Performance of B-Tree Concurrency Control Algorithms*

V. Srinivasan *Michael J. Carey*
Department of Computer Sciences
University of Wisconsin
Madison, WI 53706
srinivas@cs.wisc.edu

Abstract

A number of algorithms have been proposed for accessing B-trees concurrently, but the performance of these algorithms is not yet well understood. In this paper, we study the performance of various concurrency control algorithms using a detailed simulation model of B-tree operations in a centralized DBMS. Our study considers a wide range of data contention situations and resource conditions. Results from our experiments indicate that algorithms in which updaters lock-couple using exclusive locks perform poorly as compared to those that permit more optimistic index descents. In particular, the B-link algorithms provide the most concurrency and the best overall performance.

1 Introduction

Database systems frequently use indices to access data. These systems typically operate at a high level of concurrency, and since any transaction has a high probability of accessing an index, it is necessary to ensure that concurrent access to an index is not a bottleneck in the system. Since B-trees[1] are the most common dynamic index structures in database systems, most earlier work has concentrated on them and we will focus here on B-tree concurrency control algorithms. However, many of our results will lend insight into concurrency control for other index structures also.

Concurrency control techniques that work well for records or data pages, such as two-phase locking [Gray79], are overly restrictive when naively applied to such items as index pages. Special techniques must be employed to prevent indices and system catalogs from becoming concurrency bottlenecks. A number of algorithms have been proposed for accessing B-trees concurrently [Sama76, Baye77, Mill78, Lehm81, Kwon82, Good85, Mond85, Sagi85, Shas85, Lani86, Bili87, Moha89, Weih90], but few performance analyses exist that compare these algorithms. The earlier studies [Baye77, Bili85, Shas85, Lani86, John90] each compare only a few algorithms and have been based on simplified assumptions about resource contention and buffer management. Thus, the relative performance of these algorithms in more realistic situations is still an open question.

In this paper, we analyze the performance of various B-tree concurrency control algorithms using a simulation model of B-tree operations in a centralized DBMS. Our study differs from earlier ones in several aspects:

1. We study a representative list of algorithms including variations of the Bayer-Schkolnick, top-down, and B-link algorithms as well as a new algorithm that allows deadlock detection at a single node. Based on our analysis of these algorithms, we make further projections about the performance of other algorithms.
2. We use a closed queuing simulation model that is quite detailed and consists of a B-tree in a centralized database with a buffer manager, lock manager, CPUs and disks.
3. In our experiments, we consider tree structures with high and low fanout in a wide range of resource conditions and workloads.
4. We measure a wide variety of performance measures that help us to make precise statements about the performance of searches, deletes and inserts in the different algorithms.

Section 2 briefly reviews the set of B-tree concurrency control algorithms that have been proposed in the literature, focusing on the ones that were chosen for our study. The simulation model and performance metrics that we use in our study are described in Section 3. Section 4 presents details of our experiments and results.

*This research was partially supported by the National Science Foundation under grant IRI-8657323 and by a University of Wisconsin Vilas Fellowship.

[1] By B-tree we mean the variant in which all keys are stored at the leaves, also called B^+-trees and sometimes B^*-trees [Come79].

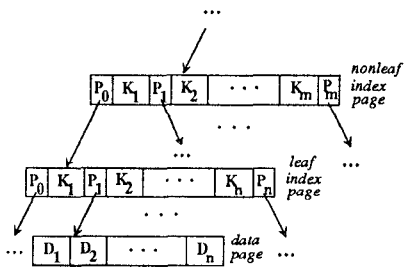

Figure 1: An example B-tree fragment

Section 5 compares this study with related work in the field. Finally, in Section 6, we summarize our key results.

2 B-trees in a DBMS

We assume that the reader is familiar with the basic B-tree index structure [Come79]. An example B-tree fragment is illustrated in Figure 1, and a B-tree page split is illustrated in Figures 2a and 2b. B-trees in real database systems usually perform page merges only when pages becomes empty; nodes are not required to be at least half-full, since in practical workloads, this is not found to decrease occupancy by much [John89]. We assume this approach to B-tree merges for this study. A node is considered *safe* for an insert if it is not full and safe for a delete if it has more than one entry. A split or merge of a leaf node propagates up the tree to the lowest safe node in the path from the root to this leaf. If all nodes from the root to the leaf are unsafe, the tree increases or decreases in height. The set of pages that are modified in an insert or delete operation is called the *scope* of the update.

mode	S	IX	SIX	X
S	√	√	√	
IX	√	√		
SIX	√			
X				

Table 1: Lock compatibility table.

2.1 B-tree Concurrency Control

Most B-tree concurrency control algorithms make use of the following relationship between a safe node and the scope of an update. When an updater is at a safe node in the tree, the only pages that can end up in the scope of this update are nodes in the path from this node to the leaf. Any locks held on nodes at higher levels can be released. Several algorithms use a technique called *lock-coupling* in their descent from the root to the leaf,

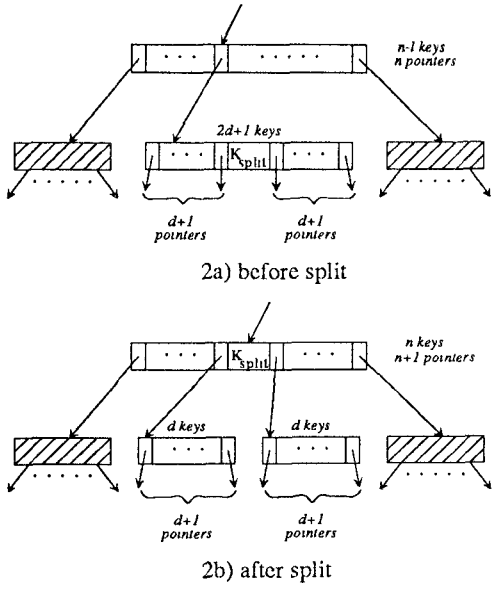

Figure 2: A B-tree page split.

releasing locks early using the above property. An operation is said to lock-couple when it requests a lock on an index page while already holding a lock on the page's parent, releasing the parent lock if the new page is found to be safe.

