A Framework for the Performance Analysis of Concurrent B-tree Algorithms

Theodore Johnson, johnsnth@csd2.nyu.edu
Dennis Shasha, shasha@cs.nyu.edu
Courant Institute of Mathematical Sciences, New York University *

Abstract

Many concurrent B-tree algorithms have been proposed, but they have not yet been satisfactorily analyzed. When transaction processing systems require high levels of concurrency, a restrictive serialization technique on the B-tree index can cause a bottleneck. In this paper, we present a framework for constructing analytical performance models of concurrent B-tree algorithms. The models can predict the response time and maximum throughput. We analyze three algorithms: Naive Lock-coupling, Optimistic Descent, and the Lehman-Yao algorithm. The analyses are validated by simulations of the algorithms on actual B-trees. Simple and instructive rules of thumb for predicting performance are also derived. We apply the analyses to determine the effect of database recovery on B-tree concurrency.

1 Introduction

Concurrency control is the activity of preventing harmful interference between asynchronous concurrent processes in a database management system. Concurrent search structures, especially concurrent B-trees, are most often used in database systems. Applications such as airlines, telecommunications, banks, and real-time databases require 1000 or more transactions per second; a transaction consists of 4-6 record accesses, most of them through indices [6]. If each transaction requires .1 seconds, the multiprocessing level will average around 100. At such high multiprocessing levels, restrictive serialization techniques result in a serialization bottleneck.

Many concurrent B-tree algorithms have been proposed ([2,5,14,16,18,23]). These algorithms have not been satisfactorily analyzed, however. Bayer and Schkolnick [2] and Ellis [5] derive formulae for the number of locks held and the maximum number of concurrent operations, respectively, for their algorithms. These analyses don't predict performance, though. Ford et. al ([20,21]) compares the different algorithms based on the number of locks that the algorithms place, and calculates performance for a bus-based architecture. This analysis does not take into account the probability that the locks conflict or the time that the locks are held, and is very dependent on the underlying computer architecture. Shasha et. al. [4] give architecture-independent formulae for calculating upper and lower bounds on the performance of various B-tree algorithms. No deletes are allowed in the operation mix, however, and the analysis depends on crude assumptions about the flow of operations in the B-tree. Carey and Cheng [3] discuss a simulation model of concurrent B-trees.

In this paper, we will discuss the framework of our analytical model. Using the framework and the analytical tools that we have developed, we can analyze concurrent B-tree algorithms by a uniform method.

2 The Concurrent B-tree Algorithms

The algorithms that we will consider are *Naive Lock-coupling*, *Optimistic Descent* and *Link-type*. The first two algorithms are described in [2]. The Link-type algorithm was first proposed in [16], then extended

*This work was partially supported by the National Science Foundation under grant number IRI-89-1699 and by the Office of Naval Research under grant number N00014-85-K-0046.

in [15,23].

The simplest B-tree concurrency control mechanism is to have each operation lock the entire tree. However, most update operations (i.e. inserts and deletes) change only a small portion of the tree. The *Naive Lock-coupling* algorithm ([2,17]) is a first step towards exploiting this fact. Each node must be locked before it is accessed. Lock-coupling requires that the child of a node must be locked before the lock on the parent may be released. Search operations read lock (R lock) the root, search the root to choose the correct child, then R lock the child. After obtaining the R lock on the child, the lock on the parent is released. Insert and delete operations follow the same protocol, except that write locks (W locks) are used, and all ancestor locks are released if and only if the child node is safe for the operation. For example, insert operations only release the ancestor locks if the child is not full. When the leaf node is locked, all nodes that will be involved in restructuring are locked, so that the operation may restructure without interfering with the correct execution of other operations.

The Naive Lock-coupling algorithm can be improved on by observing that a B-tree rarely restructures. In the *Optimistic Descent* algorithm [2], the insert and delete operations place R locks down to the leaf, then W lock the leaf using lock-coupling. If the leaf is unsafe, the operation releases all of its locks and starts over, placing all W locks on the second pass.

The *Link-type* algorithm ([16,23]) uses links to remove the need for lock-coupling. Every node (except for the rightmost) is linked to its right neighbor. Operations place R locks to the leaf, and insert and delete operations place W locks on the leaves. At most one lock is held at a time. If a node becomes too full after an insert, half of the keys in the node are transferred to a new sibling, the sibling is linked with its neighbors, the node lock is released, the parent node is locked, and a pointer to the new sibling is inserted into the parent. If the parent becomes too full, the process continues. If a node becomes lost navigating the B-tree, it can use the links to find the proper node. (In a merge-at-empty B-tree, merges are infrequent, provided inserts outnumber deletes, so we ignore them [10]).

3 The Analytical Framework

We model a concurrent B-tree as an open system of queues. Our model makes several assumptions and approximations that are justified by the model's close agreement with simulation. The queues represent the lock queues in which the operations must wait to obtain the required node locks. Each queue models a representative node on a different level of the tree (see Figure 1).

3.1 Performance Measures

To determine the performance of the various concurrent B-tree algorithms, we need performance measures. One obvious performance measure is the response time given by the algorithm. The response time of an operation is the time from the beginning to the end of an operation. An operation's response time has two components: the time spent working and the time spent waiting. The time spent working is assumed to be constant, for a given algorithm, while the time spent waiting will increase with increasing concurrency, and is the focus of study.

Unlike the analyses in [2,5,4], this analysis models the concurrent B-tree as being an open system. In an open system, the throughput is not an interesting measure: if all of the queues are stable, the throughput is equal to the arrival rate. Instead, we will calculate the *maximum throughput*, which is the maximum possible sustainable arrival rate. The maximum throughput and the response time are complementary performance measures. An algorithm with a higher maximum throughput usually has lower response times, but not always.

3.2 Assumptions

Arrival rate: The arrival rate into the queue that represents the root is proportional to the arrival rate of all operations (the rates are not necessarily equal because, depending on the algorithm, an operation might lock the root twice). The arrival rate into a lower queue is the arrival rate into the predecessor queue divided by the fanout on the predecessor's level. We assume that the root's arrival rate can be described by a Poisson process, and that the arrival rates in the succeeding queues can also be described by a Poisson process. This is a strong assumption, but not an uncommon one ([11,13]).

Service time: The service time in each queue is the time that a lock is held, and depends on the algorithm. Lock coupling is modeled by defining the service time to include the time to access the node, the time spent waiting to lock the next level, and all further time that the lock must be held if the child is unsafe. This modeling assumption, combined with the previous assumption, removes the explicit

stochastic dependence between queues. We assume that the access times can be modeled by exponential distributions, and that the lock service times can be modeled by hyperexponential distributions (another strong assumption).

Resource contention: We assume that resource contention can be described by a service-time dilation factor. This assumption allows us to separate the effects of resource contention and data contention (as, for example, in [22,26]), which in turn allows us to create a much simpler model than otherwise. We do not feel that this assumption is a strong assumption. While many of the algorithms require the performance of some extra work, the additional work is small even for the maximum sustainable throughput.

Steady state: The analysis calculates the performance of an algorithm on a B-tree of a particular size. On the other hand, our restructuring parameters model the case when inserts outnumber deletes. Blocking decreases when the root's fanout increases, and response times increase when the B-tree's height increases. The analysis assumes that the queues reach a steady state, which implies that the B-tree doesn't grow too quickly. This is usually a safe assumption, and is justified by comparison to the simulation results.

Lock types: Different types of operations place different types of locks. In Naive Lock-coupling, for example, search operations place R locks and insert operations place W locks. An operation may place different types of locks during different phases of the operation (for example, insert operations in the optimistic descent algorithm). We assume that R locks may be shared, but that W locks are exclusive (may not be shared with either R or W locks). We also assume that locks are granted in FCFS (First-Come, First Serve) order. We analyzed these types of lock queues in [8]. The assumptions made to analyze those queues must be made to analyze the concurrent B-tree algorithms. The primary assumptions are Poisson arrivals and exponential service times. The results of [8] are discussed in the Appendix.

B-trees: Bayer and McCreight proposed a B-tree where keys are stored at all levels in the tree [1]. Wedekind pointed out that the B^+-tree, where all keys are stored in the leaves, is more appropriate for database applications [25]. All of the data structures studied in this paper are B^+-tree, and the term B-tree will be an abbreviation for the term B^+-tree.

The B-tree described by Wedekind restructures a node if it becomes less than half full [25]. We call this strategy *merge-at-half*. Most B-trees implemented in practice never restructure nodes due to underflow conditions. We call this strategy *merge-at-empty*. Johnson and Shasha compare the space utilization and restructuring rates of merge-at-half and merge-at-empty B-trees [9]. They find that merge-at empty B-trees have a significantly lower restructuring rate and a slightly lower space utilization, if there are more inserts than deletes in the instruction mix. Merge-at-empty is more appropriate than merge-at-half for concurrent B-tree algorithms. All algorithms studied in this paper use merge-at-empty B-trees.

