PERFORMANCE
OF
B-TREE CONCURRENCY CONTROL ALGORITHMS

by

V. Srinivasan
Michael J. Carey

Computer Sciences Technical Report #999
February 1991

PERFORMANCE
OF
B-TREE CONCURRENCY CONTROL ALGORITHMS

V. Srinivasan
Michael J. Carey

Computer Sciences Department
University of Wisconsin
Madison, Wisconsin 53706 USA

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

PERFORMANCE OF B-TREE CONCURRENCY CONTROL ALGORITHMS

V. Srinivasan Michael J. Carey

Computer Sciences Department
University of Wisconsin - Madison
Madison, Wisconsin 53706 USA
e-mail: srini@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 algorithms using a detailed simulation model of B-tree operations in a centralized DBMS. Unlike the few earlier performance studies on this topic, ours considers a wide range of data contention situations and resource conditions. Furthermore, based on the performance of a representative set of B-tree concurrency control algorithms, including one new algorithm, we make projections on the performance of others in the literature. 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[†] 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, Shas84, 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:

[†] 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]

(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. The results presented here should therefore be useful to database system designers in a wide range of systems including single and multiple processor systems with one or more disks.

(3) In our experiments, we consider tree structures with high and low fanout, a wide range of resource conditions, and workloads which contain various proportions of searches, inserts, deletes, and appends.

(4) We measure a wide variety of performance measures like throughput, average response time of operation types, resource utilizations, lock waiting times, buffer hit rates, number of I/Os, link chases in B-link algorithms, probability of splitting and merging, frequency of restarts, etc. These measures help us to make precise statements about the performance of searches, deletes and inserts in the different versions of the algorithms.

Section 2 briefly reviews the set of B-tree concurrency 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. In Section 5, we discuss how these results can be used to predict the performance of other protocols. Section 6 compares this study with related work in the field. Finally, in Section 7, we summarize our key results and our plans for future work.

2. B-trees in a Database Environment

An index is a structure that efficiently stores and retrieves information (usually one or more record identifiers) associated with a search key. The index can be either one-to-one (unique) or one-to-many (non-unique). The keys themselves can have fixed or variable lengths. Though we shall restrict ourselves to unique indices with fixed length keys for this study, most of our results also directly apply to trees with variable length keys or duplicate keys.

2.1. B-tree Review and Terminology

A B-tree index is a page-oriented tree that has the following properties. Firstly, it is a balanced *leaf* search tree — actual keys are present only in the leaf pages, and all paths from the root to the leaf are of the same length. A B-tree is said to be of order d if every node has at most $2d$ separators[§], and every node except for the root has at least d separators. The root has at least two children. The leaves of the tree are at the lowest level of the tree (level 1) and the root is at the highest level. The number of levels in the tree is termed the tree height. A nonleaf node with

[§] A *key* is usually meant to imply that associated information for that value exists in the index. A *separator* defines a search path to leaf pages that contain actual keys and associated information.

j separators contains $j+1$ pointers to children. A <pointer, separator> pair is termed an index entry. Thus, a B-tree is a multi-level index with the topmost level being the single root page and the lowest level consisting of the set of leaf pages. Figure 1 summarizes these concepts.

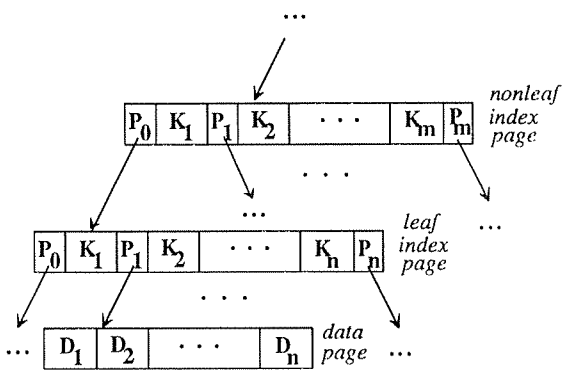

Figure 1: An example B-tree fragment.

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

Table 1: Lock compatibility table.

The index is stored on disk, and a search, insert, or delete operation starts by searching the root to find the page at the next lower level that contains the subtree having the search key in its range. The next lower level page is searched, and so on, until a leaf is reached. The leaf is then searched and the appropriate action is performed. Operations can be unsuccessful; for example, a search may not find the required key.

As keys are inserted or deleted, the tree grows or shrinks in size. When an updater tries to insert into a full leaf page or to delete from a leaf page with d entries, a page split or page merge occurs. 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 contain at least d entries, 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.

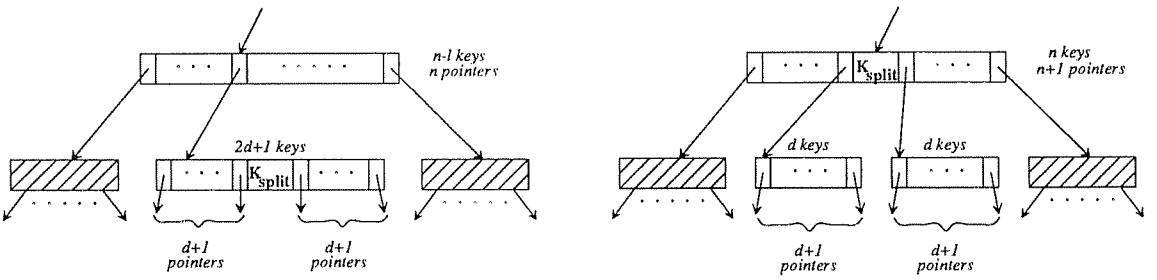

Figure 2: A B-tree page split.

2.2. B-tree Concurrency Control Algorithms

In our discussions of this section and the rest of the paper, we shall use the lock modes S, IX, SIX, and X. Their lock compatibility relationships are given in Table 1.

A naive B-tree concurrency control algorithm would treat the entire B-tree as a single data item and use locks (or latches[†]) on just the root page to prevent conflicts. Readers (searches) get S locks on the root, while updaters (inserts or deletes) get X locks on the root. Locks have to be held for the entire duration of an operation. The naive algorithm can be improved by considering every index page as an independently lockable item and making use of the following relation 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 be present 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, 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 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. This strategy ensures that when an update operation reaches a leaf, it holds X locks on all pages in its scope and no locks on any other index node. Updaters and readers whose scopes do not interfere can execute concurrently. However, a considerable number of conflicts may be caused at higher level nodes due to the use of X locks. A class of algorithms that improves on the above idea was proposed by Bayer and Schkolnick [Baye77].

2.2.1. 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.

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

[†] Latches [Moha89] can be thought of as fast locks. They are also less general than locks; eg., no deadlock detection is performed for latch waits. In this paper, latches can be used wherever locks are used.

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.

2.2.2. Top-down Algorithms

In top-down algorithms [Guib78, Care84b, 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.

2.2.3. 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. After a half-split, and before the <key, pointer> pair corresponding to the new page has been inserted into the parent page, the new page is reachable through the right link of the old page. 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 that have been proposed [Lehm81, Sagi85, Lani86] 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 study three variations of the B-link algorithms, LY, LY-LC and LY-ABUF.

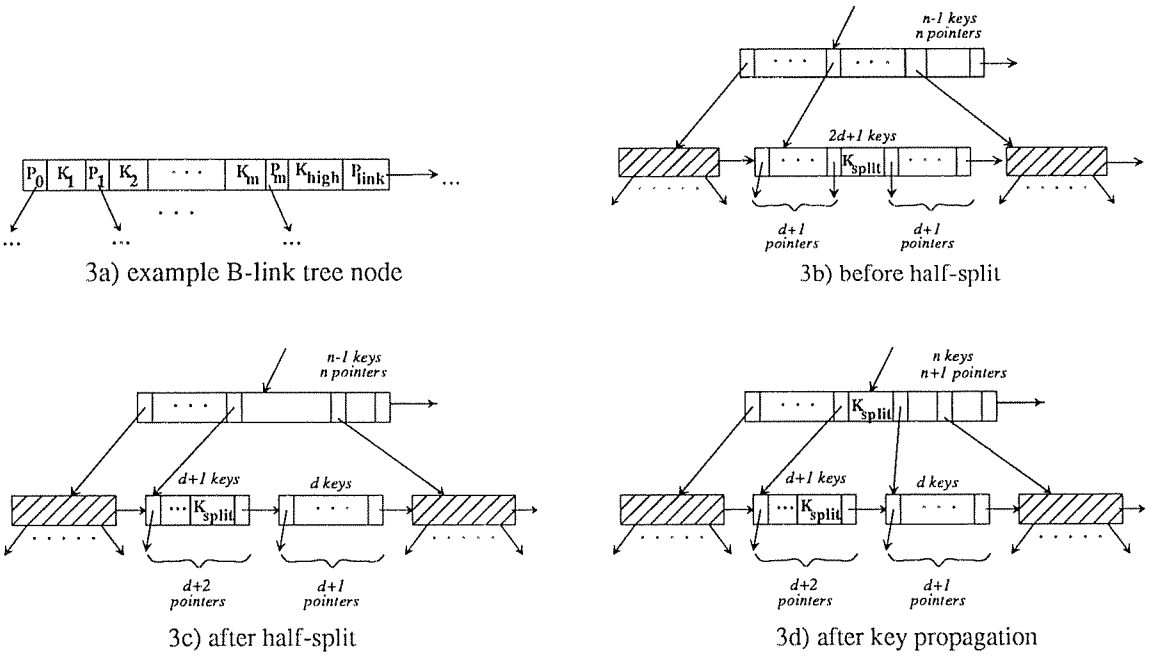

Figure 3: A B-link tree page split.

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 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.

