An Analytical Model for the Performance of Concurrent B Tree Algorithms

by

Dennis Shasha, Vladimir Lanin, Jeanette Schmidt
(shasha@nyu-csd2.edu, lanin@nyu-csd2.edu, schmidtj@nyu.edu)

Ultracomputer Research Laboratory
Courant Institute of Mathematical Sciences
715 Broadway, 10th Floor
New York, NY 10003

Ultracomputer Note #124
Computer Science Technical Report #311
July, 1987

This work was partially supported by the National Science Foundation under grant number DCR8501611 and by the Office of Naval Research under grant number N00014-85-K-0046.

ABSTRACT

A dictionary is an abstract data type supporting the actions search, insert, and delete. Search structures are data structure used to implement a dictionary, e.g. B trees, hash structures, grid files, etc. Many concurrent algorithms for search structures have been proposed, but relatively little work has addressed their performance. This paper proposes an analytic performance model for a range of concurrent algorithms on B+ trees. It uses this model to propose a high performance, simple-to-program variant of lock-coupling algorithms.

The model actually consists of two mutually exclusive models: a ‘‘constant flow’’ model in which the throughput is assumed to remain at its average value; and a ‘‘bursty flow’’ model in which the throughput is predicted to oscillate with a certain frequency and amplitude based on the number of processes, the height of the tree, and the size of the nodes. We find that the data points from simulation fall between these two predictive models, tending towards the bursty flow one as the number of processors increases. Inasmuch as the analytical model characterizes B+ trees in terms of fanout, height, probability of restructuring, and node size, it does not depend on the details of B+ trees and so can be extended, at least in spirit, to other structures.

Index terms: parallel, concurrent, asynchronous, data structure, B tree, performance.

1. Introduction

Concurrency control is the activity of preventing harmful interference among asynchronous parallel processes in a database management system. The more restrictive this control, the less parallelism there is. As multi-processor configurations become widespread, efforts to lessen the restrictiveness of concurrency control may be an important contributor to performance.

Concurrency control algorithms for database systems that view items as elements of an associatively accessible set have been widely proposed [2]. There has also been extensive performance simulation [5] and analysis [1],[39],[38] of such protocols.

However, the items of a database are usually accessed through a search structure index. A system that forced these accesses to occur serially would be slow. Therefore, there has grown up a variety of concurrent algorithms on search structures, as we will review in section 2.1. Some of these algorithms bear a close resemblance to database algorithms, especially of the tree-locking variety [22], but some new algorithms that exploit the semantics of search structures have also been invented, inspired by [18],[25],[13],[20].

Because search structure accesses are in the ‘‘inner loop’’ of queries, it is surprising that performance work on concurrent search structure algorithms is only now beginning to surface [4],[12],[36]. We review that work in section 2.2. The goal of this paper is to provide a simple analytical model for predicting the performance of a variety of algorithms.

2. Related Work

2.1. Search Structure Algorithms

Bayer and Schkolnick [3] propose a collection of *lock-coupling* protocols. In lock-coupling, a search or update execution locks the child of a node before releasing its lock on the node itself. Thus, a process always holds at least one lock during its execution.

Lock-coupling protocols [35],[8],[23],[28],[29],[22] produce conflict-preserving serializable computations [2],[7],[30]. Intuitively, the node level operations of a conflict-preserving serializable computation can be rearranged to create a possible sequential computation with the same effect on the data structure and the same return values.

Kung and Lehman [18] proposed the first non-conflict-preserving concurrency control protocol for search structures, which added forward pointers or links to the structure. Lehman and Yao [25] adapted their ''link'' technique to B trees. Search, insert, and delete executions in their algorithms release the node they hold before they obtain a lock on the next node. Thus, a process may hold no locks during some parts of its execution; this is impossible for any action using the lock-coupling protocol. The link technique therefore offers the possibility of more concurrency than lock-coupling, but requires the addition of new edges or links to the structure in order to prevent anomalies.

For example, figure 1 shows the stages of a split in the Lehman and Yao algorithm. Note the edge between n and n' following the split. Let us consider a possible computation. Insert(v) and search(x) read the root at the same time, then the insert adds v to n and splits n. The search proceeds to n but then traverses the edge from n to n'' and returns x. Thus, the computation is serializable, even though it is not conflict-preserving serializable since the search action would not go to two leaf nodes in any possible serial computation.

Other algorithms have been proposed that use the link technique. Manber and Ladner [27] use the link technique in a binary tree algorithm. Ellis, Hsu and Yang, [9],[10],[17], applies the link technique to hash structures. [36],[14] proposes algorithms for several structures [26]. [31],[24] present concurrent B+ tree algorithm that treats deletes symmetrically with inserts.

Orthogonal to the locking technique used is the question of when to issue restructuring operations. In the Manber and Ladner algorithm, maintenance actions do all restructuring. Guibas and Sedgewick [15] and Mond and Raz [28] suggest that an updater on argument x should restructure before locating x. For example, during its descent phase, an insert will split a node that is already full. A more standard algorithm would not do so, but might be forced to split the node in a separate reascent, if the insert is successful. We will analyze variants of the Bayer and Schkolnick, Mond and Raz, and Lehman and Yao algorithms (see appendix A for the detailed description of these algorithms).

2.2. Performance Studies

The few performance studies that have been done on concurrent search structure algorithms have disagreed about their metric. Bayer and Schkolnick [3] proposed the number of locks held as a metric. Ellis [10] cited the maximum number of concurrently executing transactions as a metric.

But these are still indirect measures. The real measure should be throughput. These measures depend on the data contention of a given concurrent algorithm and on the resource contention present in a given hardware configuration. For example, if the data is on only one disk, resource contention will dominate all other considerations and throughput will be bad for all algorithms. On the other hand, if there are many disks, and a fast shared memory, the performance will depend almost exclusively on the data contention present in an algorithm.

Ford, Jipping, and Shultz [12] argue for a resource-based analysis. Their analysis and simulation take the architectural details of their particular distributed system into account for the algorithm they looked at. For example, they bound the probability of certain kinds of interference by appeal to the number of bus cycles certain operations take. They argue that their method of analysis should be the paradigm for system implementors.

Carey and Cheng [4] model the resources in their system as a queuing system with terminals, a CPU, a buffer, and disk drives. They can vary these parameters in their simulation experiments from one extreme in which resource contention dominates to another extreme where data contention dominates.

Our work [36],[24] simulates throughput assuming no resource contention (two accesses to different pages may always execute concurrently), only data contention. Data contention comes about when two accesses to the same page conflict (i.e. one updates the page).

Ignoring resource contention means that the results will not reflect actual conditions in a real system (like government gasoline consumption tests). The results should reflect the relative performance of the different algorithms in resource-rich environments (e.g. shared memory multi-processors on fast buses). Removing resource contention leaves us with an understandable analytical model – an important objective of our work. However, we agree that resource contention must be modelled in order for precise predictions to be valid.

3. B+ trees

A *B+ tree* [6] is a tree with the following properties:

1) All keys are stored in the leaf nodes.

2) All paths from the root to a leaf node have the same length.

3) Each non-leaf node *n* in the tree consists of a sequence ($v_1, p_1, v_2, \ldots, p_q, v_{q+1}$). where each v_i is a value and p_i is a pointer to a child of *n* in the case of non-leaf nodes or a record in the case of leaf nodes. The values (called *separator* values) are ordered in ascending order ($v_i < v_{i+1}$ for $1 \le i \le q$). All keys in the leaves of the subtree pointed to by p_i are a subset of $\{ x \mid v_i \le x < v_{i+1} \}$.

4) The number of keys in leaf nodes or value-pointer pairs should be between $\lceil s/2 \rceil$ and s inclusive for some s.[1]

The B+ tree supports search actions (i.e. determining whether a value is in the tree) insert and delete actions. The latter two are called *updaters.* An updater is *successful* if it changes the contents of the tree, otherwise it is *unsuccessful.* A node is said to be *safe* with respect to an insert if the number of keys or value-pointer pairs it contains is less than s. That is, a new element could be put into the node without causing the node to split. Safety is defined similarly with respect to delete.