3.3 Framework

The general analytical framework is the following: Classify operations based on whether they place R or W locks. Calculate the arrival rates and service times for the R and W locks in each queue. Calculate the expected waiting times for the R and W locks in each queue. The algorithm on the B-tree can handle the arrival rate if all of the queues are stable. The response time of an operation is the time the operation would take alone plus the time spent waiting for locks.

4 The Simulator

We wrote a concurrent B-tree simulator and performed experiments. The simulator first builds a B-tree out of a sequence of insert and delete operations. Next, a sequence of concurrent B-tree operations is performed. The simulator uses the concurrent B-tree algorithm being studied to perform the concurrent operations. The proportion of search, insert and delete operations is a parameter. The proportion of insert to delete operations in the construction phase is the same as the proportion in the concurrent operation phase.

The concurrent operations arrive in a Poisson process and then perform their action on the B-tree. All service times have exponential distributions. The only upper limit on the number of concurrent operations is the amount of space allocated for them. If the simulator tries to create more concurrent operations than there is space for, the simulator crashes. If a simulator crashes with a particular set of parameters, the space allocated for concurrent operations is increased until the simulator doesn't crash on any run, if possible. Otherwise, none of the simulator runs are reported.

The simulator collects a variety of statistics, including the operation response times and the lock waiting times. The simulator might also collect some algorithm-specific statistics.

5 Analysis of Naive Lock-Coupling

We model the concurrent B-tree as a sequence of jobs that pass through a series of FCFS R/W queues. Each job represents one of the operations and each queue represents a level in the B-tree. Since insert and delete operations always place write locks in this algorithm, we model them as W jobs. Search operations always place read locks, so we model them as R jobs. The B-tree has h levels: the leaves are at level 1 and the root is at level h. The expected waiting time of an operation is the sum of the expected waiting times in all of the queues. The expected response time of an operation is the expected waiting time plus the expected service time for the operation. The expected service time is a direct calculation. In order to find the expected waiting time, we need to determine the arrival rate and the expected time that an operation holds a lock at each level in the B-tree and use these parameters on the FCFS R/W queue.

In the following list of parameters, the cost of accessing a node is the same as if the operations were to access the B-tree serially.

Parameters

$Se(i)$: The expected time needed to search an i level node.

M: The expected time needed to modify a (leaf level) node.

$Sp(i)$: The expected time needed to split an i level node (including the cost of modifying the parent node).

$Mg(i)$: The expected time needed to merge (delete) an i level node

$\Pr[F(i)]$: Probability that a level i node is insert-unsafe (full).

$\Pr[Em(i)]$: Probability that a level i node is delete-unsafe (empty).

$E(i)$: Expected number of children of level i node.

λ: Arrival rate of all operations.

q_s, q_i, q_d: Probability that an operation is a search, insert or delete operation, respectively. $q_s + q_i + q_d = 1$.

Variables

$T(o, i)$: Expected time that operation o holds a lock on a node at level i.

$\mu_{L,i}$ Service rate of lock type L at level i.

$\rho_w(i)$: Probability that a W lock is in the level i lock queue.

$r_e(i)$: Expected time that a W lock, L, must wait for a preceding R locks in a level i queue if the queue had no other W locks when L arrived.

$r_u(i)$: Expected time that a W lock, L, must wait for a preceding R locks in a level i queue if the queue had at least one other W lock when L arrived.

$R(i)$: Expected time to obtain a R lock on a level i node.

$W(i)$: Expected time to obtain a W lock on a level i node.

$Per(o)$: Expected response time of operation o (total time in the system).

Let us now analyze the expected time that an operation holds a lock on a node if some other operation might be waiting to lock the node. (If we are going to analyze the waiting time in a lock queue, these are the only service times that count.) Because the protocol for the concurrent operations uses lock-coupling, the waiting time on the ith level depends on the time to place a lock on an $i-1^{th}$ level node. Therefore, we will analyze the the waiting times from the leaf up. If a leaf might split, then the node above it will be locked. Therefore, if O2 is waiting for the leaf-level lock held by O1, O2 will only have to wait while O1 performs the search/insert/delete on the leaf. If O2 is waiting for O1's lock on the second level, O2 will have to wait while O1 searches the node, waits for a leaf lock, performs the operation if the leaf is op-unsafe, and restructures the leaf, if needed. In general, if O2 is waiting for O1's lock on a level i node, O2 will have to wait while O1 searches the level i node and locks a level $i-1$ node. If the level $i-1$ node is op-unsafe, the level i lock will be held for the level $i-1$ lock service time. In addition, the level $i-1$ node might be restructured. Using this observation, we get:

Theorem 1 *The expected time that operation $o \in \{S, I, D\}$ holds a lock on a level i node, $i = 1, \ldots, h$, when another operation might be waiting is*

level 1: *(leaf level)*

search: $T(S, 1) = Se(1)$

insert: $T(I, 1) = M$

delete: $T(D,1) = M$

level j: $(1 < j \leq h)$

search: $T(S,j) = Se(j) + R(j-1)$

insert: $T(I,j) = Se(j) + W(j-1)$
$+ \Pr[F(j-1)]T(I,j-1)$
$+ Sp(j-1)\prod_{k=1}^{j-1}\Pr[F(k)]$

delete: $T(D,j) = Se(j) + W(j-1)$
$+ \Pr[Em(j-1)]T(D,j-1)$
$+ Mg(j-1)\prod_{k=1}^{j-1}\Pr[Em(k)]$

The probabilities that nodes are insert-unsafe or delete-unsafe can be determined from [10]. In addition, [10] shows that if we use a merge-at-empty B-tree, and there are more inserts than deletes in the operation mix, then the probability that a leaf node merges is almost zero, and the probability that a merge propagates is infinitely smaller. Therefore, the formulae for $T(D,j)$ can be simplified to

Corollary 1 *If there are at least 5% more inserts than deletes,*

$$\begin{aligned} T(D,1) &= M \\ T(D,j) &= S(j) + W(j-1) \\ \Pr[F(1)] &= (1-2q)/[(1-q).68N] \\ \Pr[F(j)] &= 1/.69N \qquad 1 < j \leq h \end{aligned}$$

where N is the maximum node size.

Proof: Under the assumption, $\Pr[F(1)]$ can be determined from the rule of thumb [10] and $\Pr[F(j)]$, $1 < j \leq h$, is the same as that for a pure-insert tree [9] •

With $T(Op,j)$ calculated, we can calculate the service time parameters for the level j FCFS R/W queue. Since inserts and deletes always place exclusive locks, we model them as W customers. Search operations place shared locks, so we model them as R customers. Let $\mu_{R,i}$ be the service rate for R customers on the ith level, and let $\mu_{W,i}$ be the service rate for W customers on the ith level. Then:

Proposition 1

$$\begin{aligned} \mu_{R,i} &= 1/T(S,i) \\ \mu_{W,i} &= 1/(\tfrac{q_i}{q_i+q_d}T(I,i) + \tfrac{q_d}{q_i+q_d}T(D,i)) \end{aligned}$$

In order to finish the characterization of Naive Lock-coupling in terms of the R/W queues, we need to specify the arrival rates. Let λ_i be the arrival rate of all jobs on level i. Let $\lambda_{R,i}$ be the arrival rate of R jobs on the ith level, and let $\lambda_{W,i}$ be the arrival rate of W jobs on the ith level. The arrival rate to a level i queue is the arrival rate to the level $i+1$ queue divided by the fanout at level $i+1$. The fanout is the expected number of children of a level $i+1$ node. At the root, the fanout depends on the number of items in the tree and the node size [9]. Below the root, the actual fanout is a constant that depends only on the maximum fanout (approximately $.69N$, as described in [9]). Therefore, we have:

Proposition 2

$$\begin{aligned} \lambda_h &= \lambda \\ \lambda_i &= \lambda_{i+1}/E(i+1) \end{aligned}$$

and

$$\begin{aligned} \lambda_{R,i} &= q_s\lambda_i \\ \lambda_{W,i} &= (q_i + q_d)\lambda_i \end{aligned}$$

Now we can apply the parameters $\lambda_{R,i}$, $\lambda_{W,i}$, $\mu_{R,i}$, $\mu_{W,i}$ to a FCFS R/W queue and calculate the probability that a W operation is in the queue, $\rho_w(i)$, and the extra work caused by the R operations, $r_e(i)$ and $r_u(i)$ by using Theorem 6 (in the appendix). The throughput of Naive Lock-coupling on a particular B-tree reaches its maximum when the nodes on some level in the B-tree are always W-locked. At this point, an increase in the arrival rate will not result in an increase in throughput, and the saturated queues will become unstable. Because of lock-coupling, the bottleneck node will always be the root.