In the B-link algorithm as it was first proposed in [Lehm81], readers did not use locks at all. Instead, they relied on the atomic nature of disk I/Os and used their own consistent copies of pages. To account for buffer hits, we modified this original algorithm to use a buffer manager that provides support for such an atomic read-write model. Since readers do not lock pages, a reader instead gets a read-only copy of the most recent version of a page. Updaters in their first descent to the leaf behave just like readers, using read-only copies with no locking. However, updaters do have to acquire an X lock on a page before requesting a writable copy of the page. Finally, whenever an updater frees a writable copy, this copy is made the current version of the page and future page requests get a copy of this new version. We implemented an algorithm based on the above atomic buffer model called LY-ABUF.

2.2.4. 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 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[‡] repeatedly tries the protocol until it succeeds; starvation is avoided by assigning priorities to operations based on their first start time.

Apart from the algorithms described above, several other B-tree concurrency control algorithms have been proposed as well [Kwon82, Bili87, Moha89]. We shall discuss these other algorithms in Section 6.

3. Simulation Model

Our model is a closed queueing model with a varying number of terminals and a zero think time between the completion of one transaction submitted by a given terminal and the submission of the next one. Transactions in this study are "tree transactions," each performing a single B-tree operation (search, insert, or delete). There are three main components of the simulation model: the system model, which models the behavior and resources of the database system; a workload model, which models the mix of transactions in the system's workload; and a B-tree model, which characterizes the structure of the B-tree. Apart from these, there is the actual concurrency control algorithm that is being executed.

3.1. System Model

The system model is intended to encapsulate the resources present in a database system and the major aspects of the flow of transactions 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 and these are modeled using a module called the *resource manager*. A CPU resource can be used in several ways — it is used when a concurrency control request or a buffer page request is processed, or when a B-tree page is processed. Requests for the CPUs are scheduled using a FCFS (first-come first-served) discipline with no preemption. A common queue of pending requests is maintained and when a CPU becomes free, the first request in the queue is assigned to it. The disk resource is used when a B-tree page is read into or written out of the buffer pool. Each of the disks has its own

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

disk queue, and these queues are also managed in an FCFS fashion†. When a new I/O request is made, the disk for servicing the request is chosen randomly from among all disks (i.e., we assume uniform disk utilization). Each I/O request is modeled as having three components: a seek to a randomly chosen cylinder from the current position of the disk head, a randomly chosen time between the minimum and maximum rotational latency, and a fixed page transfer time.

The physical resource model also includes a buffer pool for holding B-tree nodes in main memory. The behavior of the buffer manager is captured via a DeNet module called the *buffer manager*. The buffer pool is managed in a global LRU fashion. Transactions *fix* each page at the buffer pool prior to processing it, and they *unfix* each page when they are done processing it and no longer need it to be memory-resident; pages are placed on the LRU stack at the point when they are unfixed. The buffer performs demand-driven writes. Fixing a page may therefore involve upto two I/Os, one to write out a dirty page and another to read the requested page in. Apart from the buffer manager, there is a module called the *lock manager* that models the acquiring and releasing of locks.

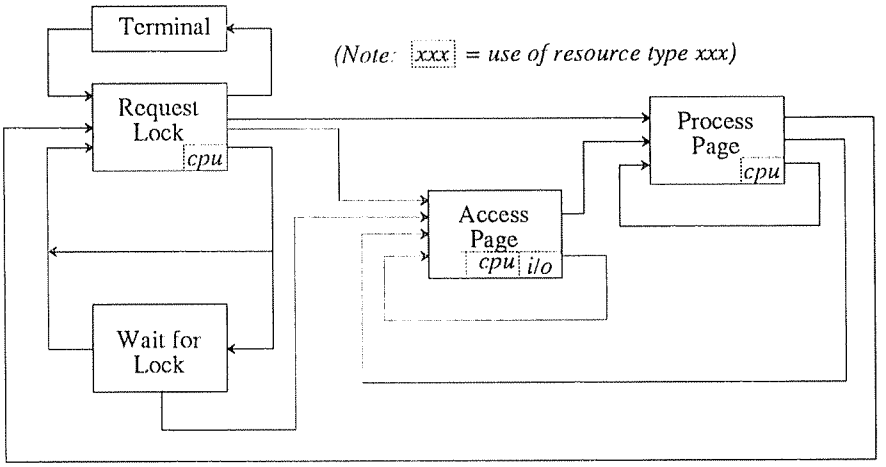

Figure 4: Transaction states.

3.2. Transaction Flow

Figure 4 shows the various states of a tree transaction in the system. Once a terminal submits a transaction, we say that the transaction is active. An active transaction is always in one of four states. The first state, the "request lock" state, is entered when it needs to set a lock, to convert the mode of an existing lock to a different lock mode, or to release a lock. A concurrency control CPU cost is associated with processing such a request. The second state, "wait for lock," is entered when a transaction requests a lock that is already held by another transaction in a conflicting mode. Transactions waiting for locks on a B-tree node are queued in order of arrival, and waiting transactions are awakened when a transaction holding a conflicting lock releases it. A transaction

† We also ran experiments with an elevator disk scheduling algorithm and the results of these experiments will be briefly described later.

locks and unlocks index pages according to the locking strategy of a particular concurrency control algorithm. The third state, "access page," is entered when a transaction wants to fix or unfix a B-tree page. In this state, in the event of a buffer pool miss, a transaction will either do the I/O itself or wait for an already pending I/O for the page to complete. The time associated with a page access consists of two components — the CPU cost in the buffer manager, and the waiting time for any disk I/O. The fourth possible state for an active transaction is the "process page" state shown in Figure 4. This state models the processing of a B-tree node (e.g., search, insert, delete, split, merge), and it has a CPU cost associated with it that depends on the type of operation being performed. The particular path that a given transaction follows through these four states therefore depends on the type of operation that it performs, the locking protocol employed, the size of the pool of pages for buffering B-tree nodes, and the degree of lock conflicts experienced by the transaction during its execution.

3.3. Workload Model

One component of the workload is the number of terminals in the system, referred to as the multi-programming level (MPL) of the system. A given terminal can submit any one of the four types of B-tree operations (search, insert, delete, or append). Another component of the workload is therefore a set of probabilities that define the proportion of searches, inserts, deletes and appends in the workload. 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 chosen from a key space that consists of integer values between 1 and 80,000. Inserts use even keys from the key space, while deletes use odd keys, thus ensuring that inserts and deletes do not interfere at the level of key values. To ensure that deletes are always successful, an initial tree is built using a random permutation of all of the odd keys in the key space. In contrast to updaters, searches can use both odd and even integers as key values. Finally, the keys for appends are chosen sequentially from 80,001 onwards. The actual series of keys for inserts and deletes are chosen from random permutations of their respective portions of the key space.

3.4. B-Tree Model

Transactions in our simulation model concurrently operate on the same B-tree. The B-tree model describes the characteristics of this index. 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 particular locking algorithm in use. The B-tree and its locking protocols are both modeled by the *B-tree manager* module. There are several versions of the B-tree manager, each corresponding to a different locking algorithm.

3.5. Performance Metrics

We use the above system, workload, and B-tree models as a platform for studying the performance of the B-tree concurrency algorithms described in Section 2. The main performance metric used for the study is the throughput rate for tree operations, expressed in units of tree transactions per second (TPS). We also monitored several other performance measures, including operation-specific measures like tree operation response times, waiting times for locks at various levels, buffer hit rates, I/O service times, link chases for B-link algorithms, restarts for optimistic protocols etc. These measures will be used to explain the results seen in the throughput curves and also to compare and contrast protocols that perform similarly in terms of throughput.

In addition to concrete performance measures such as transaction throughput, it would be nice if the *level of concurrency* provided by the protocols could be somehow characterized. For this study, we have adopted the throughput of the protocols under *infinite resources* [Fran83, Tay84, Agra87] as a measure of the level of concurrency that they provide. The resource manager simulates such a condition by replacing the CPU and disk scheduling code with pure time delays. Transactions then proceed at a rate limited only by their processing demands and locking delays, so protocols that reduce locking delays (i.e., permit higher concurrency) provide significant performance improvements.

3.6. Range of Experiments and Parameters

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

The B-tree can have one of two fanouts, a high fanout (200 entries/page) or a low fanout (8 entries/page). In all of our experiments, we started with a tree containing 40,000 keys. In the high fanout case, the initial tree is a three-level tree containing 3 index pages and 260 leaf pages. The initial tree in the low fanout case has six levels (seven for the top-down algorithms[†]) with around 1500 non-leaf pages and 7000 leaf pages.

For each of the high and low fanout trees, we consider two buffer pool size settings — one where the tree fits entirely in memory, and the other where the number of buffer pages is less than the number of pages in the B-tree. For the high fanout case, the two buffer pool sizes are 200 pages and 600 pages. A 200 page buffer pool results in around 75% of the tree being in memory, while the 600 page setting leads to an in-memory tree (even if the tree grows in size). The corresponding sizes for the buffer pool in the low fanout tree are 600 pages (7% of the tree in memory) and 12,000 pages (memory-resident tree). In cases where the buffer pool size is smaller

[†] Pre-splitting in the top-down algorithms leads to early splits for non-leaf pages, and this causes the tree built using these algorithms to have a larger height than those built using the bottom-up strategies. This effect is significant only for low fanouts.

num-cpus	Number of CPUs (1..∞)
num-disks	Number of disks (1..∞)
disk-seek-time	Min: 0 msec; Max: 27 msec
cpu-speed	20 MIPS
cc-cpu	CPU cost for a lock or unlock request (100 instructions)
buf-cpu	CPU cost for a buffer call (1000 instructions)
page-search-cpu	CPU cost for a page binary search (50 instructions)
page-modify-cpu	CPU cost for a insert/delete (500 instructions)
page-copy-cpu	CPU cost to copy a page (1000 instructions)
num-init-keys	Number of keys present in the initial tree (40,000)
fanout	Number of index entries per page (200/page, 8/page)
cc-alg	Concurrency control protocol (LY, B-X, TD-SIX, etc.)
num-bufs	Size of the buffer pool (see text)
num-operations	Number of operations in the simulation run (10,000)
mpl	Multiprogramming level (1..300)
search-prob	Proportion of searches (0.0 .. 1.0)
delete-prob	Proportion of inserts (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.

than the size of the tree, the system will be disk bound due to the large difference between the per-page CPU and disk service times.

