. If the operation is local to the node, say update of some attribute

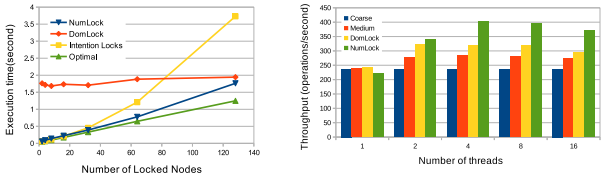

Figure 3: Performances of various locking techniques

value, then acquiring a hierarchical lock adversely affects parallelism. For instance, a fine-grained lock on node F in Figure 1, should only lock node F but not L. Presence of both hierarchical and fine-grained locking semantics is tricky. In Figure 1, more than one node can get the same interval, hence the logical intervals alone can not distinguish between two nodes uniquely. For the unique identification, we move from integer numbers to floating point intervals. Figure 2(c) shows an intermediate stage in the interval numbering process. Here, both h and i subsume the child node m and partially overlap each other. Hi-Fi modifies the locking protocol as follows: (i) A node is allowed to lock in fine-grained mode if the interval of the node is not subsumed by any existing locked interval in non-compatible mode. (ii) A node is allowed to lock in hierarchical mode if the interval of the node does not overlap with any interval locked in non-compatible mode. In this way we achieve both hierarchical and fine-grained locking.

4 EXPERIMENTATION

Our locking framework provides a way to compare different instances of hierarchical locking. Figure 3 shows the execution time taken by each technique with 32 parallel threads each executing 1000 operations by varying number of nodes locked. We observe that the time taken by intention locks (IL) grows linearly with the number of locked nodes. On the other hand, although DomLock takes more time, it remains fixed. NumLock actually balances IL and DomLock and outperforms both these techniques.

We also implement interval based hierarchical locking technique as part of STMBench7. We compare DomLock, NumLock, and built-in coarse-grained and medium-grained locking techniques in STMBench7. NumLock achieves on an average 25% throughput improvement over DomLock.

REFERENCES

[1] J. N. Gray, R. A. Lorie, and G. R. Putzolu. 1975. Granularity of Locks in a Shared Data Base. In *VLDB 1975*. ACM, New York, NY, USA, 428–451.

[2] Ganesh K, Saurabh Kalikar, and Rupesh Nasre. 2018. Multi-Granularity Locking in Hierarchies with Synergistic Hierarchical and Fine-Grained Locks. In *EuroPar 2018*. ACM, New York, NY, USA.

[3] Saurabh Kalikar and Rupesh Nasre. 2016. DomLock: A New Multi-granularity Locking Technique for Hierarchies. In *PPoPP 2016*. ACM, New York, NY, USA.

[4] Saurabh Kalikar and Rupesh Nasre. 2018. NumLock: Towards Optimal Multi-Granularity Locking in Hierarchies. In *ICPP 2018*. ACM, New York, NY, USA.