Theorem 2 *The maximum throughput of the Naive Lock-coupling algorithm on a particular B-tree is limited by the minimum arrival rate such that $\rho_w(h) = 1$*

Examining Theorem 1, we can see that the lock service times on level i, and thus $\rho_w(i)$, depend on the lock waiting times on level $i-1$ (except at the leaves). Lock coupling gives the service time distributions a large variance: an operation might or might not have to wait for the next node and the next node might or might not be full. So, we cannot model the service time distribution as being exponential; instead we model the distribution as a series of exponential distributions.

Divide the expected time that a W operation blocks other locks into three parts: the time that every W operation blocks other locks (the node search time plus the wait for readers), the wait to obtain the child's lock, and the time spent holding the child's lock, if the child is *op*-unsafe. If the queue for the child's lock has another W operation in it, then the expected wait is $r_u(i-1) + R(i-1)/\rho_w(i-1)$; otherwise the expected wait is $r_e(i-1)$. The child's queue will have a W operation in it with probability $\rho_w(i-1)$. If the W lock is due to an insert operation $(q_i/(q_i+q_d))$ and the child is insert-unsafe

$(\Pr[F(i-1)])$, then the lock will be held for an additional $T(I, i-1)$ (as noted above, [10] allows us to assume that nodes are almost never delete-unsafe). Additionally, the lock might be held while a split up to the child takes place. As a simplification, add this wait to $T(I, i-1)$, as it will occur only if the child is insert-unsafe. If we model the stages as having exponential distributions, the resulting server has a hyperexponential distribution. See Figure 2. Kleinrock [12] describes how to calculate the expected waiting time for this server. Let:

$$\begin{aligned}
p_f &= q_i \Pr[F(i-1)]/(q_i + q_d) \\
\rho_o &= \rho_w(i-1) \\
t_e &= (Se(i) + \rho_w(i) r_u(i) + (1-\rho_w(i)) r_e(i)) \\
t_f &= 1/(T(I, i-1) + Sp(i-1) \textstyle\prod_{k=1}^{j-2} \Pr[F(k)]) \\
\mu_o &= 1/(R(i-1)/\rho_w(i-1) + r_u(i-1)) \\
t_o &= \rho_o/\mu_o + (1-\rho_o) r_e(i-1)
\end{aligned}$$

Then:

Theorem 3 *The lock waiting times are, for $i = 2, \ldots, h$:*

$$\begin{aligned}
R(i) = \; & \tfrac{\lambda_w(i)}{1-\rho_w(i)} \big[t_o t_e + p_f t_f t_e + t_e^2 + p_f t_o t_f + \\
& \rho_o/\mu_o^2 + p_f t_f^2 + (1-\rho_o) r_e^2(i-1) \big] \\
W(i) = \; & R(i) + \rho_w(i) r_u(i) + (1-\rho_w(i)) r_e(i)
\end{aligned}$$

Proof: The expected waiting time for a M/G/1 server is [12]:

$$W = \frac{\lambda \overline{x^2}}{2(1-\rho)} \tag{1}$$

where x is the service time of the server. The value of $\overline{x^2}$ can be found by differentiating the Laplace transform twice and evaluating at zero.

Let:

$$\begin{aligned}
\mu_e &= 1/t_e \\
\mu_l &= 1/t_f
\end{aligned}$$

Then the Laplace transform of the server is [12]:

$$\begin{aligned}
B^*(s) = \; & \left(\tfrac{\mu_e}{s+\mu_e}\right) \left(p_f \left(\tfrac{\mu_l}{s+\mu_l}\right) + (1-p_f)\right) \\
& \left(\rho_o \left(\tfrac{\mu_o}{s+\mu_o}\right) + (1-\rho_o) \tfrac{1}{r_e(i-1)s+1}\right)
\end{aligned}$$

Take the second derivative of $B^*(s)$ and evaluate at zero:

$$\begin{aligned}
B^{*(2)}(0) = \; & 2[t_o t_e + p_f t_f t_e + t_e^2 + p_f t_o t_f + \\
& \rho_o/\mu_o^2 + p_f t_f^2 + (1-\rho_o) r_e^2(i-1)]
\end{aligned} \tag{2}$$

Combine (1) and (2) to obtain the result •

The leaves are an exception. Model their service time by an exponential distribution.

Theorem 4 *The lock waiting times at the leaves are:*

$$\begin{aligned}
R(1) = \; & \tfrac{\rho_w(1)}{1-\rho_w(1)} [1/\mu_{w,1} + \rho_w(1) r_u(1) \\
& + (1-\rho_w(1)) r_e(1)) \\
W(1) = \; & R(1) + \rho_w(1) r_u(1) + (1-\rho_w(1)) r_e(1)
\end{aligned}$$

Proof: The waiting time of an M/M/1 queue is $\rho/[(1-\rho)\mu]$ ([12]). Apply Theorem 6 to obtain the result •

After calculating all of the lock waiting times by starting at the leaves and working up, we can calculate the expected response times of the operations. The response time of an operation is the time needed to perform the operation in the absence of locking, plus all of the lock waiting times.

Theorem 5 *The response times of the search (S), delete (D) and insert (I) operations are:*

$$\begin{aligned}
Per(S) = \; & \textstyle\sum_{j=1}^{h} (Se(i) + R(i)) \\
Per(D) = \; & M + W(1) + \textstyle\sum_{j=2}^{h} (Se(i) + W(i)) \\
Per(I) = \; & M + \textstyle\sum_{j=2}^{h} Se(i) + \sum_{j=1}^{h} W(i) + \\
& \textstyle\sum_{j=1}^{h-1} (\prod_{k=1}^{j} \Pr[F(i)]) Sp(i)
\end{aligned}$$

To summarize, the sequence of calculations is: (1) Use the FCFS analysis in [8] to calculate $\rho_w(1)$ (see the Appendix), (2) compute $R(1)$ and $W(1)$ using Theorem 4, (3) use Theorem 3 to compute the R and W lock waiting times for the other levels. If $\rho_w(i) < 1$ for all levels $1 \leq i \leq h$, the queues are stable and the algorithm will give a throughput equal to the arrival rate. Use the lock waiting times $R(i)$ and the $W(i)$ to compute the response times of the operations using Theorem 5.

5.1 Analysis of Optimistic Descent and the Link-type Algorithm

For reasons of space, we will not analyze the Optimistic Descent algorithm or the Link-type algorithm in depth. Rather, we will describe how to modify the analysis of the Naive Lock-coupling algorithm to model the other two algorithms.

In the Optimistic Descent algorithm, an update (insert or delete) operation makes an initial optimistic descent, where it follows the search operation's protocol, except for placing a W lock on the leaf. If the leaf needs to be restructured, it makes a second descent, placing W locks. To account for the second descent, create a new type of operation, the *redo-insert*

operation. The rate at which redo-insert operations enter the B-tree is the rate at which update operations make second descents, $q_i \Pr[F(1)]\lambda$. Use the four types of operations to calculate the R and W lock waiting times at each level. The response time of an insert operation is the time to make the first descent, plus $\Pr[F(1)]$ times the response time of a redo-insert operation.

In the Link-type algorithm, all operations place R locks down to the leaf, then update operations W lock and search operations R lock the leaf, holding at most one lock at a time. If the node must be restructured, it is half-split, the W lock on the node released and the parent W-locked. Therefore, the arrival rate of W locks to a leaf is $E[h] \cdots E[2](q_i + q_d)\lambda$. The arrival rate of W locks to a node above the leaf is the rate at which children of the node split, $E[h] \cdots E[3] \Pr[F(1)] q_i \lambda$ and so on. Since there is no lock coupling, the service time of an R lock is the time to search the node and the service time of a W lock is the time to modify the node and potentially half-split it. A complication is that operations might cross links, which increases the arrival rates, and thus increases the probability that an operation crosses a link. However, link crossing is rare and has a negligible effect on performance (see Figure 9).

5.2 Resource Contention

Since all access times are parameterized, resource contention can be factored in as a pre-calculation dilation factor. Assuming that one has a model of the computer system, calculating the resource contention dilation factor is straight-forward. By Little's Law, the number of active (non-blocked) operations is the arrival rate times the expected serial service. Blocked transactions can be assumed to be idle. The throughput is simply the arrival rate, if the concurrent B-tree algorithm is stable. These parameters can be used to calculate the dilation factor.