A *split* of a node n entails creating a new node n', moving the greater key (or value-pointer pairs) of the data from n to n' and changing the parent of n to include a pointer to n' as well with an appropriate separator value (figure 2 shows a split in a non-link structure). This new pointer in the parent may cause the parent to split, which may cause its parent to split and so on. This is called *cascaded bottom-up splitting.* One way to prevent cascaded bottom-up splitting is to split top-down. That is, whenever an inserter finds an unsafe node (i.e. one to which an insertion would cause a split), it splits it (preemptively, as it were). This is called *top-down splitting* [15].

There is another kind of split that involves a slightly different structure called the B+ link tree, introduced by Lehman and Yao. This structure is a B tree with a singly linked list at every level from left to right.

A half_split creates a new node and transfers the half consisting of greater valued elements from the original node to the new one as illustrated in figure 1. The complete_split puts the appropriate pointer in the parent node.

Since all the concurrent algorithms we consider involve locking, we define the two lock types: *shared* and *exclusive.* (Exclusive locks are denoted *elocks* and shared locks sometimes *slocks*). Two actions may hold a shared lock on the same node concurrently, but if one action holds an exclusive lock on a node, then no other action may hold any lock on the node.

3.1. Protocol characterization

Among the classical lock-coupling protocols, the search actions always use shared locks and only hold parent-child pairs (see first page of appendix A). For updaters, there are three major orthogonal decisions:

Decision 1. Either:
(conservative) an updater always begins with an exclusive lock on a node that it locks, since it might later have to modify the node
or
(optimistic) in its first descent, the updater descends as a search action would (appendix A) except that it

[1] In practice, many systems don't merge until the number is zero. So the upper bound is more serious than the lower one.

obtains an exclusive lock at the leaf level. Normally, it will modify the leaf and exit. If it has to split a node, the updater may have to start over using exclusive locks.[2]

Decision 2. Either:
(bottom-up) split and merge if necessary after performing an insert or delete and start with the leaf level
or
(top-down) during descent, an inserter splits any node that is full and a delete merges any node that is half-full [15]. (We give credit for the concurrent version of this algorithm to Mond and Raz [28] though it was discovered independently by Carey [4] and was anticipated by Guibas and Sedgewick themselves. The optimistic version presented here comes from [24].

Decision 3. Either:
(two lock strategy) use exclusive locks and shared locks only
or
(multi-lock strategy) use exclusive locks, shared locks, and various kinds of intention locks.[3] Intention locks exclude other intention locks and exclusive locks, but are compatible with shared locks. This strategy improves on the pessimistic strategy above by allowing searchers to proceed concurrently with updaters, until the updaters perform actual changes. This is better than the conservative approach, but is almost always worse than the optimistic approach (because of updater-updater conflicts).

Our simulations and the analytic formulas we have developed explore the possibilities presented by the available combinations of the first two decisions. However, we concentrate on optimistic algorithms because they perform better and have a non-trivial analysis. We also analyze the Lehman and Yao style algorithm, which is optimistic and bottom-up. In all of these algorithms, we concentrate only on searches and inserts for simplicity. Appendix A contains the programs for the algorithms we have simulated and analyzed.

4. Simulation Conditions

We have simulated these algorithms using Concurrent Euclid [16] and the following conditions.

1. Any modified node must be written to disk.
2. The height of the B+ tree doesn't change during the simulation.
3. Every insert is successful (deletes remain an open problem).
4. In cases where there are inserts and searches, each incoming action is given a certain probability (depending on the experiment) of being an insert independently of any other actions. The arguments

[2] The term ''optimistic'' is used in a similar sense to Kung and Robinson's [21]. The difference is that our updaters will restart if they have to split or merge a node, whereas Kung and Robinson's updaters would restart whenever they discover a conflict.

[3] Many different kinds of locks are useful in other contexts, [19].

of different actions are chosen randomly.

5. Lock requests are granted in first-in first-out order, though shared lock requests are handled in batches. Thus, a shared lock request arriving when a node is held by shared locks is granted immediately, unless there is an exclusive lock request pending. A shared lock request arriving at a queue containing exclusive lock requests joins the other shared locks in the queue. If there are none, a new batch is started immediately after the last exclusive lock request. Once a lock is granted, the grantee may immediately access the node, since there is no resource contention. Thus, all contention is due to locks.

6. There is a backlog of dictionary actions keeping the processes busy forever.

4.1. Cost models

The values of the remaining parameters depend directly on the given cost model. There are two: one assumes that input/output costs dominate (called the IO cost model). The other (called the Buffer cost model) assumes the existence of a buffer for the first two levels of the tree and assumes that processing has a cost. Costs are calculated as follows:

1. IO cost model: it takes one unit of time per read or write of a node from disk. The in-memory processing of a node costs nothing.

   Buffer cost model: it takes one unit of time per read or write of a node from disk. The in-memory processing of a node cost 0.3 units [4].

2. In the buffer cost model, the processing time for adding a key is included in the time for finding a key. However the processing time for moving data from one node to another is 0.6 (for the operations on the two nodes), even if one of the nodes has had a key added to it.

3. In the buffer cost model, the highest two levels of the B+ tree are assumed to remain in memory. Also, if a process has visited a node at depth three or greater, then that process never needs to reread that node from disk. However, this rereading buffer is private: other processes would still have to get that node from disk. This allows us to ignore the memory management discipline and does not lead to substantive inaccuracies, since different nearly concurrent processes will rarely access the same node at depth three or greater.

In the IO cost model, if a process *p* accesses a node, it must first read that node unless *p* has read the node since it acquired a lock on it.

In the following, the first member of each pair will be the time assuming the IO cost model and that in-memory processing takes no time; the second member will assume the buffer cost model, that the first two levels are in memory, and that in-memory processing requires 0.3 units.

split_time - time to perform a single split for the Bayer and Schkolnick or Mond and Raz algorithms assuming node arguments are already in memory is (IO: 3, Buffer: 0.6 + 0.3 + 3), i.e. the time to do the data movement, update the parent, and write three nodes.

half_split - time to perform a Lehman and Yao half_split assuming the nodes are already in memory is (IO: 2, Buffer: 0.6 + 2), i.e. the time to do the data movement and write both children.

complete_split - time to perform a Lehman and Yao complete_split, (IO: 2, Buffer: 0.3 + 1). In the IO model, the node has to be reread. In the buffer model it doesn't.

If a node is split and its parent is subsequently split, the parent only needs to be written once. Thus, the time to write the pages and do the associated processing for x splits in any algorithm is is (IO: 2x, Buffer: 2.9x), in addition to the write of the leaf node where the insert that caused the splits took place.

Some additional reads may be required to set up the split. These require (IO: h, Buffer: h $\times$ 0.3) for the lock-coupling algorithms and (IO: x, Buffer: 0) for the Lehman and Yao algorithm, where h is the height of the tree. The reason for the discrepancy is that the lock-coupling algorithms reread from the root.

5. Analytical Model

Devising a tractable performance model for these algorithms requires approximation. We divide these into approximations about the structure and approximations about process interactions. We have one set of approximations about structure, including the behavior of individual actions on the structure.

However, we have two mutually exclusive sets of approximations about process interactions: one says that the flow of processes through the structure is constant at the mean and the other says that the flow consists of bursts of processes flowing through the structure interspersed with blocking at the root. The first one (called the constant flow model) characterizes the behavior when there are few concurrent processes. The second one (called the bursty flow model) characterizes the behavior when there are many concurrent processes.

With an intermediate number of processes, the behavior should fall in the middle. An important open problem is to characterize analytically the situations in which one approximation or the other is best.

5.1. Structure approximations

In the following, s is the maximum size of a B tree node. h is the height of the tree. Height (or level) 1 is the height (or level) of the leaves. N is the number of keys at the leaves.

1. We assume that the number of descendents of a node at level i is proportional to the number of the node's immediate children. From this and results by Yao [40] it follows that the average number of nodes at height i is $N/(0.69s)^i$ for $i < h$ (there is always exactly one node at the root level).

2. The probability that an item is a descendant (or would be a descendant if inserted) of a given node at level i is 1/number of nodes at level i, in the absence of information about the load factor of the node. However if the node is known to have r_j descendents we assume (following from assumption 1) that the probability is $r_j /$ number of nodes at level $i-1$, which (again following from assumption

1) is about $r_j/(0.69s\times$ number of nodes at level i). In particular, if the node is known to be full, the probability is assumed to be (1/0.69) $\times$ (1/number of nodes at level *i*).