In an actual database system, it is difficult to predict exactly what the operation mix is going to be. Furthermore, any system is bound to undergo changes in workload from time to time. In order to capture a wide range of operating conditions, we used four different workloads in our experiments: a search dominant workload (80% searches, 10% deletes and 10% inserts), an update dominant workload (40% inserts, 40% deletes and 20% searches), an insert workload (100% inserts), and an append workload (50% each of searches and appends).

Like the workload, the system resources in our experiments also span a wide range of conditions. For the in-memory tree case, we study three different resource settings: one CPU, eight CPUs, and infinite resources. In this case, the number of disks is immaterial since the tree is always in memory and no I/Os are performed. For the case where the buffer pool size is smaller than the size of the B-tree, where the system is disk-bound, we again study three situations: one CPU and one disk, one CPU and eight disks, and infinite resources.

A system configuration consists of a fixed value for each of the following parameters: the workload, the number of CPUs, the number of disks, the B-tree fanout, and the buffer pool size. Using the parameter values described above, there are twelve system configurations possible for each workload. In each configuration, we varied the MPL and conducted one experiment for each 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.

The experiment 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 all of the data presented here, the 90% confidence interval is within 2.5% (i.e., ±2.5%) of the mean.

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. When presenting data on performance measures, we will provide the curve for a representative algorithm of a class rather than reproduce all curves for all algorithms. We reproduce the actual curve for an algorithm only when it differs from the others in its class. 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.

Class	*Algorithms*	*How Related*
B-LINK	{LY, LY-ABUF, LY-LC}	B-link algorithms
OPT	{OPT-DLOCK, TD-OPT, B-OPT}	Optimistic descent algorithms
SIX-LC	{B-SIX, TD-SIX}	SIX lock coupling algorithms
X-LC	{B-X, TD-X}	X lock coupling algorithms

Table 3: Algorithm classification.

4.1. Experiment Set 1: Low Data Contention, Steady State Tree

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 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 bad algorithms. We shall first discuss experiments conducted on the high fanout tree (200 keys/page) followed by the results for the low fanout tree (8 keys/page). For each subset of experiments, we shall compare the performance of alternative B-tree locking algorithms at various resource levels starting with a single CPU and disk.

4.1.1. High Fanout Tree Experiments

The throughput curves for the case with a single CPU and disk are shown in Figure 5. The buffer pool in this experiment has 200 pages, or about 75% of the tree size. It is seen from the figure that the throughputs for all algorithms are only slightly greater at higher MPLs than at an MPL of 1. This is because there is only one CPU and disk, and 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 disk becomes fully utilized by an MPL of 4 for all algorithms. After that, no improvement in throughput is possible.

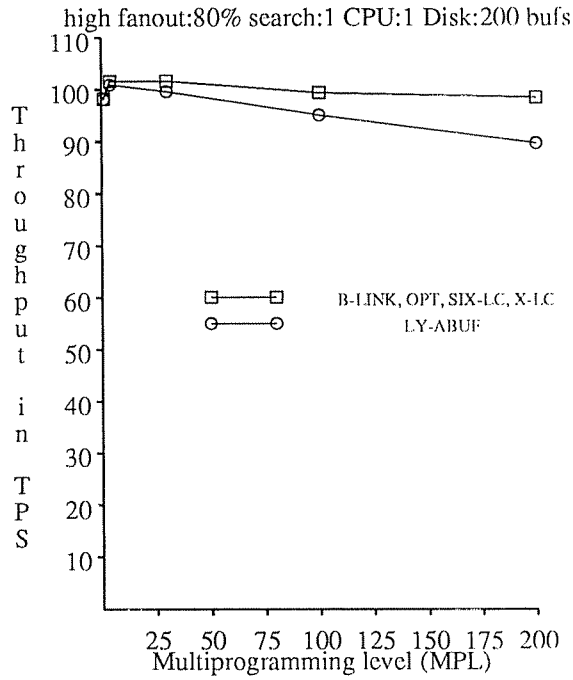

Figure 5: High fanout tree, single disk

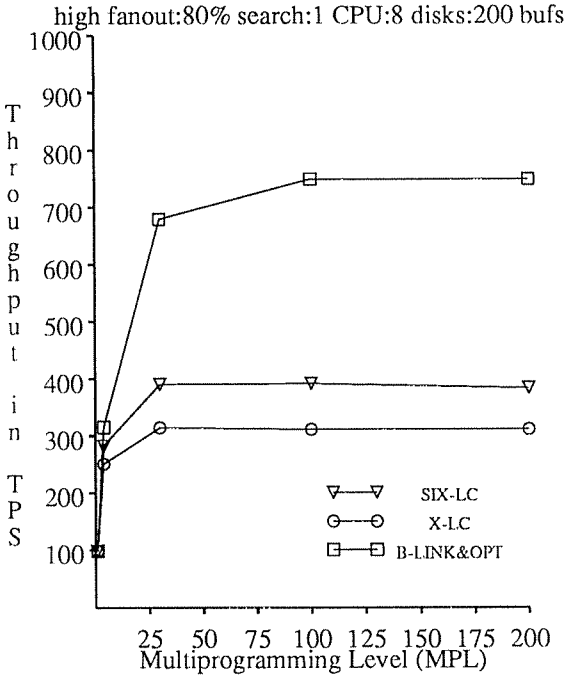

Figure 6: High fanout tree, 8 disks

With low data contention, the only performance difference visible in Figure 5 is that LY-ABUF performs worse than the other B-link algorithms at high MPLs. On further investigation, we found that while the number of I/Os performed by the other algorithms does not change much with the MPL, the number of I/Os in LY-ABUF increases. In fact, at an MPL of 200, LY-ABUF performs about 12% more I/Os than the other algorithms. This is due to the fact that the atomic read-write model used by LY-ABUF causes multiple copies of pages to exist in the buffer pool, while all other algorithms have just one copy of a page in the buffer pool. These extra copies cause the buffer pool to be used inefficiently in LY-ABUF, especially at higher MPLs, leading to more disk I/Os and extra waiting at the bottlenecked disk. Also, the LY-ABUF algorithm handles the cases when (i) a requested buffer page is not in memory and (ii) the page is in memory but is being modified by another transaction, in the same way; it must perform a disk I/O in both cases. In this experiment, case (ii) is more likely due to the size of the buffer pool being a sizeable fraction of the B-tree. Note that this phenomenon should not occur when the buffer pool is much smaller or much larger than the B-tree size, and we indeed found that there was essentially no performance difference between LY-ABUF and the other B-link algorithms in those cases. We will drop the LY-ABUF algorithm from future graphs since it never performs better than the other

two B-link algorithms.

The throughput curves for the case with 1 CPU and 8 disks is given in Figure 6. Due to the additional system resources, the throughput of all algorithms increases to a higher level before leveling off than in Figure 5. 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. It is evident from the response time curves (Figure 7) and the lock waiting times at the root (Figure 8) that the lock waiting time at the root is a significant fraction of the insert operation response time 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.

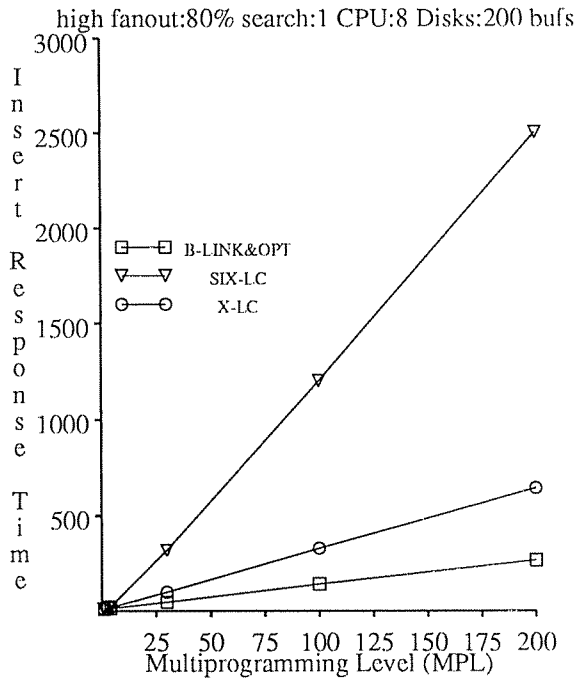

Figure 7: Insert response times (msec), 8 disks

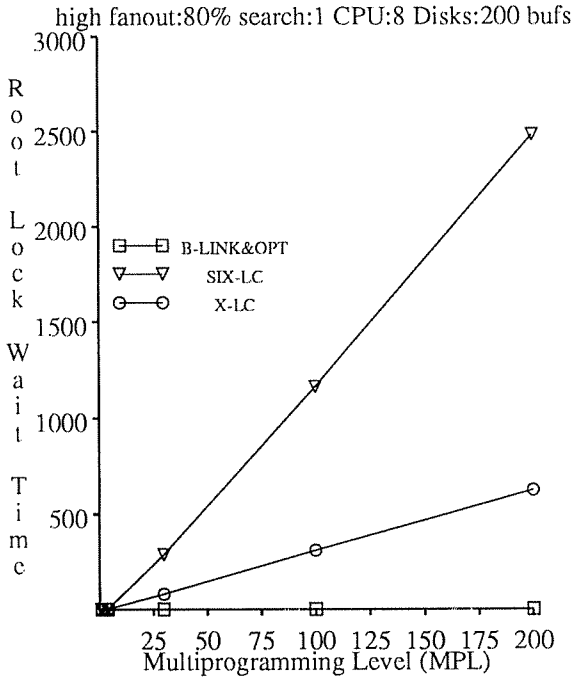

Figure 8: Insert waiting times (msec) at root, 8 disks

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 in front of 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 experiments 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 9 are very close to the insert response times in Figure 7. On the other hand, in the SIX-LC algorithms, the response time for searches increases only slightly with MPL (Figure 9), while the response time for updaters increases steeply with MPL (Figure 8). 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 6) 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 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 10. Note how the pessimistic algorithms level off at around the same maximum throughput as in the 1 CPU and 8 disks case (Figure 6), while the optimistic and B-link algorithms make excellent gains in throughput with increasing MPL. The reason the optimistic and B-link increases are slightly less than linear is due to contention at the buffer pool which results in increased buffer access times at higher MPLs.