5.3 Experiments and Comparison

We ran simulation experiments for comparison with the analysis. The simulator ran until 10,000 concurrent operations were performed. Figures 3 through 8 show the comparisons. A node in the underlying B-tree held a maximum of 13 items. The concurrent operations started when the B-tree held about 40,000 items. The root of the B-tree had about 6 children. The B-tree had 5 levels; it was assumed to have the two top levels in memory and the remaining levels on disk. The cost of disk accesses can be varied, but the figures show experiments in which the cost of accessing an on-disk node is 5 times the cost of accessing an in-memory node. The time to search the root was one time unit. The time to modify a leaf was twice the time to search the leaf, and the time to split a node was three times the time to search it. The time to modify the parent is included in the cost of a split. The proportions of concurrent operations were: $q_s = .3$, $q_i = .5$ and $q_d = .2$. At each setting of the parameters, 5 simulations were run, each with a different seed.

Figures 3 and 4 compare analytical and simulation predictions of the search and insert response times of the Naive Lock-coupling algorithm. Figures 5 and 6 show the comparison for the Optimistic Descent algorithm and figures 7 and 8 show the comparison for the Link-type algorithm. The response time of the operation is plotted against the arrival rate (or the throughput). The comparison shows that the analysis and the simulation predict the same response times.

The response time curves tend to stay level with an increasing arrival rate, then increase rapidly as the arrival rate approaches the maximum throughput. The increase in the response time is due to increases in the waiting time. The rapid increase in the response time can be predicted from standard M/M/1 queuing theory [12]. The response time of the lock-coupling algorithms increase even more rapidly than the M/M/1 queue, however. The root writer utilization increases non-linearly with the increasing arrival rate, as is shown in figure 10 (for the Naive Lock-coupling algorithm). To go from $\rho_w = .5$ to $\rho_w = 1$ requires less than a 50% increase in arrival rate. This is due to an increasing wait for a lock on one of the root's children, and hence reflects the cost of lock-coupling. Figure 11 shows the maximum throughput plotted against the cost of accessing an node stored on disk (for Naive Lock-coupling). The cost of locking nodes stored two levels below the root can have a significant impact on the performance of the algorithm.

Figure 12 shows a comparison of the response times of the three algorithms. The Optimistic Descent algorithm has significantly better performance then the Naive Lock-coupling algorithm, and the Link type algorithm has significantly better performance than the Optimistic Descent algorithm. The Link-type algorithm achieves very high concurrency because only those nodes that are actually modified are W-locked, and the W locks are held for a relatively short period of time.

6 Rules of thumb

The calculations of the previous section can generate accurate predictions of the concurrent algorithm's performance, but are somewhat cumbersome for generating intuition about performance. We will derive a rule of thumb that will predict the arrival rate where $\rho_w = .5$. Denote this arrival rate by $\lambda_{\rho=.5}$. Increasing the arrival rate beyond this point will cause a disproportionate increase in waiting. The rules of thumb will predict an 'effective maximum arrival rate'.

Naive Lock-coupling The root is the bottleneck for the Naive Lock-coupling algorithm, so we will approximate $\lambda_{\rho=.5}$ for the root. The writer utilization is the writer arrival rate divided by the aggregate customer service rate: $\rho_w = \lambda_w/\mu_a$. (The time to serve an aggregate customer is the time to serve a W lock and all R locks immediately ahead of the W lock and for which the W lock must wait. See the Appendix.) Therefore, the writer arrival rate when the writer utilization is .5 is:

$$\lambda_{w,\rho=.5} = \mu_a/2$$

We need to approximate the aggregate customer service rate, or, equivalently, the expected time to serve an aggregate customer. We will approximate the aggregate customer service time by the time to search the root, get the next lock, search the child if the child is full, and wait for the preceding readers. Define:

$T_{a,l}$: time to serve an aggregate customer on level l.

$T_{r,l}$: time to serve R customers that are ahead of a W customer on level l.

Then,

$$T_{a,h} = Se(h) + W(h-1) + T_{r,h} + \frac{q_i}{q_i + q_d}\Pr[F(i-1)]T_{a,h-1} \tag{3}$$

We will approximate the time to serve the preceding readers first. If ρ_w is known, the time to serve the preceding readers is, by Theorem 6,

$$T_r = \left(\rho_w \ln\left(1 + \frac{\rho_w\lambda_r}{\lambda_w}\right) + (1-\rho_w)\ln\left(1 + \frac{(1+\rho_w)\lambda_r}{\mu_r+\lambda_w}\right)\right)/\mu_r$$

Approximate μ_r by μ_w and approximate λ_w/μ_w by ρ_w.

$$\begin{aligned} T_r &\approx \left(\rho_w \ln\left(1 + \frac{\rho_w\lambda_r}{\lambda_w}\right) + (1-\rho_w)\ln\left(1 + \frac{\lambda_r}{\mu_w}\right)\right)/\mu_r \\ &\approx \ln\left(1 + \rho_w\frac{\lambda_r}{\lambda_w}\right)/\mu_r \end{aligned} \tag{4}$$

In the case of Naive Lock-coupling, $\lambda_r/\lambda_w = q_s/(1-q_s)$, so

$$T_r \approx \ln\left(1 + \rho_w\frac{q_s}{1-q_s}\right)/\mu_r$$

At the root, $\rho_w = .5$ and $\mu_r = 1/Se(h)$.

$$T_{r,h} \approx \ln\left(1 + \frac{q_s}{2(1-q_s)}\right)Se(h) \tag{5}$$

The wait for the lock on the child is approximately

$$W(h-1) \approx \frac{\rho_{w,h-1}}{1-\rho_{w,h-1}}T_{a,h-1}$$

Approximate $\rho_{w,h-1}$ by $\rho_w/E(h) = 1/2E(h)$, so that

$$W(h-1) \approx T_{a,h-1}/(2E(h)-1) \tag{6}$$

We need an approximation for $T_{a,h-1}$. Instead of continuing the approximations down the the leaves, approximate the time to get the next lock and search the next node, if it is full, by $Se(h-1)/2$. In addition, use the approximation $\ln(1+x) \approx x$, so

$$T_{a,h-1} \approx Se(h-1)\left(1.5 + \frac{q_s}{2E(h)(1-q_s)}\right) \tag{7}$$

If we combine all of the approximations and adjust the arrival rate to count the search operations, we get:

Rule Of Thumb 1 *The arrival rate to a Naive Lock-coupling algorithm such that the writer utilization is .5 is approximately*

$$\lambda_{\rho=.5} \approx \left[2(1-q_s)\left[Se(h)\left(1+\ln\left(1+\frac{q_s}{2(1-q_s)}\right)\right) + \left(\frac{1}{2E(h)-1} + \frac{q_i}{q_i+q_d}\Pr[F(h-1)]\right)\left(Se(2)\left(1.5+\frac{q_s}{2E(h)(1-q_s)}\right)\right)\right]\right]^{-1}$$

If the maximum node size and root fanout is large enough, we can get an even simpler formula.

Rule Of Thumb 2 (Limit) *If the maximum node size and the root fanout are large,*

$$\lambda_{\rho=.5} \approx \frac{1}{2(1-q_s)\left[Se(h)\left(1+\ln\left(1+\frac{q_s}{2(1-q_s)}\right)\right)\right]}$$

Figure 13 shows a comparison between the analytical predictions, rule of thumb 1, and the limit rule of thumb 2. The in-memory B-tree effective maximum arrival rate closely matches the predictions of

rule of thumb 1. If the disk cost is 10, however, rule of thumb 1 vastly overestimates performance when the maximum node size is small. This is due to waiting for the expensive on-disk nodes. When the maximum node size becomes large, the arrival rate, and thus the lock waiting time, on the on-disk levels becomes small. rule of thumb 1 quickly approaches the limit rule of thumb 2. Notice that in the limit rule of thumb, the effective maximum utilization doesn't increase with the maximum node size.

Optimistic Descent The Optimistic Descent algorithm uses the same mechanism as Naive Lock-coupling, so the same analysis can be applied in deriving a rule-of-thumb for the effective maximum utilization. The primary difference is that $\lambda_r = \lambda$ and $\lambda_w = q_i \Pr[F(1)]\lambda$, so that

$$\lambda_r/\lambda_w = 1/(q_i \Pr[F(1)])$$

Also, the reader arrival rate is certain to be much larger than the writer arrival rate, so we can't use the approximation $\ln(1+x) \approx x$.

Rule Of Thumb 3 *The arrival rate to an Optimistic Descent algorithm such that the writer utilization is .5 is approximately*

$$\lambda_{\rho=.5} \approx \left[2q_i \Pr[F(1)] \cdot \left[Se(h)\left(1+\ln\left(1+\frac{1}{2q_i \Pr[F(1)]}\right)\right) + \left(\frac{1}{2E(h)-1} + \frac{q_i}{q_i+q_d}\Pr[F(h-1)]\right)\left(Se(2)\left(1.5+\ln\left(1+\frac{1}{2E(h)q_i \Pr[F(1)]}\right)\right)\right)\right]\right]^{-1}$$

If the maximum node size and root fanout is large enough, we can get an even simpler formula.