3. We will assume that splits behave like independent binomial variables, i.e. that an insert will cause a split with probability *p*, independent of the splits caused by previous inserts.

Justifications and Implications

Point 1. It was proven in Yao [40] that the average number of nodes at level one is asymptotically $N/(0.69s)$. Yao assumes in his analysis that the inserted elements come from a common distribution and hence their order of insertion is a random permutation of their order at the leaves. His analysis relies on the fact that a node at level 1 which contains *k* keys has probability $(k+1)/(N)$ of being accessed by the N^{th} insert. This is not true in general at any level higher than 1, however generalizing Yao's bound to higher levels merely boils down to assuming that the number of descendents of a node at level *i* is proportional to the number of its immediate children, which is generally close to the true value as the simulations show.

Point 2. As mentioned in point 1, fuller nodes have a higher probability of being hit. Moreover the conditional probability of hitting a node, given say that that node has just been accessed, may not be 1 / number of nodes at level i, since the number of descendents of a node is not independent of the probability of accessing it. However since the information revealed seems to be minor we will assume that the correlation is negligible.

Point 3. Clearly the splits caused by successive inserts are not totally independent. However since most splits occur at very low levels in the tree, where the number of nodes is very large, the correlation between the probability of a split for successive inserts is low. Another important justification for this assumption is that it makes the analysis tractable.

5.2. Process flow approximations

Here, we describe two mutually exclusive models: the constant flow model and the bursty flow model. Before we begin, we define a few terms.

A process is *at level i* if it either holds a lock on level *i* or is waiting for a lock on level *i* and these two conditions don't hold for any level below *i*.

A lock request is *at level i* if it is in a queue at level *i* or holds a lock at level *i*. Sometimes, we drop the term ''request'' for brevity, saying *an elock is at level i* instead of *an elock request is at level i.*

There are two sources for elock requests: restructuring (splits in our case), and simple updates (inserts in our case). We will call the first kind of requests *splitting elock requests* and the others *simple elock requests*

A *splitting process* is an insert process that has discovered that the leaf into which it should have inserted was full. Initially, all search and insert processes are non-splitting.

5.2.1. Constant flow model

1. The number of non-splitting processes entering any level of the tree in one time unit is the same at every level at any time.

   Since the number of non-splitting processes entering the root at time i is equal to the throughput at time $i-1$, it follows that the number of non-splitting processes entering any given level at time i is equal to the throughput.

2. The length of time required to service an exclusive lock or shared lock once acquired at a node at level i is constant at its expected value (determined by the algorithm and the height i of the node).

3. Splitting elock requests and simple elock requests contribute independently to the waiting time of other actions.

Algorithm-Specific Assumptions

4. In the Bayer-Schkolnick and Mond-Raz algorithm, splitting elock requests only cause delays at the root.

5. In the Lehman-Yao algorithm, splitting elock requests cause delays predominantly at the lower levels of trees.

Justifications and Implications.

Points 1 and 2. The constant flow model characterizes the situation when there are no queues of processes caused by elocks and all delays are random. In that case, the flow through the various levels should be uniform. When the number of processes is small, there are few conflicts, so the assumptions of constant flow seem reasonable. For algorithms which elock the root frequently for more than one time unit, a constant flow is clearly impossible, since every time the root is locked no processes will enter level $h-1$. When the elock is released exactly one processor will enter level $h-1$. For these algorithms ‘‘an almost constant flow’’ can still hold, meaning that the flow will be constant within some small time unit which is bigger than one. When the timeunit for which the flow is constant gets large (close to the time it takes to perform a search on the B-tree) the model becomes a bursty flow model as explained in the next subsection.

From the assumption of constant flow it follows that a process which elocks the root, and hence blocks the access to the root, will not cause a long queue of processes waiting to aquire the root. If this were the case, the flow could not remain constant. On the contrary what would happen is as follows. A large number of processes would be forced to wait at the root. When the waiting processes finally run, they would generate a large group of elocks at about the same time (when some portion of them discover that they must split a node). This would cause a long block of flow in the upper levels of the tree on one hand, and on the other hand it would cause a large number of processes to wait at the root on turn. We would then get a cyclic behavior which results from positive feedback. This behavior becomes more pronounced as the number of processors increase, all else being equal. Figure 3 shows the number of processes waiting over time for the Bayer-Schkolnick algorithm on a three level 20-39 B-tree using pure inserts and only 10 processes. Figure 4 shows the waiting over time for the same conditions except with 40 processes, showing clear cyclic behavior. Figure 5 shows the results with 40 processes using the Lehman-Yao algorithm (which seldom locks the root).

Point 3. Simple elocks only occur at the leaves, and occur even in the absence of splits. The delays they cause seem to have little correlation with the delays caused by splitting elocks.

Point 4. An action that acquires an elock on the root using the Bayer-Schkolnick or Mond-Raz algorithms only releases that elock after it has acquired and released the child of the elock. A shared lock following such an action will only rarely catch up to it and have to wait for that process again. Simulation bears this out.

Since all shared locks have to go through the root, whenever the throughput is greater than one it is likely that whenever an elock arrives, a shared lock will have just preceded it.[4] Simulation bears this out.

Point 5. The Lehman-Yao algorithm acquires most of its elocks low in the tree.

5.2.2. Bursty flow model

We use the bursty flow model for the optimistic Bayer-Schkolnick and Mond-Raz algorithms to characterize these algorithms when they cause splits at high rates.

1. Concurrent processes work and wait in a cyclic fashion. At the beginning of a cycle, all processes descend the tree. They all complete except for say m inserts that cause splits.

A possible scenario would then look as follows: all the m splitting inserts acquire the root lock sequentially causing all the other $p - m$ processes to form a queue behind them. When the splitting inserts release the root lock, the cycle ends. This would imply that each process waits for m exclusive locks at the root when performing any action.

2. There is no additional waiting due to simple elock requests.

This model is a caricature of the situation when there are many concurrent processes. It is not accurate even when most of the waiting is at the root. The cycle is not always the same (see figures 3, 4, and 5), nor is the flow entirely bursty.

6. Predictions

In this section, we describe the quantitative predictions of each model. First, we describe the parameters we will use in these predictions and the subsequent detailed calculations.

[4] Remember that many shared locks may hold a node at the same time. An elock request will therefore never have to wait more than the time it takes for a shared lock to hold a node.

6.1. Parameters

B+ tree height = h. (height of 1 is leaf level)

maximum size of nodes = s (minimum size $\lceil s/2 \rceil$, average size of non-root nodes 0.69s, average size of root node s/2).

number of processes = p

throughput = e, in units of number of dictionary actions completed per unit time. Sometimes we specify the components. For example, e_{insert} is the throughput of inserts. If inserts and searches are mixed, the total throughput is simply the sum $e_{insert} + e_{search}$.

There are three parameters that determine data contention due to restructuring. The expressions following the definition of the parameters assume that only inserts (or inserts and searches) take place.

p_split(i) is the probability that a given insert action will cause a split to some node at height i. According to the structure assumptions of section 5.1, p_split(i) = $(1/(0.69s))^i$.

p_highest(i) is the probability that the highest node that will be split by a given insert action is at height i. In other words, it is the probability that a node at height i will be split and no higher ones will be. Thus p_highest(i) = p_split(i) − p_split(i+1).

p_hit(i, r) is the probability that a process will access a node *n* at height *i*, given that *n* has *r* entries. As explained in the structure assumptions of section 5.1, we approximate $p\_hit(i,r)$ by $r/$number of nodes at height $i-1$, which is about $r/$((number of nodes at height i) $\times$ 0.69s).

p_hit(i) is the probability that a process will access a given node at height *i*, where the length of the node is unknown. Thus, p_hit(i) = 1/(number of nodes at height i).

6.2. Constant Flow Model

The constant flow model will be applied to all the algorithms for light insert loads and to the Lehman-Yao algorithms for all loads. The goal is to find the average time that a job takes. This can be broken down into the time the job is active and the time it spends waiting for locks.

For the Bayer-Schkolnick and Mond-Raz algorithms, the analysis is divided into two parts : one part deals with splitting elocks at the root and the other with simple elocks at the leaves. The main conceptual difference is that the root analysis assumes that a shared lock will hold the root when an elock arrives. The Lehman-Yao algorithm uses a generalization of the analysis for simple elocks at the leaves.

In both analyses, we try to find the waiting time of a process in its first descent. Since very few processes cause splits we ignore the waiting time in the second descent. Thus, our goal is to determine the time that a process waits for exclusive locks at each level in its first descent. These waiting times are summed to get the total waiting time.

For simplicity, we derive the formulas for the root and for the non-root levels separately. It turns out that the formulas are quite similar, but certain subtleties in their motivation and applicability suggest the separation.

6.2.1. Root Analysis

This analysis is only concerned with splitting elocks at the root. It applies to any algorithm that elocks the root frequently. In our case, it applies to the optimistic Bayer-Schkolnick and Mond-Raz algorithms.

The time during which an elock is at the root consists of two parts: the service time of the shared lock for which the elock is waiting, and the service time of the elock itself. The first quantity is called sservice(h), the second eservice(h). We call this sum the *duration(h)*.[5]

We now want to calculate the probability of an elock at the root. Let *numelock(h)* be the arrival rate of elocks at the root.

The likelihood that there is an elock at the root is the expected fraction of time that elocks hold the root. Thus, the likelihood that there is an elock at the root equals the product of the arrival rate of elocks at the root and the length of time that each elock is at the root, i.e. numelock(h) × duration(h). (Strictly speaking, this expression gives the expected number of elocks at the root and might exceed 1. However, the constant flow model will only be valid when the expression is substantially less than 1, so we interpret it as a probability and assume there is never more than one elock at the root.)

Shared locks will arrive in a constant flow following the elock and will be forced to wait. The average waiting time for these shared locks is duration(h) × 1/2. The expected time from the request to beginning of service of a shared lock request at the root is then:
prob(finding an elock at the root) × (expected waiting time | there is an elock at the root)
= numelock(h) × duration(h) × duration(h) × 1/2.

To complete this derivation, we note that numelock(h), the arrival rate of elocks at the root is equal to the average number of splitting processes per unit time which is on turn equal to prob of an insert to cause a split × throughput of inserts.

[5] Since sservice(h) is the length of time that a root lock is held, one might think that it depends on the waiting time of the process holding the lock at lower levels. This is unlikely in the lock-coupling algorithms, since an elock at height $h-1$ would be due to a split and there are very few of those.

The only parameters that will vary from algorithm to algorithm are the service times.

Theorem Root: According to the constant flow model, the waiting period due to splitting elocks in the Bayer-Schkolnick and Mond-Raz algorithms is prob of split per insert × throughput of inserts × duration(h) × duration(h) × 1/2, where duration(h) is the service time of shared locks + the service time of elocks at the root. □

Note that in order to derive the above theorem, we did not have to use any of the structure approximations from section 5.1. The only assumption is one of constant flow.

6.2.2. Non-root analysis

This analysis applies to all elock requests in the Lehman-Yao algorithm and simple elocks in the other algorithms. (Recall that simple elocks are elocks obtained on the leaves by inserts on their first descent.) The main difference between the two kinds of elocks is that for simple elocks, we assume that the nodes are not full, whereas nodes that are split are full.

Because collisions are very rare at the leaf level it is rare that a lock will hit a node during the short time that it is exclusively locked. Therefore elocks lower in the tree cause delays much less often. We therefore parametrize the elocks by the height they hit. As usual, height 1 corresponds to the leaves and height h to the root.

Recall that eservice(i) is the time an elock is held at height i, where $1 \le i \le h$. At the root, duration(h) = eservice(i) + service time of a shared lock. Our reasoning was that there are far more shared locks and all processes must go through the root.

At lower levels, we make the assumption that splitting elocks never wait for other locks and that a waiting process never delays a third process. There are three justifications for these assumptions. First, since in any lock-coupling algorithm, an elock obtained in a second descent at a non-root node is preceded by an elock at the node's parent and by processing the parent, the elock request will probably be immediately satisfied. Second, there are many more nodes at the lower levels and relatively few splitting inserts, so such waiting has a minor effect on throughput. (In fact, earlier in our study, when we included an estimate of this waiting, the expected throughput was affected by less than 1%.) Third, making a simple assumption that introuces a small inaccuracy is better than proposing a complicated assumption that might introduce inaccuracies of the same order of magnitude.

We now want to calculate the probability of an elock at a given node per unit time. Let the arrival rate of elocks at height i be numelock(i). The expected arrival rate of elocks at a given node at height i is numelock(i) × p_hit(i). (If the node is full, it is numelock(i) × p_hit(i,s), which is approximately numelock(i) × p_hit(i) × 1/0.69.)

The likelihood that a given node at height i has an elock = arrival rate of elocks at a given node at height i × length of time that an elock is at a node at height i
= numelock(i) × p_hit(i) × duration(i). (Again, multiplied by 1/0.69 if the node is full.)

So, the expected time from the request to the beginning of service of a shared lock request (or a simple elock request) is

likelihood that a given node at height i has an elock $\times$ duration(i) $\times$ 1/2

= numelock(i) $\times$ p_hit(i) $\times$ duration(i) $\times$ duration(i) $\times$ 1/2 (Multiplied by 1/0.69 if the node is known to be full.)

Theorem Non-Root: The expected time a shared lock request or simple elock request will wait at height i (<h) is numelock(i) $\times$ p_hit(i) $\times$ duration(i) $\times$ duration(i) $\times$ 1/2. If the node it must wait for is known to be full, then the expected time is numelock(i) $\times$ p_hit(i) $\times$ 1/0.69 $\times$ duration(i) $\times$ duration(i) $\times$ 1/2. □

For simple elocks, numelock(1) = throughput of inserts. For splitting elocks in the Lehman-Yao algorithm, numelocks(i) depends on the height i and will be derived later. (However, it is again a linear function of the insert throughput.)

Note that in order to derive Theorem Non-Root, we had to use all the structure approximations of section 5.1.

6.2.3. Throughput in constant flow model

Let p be the number of processes. Let q be the average proportion that are inserts at any given time. (This may differ from the probability of inserts as we explain in the next subsection.) The throughput of inserts is

$$e_{insert} = \text{qp/(active time of inserts + waiting time of inserts)}$$

Recall that the waiting time is a linear function of numelock, which in turn is a linear function of the insert throughput. Also, recall that the two main contributors to waiting (root elocks and simple elocks) are assumed to be independent (see assumptions of constant flow model). Therefore, there is an algorithm-dependent constant k such that

waiting time = k e_{insert}. (We can obtain k from theorems root and non-root by adding the contributions of waiting times given by each theorem. k is a function of parameters such as p_hit(i) and duration(i) which depend on the algorithm.)

Here, duration is the service time of shared locks + the service time of exclusive locks at the root

If we let a_{insert} be the active time for insert, we derive the following formula for the insert throughput:

$$e_{insert} \sim \text{qp}/\ (a_{insert} + \text{k}\ e_{insert})$$

Treating this an equation, we obtain the throughput as a root of k

$$e_{insert}^2 + a_{insert}\ e_{insert} - \text{qp} = 0.$$

Since a_{insert} is positive, there is only one non-negative root. To predict the performance of a specific algorithm on a specific structure using a particular cost model, we need only determine k and a_{insert}.

A search waits as much as an insert (since the additional time those inserts which acquire elocks wait, adds only a negligible amount to the average waiting time per insert). So,

$$e_{search} \sim (1 - \text{qp})/(a_{search} + \text{k}\ e_{insert}).$$

6.3. Calculating the number of inserts

The throughput depends on the average proportion of processes that are performing inserts at any given time, as in the formulas of the previous section. These are not only a function of the probability of searches relative to the probability of inserts, but also the relative length of these two operations. For example, if an insert took infinitely longer, then eventually all processes would be executing inserts, provided inserts had a non-zero probability.

If the search time and insert time were constant (though perhaps different) as we assume, the expected value of the number of processes doing searches and inserts at any time would obey the following.

number of search processes + number of insert processes = number of concurrent processes altogether.

number of search processes α the number of concurrent processes altogether $\times$ prob of search $\times$ time of search.

number of insert processes α the number of concurrent processes altogether $\times$ prob of insert $\times$ time of insert.

Assuming the same constant of proportionality for the numbers of searches and inserts, we have

number of search processes = (prob of search $\times$ time for search $\times$ number of concurrent processes altogether) / (time for insert $\times$ prob of insert + time for search $\times$ prob of search).

number of insert processes = (prob of insert $\times$ time for insert $\times$ number of concurrent processes altogether)/ (time for insert $\times$ prob of insert + time for search $\times$ prob of search).

Of course, the times for insert and search are unknown to us. We compute those as we compute the throughput. So, we calculate the number of search and number of insert processes and their times iteratively. We start by estimating the number of insert and search processes based on their probability. Running our model gives us the times for insert and search. We enter these into the above equations, get new predictions of the number of insert and search processes and rerun the analytical model. The iteration stabilizes within 3 or 4 iterations. The result is approximately the following: when the probability of an insert is 0.7, the actual proportion of insert processes is 0.75; when the probability is 0.3, the actual proportion is 0.34.

6.4. Bursty flow predictions

The bursty flow model has far more straightforward predictions. A ''cycle'' in this model is the time needed for all processes to complete one action each. That is, search time = insert time = one cycle. The model applies only to the Bayer-Schkolnick and Mond-Raz algorithms. It assumes all waiting is due to splitting elocks at the root. The active time is just the time for non-splitting inserts to complete.

The cyclic behavior that develops is as follows. There is a constant queue of elocks at the root interrupted by a flow of processes. If the total number of processes is p and qp is the portion of processes actually executing inserts, each time these qp inserts reach the leaves they cause

$$q \times p \times \delta \text{ splits,}$$

where δ, the average number of splits at the leaves per insert, is $1/(0.69s)$.

Each split causes a delay of $eservice(h)$. So a complete cycle of p processes takes time to perform a non-splitting insert + $q \times p \times \delta \times eservice(h)$.

Theorem bursty: The throughput of an algorithm satisfying the bursty flow prediction is $p/(q \times p \times \delta \times eservice(h)$ + time to perform a non-splitting insert). □

7. Application to the Algorithms

To specialize the predictions to actual algorithms and cost models, we must calculate the waiting time for processes on their first descent and the active time. (Recall that for simplicity's sake we ignore the waiting time of splitting inserts on their second descent, because there are very few of them ~ number of inserts/(0.69s).) These depend (according to theorem root and non-root) on the following algorithm and model-dependent parameters:

the service time for a splitting elock,
the service time for a simple elock,
the service time for a shared lock at the root, and
the active time of each kind of action.

We also calculate the following structure- dependent parameters: probability of split and of hitting a node at a level (the latter also depends on how full the node is).

A few general statements can be made. The service time for a shared lock is 1 in the IO cost model and either 0.3 or 1.3 in the buffer cost model depending on whether the shared lock is at the upper two levels of the tree or not. (A process obtains a lock on a node, releases the lock on its parent, accesses the node to determine on which child to obtain a lock.) The service time for a simple elock is 2 in the IO cost model and 2.3 in the buffer cost model. (The time to read and write a node.) The active time to perform a search is (IO: h, Buffer: (2* 0.3) + (h - 2)(1.3)). The active time to perform an insert without splitting is (IO: h + 1, Buffer: (2* 0.3) + (h - 2)(1.3) + 1). These metrics are useful in the bursty flow model.

It remains to determine the service time at the root and the active time of inserts, except in the case of the Lehman and Yao algorithm, where more parameters are needed.

7.1. Optimistic Bayer and Schkolnick

1. *eservice*(h). The service time at the root assumes at least one split is necessary, since that is why the root is elocked. The strategy that Bayer-Schkolnick uses when performing a second descent is the following: elock the root; read the root; determine where appropriate child is; acquire elock on child; read child; process child: if child is safe, then release elock on root, else continue descent.

We give the result for the IO cost model case first, then explain the terms. (Recall that in the IO cost model, all nodes must be reread in the second descent, but processing time is free.) By the structure assumptions 5.1, the proportion of full nodes at any level (except the root) is assumed to be (1/0.69s).

For the IO cost model, eservice(h) is

$$\sum_{i=1}^{h-2} (i+1)(1-(1/(0.69s)))((1/(0.69s))^{i-1}) + (h+1)((1/(0.69s))^{h-2})$$

The ith summand of the first term gives the time the root is locked if there are i − 1 consecutive unsafe nodes on the way to the split node, starting from a child of the root. The common case when the child of the root is safe corresponds to $i = 1$. The second term represents the time the root is held assuming there are splits from the root to the leaves starting with the child of the root, given that the leaf is split. This is the only case in which the root is changed. When the root is changed, it is the first node written and the elock on it is released immediately. That is why the time held is $h + 1$: h to descend the structure a second time and 1 to write the root.

For the buffer cost model, the time the root is held, *eservice*(h), becomes

$$\sum_{i=1}^{h-2} ((i+1)0.3)(1-(1/(0.69s)))((1/(0.69s))^{i-1}) + ((0.9(h-1)) + 1.3)((1/(0.69s))^{h-2})$$

2. Active time of insert. The time to do the actual work of an insert is:

(IO): $h + 1 + p\_split(1) \times h + \sum_{i=1}^{h-1} 2i \times p\_highest(i)$,

(Buffer): $(h-2) + h \times 0.3 + 1 + p\_split(1) \times h \times 0.3 + \sum_{i=1}^{h-1} p\_highest(i) \times (2.6i)$.

For the IO model, the term h is the time to descend the structure, 1 is the time to write the leaf node whether there is a split or not. $p\_split(1) \times h$ reflects the time to descend the structure the second time if there is a split. The summation reflects the extra writes necessary as the splits propagate up the tree.

7.2. Optimistic Mond and Raz

1. *eservice*(h). The procedure in the Mond-Raz algorithm is: elock the root; read the root; determine where appropriate child is; acquire elock on child; read child; process child: if child is safe, then release elock on root, else split child and release the root. As in the Bayer-Schkolnick algorithm, the root is written first whenever a split occurs. The expression in the IO cost model is: $2 + 1 \times$ probability of a node at level $h - 1$ to be full, given that there is a split at the leaves

Buffer cost model: $(2\times 0.3)+2.2\times$ probability of a node at level $h-1$ to be full, given that there is a split at the leaves.

The first term is the time the elock holds the root if the child it accesses is safe. The second term is the extra time to split a child if the child is full.

A caveat is in order here. In the optimistic Mond and Raz algorithms the expected number of elements per node is only approximated by our structure assumptions and might be less than $0.69s$. The reason is that nodes are sometimes split when they are full, but before anything is inserted into them. Since no node splits when it has only $s-1$ nodes, the number of elements must be greater than $0.69(s-1)$. So, the inaccuracy of our structure assumption is small.

IO cost model is : $2+(1/0.69s)^{h-2}$.
Buffer cost model: $(2\times 0.3)+2.2\times(1/0.69s)^{h-2}$.

One advantage of the Mond and Raz algorithm is that as soon as an insert in its second descent encounters a full node it splits it. The delay for this insert is greater than in the Bayer and Schkolnick case, but only one insert is so delayed, making full nodes at high levels very improbable. However, for the common case of a height 3 tree, the two algorithms will split in the same instances, though the Bayer-Schkolnick algorithm will descend to the leaves before splitting the level 2 node.

2. Active time of insert. Like the Bayer and Schkolnick algorithm, the optimistic Mond and Raz algorithm redescends the tree starting at the root when it encounters a split at the leaves. Whereas it may split certain nodes earlier, simulation shows that the difference in the number of nodes as a function of the number of items in the tree is small. Therefore, the active times of the two algorithms are the same. We repeat it for convenience.

   (IO: $h+1+p\_split(1)\times h+\sum_{i=1}^{h-1} 2i\times p\_highest(i)$,

   Buffer: $(h-2)+h\times 0.3+1+p\_split(1)\times h\times 0.3+\sum_{i=1}^{h-1} p\_highest(i)\times(2.6i)$).

7.3. Lehman and Yao algorithm

In the Lehman and Yao algorithm, a splitter will rarely elock the root. Instead, it first performs a half_split at the leaf level, then does a complete_split at the level above, continuing with a half_split at that level only if necessary. The root will not become a bottleneck and the constant flow model is the only one that makes sense.

1. $numelock(i)$ and $eservice(i)$ due to splits. (For simple elocks, numelock(1) = e_{insert} and eservice(1) is as in the general remarks. What we compute here adds values to numelocks(i).) Half_splits and complete_splits contribute additively to these terms. Due to half_splits, numelocks(i) = $e_{insert}\times p\_split(i)$ and eservice(i) = half_split − 1. This is 1 for the IO cost model and (2 × 1.3) − 1 for the buffer cost model. We subtract 1 because the half_split includes the time to write the node that would be written by a simple elock or a previous complete_split (in the case of a split of a

higher level node).

Due to complete_splits, numelocks(i+1) = $e_{insert} \times p\_split(i)$ and duration(i) = eservice(i) = complete_split. This is 2 in the IO cost model, because we must reread the parent node and 1.3 in the buffer cost model.

Applying theorem Non-Root, we get the following formula. The waiting time due to splitting elocks is

$$\sum_{j=1}^{j=h-1} e_{insert} \times 1/2 \times (1/0.69) \times p\_hit(j) \times p\_split(j) \times (half\_split - 1)^2$$

$+ \sum_{j=1}^{j=h-1} e_{insert} \times 1/2 \times p\_hit(j+1) \times p\_split(j) \times (complete\_split)^2$, where p_split(j) is $(1/(0.69s))^j$.

Note that the split node is full, but its parent is not. That is why 1/0.69 is only in the first term.

2. Active time. The active time for an insertion is $h+1+\sum_{i=1}^{h-1} p\_highest(i) \times 3i$ for the IO cost model.

For the buffer cost model, this reduces to $(h-2)+h \times 0.3+1+\sum_{i=1}^{h-1} p\_highest(i) \times 2.9i$.

8. Validation

We wrote the three algorithms described above (whose code is in appendix A) in Concurrent Euclid [Holt83] and ran them using the simulation parameters specified in the previous section. Ideally, to determine throughput, we should let the computation reach its average throughput and then observe the throughput for a given period of time (1000 steps). That is the measurement approach we took. The standard deviation of the throughput of different runs (brought about by different random seeds) was about 1%.

We examined the algorithms using three level trees with node sizes 13 and 39. We ran the tests using 1, 10, 20, 30, and 40 processes.

The average throughput as a function of the number of processes for the optimistic lock-coupling algorithms is graphed in figures 6.1 to 6.12 for the IO cost model and 3 level 20-39 tree, 7.1 to 7.8 for the IO cost model and 4 level 7-13 tree, 8.1 to 8.8 for the buffer cost model and 3 level 20-39 tree, 9.1 to 9.4 for the buffer cost model and 4 level 7-13 tree. For these algorithms the simulation data are circles, the dashed curve is the oscillatory prediction and the solid curve is the analytical prediction. For the Lehman and Yao algorithm, the simulation results are circles and the analytical result is the solid curve.

The constant flow prediction gives a close fit for the Lehman and Yao and Quick split (described below) algorithms. For the Bayer-Schkolnick and Mond-Raz algorithms, the values lie between the two curves (except in two graphs), but tend towards the bursty flow prediction as the number of processes increases.

Since the bursty flow prediction is very pessimistic, the exceptional graphs (for the buffer model Bayer-Schkolnick algorithm) are surprising. One reason is that the bursty flow model assumes that all waiting takes place at the root. This idealization is an excellent approximation for the IO cost model, where simulation shows that only 8% of the waiting time (0.2 units out of 2.58) takes place at non-root nodes (Bayer-Schkolnick, 20-39 tree, 40 processes, 100% inserts). However, with a buffer for the nodes of the first two levels, 23% of the waiting takes place at non-root nodes (0.24 units out of 1.03), so the approximation is not as good. These waiting times correspond to 3% and 7% of the total average job time, respectively.

9. Improving the Standard Algorithms

From these results it is easy to conclude that an algorithm should avoid exclusively locking the root whenever possible and hold an exclusive lock on the root for as short a time as possible. This suggests a simple modification of lock-coupling optimistic algorithms that will have them lock the root much less frequently.

Modify the optimistic algorithm so that an insert(x) in its first descent remembers the address of the height 2 node that it visited. Instead of exclusively locking the root upon finding a full leaf node, the insert(x) should release all its locks, then exclusively lock the height 2 node. If the height 2 node is still not full, then the insert(x) exclusively locks the appropriate leaf (the one that should contain x) and performs the split if necessary. If the height 2 node had been full in the first descent or is full or if the old height 2 node is no longer the parent of a leaf that should contain x, then the inserter would do a conservative descent from the root (as is done now whenever there is a split).

Because this technique makes exclusive locking of the root rare, it should should help the performance of lock-coupling algorithms substantially. The algorithm for delete is analogous.

This algorithm style, which we call *Optimistic Lock-Coupling with Quick Split* is original in its details, but bears a close resemblance to [23], the intention lock variant of [3], and the give-up style algorithms in [36]. The version we analyzed used the Mond-Raz technique for the final conservative descent, when necessary.

This algorithm is much easier to program than the Lehman-Yao algorithm. It gives approximately the same performance. (At 40 insert processes, Lehman-Yao is at most 10% better in the simulations we have run; for fewer processes, the Lehman-Yao advantage decreases.). See figures 6 through 9.

9.1. Optimistic Lock-Coupling with Quick Split

We give the analytical expressions for this algorithm. Because most of its locking occurs at the leaves, the constant flow model should be quite accurate. In fact, we will use the non-root analysis (except that we will add sservice(h) to eservice(h)).

1. *eservice*(h). Since an insert only exclusively locks the root when the parent of the leaves is full, the likelihood that such an elock will result in a split all the way to the leaves is higher than in the Mond-Raz algorithm. Here are the expressions. IO cost model: $2+1\times(1/(0.69s))^{h-3}$. Buffer cost model: $(2\times0.3)+1.6\times(1/(0.69s))^{h-3}$.

The first term is the time the elock holds the root if the child it accesses is sparse. The second term is the extra time to split a child if the child is full.

The total service time is higher than in the Mond-Raz algorithm, however the likelihood of splitting the root is related to p_split(2) instead of p_split(1) as before. When calculating $duration(h)$, we include the service time of a shared lock.

2. The service time at height 2 in the IO cost model is 5 and in the buffer model it is 4.2. The break-down for the IO cost model is 1 to reread the parent, 1 to reread the child, and 3 to perform all the writes. The break-down for the buffer cost model is 0.3 to add the key, 0.6 to move the data, 2 to write the leaves, and 1.3 to change and write the parent.

   The reader may recall that the service time at the root for the original Mond-Raz algorithm was close to 2 in the IO model, because splits were very unlikely at the top two levels (so the 2 just represented the time to read the top two nodes and then release the elock on the root). The time of 5 here is due to the fact that a split is considered to be certain.

3. Active time of insert. IO cost model: $(h+1) + p\_highest(1) \times 4 \quad + p\_split(2) \times h + \sum_{i=2}^{h-1} 2i \times p\_highest(i)$. Buffer cost model: $((h-2) + h \times 0.3 + 1) \quad + p\_highest(1) \times 2.2 + p\_split(2) \times h \times 0.3 + \sum_{i=2}^{h-1} p\_highest(i) \times (2.9i)$.

The first term is the initial descent and non-splitting insertion time. The term with p_highest is the quick split time. The third term corresponds to the descent due to a split at height 2 or higher. The fourth term has to do with the actual splitting work.

4. $numelock(2)$ = $e_{insert} \times p\_highest(1)$. (Simple elocks are handled by the general case.) $numelock(h)$ = $e_{insert} \times p\_split(2)$.

   Applying theorem Non-Root, we get the following formula. The waiting time due to splitting elocks is

$$e_{insert} \times 1/2 \times p\_hit(2) \times p\_highest(1) \times duration^2(2) + e_{insert} \times 1/2 \times p\_split(2) \times duration^2(h).$$

10. Conclusion

We have proposed an analytical characterization of four algorithms and validated it through simulation. For the optimistic lock-coupling algorithms that lock the root upon every split, the analytical model consists of two sets of formulas. One set assumes constant throughput, whereas the other postulates bursty behavior. The data points lie between these curves and tend towards the bursty curve as the number of processes increase. Considering the intricacies and strange states these algorithms can reach, the accuracy of our predictions is gratifying.

It is also relatively straightforward to derive the equations of new algorithms, since the form of the calculation for the expected time a process waits at a given height remains the same and is given by theorems root, non-root, and bursty. In fact, the model suggested a simple yet better algorithm, which we implemented an analyzed (Optimistic Lock-Coupling with Quick Split).

In all cases, the optimistic Bayer-Schkolnick algorithm was slightly slower (about 5% at 40 insert processes) than the Mond-Raz algorithm. The Mond-Raz in turn was at least 30% slower than the Optimistic Lock-coupling with Quick Split, which was within 10% of the Lehman-Yao algorithm, all at 40 insert processes. As the portion of insert processes decreases, the differences among the algorithms diminishes.

We have shown how to extend the model to handle a buffer. The model correctly predicted the improvement due to a buffer of about 30% for a three-level 20-39 tree.

The model is limited to transactions consisting of single dictionary actions. Nevertheless it applies to a two-level transaction model in which data items are locked outside the structure and the dictionary actions constitute sub-transactions. Locks on nodes that are split and merged should be released before the end of the transaction, since there is no need to undo them for transaction consistency reasons [37].

Another limitation is that the model ignores resource contention. One may view resource contention as simply an added delay factor [38]. For example, instead of saying that reading a node costs one time unit, make the time be a positive function of the number of active transactions. This a crude approximation for search structures, that requires expermental validation.

There are a number of hard open problems. In the presence of deletes very little is known about B-trees, in particular we don't know what the average load factor in this case is and hence can not estimate the probability of a split.

We also don't know how to predict how many processes are needed to make the bursty flow model formulas a better approximation to the throughput than the constant flow formulas for a given algorithm. We think it is a function of the size of the nodes in the B-tree.

References

[1] ACL85 R. Agrawal, M. Carey, and M. Livny, ‘‘Models for studying concurrency control performance: alternatives and implications’’, Proceedings of the ACM SIGMOD International Conference on Management of Data, Austin, Texas, May 1985.

[2] BG81 P. A. Bernstein and N. Goodman, ‘‘Concurrency Control in Distributed Database Systems,’’ Computing Surveys 13, 2 (June 1981), pp. 185-221.

[3] BS77 R. Bayer and M. Schkolnick, ‘‘Concurrency of operations on B-trees,’’ Acta Informatica, vol. 9, pp. 1-21, 1977.

[4] CC86 M. Carey and C-Y Cheng, ‘‘B+ Tree Locking: a performance perspective’’ manuscript, 1986.

[5] CS84 M. Carey and M. Stonebraker, ‘‘The Performance of Concurrency Control Algorithms for Database Management Systems,’’ Proceedings of the Tenth International Conference on Very Large Data Bases, Singapore, August 1984.

[6] Comer79 R. Comer, ‘‘The ubiquitous B-tree,’’ ACM Computing Surveys, vol. 11, pp. 121-138, 1979.

[7] EGLT76 Eswaran, K. P., Gray, J.N., Lorie, R. A. and Traiger, I.L. "The Notions of Consistency and Predicate Locks in a Database System," CACM Vol. 19, No. 11, November, 1976.

[8] Ellis80 C. S. Ellis, ‘‘Concurrent Search and Inserts in 2-3 trees,’’ Acta Informatica, vol. 14, no. 1, pp. 63-86, 1980.

[9] Ellis82 C. S. Ellis, ‘‘Extendible Hashing for Concurrent Operations and Distributed Data,’’ ACM-SIGMOD, pp. 106-115, 1982.

[10] Ellis85 C. S. Ellis, ‘‘Concurrency control and linear hashing,’’ Fourth annual ACM SIGACT/SIGMOD symposium on Principles of Database Systems, Portland, Oregon, pp. 1- 7, 1985.

[11] FC84 R. Ford and J. Calhoun, ‘‘Concurrency Control Mechanisms and the Serializability of Concurrent Tree Algorithms’’, Proceedings of the ACM Symposium on the Principles of Database Systems, 1984.

[12] FJS85 R. Ford, M. Jipping and R. Shultz, ‘‘On the Performance of an Optimistic Concurrent Tree Algorithm,’’ Technical Report 85-07, Dept. of Computer Science, University of Iowa, 1985.

[13] Garc83 H. Garcia-Molina, ‘‘Using Semantic Knowledge for Transaction Processing in a Distributed Database,’’ ACM Transactions on Database Systems, vol. 8, no. 2, June, 1983.

[14] GoSh85 N. Goodman and D. Shasha, ‘‘Semantically-based concurrency control for search structures’’ Fourth annual ACM SIGACT/SIGMOD symposium on Principles of Database Systems, Portland, Oregon, pp. 8 - 19, 1985. To appear in ACM Transactions on Database

Systems.

[15] GuiSed78 L. J. Guibas and R. Sedgewick, ‘‘A Dichromatic Framework for Balanced Trees,’’ Proc. 19th Annual Symposium of Foundations of Computer Science, 1978, pp. 8-21.

[16] Holt83 R. C. Holt, *Concurrent Euclid, The Unix System, and Tunis* Addison-Wesley, Reading, Mass., 1983.

[17] HsuYan M. Hsu and W. P. Yang, ‘‘Concurrent Operations in Extendible Hashing,’’ Proceedings of the Twelfth Very Large Data Bases Conference, Kyoto, Japan, August 25-28, 1986 pp. 241-247.

[18] KL80 H. T. Kung and P. L. Lehman, ‘‘Concurrent Manipulation of Binary Search Trees,’’ ACM Trans. Database Syst., vol. 5, no. 3, pp. 339 -353, 1980.

[19] Korth83 H. F. Korth, ‘‘Locking Primitives in a Database System,’’ Journal of the ACM, volume 30, no. 1, January 1983, pp. 55-79.

[20] KP83 H. T. Kung and C. H. Papadimitriou, ‘‘An Optimality Theory of Concurrency Control for Databases,’’ Acta Informatica, vol. 19, no. 1 (1983), pp. 1-11.

[21] KR81 H. T. Kung and J. T. Robinson, ‘‘On optimistic concurrency control,’’ ACM Transactions on Database Systems 6: 2, 1981, pp. 213-226

[22] KS83 Z. Kedem and A. Silberschatz, ‘‘Locking Protocols: From Exclusive to Shared Locks,’’ Journal of the ACM, Vol. 30, No. 4, Oct. 1983, pp. 787-804.

[23] KW82 Y. S. Kwong and D. Wood, ‘‘Method for concurrency in B-trees,’’ IEEE Transactions on Software Engineering, vol. SE-8, no. 3, pp. 211-223, 1982.

[24] LS86 V. Lanin and D. Shasha, ‘‘A symmetric concurrent B tree algorithm’’ New York University, Technical Report number 210, April, 1986.

[25] LY81 P. L. Lehman and S. B. Yao, ‘‘Efficient locking for concurrent operations on B-trees,’’ ACM Trans. on Database Systems, vol. 6, no. 4, December 1981, pp. 650-670.

[26] Lomet83 D. B. Lomet, ‘‘Bounded Index Exponential Hashing,’’ ACM Transactions on Database Systems, vol. 8, no. 1, March, 1983.

[27] ML82 U. Manber and R. E. Ladner, ‘‘Concurrency control in a dynamic search structure,’’ ACM Proceedings on Database Systems, pp. 268-282, 1982.

[28] MR85 Y. Mond and Y. Raz, ‘‘Concurrency control in B+ trees using Preparatory Operations’’ 11th International Conference on Very Large Data Bases, August 1985, pp 331 - 334.

[29] MS78 R. E. Miller and L. Snyder, ‘‘Multiple access to B-trees,’’ (preliminary version), Proc. of the 1978 Conf. on Information Sciences and Systems, J. Hopkins Univ., Baltimore, Maryland (1978).

[30] Papa79 C.H. Papadimitriou, ‘‘Serializability of Concurrent Database Updates,’’ JACM, vo. 26, no. 4, pp. 631-653. October, 1979.

[31] Sa85 Yehoshua Sagiv, ‘‘Concurrent operations on B*-trees with overtaking,’’ Fourth annual ACM SIGACT/SIGMOD symposium on Principles of Database Systems, Portland, Oregon, pp. 28 - 37, 1985.

[32] Sa86 Yehoshua Sagiv, ‘‘Concurrent operations on B*-trees with overtaking,’’ Journal of Computer Science and Systems, 1986.

[33] Salz85b B. Salzberg, ‘‘Restructuring the Lehman-Yao tree,’’ College of Computer Science, Northeastern University, 1985.

[34] Salz85a B. Salzberg, ‘‘Grid File Concurrency,’’ Technical Report No. BS-85-27, College of Computer Science, Northeastern University, June 1985.

[35] Sama76 B. Samadi, ‘‘B-trees in a System with Multiple Users’’, Information Processing Letters, Vol. 5, pp. 107-112, October 1976.

[36] S84 D. Shasha, ‘‘Concurrent Algorithms for Search Structures,’’ Ph.D. Thesis, Harvard University, TR-12-84, June, 1984.

[37] S85 D. Shasha, ‘‘What good are concurrent search structure algorithms for databases anyway?,’’ Database Engineering, June 1985, vol. 8, no. 2, pp. 84-90.

[38] T84 Y. C. Tay, ‘‘A mean value performance model for locking in databases,’’ Ph.D. dissertation, Harvard Univ., Cambridge, Mass. Feb. 1984.

[39] TGS85 Y.C. Tay, N. Goodman, and R. Suri, ‘‘Locking Performance in Centralized Databases,’’ ACM Transactions of Database Systems, December 1985, vol. 10, no. 4, pp. 415-462.

[40] Y78 A. Yao, ‘‘On random 2-3 trees,’’ Acta Informatica, 9 (1978), pp. 159-170.

[41] W86 G. Weikum, ‘‘A theoretical foundation of multi-level concurrency control’’ Fifth annual ACM SIGACT/SIGMOD symposium on Principles of Database Systems, Cambridge, Ma., pp. 31-42, 1986.

Appendix A - algorithms tested

Here is the basic lock-coupling member action. Of the updaters, we have only written down the insert programs. The deletes are symmetric.

```
procedure member(x)
begin
place shared lock on the root
get root and make it current
while current node is not a leaf do
begin
  place shared lock on appropriate child of current node
  release shared lock on current node
  make that child the current node
end
return true iff x is in current node
end
```

Conservative, bottom-up insert. In this algorithm, each updater holds an exclusive lock on all nodes, but releases its lock on a node *n* upon discovering that a descendant of *n* is safe. If splits are necessary, they are performed from the leaves up.

```
procedure insert(x)
begin
place exclusive lock on the root
get root and make it current
while current node is not a leaf do
begin
  place exclusive lock on appropriate child of current node
  make that child the current node
  release locks on all ancestors of current node if
            current node is safe
end
if x is not in current node then
  add x to current node
  while current node has 2k elements do
    split current node in memory
    make parent of current node the current node
  end
  write the nodes in order from the node at the highest
            level to the node at the lowest level
end
```

Conservative, top-down insert. In this algorithm, each updater holds exclusive locks on two nodes, a parent and a child. If the child is unsafe, the parent splits it (in the insert case) or merges it (in the delete case).

```
procedure insert(x)
begin
place exclusive lock on the root
get root and make it current
while current node is not a leaf do
begin
  place exclusive lock on appropriate child of current node
  if child is unsafe
   then split child
            write parent node first
            make new child be appropriate node of split pair
            discard lock on other node of the pair
  end if
  make the child the current node
  release lock on parent of child
end
if x is not in current node then
  add x to current node
end
```

In the optimistic versions of the top-down and bottom-up lock-coupling protocols, updaters use the protocol of member actions except

1. at the leaves they acquire exclusive locks instead of shared locks; and
2. If they discover that they need to perform splits or merges, they start over again with the conservative method. This means they may have to read certain nodes a second time in their second descent. The buffer is however assumed to be big enough to hold the nodes that they read the second time.

Next, we look at the protocol in the style of the Lehman and Yao [LY81] algorithm. In this algorithm, an updater only obtains exclusive locks on nodes it actually must split. Links to right siblings enable an update or member to find the leaf node that it must go to.

```
procedure member(x)
begin
place shared lock on the root
get root and make it current
while current node is not a leaf do
begin
 if there is an appropriate child of current node then
  release shared lock on current node
  place shared lock on appropriate child of current node
  make that child the current node
 else
  release shared lock on current node
  place shared lock on right sibling of current node
  make that sibling the current node
 end if
```

```
end
{member is at the leaf level.}
if x does not belong in current node then
  release shared lock on current node
  place shared lock on right sibling of current node
  make that sibling the current node
end if
if x is in current node then
  return true
 else return false
end if
end


procedure insert(x)
begin
place shared lock on the root
get root and make it current
while current node is not a leaf do
begin
 if there is an appropriate child of current node then
  release shared lock on current node
  if children of current node are leaves then
   place exclusive lock on appropriate child of current node
   else place shared lock on appropriate child of current node
  end if
  make that child the current node
 else
  release shared lock on current node
  place shared lock on right sibling of current node
  make that sibling the current node
 end if
end
{insert is at the leaf level.}
if x does not belong in current node then
  release exclusive lock on current node
  place exclusive lock on right sibling of current node
  make that sibling the current node
end if
if x is not in current node then
  add x to current node
  if that causes current node to contain 2k elements
            then half_split the current node
               yielding the current node and a new node
                complete_split(current: node, new: node)
end
```

The operation ‘‘complete_split’’ referred to in the last line is a separate process that tries to perform the complete_split by putting a pointer and a value in the level above the leaves. This may entail traversing several links. However, that is an extremely rare occurrence (since each link corresponds to a split made at height two after the insert was at height two during its descent and before it did its half_split), so we ignore the intricacies of this procedure. The complete_split may entail a new half_split at height 2, generating a new complete_split at height 3 and so on.

```
complete_split(n,new: node)
begin
 n’ := last node visited at level above n
 exclusive lock n’
 while n’ is not the parent of n do
  n’’ := n’.rightlink
  release lock on n’
  obtain exclusive lock on n’’
  n’ := n’’
 end while
 add pointer to new in n’
 if this causes n’ to have 2k pointer-value pairs
  then
    half_split n’ yielding n’ and new’
    complete_split(n’,new’)
 end if
end
```

These protocols never deadlock. The reason is that locks are always obtained from top to bottom or left to right order. Therefore every lock that an action requests, it eventually obtains.

The last algorithm is the optimistic lock-coupling with quick split. The member is the standard lock-coupling version. The insert performs an optimistic descent. If it has to split a leaf node, it first tries to split the immediate parent of that node. Otherwise, it redescends in a conservative fashion.

```
procedure insert(v) =
begin
  place shared lock on root
  get root and make it current
  while current is not parent of leaf
    place shared lock on appropriate child of current node
    make child the current node
    release lock on parent node
  end
  place exclusive lock on appropriate leaf
  note if parent of leaf is safe
  release lock on parent of leaf
```

```
  if leaf is safe then
    if v in leaf then
      add v to leaf
    end if
    relase lock on leaf
  else
    relase lock on leaf
    if parent of leaf was safe then
      {try quick split}
      place exclusive lock on old parent of leaf
      if old parent is unsafe or not the parent of the appropriate leaf then
        release lock on old parent {quick split did not work}
      else
        place exclusive lock on appropriate leaf
        if leaf is safe then
          release lock on parent
             else
          split leaf
          decide which leaf of pair is now appropriate
          write parent node
          release lock on parent
          write inappropriate leaf
          release lock on inappropriate leaf
        end if
        if v is not in leaf then
               add v to leaf
             end if
        write leaf
        release lock on leaf
      end if
    end if
    if quick split did not work then
      perform a conservative Mond and Raz insert
    end if
  end if
end insert
```