In a simple algorithm proposed in [Sama76], all operations get an X lock[2] on the root and then lock-couple their way to the leaf using X locks, releasing locks at higher levels whenever a safe node is encountered. A class of algorithms that improves on the above idea was proposed by Bayer and Schkolnick [Baye77].

2.1.1 Bayer-Schkolnick Algorithms

In all Bayer-Schkolnick algorithms, searches always follow the following locking protocol. A search gets an S lock on the root and lock-couples to the leaf using S locks. The various algorithms differ in the locking strategy used by updaters. We shall describe three representative algorithms, B-X, B-SIX, and B-OPT.

In the first algorithm, called B-X, updaters get an X lock on the root and then lock-couple to the leaf using X locks. The X locks of updaters on the path from the root to the leaf may temporarily shut off readers from areas of the tree not in the actual scope of an update. The above problem can be rectified if updaters lock-couple using SIX locks in their descent to the leaf. This algorithm, called B-SIX, allows readers to proceed faster (since SIX locks are compatible with S locks) but updaters, on reaching the leaf, have to convert the SIX locks in their scope to X locks. This top-down conversion drives away any readers in the updater's scope.

[2]The lock modes discussed in this paper and their compatibility relationships are given in Table 1.

In both algorithms above, updaters that do not conflict in their scope may still interfere with each other at higher level nodes. Moreover, in most B-trees, especially ones with large page capacities, page splits are rare. The third algorithm, which we call B-OPT, makes use of this fact, letting updaters make an optimistic descent using IX locks. They take an IX lock on the root and then lock-couple their way to the leaf with IX locks, taking an X lock at the leaf. Here, regardless of safety, the lock at each level of the tree is released as soon as the appropriate child has been locked. If updaters find the leaf to be safe, the operation succeeds. Otherwise, the updater releases its X lock on the unsafe leaf and makes a pessimistic descent using SIX locks, as in the B-SIX algorithm. If very few updaters make a second pass, this algorithm is expected to perform well.

Updaters in the Bayer-Schkolnick algorithms essentially update the entire scope at one time, making it necessary for them to hold several X locks at the same time. Several algorithms have been proposed that instead split the updating of the scope into several smaller, atomic operations. We consider two of these next, the top-down and B-link algorithms.

2.1.2 Top-down Algorithms

In top-down algorithms [Guib78, Care84, Mond85, Lani86], updaters perform what are known as preparatory splits and merges. If an inserter encounters a full node during its descent, it performs a preparatory page split and inserts an appropriate index entry in the parent of the newly split node. Similarly, a deleter merges a node that contains one entry with its sibling, deleting the appropriate entry from the parent. Leaf level insertion or deletion is similar except that the preparatory operations ensure that the parent will always be safe. As always, a merge or a split of the root page leads to an increase or decrease in tree height.

Based on the preparatory operations described above, we consider three top-down algorithms that correspond to the Bayer-Schkolnick algorithms in terms of the type of locking that updaters do. In the first algorithm, TD-X, updaters get an X lock on the root and then lock-couple using X locks to the leaf. At every level, before releasing the lock on the parent, an appropriate merge or split is made. The above algorithm can be improved by using SIX locks and converting them to X locks only if a split or merge is actually necessary. This variation is called TD-SIX. In the optimistic top-down algorithm, TD-OPT, updaters make an optimistic first pass, lock-coupling from the root to the leaf using S locks and then getting an X lock on the leaf. If the leaf is unsafe, the updater releases all locks and then restarts the operation, making a second descent à la TD-SIX [Lani86]. Readers use the same locking strategy as in the Bayer-Schkolnick algorithms.

The top-down algorithms break down the updating of a scope into sub-operations that involve nodes at two adjacent levels of the tree. The B-link algorithms go one step further and limit each sub-operation to nodes at a single level. They also differ from the top-down algorithms in that they do their updates in a bottom-up manner.

2.1.3 B-link Tree Algorithms

A B-link tree [Lehm81, Sagi85, Lani86] is a modification of the B-tree that uses links to chain all nodes at each level together. A page in a B-link tree contains a high key (the highest key of the subtree rooted at this page) and a link to the right sibling. The link enables a page split to occur in two phases: a half-split, followed by the insertion of an index entry into the appropriate parent. A B-link tree node and an example page split are illustrated in Figure 3. Operations arriving at a newly split node with a search key greater than the high key use the right link to get to the appropriate page. Such a sideways traversal is termed a *link-chase*. Merges can also be done in two steps [Lani86], via a half-merge followed by the entry deletion at the next higher level. The B-link algorithms differ from the Bayer-Schkolnick and top-down algorithms in that neither readers nor updaters lock-couple on their way down to a leaf. We describe two variations of the B-link algorithms, LY and LY-LC.

In the LY algorithm (LY stands for Lehman-Yao), a reader descends the tree from the root to the leaf using S locks. At each page, the next page to be searched can either be a child or the right sibling of the current page. Here, readers release their lock on a page **before** getting a lock on the next page. Updaters behave like readers until they reach the appropriate leaf node. On reaching the appropriate leaf node, updaters release their S lock on the leaf and then try to get an X lock on the same leaf. After the X lock on the leaf is granted, they may either find that the leaf is the correct one to update or that they have to perform one or more link-chases to get to the correct leaf. Updaters use X locks while performing all further link chases, releasing the X lock on a page before asking for the next. If a page split or merge is necessary, updaters perform a half-split or half-merge. They then release the leaf lock **before** they search for the parent node starting from the last node (at the next higher level) that they used in their descent. That is, updaters that are propagating splits and merges use X locks at higher levels and do not lock-couple. In the basic LY algorithm, operations lock a maximum of one node at a time.

Due to the early lock releasing strategy in the basic LY algorithm, updaters that propagate index entries after completing half-splits or half-merges can encounter "inconsistent" situations; eg., a deleter at a higher level

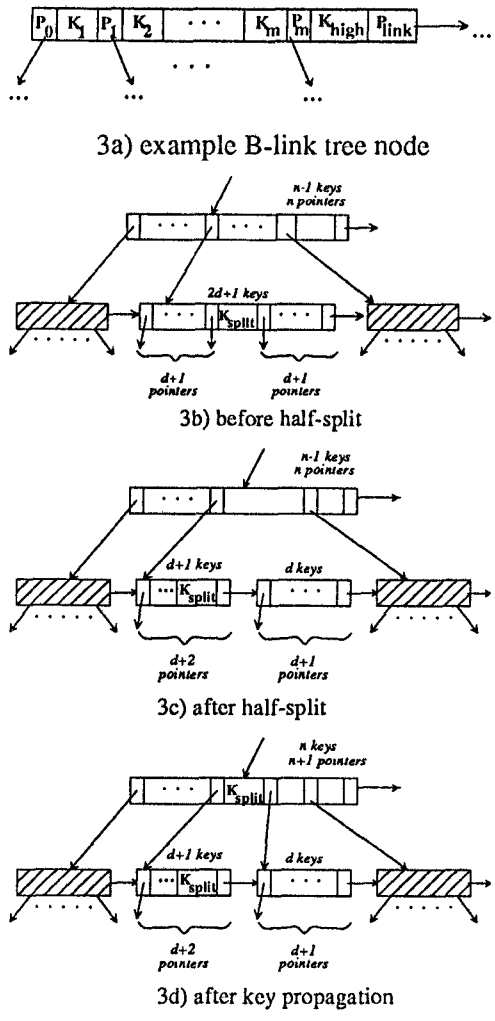

Figure 3: A B-link tree page split.

may find that the key to be deleted does not exist there (yet), and an inserter at a higher level may find that the key it is trying to insert already (still) exists. An updater that encounters such inconsistencies may be required to restart repeatedly (see [Lani86] for details). These inconsistent situations are avoided by the LY-LC algorithm, in which updaters hold an S lock on a newly split or merged node while acquiring an X lock on the appropriate parent node (in essence, lock-coupling on the way up). That is, the LY-LC algorithm differs from the LY algorithm in that updaters release their lock on a node that is half-split or half-merged only **after** getting an X lock on its current parent.

2.1.4 A New Optimistic Descent Algorithm

Updaters in the optimistic descent algorithms described earlier (TD-OPT and B-OPT) restart operations if they encounter a full leaf node rather than restarting them due to actual conflicts with other updaters. In a new optimistic algorithm that we designed, called OPT-DLOCK, restarts depend solely on deadlock-causing lock conflicts. OPT-DLOCK detects such conflicts by watching for circular waits of lock upgrade requests for the same page. A comparison of the performance of this algorithm with that of the other optimistic algorithms will provide interesting insights on the efficacy of the two restart strategies under various system and workload conditions.

In the OPT-DLOCK algorithm, readers follow the same locking strategy as in the Bayer-Schkolnick and top down algorithms. Updaters descend using S locks, keeping their scope locked until a safe node is reached, like updaters in the algorithms B-X and TD-X; they take an X lock on the leaf. In this algorithm, however, a node is considered safe only if it is both insertion safe and deletion safe. If the leaf is safe, the update is performed and all locks are released. An updater that reaches an unsafe leaf node will have at least all of the nodes in its scope (and possibly more, due to the new definition of safe node) locked with S locks, in addition to having the leaf itself locked with an X lock. Updaters reaching an unsafe leaf node release the leaf lock and then try to convert the S lock on the topmost node of their scope to an X lock. If this lock is granted, upaters proceed to drive away readers by getting X locks on the other nodes in their scope and then perform the actual update.

The new definition of safe node used in the OPT-DLOCK algorithm ensures that two updaters whose scopes intersect will always have the same top level safe node. Thus, two updaters with the same top level safe node will both try to convert their S locks on that node to X locks and will create a local deadlock at that node. Only one of them will succeed, with the others being restarted after releasing all locks associated with the failed B-tree operation. A restarted updater[3] repeatedly tries the protocol until it succeeds; starvation is avoided by assigning priorities to operations based on their first start time.

3 Simulation Model

Our model is a closed queueing model with a varying number of terminals. Transactions in this study are "tree transactions," each performing a single B-tree operation. Figure 4 shows the various possible states of a tree transaction in the system.

The system on which tree transactions operate is modeled using the DeNet simulation language [Livn90]. The system can have one or more CPUs and disks. Requests for the CPUs are scheduled using a FCFS (first-come first-served) discipline with no preemption. Each of the disks has its own disk queue, and these queues

[3]Note that a restart just involves re-trying the B-tree operation and is not a transaction abort.

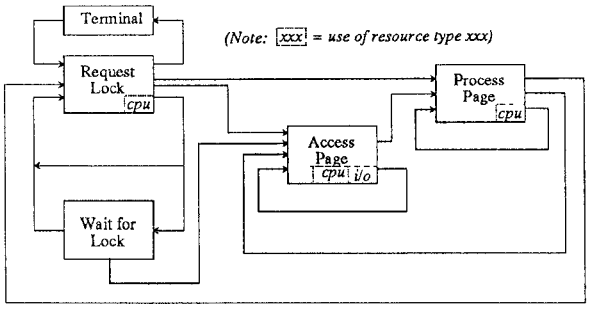

Figure 4: Transaction states.

are also managed in an FCFS fashion[4]. We assume uniform disk utilization. The buffer pool is managed in a global LRU fashion, and the buffer manager performs demand-driven writes.

The workload model consists of the number of terminals in the system, referred to as the multi-programming level (MPL) of the system, as well as the proportion of searches, inserts, deletes and appends in the workload. A given terminal can submit any one of four types of B-tree operations (search, insert, delete, or append). A terminal submits transactions one at a time. As soon as a transaction completes, it returns to the terminal. The terminal immediately generates another operation whose type is randomly determined using the set of probabilities given for the workload.

Keys for the search, insert, and delete operations are randomly chosen from a key space that consists of integer values between 1 and 80,000. The keys for appends are chosen sequentially from 80,001 onwards.

An important parameter of the B-tree is the maximum fanout of a B-tree page, the page capacity. This gives the maximum number of <key, pointer> entries in a page. In our model, the physical size of a B-tree page is always the same (in bytes), so a variation in fanout should be viewed as being due to different key sizes. For simplicity, all keys are of the same size, and no duplicates are allowed. Another parameter of the B-tree model is the locking algorithm in use.

In our experiments three factors will be varied — the workload (the percentage of searches, inserts, deletes, and appends), the system resources (the number of CPUs, disks, and buffers), and the structure of the B-tree (the fan-out and the initial number of keys). The simulation parameters for our experiments are listed in Table 2.

A system configuration is defined by values for the workload parameters, the number of CPUs, the number of disks, the B-tree fanout, and the buffer pool size. In each configuration, we varied the MPL and the concurrency control algorithm. At the start of each experiment, the buffer pool is initialized with as many B-tree pages as will fit; higher level pages are given priority in this initialization. Each simulation is stopped after 10,000 operations have completed. Batch probes in the DeNet simulation language are used with the response time metric to generate confidence intervals. For the results presented here, the 90% confidence interval is within 2.5% (i.e., ±2.5%) of the mean.

[4]We also ran experiments with an elevator disk scheduling algorithm and the results obtained were qualitatively similar to those presented here [Srin91].

num-cpus	Number of CPUs (1..∞)
num-disks	Number of disks (1..∞)
seek-time	Min: 0 msec; Max: 27 msec
cpu-speed	20 MIPS
cc-cpu	Cost for lock/unlock (100 inst.)
buf-cpu	Cost for buffer call (1000 inst.)
search-cpu	Cost for page search (50 inst.)
modify-cpu	Cost for key ins/del (500 inst.)
copy-cpu	Cost for page copy (1000 inst.)
init-keys	Keys in initial tree (40,000)
max-fanout	Entries/page (200/page, 8/page)
cc-alg	CC algorithm (LY, B-X, etc.)
num-bufs	Size of the buffer pool (see text)
num-operations	Ops per simulation run (10,000)
mpl	Multiprogramming level (1..300)
search-prob	Proportion of searches (0.0 .. 1.0)
delete-prob	Proportion of deletes (0.0 .. 1.0)
insert-prob	Proportion of inserts (0.0 .. 1.0)
append-prob	Proportion of appends (0.0 .. 1.0)

Table 2: Simulation parameters.

4 Performance Results

The relative performance of the various B-tree algorithms depends on characteristics like the workload composition, the system resources available, the B-tree structure and the multiprogramming level. We divide the algorithms into four classes and analyze the performance of these classes. The groupings of algorithms into classes are indicated in Table 3. The classes SIX-LC and X-LC are referred to collectively as *pessimistic algorithms* in the following discussion. We discuss the results of our experiments organized by workload. Due to space limitations, we shall only discuss the results of the experiments for the high fanout tree (200 keys/page); the corresponding low fanout results may be found in [Srin91].

4.1 Expt. 1: Low Contention

In our first set of experiments, we used a workload that consists of 80% searches and 10% each of inserts and deletes. The use of an equal proportion of random inserts and deletes ensures that a negligible number of

Class	*Algorithms*
B-LINK	{LY, LY-ABUF, LY-LC}
OPT	{OPT-DLOCK, TD-OPT, B-OPT}
SIX-LC	{B-SIX, TD-SIX}
X-LC	{B-X, TD-X}

Table 3: Algorithm classification.

splits and merges take place. The presence of few updaters and a small number of splits, therefore, creates a low data contention situation, and we use this workload as a filter to eliminate poor algorithms.

The buffer pool in this experiment has 200 pages, or about 75% of the tree size. For the case with a single CPU and disk, all algorithms showed virtually identical behavior and we omit the graphs due to lack of space. We found that the throughputs for all algorithms saturated at an MPL of 4 at a value that is only slightly greater than the throughput value at an MPL of 1. This is because, with only one CPU and disk, the disk rapidly becomes a bottleneck. We found that for all algorithms, at an MPL of 1, the disk is around 90% utilized, thus making some I/O and CPU parallelism possible. The left-over bandwidth is used up when the MPL is increased, and the system saturates by an MPL of 4 for all algorithms.

The throughput curves for the case with 1 CPU and 8 disks is given in Figure 5. Due to the additional system resources, the throughput of all algorithms increases to a higher level than in the single disk case before leveling off. In this case, the SIX-LC and X-LC lock-coupling algorithms only reach a maximum throughput of about half that of the optimistic (OPT) and B-link (B-LINK) algorithms. In trying to explain this, we found that the pessimistic algorithms (SIX-LC and X-LC) have a peak utilization of less than half of the available disk capacity, while the optimistic algorithms utilize the disk completely at high MPLs. This suggests that the throughputs for the pessimistic algorithms level off due to data contention, rather than due to resource contention. We also found that the lock waiting time at the root is a significant fraction of the insert operation response time (Figure 6) for the X-LC and SIX-LC algorithms, indicating that searching the root is the bottleneck. For the OPT and B-link algorithms, the response time increases at higher MPLs are due only to contention for the disks.

The bottleneck that the X-LC and SIX-LC algorithms form at the root can be understood intuitively by considering a system in which operations have to execute in several stages with the restriction that no two operations can execute the first stage simultaneously (though any number of operations can execute subsequent stages in parallel). Assume that it takes exactly 1 second to run through all stages, and that the first stage takes a fraction k $(0 < k < 1)$ of the total time to execute. Now, at an MPL of 1, the throughput will be 1 operation/second. At an MPL of 2, both operations may be phase-shifted and may not collide at the first stage. In that case, they will both have a response time equal to 1. However, the worst case is when both operations arrive at the first stage simultaneously, and one of them has a response time of 1 while the other has $1 + k$. Assuming equal probability for the collision and non-collision cases, we get an average response time of $(1 + k/2)$ and an average throughput of less than 2. At higher and higher MPLs, more and more collisions will occur. In fact, it can be shown that the asymptotic throughput at very high MPLs is $1/k$. Consequently, a bottleneck will form at very high MPLs at the first stage.

Searching the root page in the X- and SIX-locking algorithms is analogous to the first stage in the example system, and at high MPLs a bottleneck forms at the root. The bottlenecks are accelerated at higher MPLs due to the lock-coupling overhead of waiting for the lock at the next level. Notice that if k is very small, then bottlenecks will form at much higher MPLs than if k were large. We indeed noticed that in the memory-resident tree experiments, the bottlenecks formed earlier than in the experiments where the response time includes I/O This is because the overhead of searching the root (and waiting for a lock at the next level) in the memory-resident case is a significant proportion of the actual response time, while in situations that require disk I/Os for non-root nodes, it is a much smaller fraction of the response time.

The bottleneck at the root can affect the response time of operations symmetrically or asymmetrically. In the X-LC algorithms the bottleneck affects all types of operations equally; the search response times in Figure 7 are very close to the insert response times in Figure 6. On the other hand, in the SIX-LC algorithms, the response time for searches increases only slightly with MPL (Figure 7), while the response time for inserts increases steeply with MPL (Figure 6). This is because, in the SIX-locking algorithms, searches can overtake updaters on their way to a leaf and hence escape the bottleneck at the root. However, the system then gets filled with slower updaters, and the contention levels are much higher than in X-locking (where searches and updaters take approximately equal times to complete). In fact, the increase in response time for updaters is so large that the throughput of the SIX-locking algorithms is only slightly better than that of the X-locking algorithms (Figure 5) in spite of the almost constant search response times of the SIX algorithms. It should be noted that the phenomenon of a bottleneck at the root for pessimistic algorithms has been mentioned in earlier papers [Bili85, John90]; our contribution is to the understanding of how bottlenecks affect the response times of different B-tree operation types.

Finally, to get an idea of the extent to which the different algorithms can take advantage of the concurrency available in the workload and the high fanout B-tree structure, we present their throughput curves for the case of infinite resources in Figure 8. Note how the pessimistic algorithms level off at the same maximum throughput as in the 1 CPU and 8 disks case (Figure 5), while the optimistic and B-link algorithms make excellent gains in throughput with increasing MPL.

In addition to the above experiments, we also performed experiments in which the entire tree is in memory. The only difference between the disk bound experiments described above and the experiments with the memory-resident tree was that the CPU resource became the bottleneck at earlier MPLs than the disks did in the disk-bound experiments, as would be expected. However, the qualitative results were similar to those for the disk bound case. We omit these graphs due to space limitations.

It should be noted that in the above experiments, there was no significant performance difference between the top-down algorithms and the corresponding Bayer-Schkolnick algorithms. This is to be expected since the tree has only three levels; the number of exclusive locks held at one time on the scope of an update is hardly different in the two cases (due to the rarity of splits and merges).

4.2 Expt. 2: Moderate Contention

The 100% insert workload used in this experiment differs from the one used in the first set of experiments in that it creates higher data contention due to the significant number of splits required to accommodate the new keys being inserted. In particular, nodes one level higher than the leaf are modified frequently under this workload. Again, we will discuss only the high fanout tree experiments here.

We first consider experiments that were conducted with a buffer pool size of 200 pages, or about 75% of the initial B-tree size. In the single CPU and disk case, we again found that there is not much difference between the algorithms. Figure 9 contains the throughput curves for the 1 CPU and 8 disks case. As in the earlier set of experiments, the pessimistic algorithms again perform much worse than the optimistic and B-link algorithms. In addition, for the first time we see differences between the B-link and optimistic algorithms. We shall comment on three important aspects of these differences.

Firstly, the optimistic algorithms perform worse than the B-link algorithms. The reason is that the optimistic algorithms are only able to utilize a maximum of 80% of the disk resources due to data contention, while the B-link algorithms are still able to saturate the disks at high MPLs. As with the pessimistic algorithms in the first set of experiments, the algorithms TD-OPT and B-OPT lose their performance here due to lock waiting at the root.

Secondly, we notice in Figure 9 that the algorithm TD-OPT achieves a peak throughput higher than B-OPT. This is because the lock waiting time for the B-OPT algorithm increases faster than that of TD-OPT, so TD-OPT performs better than B-OPT. Recall that, in both algorithms, inserters make a second pass with SIX locks if they encounter a full node. Since SIX locks are incompatible with each other, two updaters in their second phase interfere with each other if both try to lock the root at the same time. Furthermore, in B-OPT, an inserter in the second pass can also interfere with an inserter in the first pass[5]. This extra interference causes the average waiting time at the root for B-OPT to be greater than that of TD-OPT. Moreover, in the TD-OPT algorithm, inserters in their first pass can overtake those in their second pass. Such overtaking, while leading to less waiting time at the root, could increase the number of restarts; i.e., overtaking allows more than one transaction to reach the same full node. A look at the restart counts for the TD-OPT and B-OPT algorithms (Figure 10) indeed shows that TDOPT at high MPLs performs about 4 times as many restarts as B-OPT. However, this is not very expensive in this disk bound case, as all pages needed after a restart are most likely in memory.

Thirdly, we find that the throughput of OPT-DLOCK increases quickly at low MPLs and then more slowly at high MPLs. Unlike the other optimistic algorithms, however, its throughput does not saturate. The reason for this behavior is the difference in its handling of restarts. Since the conflict level is high, OPT-DLOCK performs many restarts (Figure 10). However, the restarts in TD-OPT and B-OPT conflict at the root node, while those in OPT-DLOCK conflict at the level above the leaf. There are two nodes at this level in the high fanout tree, so waiting is split between the two non-leaf index nodes at this level. Thus, the waiting times for OPT-DLOCK increase more slowly than those for B-OPT and TD-OPT, and hence the throughput of OPT-DLOCK increases slightly even at high MPLs. The presence of more nodes at the level of the tree above the leaf would make this algorithm perform better due to less waiting at this level.

Just to give a flavor of the in-memory tree results for this workload, we reproduce the throughput curves for the in-memory infinite resources case in Figure 11. We find that TD-OPT performs worse than B-OPT here due to the greater number of restarts, as the overhead of a restart is now comparable to the response time of the operation itself. The number of restarts here are

[5]The first pass in TD-OPT is done by lock-coupling with S locks, while in B-OPT, the lock-coupling is done with IX locks. IX locks are compatible with other IX locks but not SIX locks, while S locks are compatible with both.

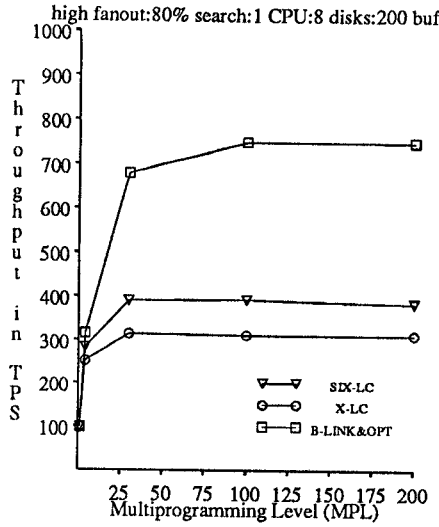

Figure 5: Expt. 1, eight disks

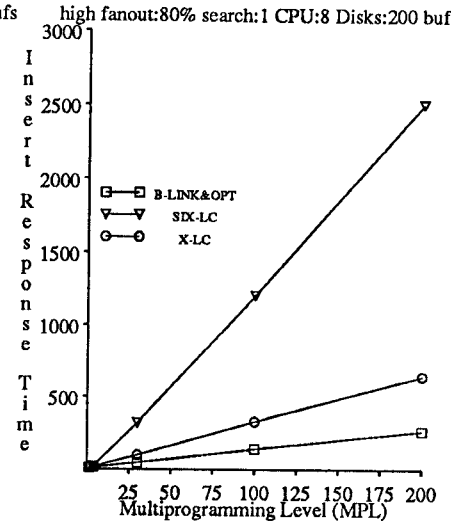

Figure 6: Expt. 1, Insert times (msec)

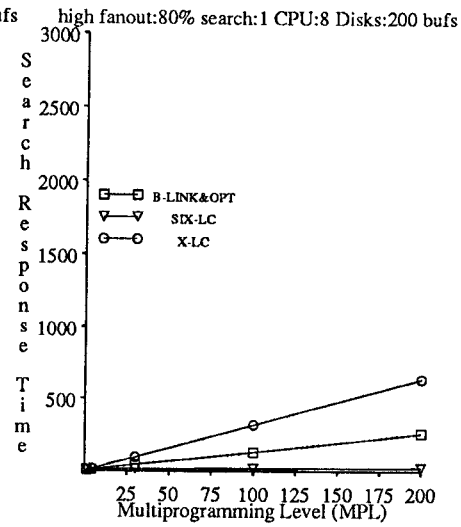

Figure 7: Expt. 1, Search times (msec)