Rule Of Thumb 4 (limit) *If the maximum node size and the root fanout are large,*

$$\lambda_{\rho=.5} \approx \frac{1}{2q_i \Pr[F(1)]\left[Se(h)\left(1+\ln\left(1+\frac{1}{2q_i \Pr[F(1)]}\right)\right)\right]}$$

Figure 14 shows a comparison between the analytical predictions, rule of thumb 3, and the limit rule of thumb 4. Again, the rule of thumb makes better predictions as the maximum node size increases.

Notice that for Optimistic Descent, the maximum arrival rate inversely depends on $\Pr[F(1)]$, which is inversely proportional to the maximum node size. This is in contrast to Naive Lock-coupling, where the effective maximum throughput is independent of the maximum node size. Thus, as the maximum node size increases, Optimistic Descent becomes increasingly better than Naive Lock-coupling.

The rules of thumb suggest a design strategy for concurrent B-trees that use Naive Lock-coupling or Optimistic Descent. The bottleneck operation in Optimistic Descent and Naive Lock-coupling is searching the root. Note that in Figures 13 and 14, the effective maximum throughput is plotted in units of time equal to the time to search the root. In general, the time to search the root will increase as the maximum node size increases. If a binary search is used to search for the proper child of the root, then the time to search the root is of the form $a + b \log N$, where N is the maximum size of the root. The maximum throughput of the Naive Lock-coupling algorithm depends on the time to search the root and not on the probability of splitting a leaf, so the maximum node size should be small. The maximum throughput of the Optimistic Descent algorithm, however, is proportional to $N/\log^2 N$, so the maximum node sizes should be as large as possible.

Link-type algorithm The maximum throughput of a Link-type algorithm is limited by the level that first becomes saturated. Because the Link-type algorithms don't use lock-coupling, this level is not necessarily the root. If we examine figure 12, we can see that a Link-type algorithm allows enormous concurrency, enough to invalidate steady-state assumptions. Therefore, the Link-type algorithm has no effective maximum throughput.

7 An Application of the Analysis - Recovery

In [24] the issue of database recovery as applied to concurrent data structures is discussed. A database manager will typically hold the exclusive locks that a transaction requests until the transaction commits ([19,7]). If a transaction aborts, the previous values of the W-locked records can simply be written back without interfering with other transactions.

If this algorithm is applied to concurrent B-trees, then any W lock placed on any node is held until the transaction commits. We will call this the *Naive recovery* algorithm. Shasha [24] points out that only the leaf locks need to be held for correct recovery. Retaining W locks on non-leaf nodes is called *recovery overlock*. We will call the protocol of releasing non-leaf W locks as soon as possible, but retaining leaf-level W locks until the transaction commits the *Leaf-only recovery algorithm.*

The Leaf-only recovery algorithm is clearly better than the Naive recovery algorithm. The question is, how much better? In particular, is it good enough to justify having one protocol for data locking and another for index locking? In order to answer this question, we modeled the effects of recovery on the Optimistic Descent algorithm.

The modifications to account for recovery are simple. Let T_{trans} be the expected time until the transaction commits. Calculate $T(OP, i)$ (as in Theorem 1). At the leaf level, let $T'(OP, 1) = T(OP, 1) + T_{trans}$ for every OP that places a W lock for the Naive recovery algorithm or the Leaf-only recovery algorithm. Above the leaf, let $T'(OP, i) = T(OP, i) + \Pr[F(i)]T_{trans}$ for every OP that places a W lock, for the Naive recovery algorithm. Let $T'(OP, i) = T(OP, i)$ otherwise. Use the $T'(OP, i)$ to calculate the lock waiting times.

Figures 15 and 16 show a comparison of the response times of the Naive recovery algorithm and the Leaf-only recovery algorithm. A B-tree that does not support recovery is also shown. In the comparison, the cost of accessing an on-disk node as compared to an in-memory node is $D = 10$ and $T_{trans} = 100$, a conservative estimate of the remaining transaction time. In Figure 15, the maximum node size is 13, and the number of B-tree levels is 5 (as in figures 3 through 8). In Figure 16, the maximum node size is 59, and the number of B-tree levels is 4. The Leaf-only recovery algorithm has slightly worse performance than the no-recovery algorithm. In contrast, the Naive recovery algorithm has significantly worse performance than the Leaf-only algorithm. Using the Leaf-only recovery algorithm is thus a simple method to significantly increase the performance of a concurrent B-tree.

8 Conclusions

This paper discusses a framework for analyzing the performance of concurrent B-tree algorithms. The framework allows concurrent algorithms to be analyzed in a uniform manner. The analysis agrees with simulation results.

Rules of thumb derived from the analysis provide intuition into the performance of concurrent data structure algorithms. The analyses can be applied to a variety of practical problems, including ranking and comparing the performance of the different concurrent B-tree algorithms and determining the performance implications of recovery algorithms.

The analysis shows that the Link-type algorithm is significantly better than the optimistic descent algorithm, which is significantly better than the Naive Lock-coupling algorithm. The rules of thumb show that the Naive lock-coupling algorithm works best on B-trees with small nodes and the Optistic Descent algorithm works best on B-trees with large nodes. An extension of the analysis shows that the Leaf-only recovery algorithm is significantly better than the Naive recovery algorithm.

Results that will appear in the full version of this paper include analyses of additional concurrent B-tree algorithms, including Two-Phase locking; a discussion of performance implications, LRU buffering, and extensions to other concurrent data structure algorithms.

References

[1] R. Bayer and E.M. McCreight. Organization and maintainance of large ordered indices. *Acta Informatica*, 1(3):173–189, 1972.

[2] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9:1–21, 1977.

[3] M. Carey and C.Y. Cheng. B^+-tree locking: a performance perspective. 1986. manuscript.

[4] V. Lanin D. Shasha and J. Schmidt. *An Analytical Model for the Performance of Concurrent B-tree algorithms.* NYU Ultracomputer Note 311, NYU Ultracomputer lab, 1987.

[5] C.S. Ellis. Concurrent search and inserts in 2-3 trees. *Acta Informatica*, 14(1):63–86, 1980.

[6] J. Gray et. al. One thousand transactions per second. In *IEEE Compcon*, pages 96–101, 1985.

[7] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):12–83, 1983.

[8] T. Johnson. Approximate analysis of reader and writer access to a shared resource. to appear in ACM SIGMETRICS '90.

[9] T. Johnson and D. Shasha. *Random B-trees with Inserts and Deletes.* Technical Report 453, NYU Dept. of C.S., 1989.

[10] T. Johnson and D. Shasha. Utilization of B-trees with inserts, deletes and modifies. In *ACM SIGACT/SIGMOD/SIGART Symposium on Principles of Database Systems*, pages 235–246, 1989.

[11] L. Kleinrock. *Communication Nets; Stochastic Message Flow and Delay.* McGraw Hill, reprinted by Dover Publications, New York, 1964.

[12] L. Kleinrock. *Queueing Systems.* Volume 1, John Wiley, New York, 1975.

[13] L. Kleinrock. *Queueing Systems.* Volume 2, John Wiley, New York, 1976.

[14] H.T. Kung and P.L. Lehman. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems*, 5(3):354–382, 1980.

[15] V. Lanin and D. Shasha. A symmetric concurrent B-tree algorithm. In *1986 Fall Joint Computer Conference*, pages 380–389, 1986.

[16] P.L. Lehman and S.B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, 1981.

[17] J.K. Metzger. *Managing Simultaneous Operations in Large Ordered Indexes.* Technical Report, Technische Universitat Munchen, Institut fur Informatik, 1975. TUM-Math. report.

[18] Y. Mond and Y. Raz. Concurrency control in B^+-trees databases using preparatory operations. In *11th International Conference on Very Large Databases*, pages 331–334, Stockholm, Aug. 1985.

[19] N. Goodman P.A. Bernstein and V. Hadzilacos. Recovery algorithms for database systems. In *Proc. IFIP Congress*, 1983.

[20] M. Jipping R. Ford and R. Schultz. *On the Performance of Concurrent Tree Algorithms.* Technical Report 85-07, University of Iowa, 1985.

[21] M. Jipping R. Ford, R. Schultz, and B. Wenhardt. On the performance of concurrent tree algorithms. submitted for publication.

[22] I.K. Ryu and A. Thomasian. Performance analysis of centralized databased with optimistic concurrency control. *Performance Evaluation*, 7:195–211, 1987.