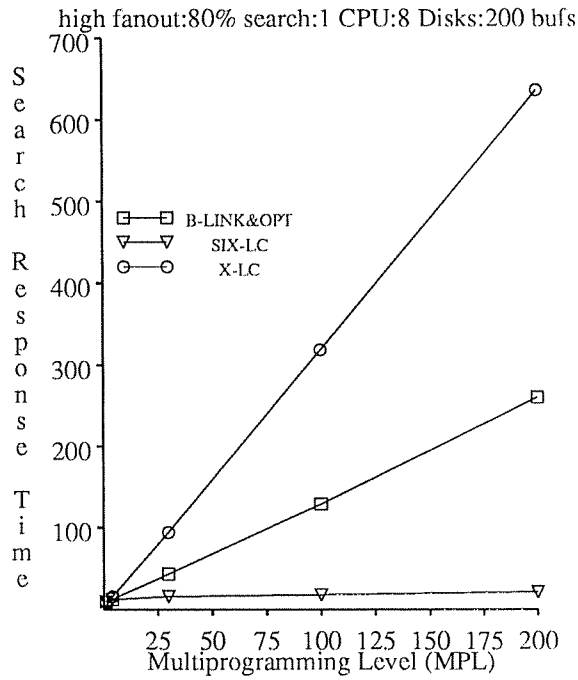

Figure 9: Search response times (msec), 8 disks

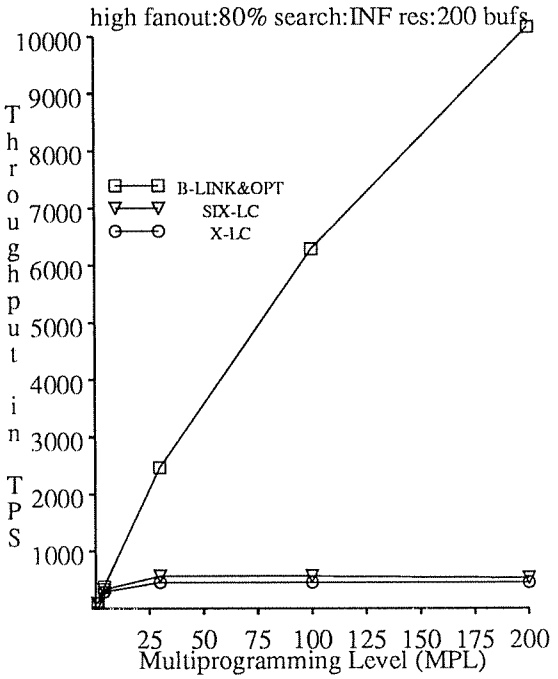

Figure 10: High fanout tree, infinite resources

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.

Notice 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).

4.1.2. Low Fanout Tree Experiments

Recall that the initial B-tree index in the low fanout case has 40,000 keys, 7,000 leaf pages and 1,500 non-leaf pages. The tree has 6 levels (7 in the top-down algorithms, as explained earlier). As in the high fanout experiments described above, the interesting cases are experiments in which the size of the buffer pool is less than the B-tree size. Here the buffer pool size is 600 pages, or about 7% of the B-tree size.

In the single CPU and disk case, there was little difference between the various algorithms, so we omit these results here. The throughput curves for the 1 CPU and 8 disks case is shown in Figure 11, where there is still not much difference in throughput between the various algorithms. In particular, the throughput of the pessimistic algorithms differs little from the throughput of the optimistic and B-link algorithms, unlike in the high fanout tree (Figure 6). This is because

the bottleneck at the root forms at higher MPLs here due to the fact that a lesser proportion of the response time is spent searching the root than in the high fanout case described earlier.

An interesting point to note is that the peak throughput for the top-down algorithms in Figure 11 is somewhat less than that of the corresponding Bayer-Schkolnick algorithms, eg., the peak throughput of TD-SIX is slightly lower than that of B-SIX in Figure 11. This is because early splitting in the top-down trees results in less occupancy for pages at the non-leaf levels. In the low fanout case, with more than a thousand non-leaf index nodes, this reduced occupancy causes a significant increase in the number of pages in the tree; the result is a reduced hit rate for index pages in the buffer pool. The reduced hit rate translates into more I/Os and, since the disk is a bottleneck in this experiment, the top-down algorithms perform slightly worse. The relatively small difference is due to the fact that the top-down buffer hit rates, though uniformly lower than those for the other algorithms, are still fairly close to the other hit rates since the leaf node hit rate dominates (the number of leaf nodes form approximately four-fifth of the total number of nodes).

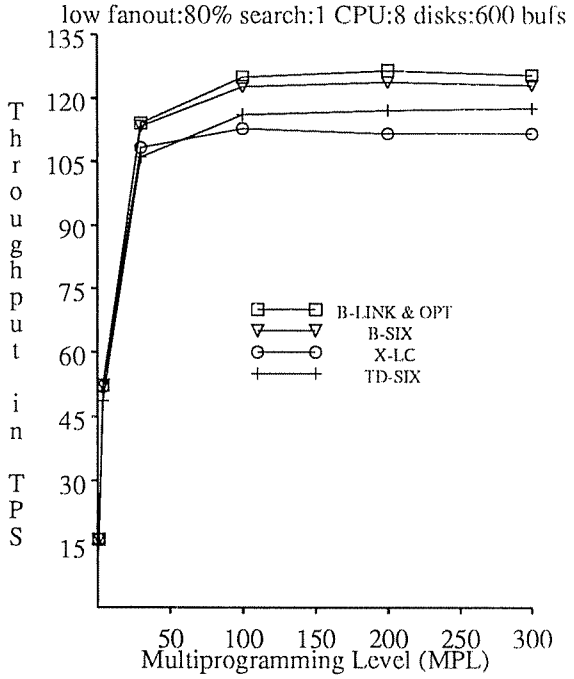

Figure 11: Low fanout tree, 8 disks

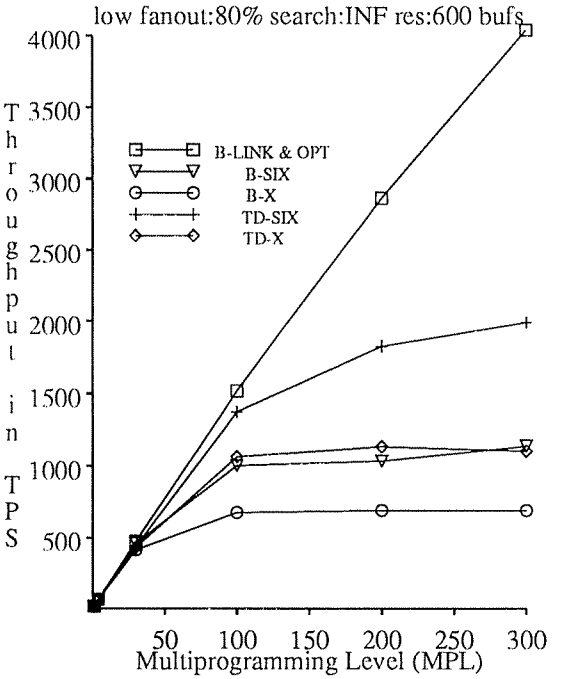

Figure 12: Low fanout tree, infinite resources

Figure 12 shows the throughput curves for the infinite resources, low fanout case with the 600 page buffer pool. Note that the optimistic and B-link algorithms perform much better than the pessimistic lock-coupling algorithms. Among the pessimistic lock-coupling algorithms, the top-down algorithms perform better than the corresponding Bayer-Schkolnick algorithms, i.e., TD-X is better than B-X and TD-SIX is better than B-SIX. These differences are due to the varying amounts of lock waiting at the root, as the lock waiting at all other levels is very small. The top-

down algorithms benefit from having to lock less of the scope in exclusive mode at any one time than the Bayer-Schkolnick algorithms, and therefore perform much better in Figure 12 in spite of a slightly lower hit rate for non-leaf index pages. As before, search operations are favored by the SIX-locking algorithms and since they are in the majority, the SIX-locking algorithms outperform the X-locking algorithms.

4.1.3. Summary

In a relatively low data contention situation, except for the single CPU and disk case, the optimistic and B-link algorithms performed much better than the pessimistic lock-coupling algorithms. Even in a system with relatively few resources, the throughputs of the optimistic and B-link algorithms were better than those of the pessimistic ones. This is because the pessimistic algorithms are unable to take advantage of the low data contention of the workload due to their exclusive locking at the root. Using a SIX policy that allows readers to overtake updaters does not alleviate this problem because even the few updaters that are present take a long time to complete; the overall throughput is therefore not increased much beyond the X-locking situation. The differences between the top-down and Bayer-Schkolnick algorithms were greater in the low fanout case, but the high fanout tree is much more likely to occur in practice. In future graphs, we will represent the performance of the pessimistic algorithms by that of the best lock-coupling algorithm, as they will be seen to perform much worse than the optimistic and B-link algorithms.

To see how a larger percentage of updaters affects the above results, we also experimented with a workload of 20% searches and 40% each of inserts and deletes. Since these results differ only in a few respects from the first set of experiments, we omit the graphs and summarize the key results. The pessimistic algorithms performed even worse for this workload than in the search-dominated workload. For all conditions except the low fanout, infinite resources case, the optimistic algorithms performed quite close to the B-link algorithms. In the low fanout case, under infinite resource conditions, the algorithms TD-OPT and B-OPT performed somewhat worse than OPT-DLOCK and the B-link algorithms. The reason is their relatively larger probability of restarts. We shall characterize this restart behavior in the next set of experiments.

4.2. Experiment Set 2: High Data Contention, Growing Tree

To further study the performance differences between the optimistic algorithms and the B-link algorithms, our next set of experiments uses a workload consisting only of inserts. This 100% insert workload 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.

4.2.1. High Fanout Tree Experiments

The results in this section are based on a subset of the experiments that were performed on high fanout trees. We first consider experiments that were conducted with a buffer pool size of 200 pages, or about 75% of the initial B-tree size. We omit the curves for the single CPU and

disk case, since there is again not much difference between the algorithms. Figure 13 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 13 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[†]. 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 14) 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 14). 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. Recall that 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.

In the infinite resources situation (Figure 15), OPT-DLOCK performs better at high MPLs than the other optimistic algorithms since its restarts become more inexpensive due to the increase

[†] 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.

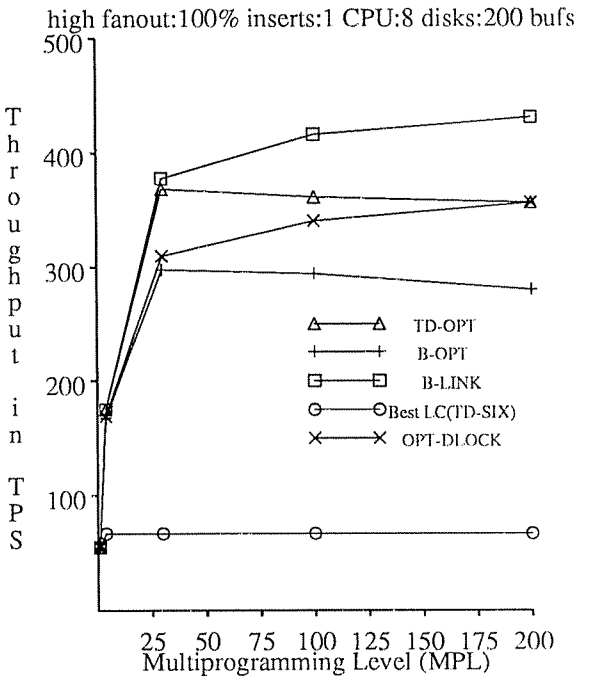

Figure 13: High fanout tree, 8 disks

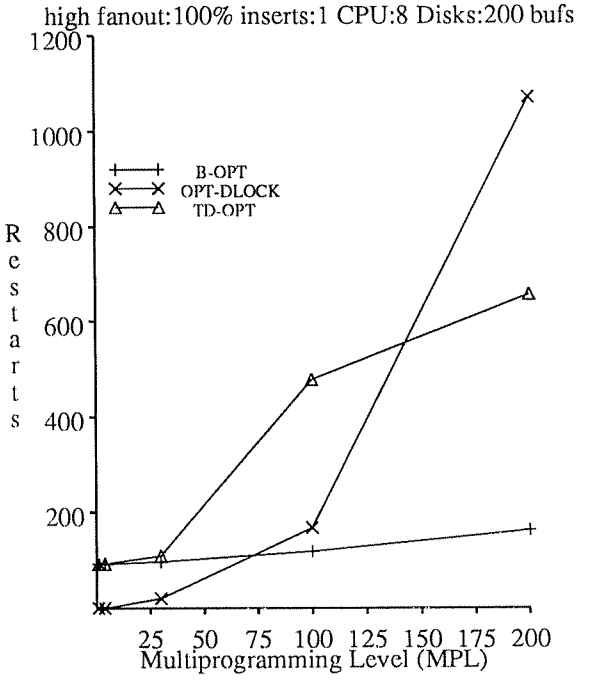

Figure 14: Restarts per 10,000 operations

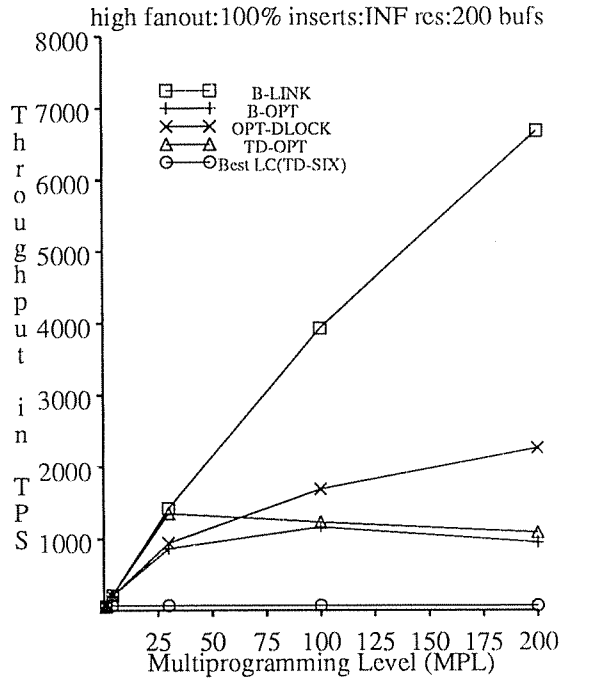

Figure 15: High fanout tree, infinite resources

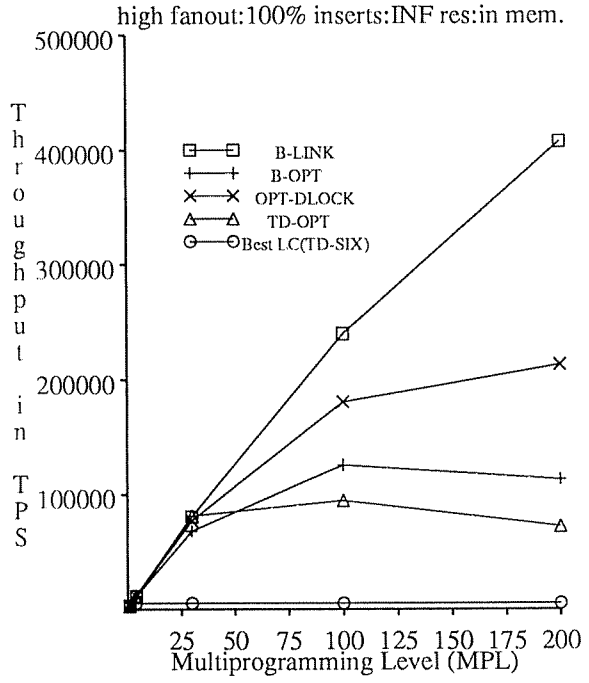

Figure 16: High fanout, in-memory tree

in the number of CPUs from 1 (in Figure 13) to infinity. As before, the B-link algorithms perform much better than all the other algorithms.

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 16. 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 essentially the same as in the earlier disk bound case (Figure 14).

4.2.2. Low Fanout Tree Experiments

The second subset of experiments using the 100% insert workload is performed on a low fanout tree. Just as in the low fanout experiments of experiment set 1, we first consider configurations in which the buffer pool contains 600 pages.

The 100% insert throughput curves with 1 CPU and 8 disks are given in Figure 17. As in the high fanout experiments discussed for this workload, TD-OPT and B-OPT perform worse than the B-link algorithms. However, the OPT-DLOCK algorithm performs close to the B-link algorithms at low MPLs and actually does slightly better at high MPLs. OPT-DLOCK is better than the other optimistic algorithms because the number of restarts in the case of OPT-DLOCK decreases drastically (to just 100 from the maximum of 1200 in Figure 14) due to a reduction in lock conflicts from the earlier high fanout case. On the other hand, due to the increased probability of finding a full leaf page, the restarts for B-OPT and TD-OPT increase to a maximum of around 1,600 from the much smaller values (100 and 600 respectively) found in Figure 14. The reason that OPT-DLOCK performs better than the B-link algorithms at high MPLs is that it is able to better utilize the buffer pool than B-link algorithms. Specifically, the B-link algorithms have to reacquire buffers for propagation of splits, while OPT-DLOCK always keeps them pinned. The B-link algorithms therefore perform more buffer calls, resulting in more I/Os at high MPLs.

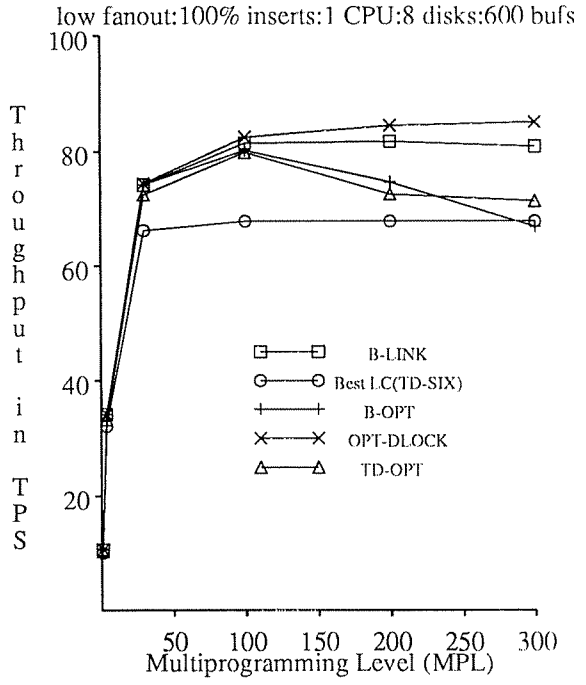

Figure 17: Low fanout tree, 8 disks

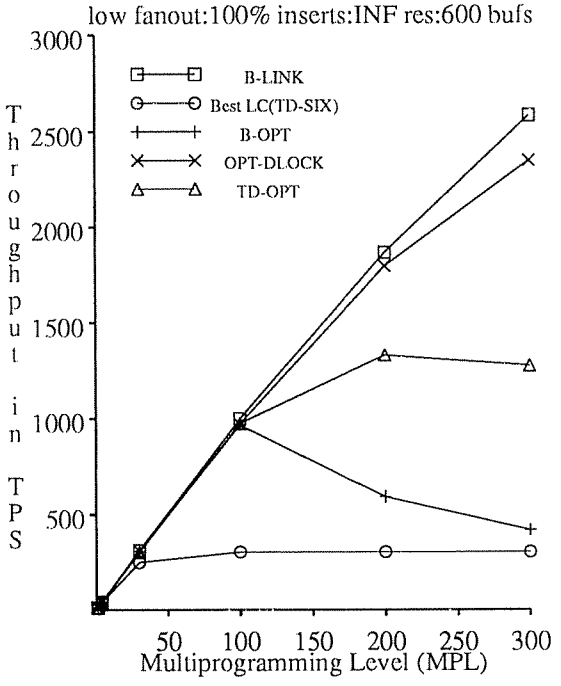

Figure 18: Low fanout tree, infinite resources

To further illustrate the above concepts, the results for the infinite resources case (Figure 18) shows that the B-link and OPT-DLOCK algorithms keep making gains in throughput, while the throughputs of the B-OPT and TD-OPT algorithms saturate at MPLs of 100 and 200 respectively. At even higher MPLs (not shown in the figure), both the TD-OPT and B-OPT algorithms end up performing close to (but slightly better than) their respective pessimistic versions. This is a surprising result, and the explanation is as follows. The probability that an operation will undergo a restart in the optimistic algorithms is the same at all MPLs, and is equal to the probability of finding a full leaf page. Therefore, at high MPLs, more restarters are active in their second (pessimistic) descent at the same time than at low MPLs. In B-OPT, as discussed earlier, restarts slow down operations in their first descent also, and this causes a bottleneck much like that of the X-LC case discussed in Section 4.1.1.

In the TD-OPT algorithm the loss in throughput is due to a slightly different reason. Recall that in TD-OPT, only operations in their second pass interfere with each other; operations in the first pass are allowed to overtake those in their second pass. Operations that succeed in their first descent at high MPLs execute much faster than operations which have to restart, much like searches and inserts behaved in the SIX-LC algorithms (Section 4.1.1). As a result, the system eventually becomes filled with updaters in their second pass, which causes a bottleneck so serious that the throughput starts to fall. We verified that this was indeed the case by looking at the standard deviation of the insert response time at low and high MPLs for TD-OPT. As expected, we found a significant difference between the standard deviations at low and high MPLs (the standard deviation varied from 50% of the value of the mean at low MPLs to 120% the value of the mean at high MPLs), indicating that the response time for restarted and non-restarted updaters vary widely at high MPLs. For B-OPT, there was no such variation in standard deviation, as expected. Both optimistic algorithms achieve a final throughput slightly greater than that of their pessimistic counterparts for exactly the same reason that the SIX-LC algorithms performed slightly better than the X-LC algorithms in the high fanout tree experiment with a search dominant workload (Figure 6); the average of the fast and slow response times for the optimistic algorithms is slightly less than the average for the pessimistic cases.

Just as the optimistic algorithms perform restarts, the B-link algorithms also involve extra overhead in high contention situations due to link-chases. The numbers of link-chases for the B-link algorithms in the 100% insert workload for high and low fanout trees are presented in Figure 19. (The figures for a particular tree do not differ significantly with the resource level.) Unlike the optimistic algorithms, where the number of restarts varied widely with the fanout of an index node, the link-chases are 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. It should also be noted that a link-chase in the B-link algorithms is much less expensive than a restart in the optimistic algorithms.

4.2.3. Summary

To summarize, in the experiments 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. Surprisingly, however, the OPT-DLOCK algorithm performed as well as the B-link algorithms in the low fanout tree but worse for trees with high fanout. Link-chases for the B-link algorithms were too infrequent in both the high and low fanout cases to affect their performance significantly.

So far, we have found the B-link algorithms to be consistently quite a bit better than the other algorithms in their ability to make use of the concurrency available in the workload and B-tree structure. The only exceptions are the restricted resource situations and situations with a low percentage of updaters, where there is little difference in performance between any of the algorithms. The B-link algorithms are therefore strong candidates for use in a practical system. Since there have been several variations proposed for B-link algorithms, it is interesting to see how these perform among themselves. We already determined that the algorithm LY-ABUF makes inefficient use of the buffer pool, and hence is not a suitable alternative. We are therefore interested in performance differences between the algorithms LY and LY-LC, which performed identically in both the first two sets of experiments. Our next set of experiments uses a workload that generates a large amount of localized contention so as to measure any performance differences between these two B-link algorithms.

4.3. Experiment Set 3: Extremely High Data Contention

In our third and final set of experiments, we use a workload that consists of 50% appends and 50% searches. Such a workload may arise, for example, given a history index to which appends are being made all the time while searches are being done to find old entries. 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. In situations with the buffer space being less than the tree size, the searches serve to keep the system disk bound. In such disk bound situations, only the searches perform I/Os, as the appends always find the pages that they need in the buffer pool. Thus, in this workload we have a situation where two operations with widely varying response times are interacting in the system.

In our graphs for the set of experiments in this section, we compare LY and LY-LC with the best optimistic algorithm (Best OPT) and the best pessimistic algorithm (Best LC). As usual, we divide the results into the high fanout and low fanout cases.

4.3.1. High Fanout Tree Experiments

The first subset of experiments are for the high fanout tree with a buffer pool size of 200

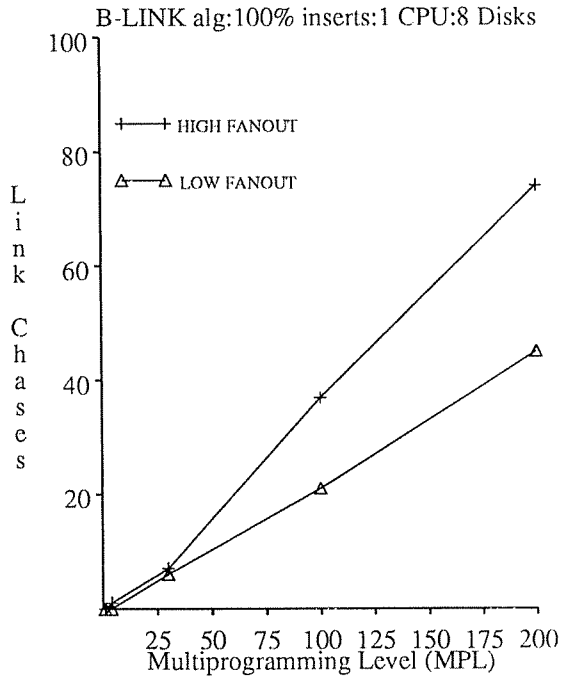

Figure 19: Link-chases (per 10,000 ops)

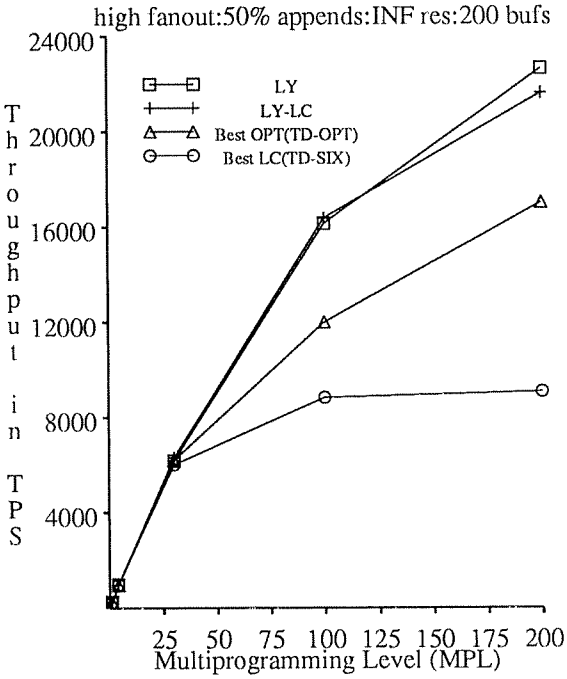

Figure 20: High fanout tree, infinite resources

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. We therefore omit the graphs for those experiments.

In the infinite resource situation (Figure 20), the B-link algorithms perform better than all of the other algorithms, and their throughputs continue to increase at high MPLs. Notice that there is still not much difference between the LC 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 21). The reason is that the very high contention between appends causes the number of link-chases to increase enormously at high MPLs for the B-link algorithms (Figure 22). 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.

We find from Figure 22 that the rate of *increase* of link-chases at high MPLs is about half of that at lower MPLs. Since searches do not interfere with appends, all link-chases are due to appends. Recall that a link-chase has a high probability of occurring when a split happens, and a reduction in the number of splits will reduce link-chases. The rate of increase of link-chases reduces at high MPLs due to a randomization of the actual sequence of the appends and

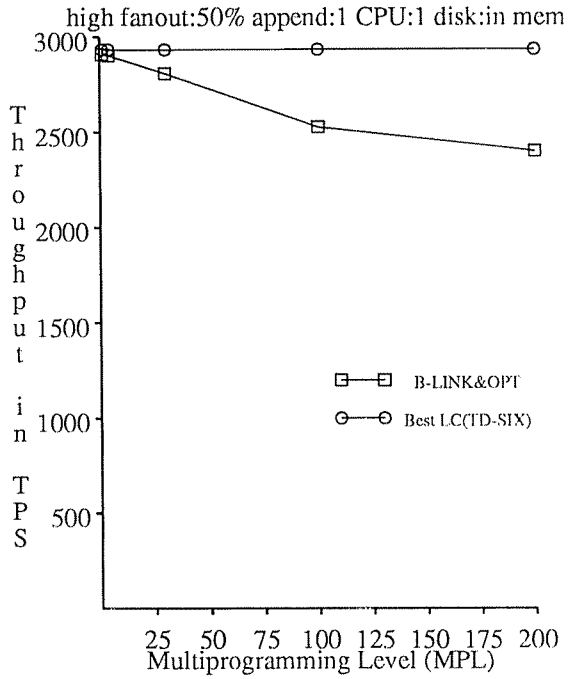

Figure 21: High fanout, in-memory tree

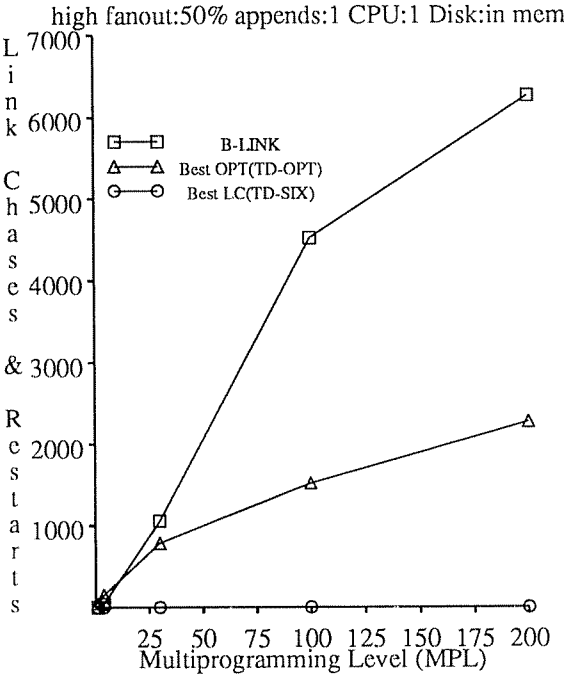

Figure 22: High contention overhead (per 10,000 ops)

a consequent reduction in the number of splits[†]. The reduction in the number of splits occurs for optimistic algorithms (TD-OPT and B-OPT) as well, but the optimistic algorithms are unable to take advantage of it because of the bottleneck that forms at the root. Unlike TD-OPT and B-OPT, which we found saturated at throughput values close to those of their pessimistic counterparts, OPT-DLOCK exhibited extreme thrashing behavior in the append experiments due to unbounded restarts.

4.3.2. Low Fanout Tree Experiments

The qualitative results for the low fanout tree are mostly similar to those of the high fanout case, so the graphs are not shown. We found no significant performance differences between the LC and LY-LC algorithms in situations where the buffer pool size was 600 pages (much less than the B-tree size). Even in heavily disk bound cases, link-chases did not significantly affect the hit rates of the buffer pool. This suggests that link chasing occurs almost immediately after a page is modified and is therefore very inexpensive.

In experiments with the entire tree in memory, we found that the LY-LC algorithm performed slightly worse than the LY algorithm at high MPLs (providing around 20% less throughput). The reason is the lock waiting introduced by holding S locks while searching for the parent of a leaf node. Still, both the LY and LY-LC algorithms performed an order of magnitude better than the rest of the algorithms.

[†] A strategy that causes the appends to be inserted in the increasing order of key values will attain a leaf page occupancy of 50% for new pages, while a randomization of the appends will cause the page occupancy to be around 69% [Yao78].

4.3.3. Summary

To summarize, even in this very high contention workload, the B-link algorithms increased in throughput except in low resource situations. For the B-link and optimistic algorithms, a large MPL randomizes an append workload and reduces the number of splits. The B-link algorithms effectively use the reduced number of splits to lower the rate of increase in link-chases at high MPLs. Moreover, link-chases are extremely cheap and cause no bottlenecks to form, unlike restarts that either cause a bottleneck at the root (TD-OPT and B-OPT) or thrashing behavior (OPT-DLOCK). There was no significant difference between the LC and LY-LC algorithms across a wide range of resource conditions and tree structures; the exception was a system configuration with a low fanout tree, infinite resources, and a buffer pool as large as the tree, where the LY-LC algorithm performed slightly worse than the LY algorithm at high MPLs. This sort of situation can likely be ruled out in practice, and even if it occurs, the LY-LC algorithm performed reasonably close to the LY algorithm and much better than all non-B-link algorithms.

5. Discussion of Performance Results

We can broadly summarize the results of the previous section into the following points.

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 even the small amount of parallelism available in systems with a few CPUs and disks.
3) Optimistic algorithms that restart upon encountering a full leaf node attain only a limited amount of performance improvement over their pessimistic counterparts. In fact, the algorithms TD-OPT and B-OPT perform close to their corresponding pessimistic algorithms at high MPLs. Since the probability of a restart is fixed by the tree structure, the number of restarts interfering with each other increases at higher MPLs, irrespective of whether there is any actual data contention or not. Therefore, these algorithms cannot adapt to dynamically changing workload conditions. A naive algorithm like OPT-DLOCK, in which restarts are based on actual lock conflicts, is much better than algorithms that restart based on leaf node occupancy in most situations.
4) 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. For example, in the append experiments, the overhead for link-chases in the B-link algorithms was comparable to that of restarts in the optimistic algorithms. However, the B-link algorithms continued to increase in throughput, given enough resources, while the optimistic algorithms saturated at a throughput level close to that of their pessimistic counterparts. This is because the restarts in the optimistic algorithms caused a bottleneck at the root (TD-OPT and B-

OPT) or at the level above the leaf (OPT-DLOCK), while the link-chases of the B-link algorithms were spread over many leaf nodes and did not cripple performance.

5) The fanout of index pages can greatly affect performance, as one would expect. For example, the performance of the OPT-DLOCK algorithm relative to the B-link algorithms in high and low fanout situations varies quite widely (Figures 15 and 18).

The only algorithms that perform consistently well are the B-link algorithms. In all workloads tried so far, the B-link algorithms can make use of extra resources and increase their throughput. They treat searches and inserts symmetrically, unlike (for example) the SIX algorithms, which speed up search response times at a heavy cost to inserts. Moreover, the overhead in the B-link algorithms in terms of page splits can actually reduce at high MPLs for a high contention situation like our append experiments. A final important observation is that the B-link algorithms perform the best in both high and low fanout trees. This leads us to conclude that B-link algorithms will also perform well for trees that have variable length keys, in which the fanout may vary widely from node to node.

In addition to the above experiments, which used a FCFS scheduling algorithm at the disk, we also performed experiments in which disk requests were scheduled using an elevator algorithm. The elevator algorithm differs from FCFS disk scheduling by significantly reducing the average seek time for disk requests when there are many concurrent requests. We found that this change in seek times affected the results of the disk bound experiments quantitatively but not qualitatively. The reason is as follows. At higher MPLs, algorithms that do not bottleneck at the root are able to queue multiple requests at the disk at the same time, thus attaining better throughput than in the FCFS case due to the reduction in response time caused by faster seeks. Algorithms that do bottleneck at the root allow much less concurrent operation and hence their disk seek times do not change much from the FCFS case; they attain the same throughput as before. Thus, the differences in throughput between the bottlenecked algorithms and the other algorithms increases, but the qualitative results are the same as before. For example, in a low contention workload on the high fanout tree with a system configuration of 200 buffers, 1 CPU, and 8 disks using the elevator scheduling algorithm, we found that while the throughput of the pessimistic algorithms was close to that in the corresponding FCFS case (Figure 6), the optimistic and B-link algorithms reached a 70% higher peak throughput than in Figure 6.

Based on the above observations and the results of the previous section, we can also comment on the performance of algorithms that have not been explicitly simulated in our system relative to the performance of B-link algorithms.

5.1. Side-Branching Technique

The *side-branching* solution proposed in [Kwon82] is a modification to the SIX locking Bayer-Schkolnick algorithm, B-SIX. Updaters in this algorithm perform the allocation of new pages and the copying of keys from old pages to new ones while holding SIX locks on the scope of the update. After making changes on the side, the updaters then make a quick pass down the scope

with X locks and patch up the nodes. This strategy aims to minimize the time during which X locks are held on nodes in order to speed up searches more than the B-SIX algorithm. However, in our low data contention experiments we found that there are so few updaters that are not waiting at the root at high MPLs, that searches basically have a free run of the B-tree. The side-branching technique, which endeavors to minimize interference between updaters and searches, is thus unlikely to have much of an effect on performance since there is already little interference (as evidenced by the almost constant search response time in Figure 9) due to updaters experiencing a bottleneck at the root.

5.2. The mU Protocol

An interesting algorithm called the mU protocol was proposed in [Bili87]. An important feature of this algorithm is that the compatibility graph for locks on a node is dependent on the occupancy of the node. The algorithm uses special insert and delete lock nodes (distinct from S and X locks) to reserve slots in a node for later insertions or deletions. (The exclusive lock mode used by pessimistic algorithms can be thought of as locking all slots.) The maximum number of insert locks that can be held on a node at any one time is equal to the number of empty key and pointer slots in the node; similarly, the maximum number of delete locks on a node is equal to the occupancy of the node. Insert and delete lock modes are incompatible with each other. This algorithm not only uses high keys and right links like the B-link algorithms, but it also uses a low key and a left link at all nodes.

There are workloads and system conditions under which this algorithm is likely to perform as well as the B-link algorithms, but there are others where it will surely perform worse. The mU algorithm is sensitive to the occupancy of the index pages, in particular that of the root page, since the number of simultaneous updaters that can be reading a page is limited by the number of empty or full slots in the node. The mU algorithm is therefore likely to perform badly for trees with mostly full or empty nodes as well as for trees with a small number of keys per page. We have seen in our experiments that the probability of finding full pages in a low fanout tree can be quite high, and in such cases the mU protocol will perform worse compared to the B-link algorithms.

Apart from the above problem with low fanout trees, inserts and deletes interfere with each other at the root in the mU protocol since their respective lock modes are incompatible. In a workload with an equal proportion of inserts and deletes, we can therefore expect this interference to cause a loss of throughput at sufficiently high MPLs due to a bottleneck forming at the root, much like in the TD-OPT and B-OPT algorithms.

5.3. ARIES/IM Algorithm

A detailed algorithm for high-concurrency index management was described in [Moha89]. We will not describe the details of this algorithm, except to state that it does not suffer from any of the drawbacks that we listed at the beginning of this section.

ARIES/IM has both left and right pointers linking nodes at the leaf level, but unlike the

B-link algorithm, nodes at higher levels do not have right links. Updaters in ARIES/IM make an initial descent to the leaf using S locks, and at the leaf level they may perform link-chases just as in the B-link algorithms. However, while the B-link algorithms perform link-chases at all levels, updaters in ARIES/IM instead use a complex protocol based on recursive restarts. These restarts do not have the disadvantage of creating any bottlenecks, however, as the operations use extra information stored in the B-tree nodes to ensure consistency rather than using exclusive locks. In our experiments, we observed that most of the link-chases in the B-link algorithms were at the leaf level (100% in the high fanout case and over 90% in the low fanout case). Since the leaf level ARIES/IM algorithm resembles the B-link algorithm in many respects, we can expect the ARIES/IM algorithm to perform close to the LY algorithm for most workloads (including the append workload).

An important difference between the ARIES/IM algorithm and the other algorithms discussed so far is that ARIES/IM allows only one page split (or merge) at a time. To find out exactly what impact this has on performance, we modified the LY algorithm to limit itself to one page split at a time (using a tree latch) and ran experiments with the 100% insert workload in infinite resource conditions. In trees with high fanout, we found that the modified algorithm performed slightly worse than the B-link algorithms; the throughput of the modified algorithm performed in between the B-link and OPT-DLOCK algorithms in Figure 15 and, at an MPL of 200, the throughput of the modified algorithm was around 25% less than that of the B-link algorithm. The waiting time for the exclusive page split lock contributed to the increase in response time, leading to a loss in throughput. In trees with low fanout, however, the modified algorithm performed much worse when compared to the B-link algorithms. In fact, it performed even worse than the optimistic algorithms in Figure 18. The cost of waiting for the tree latch is much higher in the low fanout case due to an increased number of splits.

A modification to the ARIES/IM algorithm to handle more than one page split at a time is suggested in [Moha89] and, if implemented, this should allow ARIES/IM to perform closer to the B-link algorithm. It should be noted, however, that we have looked at ARIES/IM only from the concurrency perspective of single B-tree operations; the ARIES/IM algorithm also describes how to hold extended locks on records (to allow serializability of transactions that perform more than one B-tree operation) as well as how to perform recovery using write-ahead logging.

6. Comparison With Related Work

An approximate analysis of the Bayer-Schkolnick algorithms was included in [Baye77]. The formulas provided calculate quantities like the number of locks held by operations and the maximum MPL that can be handled without creating a bottleneck at the root. This is a static analysis and does not provide insight into the dynamic performance of the various algorithms.

Biliris [Bili85] described a model for the evaluation of B-tree algorithms and a set of experiments comparing four algorithms that included the Samadi algorithm [Sama76], the B-SIX algorithm, the side-branching algorithm [Kwon82], and the mU algorithm [Bili87]. This study

found that the pessimistic algorithms bottleneck at the root and that the mU algorithm performed better in the situations considered. The side-branching technique was found not to give any improvement over B-SIX. The main shortcomings of this study are that the optimistic and B-link algorithms were not studied, response time for individual operations was not given, and no detailed analysis of the results was provided.

The most recent performance analysis of B-tree concurrency control algorithms was based on analytic modeling of an open queueing system [John90]. This study assumed infinite resource conditions and did not model buffer management. The algorithms compared in the study are a naive lock-coupling algorithm (B-EX), an optimistic algorithm (B-OPT modified with the second phase using X locks instead of SIX locks), and the LY version of the B-link algorithm. The key results of this study are that the root will become 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 B-link algorithms. We have also analyzed differences between several variants of the B-link algorithm. Some of our results differ from theirs; for example, they found that the optimistic descent algorithm always performed much better than the naive lock-coupling algorithm, while we found that the optimistic algorithms sometimes perform close to their corresponding pessimistic versions at high MPLs (eg., in a 100% insert workload on a low fanout tree), though they indeed provide much better performance at intermediate MPLs. Finally, their model allows only S and X locks, while we have considered more complicated algorithms that use SIX locks to enable certain tree operations to overtake others on their descent to the leaf.

7. Conclusions and Future Work

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. The reason for the excellent performance of the B-link algorithms is the absence of any bottleneck formation (except, of course, at the CPUs or disks in resource-constrained situations). In contrast, in all of the other algorithms, locking bottlenecks form at high MPLs if the workload contains a significant percentage of updaters. Moreover, the overhead that the B-link algorithms incurs in very high data contention situations are link-chases, which turn out to be inexpensive. We also found interesting differences in the behavior of the optimistic and pessimistic algorithms among themselves.

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 (inconsistent) situations encountered in the latter. Before using B-link algorithms in real systems, however, recovery strategies have to be designed for them, and a way has to be found to handle variable length keys. A solution for either has not yet been published in the literature.

In this study we have considered only one aspect of concurrency control on B-tree indices, namely, transactions that perform single B-tree operations. When B-tree operations are part of larger transactions, there are several strategies available for holding long term locks for recovery purposes. For example, a transaction could hold an extended lock on the index leaf page, on just one slot in the leaf page, or on just a record id or a key value. Also, these lock holding strategies are closely linked to recovery strategies; few comprehensive recovery strategies have been proposed for B-trees. Furthermore, special types of queries such as range scans may interact differently with long term lock holding strategies and recovery strategies than single operation transactions would. Finally, another facet of B-tree concurrency control and recovery involves the problem of building a B-tree index on a relation while the relation is being concurrently accessed by other transactions. The above are all candidates for future work.

Acknowledgements

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

We would like to thank Miron Livny for his great help in using the DeNet simulation environment. We would like to thank both David Dewitt and Miron Livny for suggestions that helped us to improve the presentation of the results. We would also like to thank Rajesh Mansharamani, Manolis Tsangaris, and S. Seshadri 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 Trans. on Database Sys.*, **12, 4** Dec. 1987.

[Baye72] Bayer, R. and McCreight, E.M. "Organization and Maintainance of Large Ordered Indices" *Acta Informatica*, **1(3)** 173–189 (1972).

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

[Bern81] Bernstein, P., and Goodman, N. "Concurrency Control in Distributed Database Systems" *ACM Computing Surveys*, **13(2)** June 1981.

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

[Bili87] Biliris, A. "Operation Specific Locking in B-trees" *Proceedings of the Sixth ACM Symposium on PODS* , San Diego, California, 159–169. March 1987.

[Care84a] Carey, M., and Stonebraker, M. "The Performance of Concurrency Control Algorithms for Database Management Systems" *Proceedings of the Tenth VLDB Conference*, Singapore, August 1984.

[Care84b] Carey, M., and Thompson, C. "An Efficient Implementation of Search Trees on $\lceil \lg N + 1 \rceil$ Processors" *IEEE Transactions on Computer Systems*, **11(2)** November 1984.

[Chen84] Cheng, J.M., Loosley, C.R., Shibamiya, A. and Worthington, P.S. "IBM Database 2 performance: Design, implementation, and Tuning" *IBM Systems Journal,*, **23(2)** 189–210 (1984).

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

[Elli80a] Ellis, C. "Concurrent Search and Insertion in 2-3 Trees" *Acta Informatica*, **14(1)** (1980).

[Elli80b] Ellis, C. "Concurrent Search and Insertion in AVL Trees" *IEEE Transactions on Computers*, **C-29(9)** September 1980.

[Elli83] Ellis, C. "Extendible Hashing for Concurrent Operations and Distributed Data" *Proceedings of the 2nd ACM Symposium on PODS*, Atlanta, Georgia, March 1983.

[Fran83] Franaszek, P., and Robinson, J. "Limitations of Concurrency in Transaction Processing" *IBM T.J. Watson Research Center*, **RC10151** August 1983.

[Good85] Goodman, N., and Shasha, D. "Semantically-based Concurrency Control for Search Structures" *Proceedings of the 4th ACM Symposium 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" *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, (1978).

[Haer83] Haerder, T., and Reuter, A. "Principles of Transaction-Oriented Database Recovery" *ACM Computing Surveys*, **15(4)** December 1983.

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

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

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

[Kung80] Kung, H., and Lehman, P. "A Concurrent Database Manipulation Problem: Binary Search Trees" *ACM Transactions on Database Systems*, **5(3)** September 1980.

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

[Lani86] Lanin, V. and Shasha, D. "A Symmetric Concurrent B-tree Algorithm" *Proceedings of the Fall Joint Computer Conference* , 380–389. (1986)

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

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

[Mill78] Miller, R., and Snyder, L. "Multiple Access to B-trees" *Proceedings of the Conference on Information Science and Systems*, Johns HopkinsUniversity, 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).

[Moha90] Mohan, C. "ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-tree Indexes" *Proc. of VLDB Conf.* , 392–405 (1990).

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

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

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

[Shas84] Shasha, D. "Concurrent Algorithms for Search Structures" *Ph.D. Thesis, Aiken Computation Laboratory*, Harvard University, June 1984.

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

[Tay84] Tay, Y. "A Mean Value Performance Model For Locking in Databases" *Ph.D. Thesis, Computer Science Department, Harvard University*, February 1984.

[Verh78] Verhofstad, J. "Recovery Techniques for Database Systems" *ACM Computing Surveys*, **10(2)** June 1978.

[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).