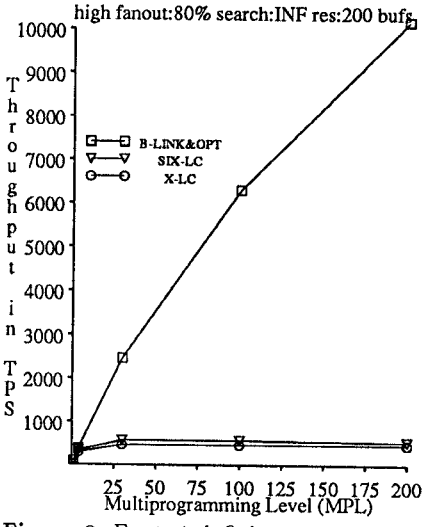

Figure 8: Expt. 1, infinite resources

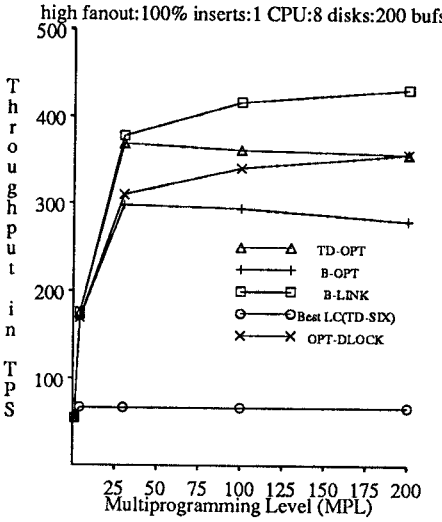

Figure 9: Expt. 2, eight disks

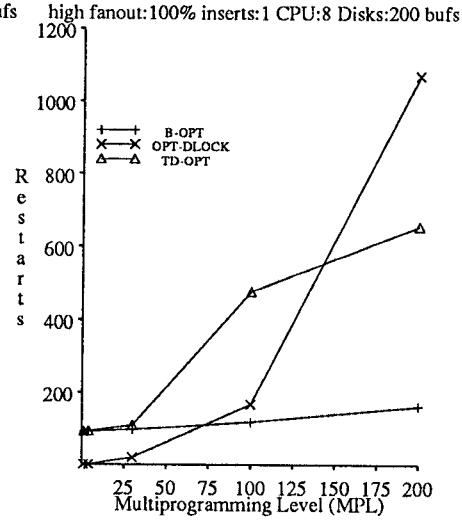

Figure 10: Expt. 2, Restarts/10,000 ops

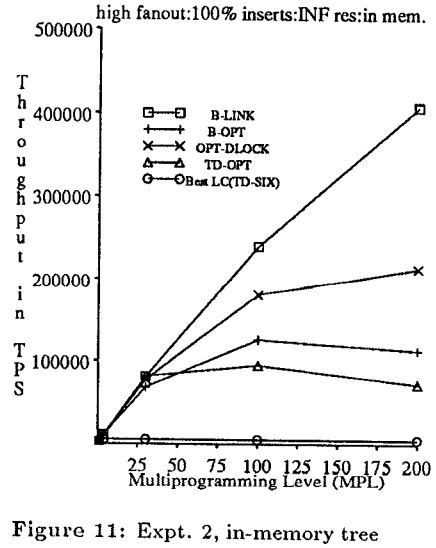

Figure 11: Expt. 2, in-memory tree

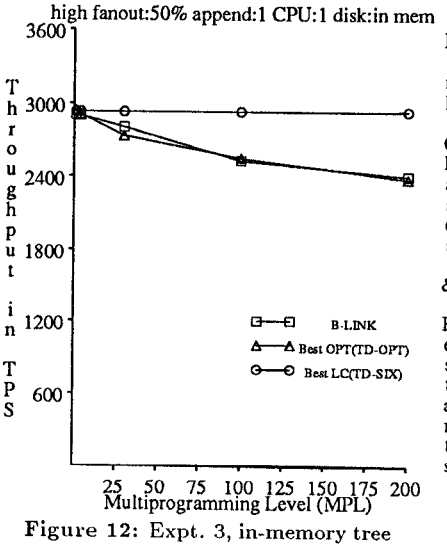

Figure 12: Expt. 3, in-memory tree

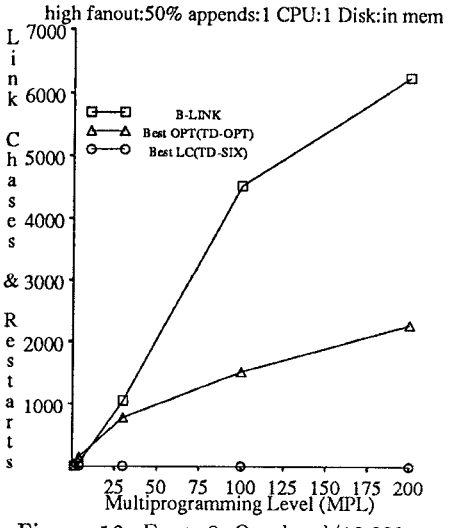

Figure 13: Expt. 3, Overhead/10,000 ops

essentially the same as in the earlier disk bound case (Figure 10).

Just as the optimistic algorithms perform restarts, the B-link algorithms also involve extra overhead in high contention situations due to link-chases. We found that unlike the optimistic algorithms, where the number of restarts varied widely with the fanout of an index node, the link-chases were not very different between trees with high and low fanout. In the case of low fanout trees, splits are more frequent, but the probability of an operation visiting a leaf that is being split by an earlier operation is very small due to the presence of thousands of leaf nodes. In trees with high fanouts, even though the probability of an operation visiting a leaf node while it is being split is fairly high, splits are relatively infrequent, thus keeping link-chases down. These two effects seem to balance each other and keep the number of link-chases fairly independent of the fanout.

To summarize, with a 100% insert workload, the pessimistic algorithms performed worse than other algorithms, as in experiment set 1. The optimistic algorithms TD-OPT and B-OPT performed worse than the B-link algorithms for all system conditions except the single CPU and single disk case.

4.3 Expt. 3: High Contention

In our third and final set of experiments, we use a workload that consists of 50% appends and 50% searches. The appends create extremely high contention for the few right-most leaf nodes in the tree. The searches are random, however, and do not interfere with the appends that are taking place. Due to space limitations, we present only a small subset of these results from [Srin91].

For experiments on a high fanout tree with a buffer pool size of 200 pages, we found little or no difference in throughput between the algorithms in the single CPU and single disk case, or even in a system with 1 CPU and 8 disks. In the infinite resource situation, we found that the B-link algorithms performed better than all of the other algorithms, and their throughputs continued to increase at high MPLs. We still found no significant difference between the LY and LY-LC algorithms.

We now switch to experiments where the entire tree is in memory. One of the few cases where a pessimistic algorithm performs better than the B-link and optimistic algorithms is the case of a 1 CPU and 1 disk system with the entire tree in memory (Figure 12). The reason is that very high contention between append operations causes the number of link-chases to increase enormously at high MPLs for the B-link algorithms (Figure 13). Also, notice that the optimistic algorithms perform an increasing number of restarts. Due to the very fast response time of operations on a memory-resident tree, the overheads for a link-chase or a restart are of the same order as the response time of an operation, and the B-link and optimistic algorithms show thrashing behavior at high MPLs. When more CPU resources are available in the system, however, the B-link and optimistic algorithms once again perform better than the pessimistic algorithms.

4.4 Summary of Results

We can broadly summarize the results of the performance experiments as follows:

1. In a system with a single CPU and disk, there is no significant performance difference between the various algorithms.
2. Lock-coupling with exclusive locks is generally bad for performance. Even for workloads dominated by searches, algorithms in which updaters use such a lock-coupling strategy cannot take full advantage of CPU and I/O parallelism, even in systems with only a few CPUs and disks.
3. The extent to which an overhead like a restart or a link-chase directly affects performance depends more on the number of conflicts that it creates than on the extra resources used.

Apart from the above experiments, we also ran several other experiments with low fanout trees whose results yielded interesting insights about the relative performance of the various algorithms. We summarize some of the results here[6].

1. The algorithms TD-OPT and B-OPT performed close to their corresponding pessimistic algorithms at high MPLs.
2. The OPT-DLOCK algorithm performed close to (and sometimes better than) the B-link algorithms.

Based on the results of the previous section, we can also predict the performance of algorithms that have not been explicitly simulated in our system relative to the performance of B-link algorithms. The following are some of the predictions from [Srin91].

1. The mU protocol [Bili87] is likely to perform worse than the B-link algorithms in certain conditions, and never better.
2. The ARIES/IM algorithm [Moha89], is likely to perform close to the B-link algorithm if more than one page split at a time is allowed.

5 Related Work

Due to space limitations, we will compare our work only with the most relevant performance study published thus far [John90]. This study compared three algorithms (B-X, B-OPT and LY) assuming infinite resource

[6] See [Srin91] for a detailed description of low fanout tree results as well as results for high fanout trees not mentioned here due to space limitations.

conditions and did not model buffer management[7]. The key results of [John90] are that the root is a bottleneck for the lock-coupling algorithms, and that, in situations with a negligible amount of link-chases, the B-link algorithm performs much better than the other algorithms. Our simulation model differs from their analytical model in that we take into account resource contention and buffer management. In addition to the low contention situations analyzed in their model, we have studied very high concurrency situations where a large number of link-chases are performed by the B-link algorithms. We have also analyzed differences between variants of the B-link algorithm.

6 Conclusions

The most important conclusion of this study is that the B-link algorithms perform the best among all of the algorithms that we studied over a wide range of resource conditions, B-tree structures, and workload parameters. Even in a high contention workload of appends, the B-link algorithms show gains in throughput under plentiful resource conditions. Among the B-link algorithms, we have further shown that a slightly conservative update algorithm that locks a maximum of three nodes at one time generally performs as well as one that locks a maximum of only one node at a time. The former variation of the B-link algorithm is more suitable for use in practice, as it avoids some rather complex situations encountered in the latter.

Acknowledgements

We would like to thank David Dewitt, Miron Livny, Rajesh Mansharamani, S. Seshadri, and Manolis Tsangaris for their constructive comments on a preliminary version of this paper.

References

[Agra87] Agrawal, R., Carey, M., and Livny, M. "Concurrency Control Performance Modeling: Alternatives and Implications" *ACM TODS,* **12, 4** Dec. 1987.

[Baye77] Bayer, R. and Schkolnick, M. "Concurrency of Operations on B-trees" *Acta Informatica,* **9** 173–189 (1977).

[Bili85] Biliris, A. "A Model for the Evaluation of Concurrency Control Algorithms on B-trees" *Boston University Rep.,* **85-015** (1985).

[Bili87] Biliris, A. "Operation Specific Locking in B-trees" *Proc. 6th ACM Symp. on PODS ,* San Diego, California, 159–169, March 1987.

[7] In parallel with this work, the authors of [John90] have enhanced their analytical model to include other algorithms, buffer management, etc. It will be interesting to compare our simulation results with their new analytical results [John91].

[Care84] Carey, M., and Thompson, C. "An Efficient Implementation of Search Trees on ⌈lg N + 1⌉ Processors" *IEEE TOCS,* **11(2)** November 1984.

[Come79] Comer, D. "The Ubiquitous B-Tree" *ACM Computing Surveys,* **11(4)** 412 (1979).

[Good85] Goodman, N., and Shasha, D. "Semantically-based Concurrency Control for Search Structures" *Proc. 4th ACM Symp. on PODS,* March 1985.

[Gray79] Gray, J. "Notes On Database Operating Systems" *Operating Systems: An Advanced Course,* Springer-Verlag, 1979.

[Guib78] Guibas, L., and Sedgewick, R. "A Dichromatic Framework for Balanced Trees" *Proc. 19th Symp. on FOCS,* (1978).

[John89] Johnson, T. and Shasha, D. "Utilization of B-trees with Inserts, Deletes and Searches" *ACM Symp. on PODS,* 235–246, 1989.

[John90] Johnson, T. and Shasha, D. "A Framework for the Performance Analysis of Concurrent B-Tree Algorithms" *Proc. 9th Symp. on PODS,* (1990).

[John91] Johnson, T., Personal communication, March 1991.

[Kell88] Keller, A. and Wiederhold, G. "Concurrent Use of B-trees with Variable-Length Entries" *SIGMOD Record,* **17(2)** June 1988.

[Kwon82] Kwong, Y., and Wood, D. "A New Method for Concurrency in B-trees" *IEEE Trans. on Soft. Engg.,* **SE-8(3)** May 1982.

[Lani86] Lanin, V. and Shasha, D. "A Symmetric Concurrent B-tree Algorithm" *Proc. FJCC,* 380–389. (1986)

[Lehm81] Lehman, P., and Yao, S. "Efficient Locking for Concurrent Operations on B-trees" *ACM TODS,* **6(4)** December 1981.

[Livn90] Livny, M. "DeNet User's Guide" *ver.,* **1.5** 1990.

[Mill78] Miller, R., and Snyder, L. "Multiple Access to B-trees" *Proc. Conf. on Inf. Sci. and Systems,* Johns Hopkins University, Baltimore, MD, March 1978.

[Moha89] Mohan, C. and Levine, F. "ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging" *IBM Research Report,* **RJ 6846** (1989).

[Mond85] Mond, Y. and Raz, Y. "Concurrency Control in B^+-trees Databases Using Preparatory Operations" *Proc. 11th VLDB Conf.,* 331–334 (1985).

[Sagi85] Sagiv, Y. "Concurrent Operations on B^*-trees with Overtaking" *Proc. 4th Symp. on PODS,* 28–37 (1985).

[Sama76] Samadi, B. "B-trees in a System With Multiple Users" *Information Processing Letters,* **5(4)** (1976).

[Shas85] Shasha, D. "What Good are Concurrent Search Structure Algorithms for Databases Anyway?" *Database Engineering,* **8(2)** June 1985.

[Srin91] Srinivasan, V. and Carey, M. J. "Performance of B-tree Concurrency Control Algorithms" *University of Wisconsin Report,* **TR 999** February 1991.

[Weih90] Weihl, W. E., Wang, Paul. "Multi-Version Memory: Software Cache Management for Concurrent B-trees" *MIT Laboratory of Computer Science Manuscript,* (1990).

[Yao78] Yao, A. C. " On Random 2–3 Trees" *Acta Informatica,* **9** 159–170 (1978).