[23] Y. Sagiv. Concurrent operations on B^*-trees with overtaking. In *Fourth Annual ACM SIGACT/SIGMOD Symposium on the Principles of Database Systems*, pages 28–37, ACM, 1985.

[24] D. Shasha. What good are concurrent search structure algorithms for databases anyway? *Database Engineering*, 8(4):84–90, 1985.

[25] H. Wedekind. On the selection of access paths in a data base system. In J.W. Klimbie and K.L. Koffeman, editors, *Database Management*, pages 385–397, North Holland Publishing Company, 1974.

[26] R. Suri Y.C. Tay and N. Goodman. Locking performance in centralized databases. *ACM Transactions on Database Systems*, 10(4):415–462, 1985.

9 Appendix - the FCFS R/W Queue

In [J90], Johnson describes a FIFO queue in which readers are granted shared locks and writers are granted exclusive locks. Readers and writers arrive at rates λ_r and λ_w and are served at rates μ_r and μ_w, respectively.

The queue is analyzed approximately using *aggregate customers*, which are writers together with all readers immediately ahead of the writer and for which the writer must wait. The analysis calculates the service time of an aggregate customer. There are two cases: either there are no writers in the queue when the current writer arrives, or there is another writer in the queue.

Let ρ_w be the probability that there is a writer in the queue. Let r_u be the expected time the current writer must wait for the readers immediately ahead of it if there is another writer in the queue when the current writer arrives, and let r_e be the time that a writer must wait for the readers if there was no other writer in the queue. The analysis involves the fact that the time to serve n concurrent readers grows logarithmicly with n. The analysis shows that

Theorem 6

$$\begin{aligned} r_u &= \ln(1+\rho_w\lambda_r/\lambda_w)/\mu_r \\ r_e &= \ln(1+(1+\rho_w)\lambda_r/(\mu_r+\lambda_w))/\mu_r \end{aligned}$$

where ρ_w is the root of

$$\rho_w = \lambda_w\left(\frac{1}{\mu_w}+\frac{\rho_w}{\mu_r}\ln\left(1+\frac{\rho_w\lambda_r}{\lambda_w}\right)+\frac{1-\rho_w}{\mu_r}\ln\left(1+\frac{(1+\rho_w)\lambda_r}{\mu_r+\lambda_w}\right)\right)$$

The service time of the aggregate customer is

$$T_a = 1/\mu_w + \rho_w r_u + (1-\rho_w)r_e$$

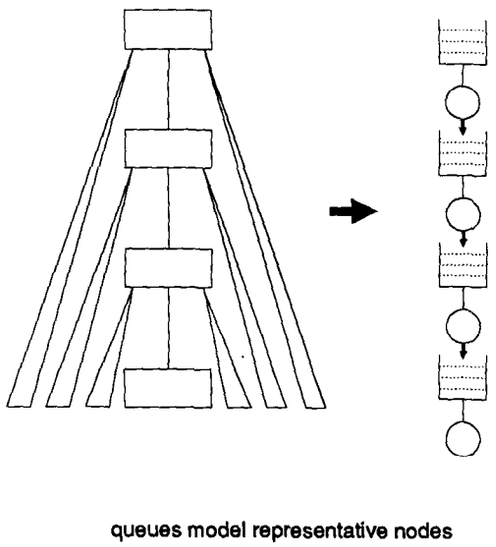


Figure 1: Model of a concurrent B-tree.

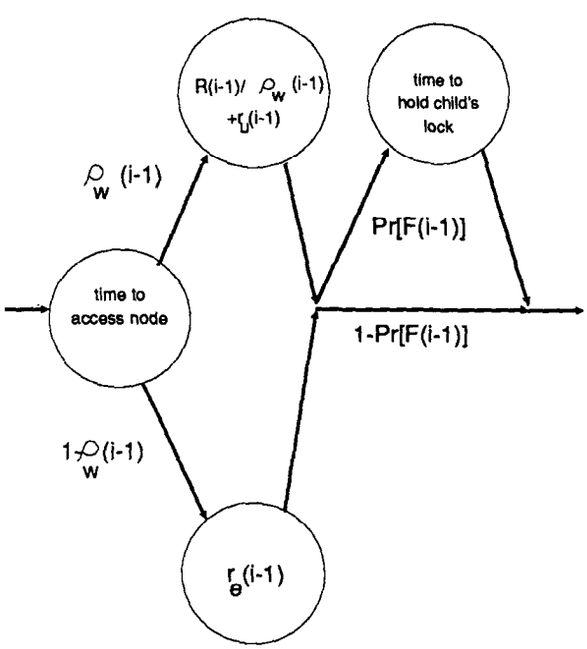


Figure 2: Model of a Naive Lock-coupling server

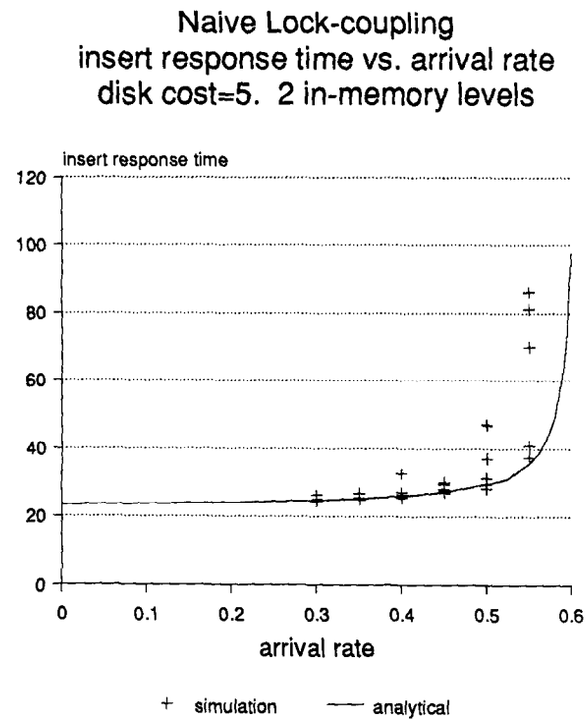


Figure 3:

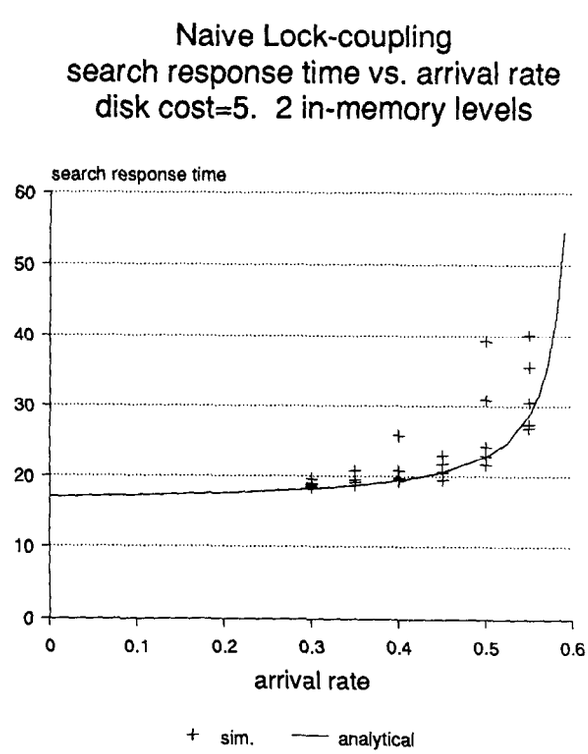


Figure 4:

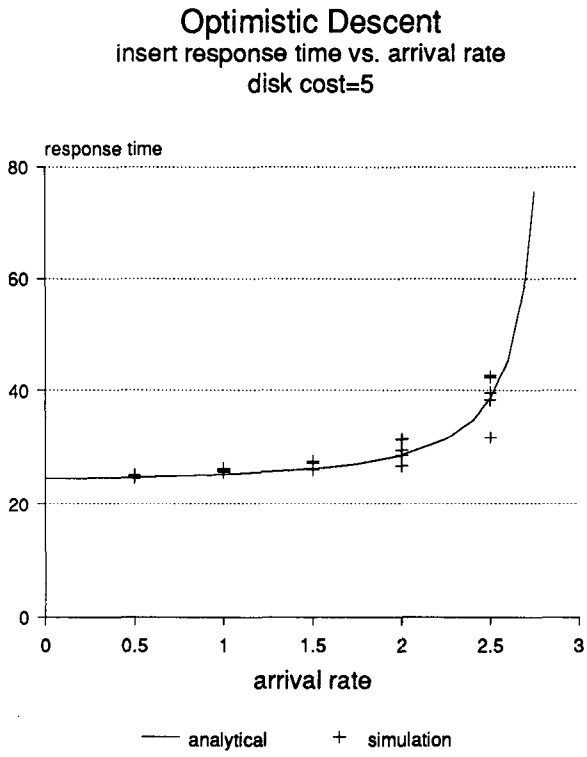


Figure 5:

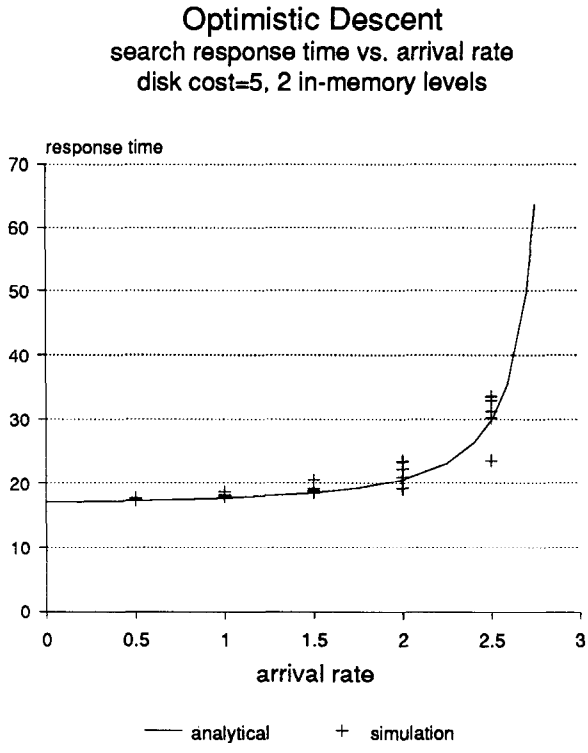


Figure 6:

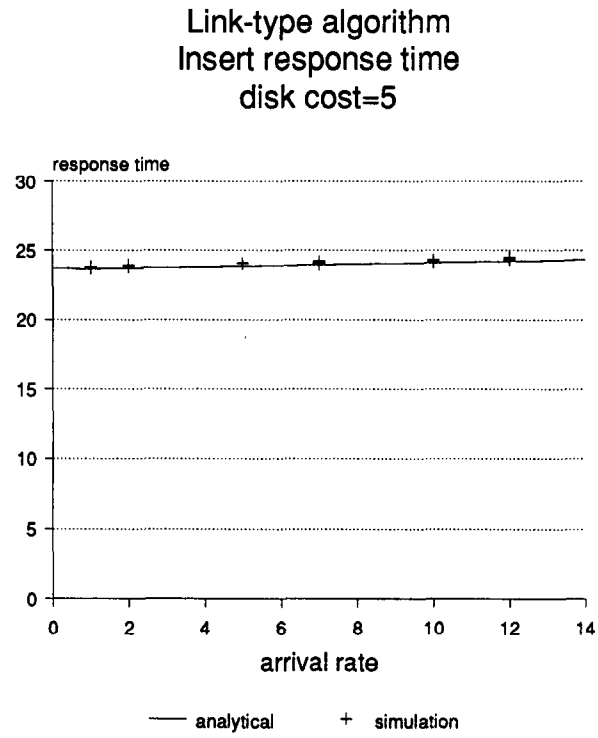


Figure 7:

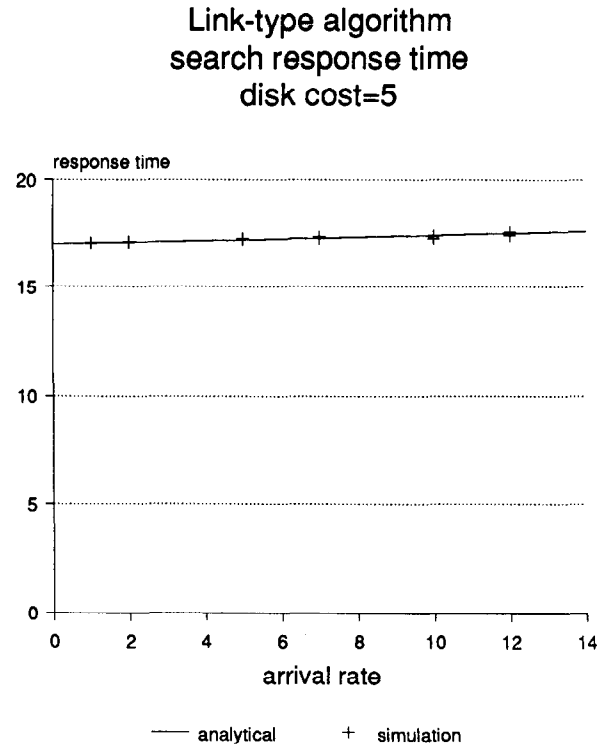
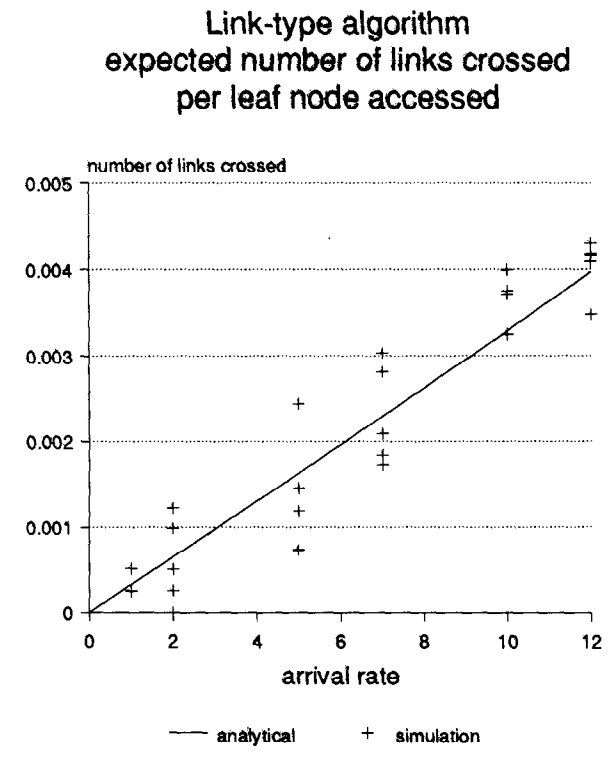


Figure 8:



disk cost=10

Figure 9:

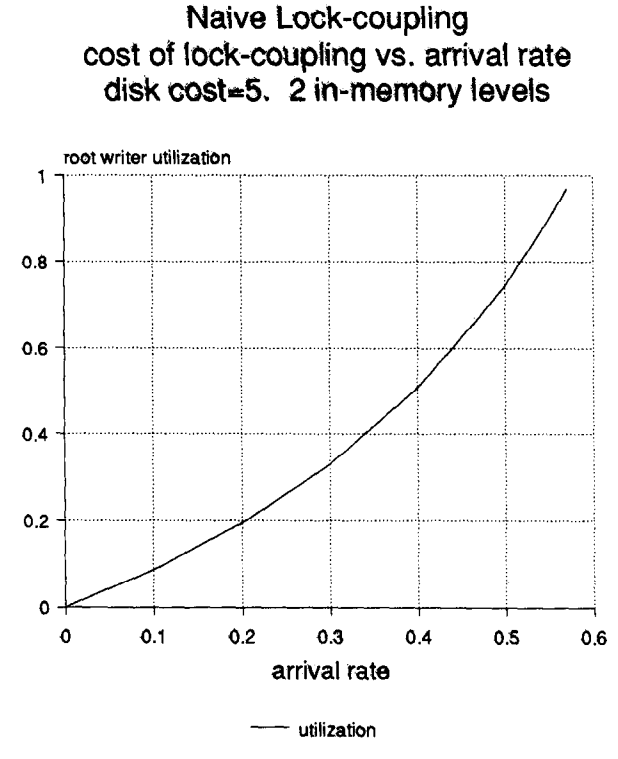


Figure 10: Increasing root writer utilization in Naive Lock-coupling

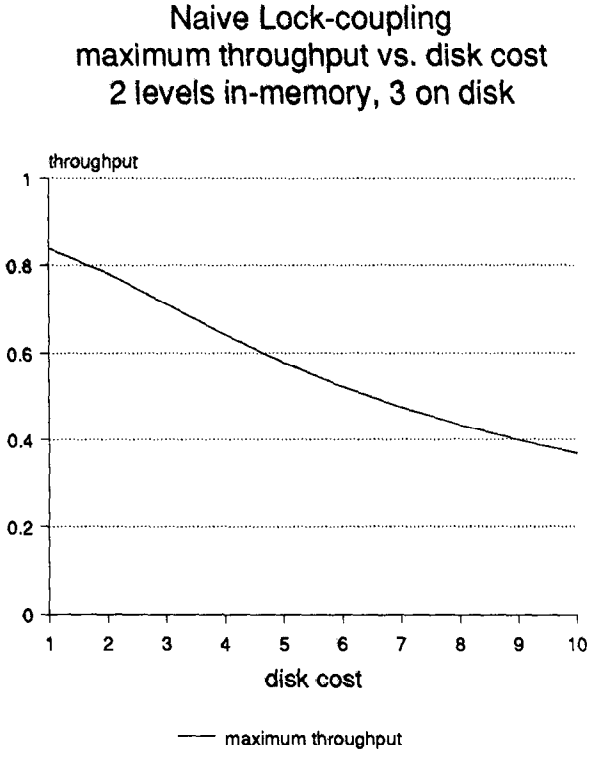


Figure 11:

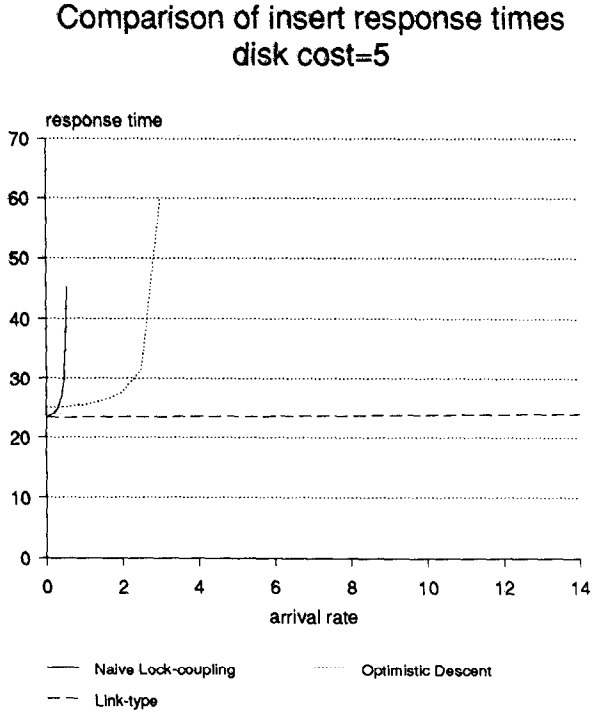


Figure 12: Comparison of response times of Naive Lock-coupling, Optimistic Descent and the Link-type algorithm

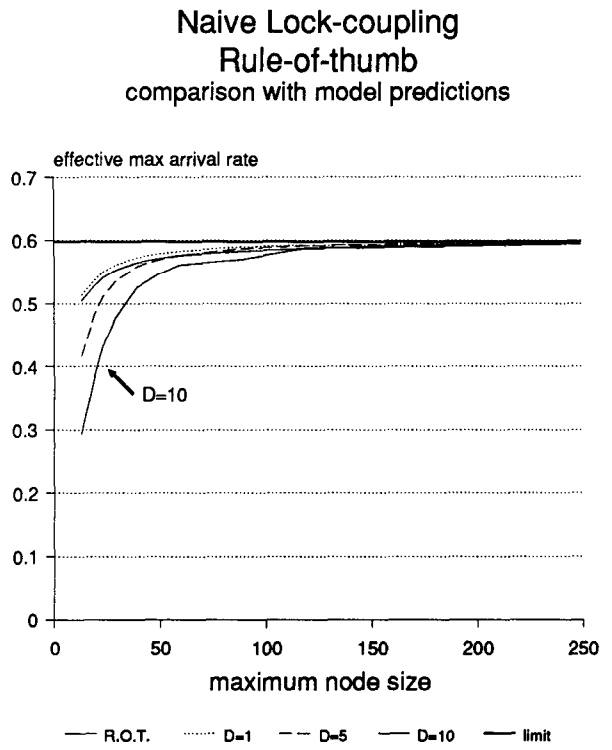


Figure 13: Comparison of the Naive Lock-coupling rule-of-thumb against analytical predictions with varying disk cost, D

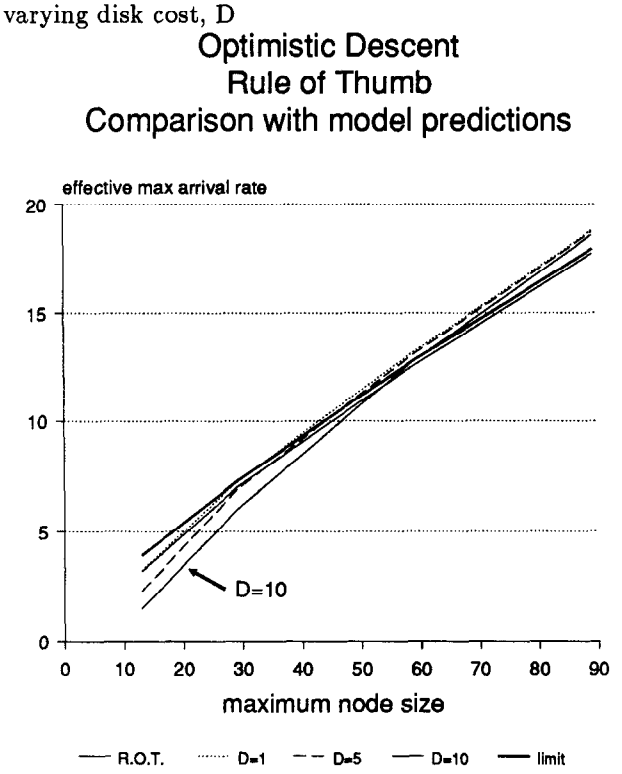


Figure 14: Comparison of the Optimistic Descent rule-of-thumb against analytical predictions with varying disk cost, D

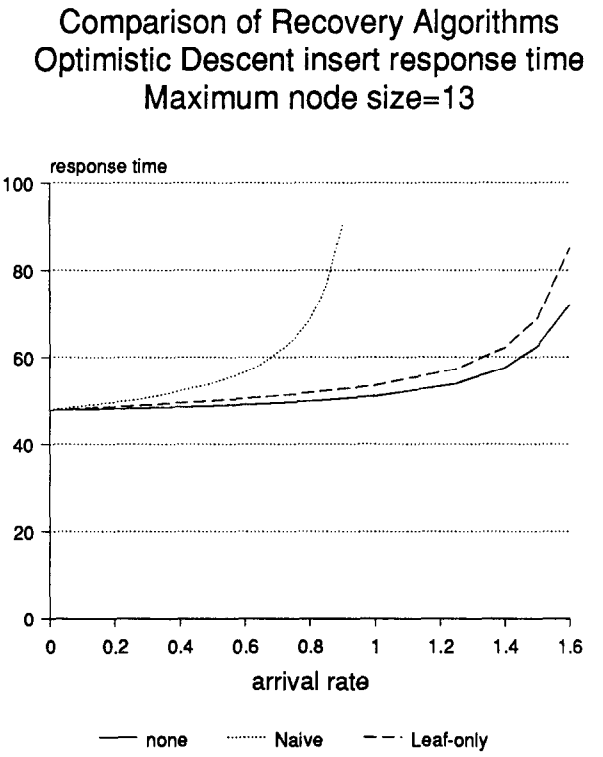


Figure 15:

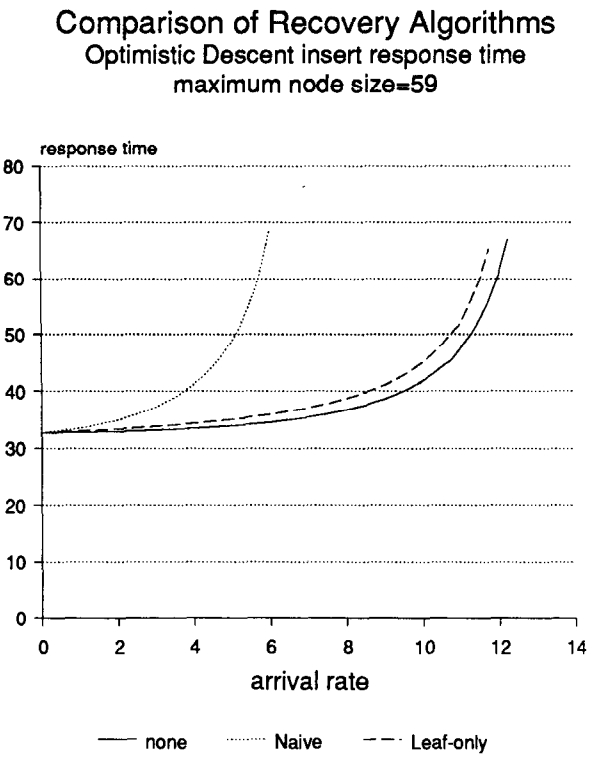


Figure 16: