A Study of Throughput Degradation Following Single Node Failure in a Data Sharing System

Nicholas S. Bowen
IBM T. J. Watson Research Center
Hawthorne, NY 10532
Phone: 914 784 7331 Fax: 914 784 6201
Internet: bowenn@ibm.watson.com

Amber Roy-Chowdhury
University of Illinois*
Urbana, IL 61801
Phone: 217 333 4767 Fax: 217 244 5685
Internet: amber@crhc.uiuc.edu

Abstract

The data sharing approach to building distributed database systems is becoming more common because of its potentially higher processing power and flexibility compared to a data partitioning approach. However, due to the large amounts of hardware and complex software involved, the likelihood of a single node failure in the system increases. Following a single node failure, some processing has to be done to determine the set of locks held by transactions which were executing at the failed node. These locks cannot be released until database recovery has completed on the failed node. This phenomenon can cause throughput degradation even if the processing power on the surviving nodes is adequate to handle all incoming transactions. This paper studies the throughput dropoff behavior following a single node failure in a data sharing system through simulations and analytic modeling. The analytic model reveals several important factors affecting post-failure behavior and is shown to match simulations quite accurately. The effect of hot locks (locks which are frequently accessed) on post-failure behavior is observed through simulations and analytic modeling. Simulations are performed to observe system behavior after the set of locks held by transactions on the failed node has been determined and show that if the delay in obtaining this information is too large, the system is prone to thrashing.

Keywords: Distributed Database Systems, Data Sharing Systems, Availability, Reliability, Modeling and Simulation

1 Introduction

Distributed database systems are increasingly being used in environments which require high throughput and high availability. The multiple nodes can be used to provide data availability after a single node failure. The distributed database system is assumed to conform to the transaction execution model [1, 2]. In order to aid in system recovery following system failure, each transaction writes a log which contains information which allows both the before and after images of the pages accessed by the transaction to be determined. Database recovery to a consistent state following a crash may be accomplished by reading the log and redoing the effects of some transactions, undoing the effects of some transactions or both.

In this paper, we consider a specific realization of a distributed database system, called a data sharing system. Data sharing systems are built upon a distributed architecture where individual nodes are coupled to each other through high-speed interconnect. Communication between nodes is via messages. The database is not partitioned, as in a data partitioning approach; all nodes can directly access all disks on which the physical database is located. Another name for data sharing is therefore shared disk. Database recovery in data sharing systems is very complex, although it builds upon the basic principles of logging and undo or redo of selected transactions. It depends on the strategies for update propagation, page replacement, concurrency control and coherence control (see [3] for background and techniques).

Locking is a critical aspect of data sharing systems. However, designing a locking scheme has different design points for mainline (i.e., non-failure) and recovery cases. For example, a centralized scheme may perform well during recovery (because the global state is known in one place) while a distributed scheme may work better for mainline performance. In general, locks held by transactions that were active when a node fails must be retained until the database manager has gone through recovery processing. In a single node system, this is not an issue because the database is not avail-

*This work was done on leave at IBM T. J. Watson Research Center, Hawthorne, NY 10532 able during recovery. However, a goal of distributed database systems is to tolerate the failure of a single node by allowing the remaining nodes to continue operation. In the distributed system, as with the single node system, the records locked by failed transactions cannot be unlocked until the database manager at the failed node has completed recovery. In practice, transactions that were active on the failed node often hold a very small fraction of the total database records. Once the locks held by the failed node are known it is desirable to abort transactions that access these locks. This is because transactions often obtain many locks, and if they were detained for the entire period preceding node recovery then there would be cascading delays due to further transactions waiting on locks acquired by waiting transactions. The number of waiting transactions could thus grow large rapidly, leading to total data unavailability.

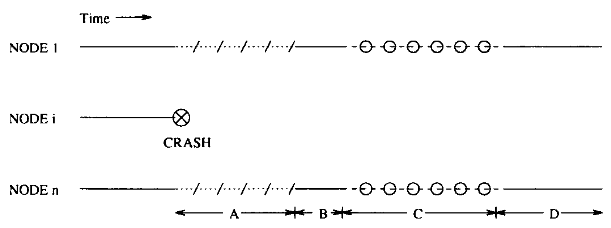
For this paper we assume some form of a distributed lock manager wherein the total set of locks is not known in a single location and following a failure there is some delay in determining the set of locks held by the failed system. We use the term *lock recovery* to refer to the process of determining these locks. these locks lock recovery. During this time period transactions requesting locks from the failed system are delayed. Once lock recovery has completed, database recovery must be done. However, prior to the completion of lock recovery, transactions requesting locks which happen to be held by failed transactions will be queued since it is not known that these requested locks are held by failed transactions. Even though the time to lock recovery can be expected to be much smaller than the time to database recovery, it could be sufficient to cause cascading lock delays under some circumstances. We study this phenomenon through analytic modeling and simulations in this paper. We show that if the time to lock recovery is too large, then the system throughput degrades rapidly due to transactions waiting on unavailable locks. We also study the effect of skewed lock access patterns to determine if these significantly alter the post-failure behavior of the system. We further study the system behavior following lock recovery and show that the throughput is subject to severe degradation in the post lock recovery phase if the lock recovery time exceeds a threshold. We finally discuss some strategies to improve the system throughput following single node failure and subsequent lock recovery.

2 System Model

The system we are interested in modeling is a distributed database system with provisions for log recovery as discussed in the introduction. The system is modeled as a loosely coupled system with identical processors in which each processor executes transactions in parallel with other processors. We assume that the entire physical database is directly accessible to each processor, a characteristic of data sharing systems. Associated with each processor is a database manager responsible for performing all bookkeeping tasks necessary for transaction execution on the processor. We refer to the database manager and its associated processor as a *node*. An important constituent of the database manager for the purposes of this paper is the lock manager. The lock manager is responsible for keeping track of information about locks owned by transactions executing at a particular node.

A node failure refers to a state where a node becomes temporarily unable to execute transactions following a fault in either the software or hardware components. The system is assumed to possess sufficient unutilized processing capability to execute transactions without loss both in the normal mode as well as following a single node failure. We also assume that surviving nodes can still access the entire database following a single node failure, i.e., we exclude disk failures from the failure scenarios under consideration.

Following a node failure, it is assumed that no knowledge is available about the locks held by transactions on the failed node at the remaining nodes. Locks owned by transactions at the failed node are referred to as *lost locks*. Knowledge about which locks are lost is assumed to be available after some processing and exchange of information by lock managers at surviving nodes. We call this process *lock recovery*. The results and conclusions in our paper are independent of the specific algorithm used to achieve lock recovery, as long as the the time to lock recovery is non-zero. Thus, they apply to any locking scheme where the lock information is distributed over all nodes and where no single node possesses information about all the locks in the system. Following lock recovery, database recovery is performed, which may include aborting the transactions which were executing on the failed node and releasing the locks held by them. We refer to this process as ***database recovery*** to distinguish it from lock recovery. Following database recovery, transactions destined for the failed node are instead processed by the surviving nodes. The time to database recovery is assumed to be much larger than the time to lock recovery. After lock recovery, all transactions on working nodes which were waiting on lost locks are aborted and all locks held by these transactions are released. Also, any transaction which subsequently attempts to



	Actions taken when lock contention occurs for	
	Locked data locked by failed transactions	Locked data locked by other transactions
A Lock Recovery	WAIT	WAIT
B Post Lock Recovery	ABORT	WAIT
C Database Recovery	ABORT	WAIT
D Post DB-Recovery	N/A	WAIT

Figure 1: Actions following a node crash

acquire a lost lock is aborted and all its locks released. The actions taken during lock recovery and database recovery are summarized in Fig. 1. A transaction is scheduled to execute on a single node and once scheduled, is never rescheduled on another node. Transactions are assumed to be of a fixed length and to possess a fixed number of locks. Locks are assumed to be held in exclusive mode and are not released until the transaction has completed executing or has been aborted. It is reasonably easy to extend subsequent analyses to the case when locks can be held in either shared or exclusive mode. If a transaction attempts to acquire a lock which is owned by another transaction, it is suspended from executing and is added to a queue of waiters for the lock. We assume that a transaction acquires all the locks that it needs at the beginning, although subsequent analyses may be easily modified for other lock acquisition strategies. When a transaction finishes executing, it releases all its locks. If there are any waiters on a lock when it is released, the lock is passed to the first waiter. Transactions are assumed to arrive as a Poisson stream. A transaction is said to be active if it is part of the pool of transactions currently executing on the system as opposed to waiting to be scheduled or waiting for a lock to be released. There is a maximum number of transactions which can be simultaneously active in the system. We assume a finite overhead cost in terms of CPU instructions executed when a transaction tries to acquire a lock which is unavailable. This is, however, small compared to the time required by the CPU to execute a transaction. We also assume an overhead cost incurred when a transaction has to be aborted and its locks released which is assumed to be of the order of the cost of transaction execution.

3 Post-Failure Behavior

3.1 Analytic Model

In this subsection we derive an ordinary differential equation (ODE) which can be iteratively solved to obtain the system throughput following a single node failure. We compare it to a simulation of the system described in Section 2 to show that it predicts the post-failure system throughput dropoff very accurately. In all subsequent discussion, all references to steady state imply steady state in the normal mode of operation.

The following notation is used in the model

Number of nodes $= N$
Transaction arrival rate $= \lambda$
Locks per transaction $= l$
Size of lock space $= L$
Steady state transaction response time $= r$

We make the reasonable assumption that $l \ll L$. We also assume that the steady state lock contention probability is very small (1% or less) since this is an important design goal of all database systems. We assume that the system reaches steady state at some time prior to node failure.

Following a single node failure, when a transaction acquires part of its locks and attempts to acquire a lost lock the transaction needs to be blocked until the lock recovery is complete. In addition, prior to being blocked, the locks acquired also become unavailable for use until the lock recovery is complete and the transactions waiting on lost locks are aborted and their locks released. Further transactions may need to wait until lock recovery if they attempt to acquire an unavailable lock (though it may not be a lost lock). Any locks which are not unavailable are referred to as available locks.

Let u_t denote the unavailable locks at time t as a fraction of the total number of locks. We now derive an ODE describing the rate of increase of u_t at t seconds following failure of a single node.

Since we assume Poisson arrivals [4], the probability of a single transaction arriving in $(t, t + \Delta t)$ is $\lambda \Delta t + o(\Delta t)$ where $o(\Delta t)$ represents terms whose ratio with Δt tends to 0 as $\Delta t \to 0$. The probability of no transaction arriving in $(t, t + \Delta t)$ is given by $1 - \lambda \Delta t + o(\Delta t)$ while the probability of more than one transaction arriving in $(t, t+\Delta t)$ is given by $o(\Delta t)$. Let the expected number of available locks acquired by a transaction at time t before attempting to acquire an unavailable lock be given by E_t. We then have

$$E_t = \sum_{k=1}^{l-1} k(1-u_t)^k u_t \tag{1}$$

since the probability that a transaction causes the number of unavailable locks to increase by k is the probability that it first acquires k available locks and then attempts to acquire an unavailable lock. Eq. (1) simplifies to

$$E_t = (1-u_t) - l(1-u_t)^l + \frac{(1-u_t)^2\{1-(1-u_t)^{l-1}\}}{u_t} \tag{2}$$

In the above expression we have made the approximation that u_t remains constant as a single transaction acquires locks, which may be justified by our initial assumption that $l \ll L$. The expected fraction of unavailable locks at time $t + \Delta t$ is then given by

$$u_{t+\Delta t} = (1-\lambda\Delta t)u_t + \lambda\Delta t(u_t + \frac{E_t}{L}) + o(\Delta t) \tag{3}$$

The above equation, upon simplification and in the limit as $\Delta t \rightarrow 0$ yields the following ODE

$$\dot{u}_t = \frac{\lambda E_t}{L} \tag{4}$$

where the expression for E_t is given by Eq. (2). An explicit solution for Eq. (4) is not easy to find because of the complicated expression for E_t which also depends on u_t. However, a knowledge of the initial condition of the ODE given by Eq. (4) makes it possible for it to be solved by iterative techniques. The initial condition may be determined by the following argument.

Let the expected number of transactions active or waiting in the system in the steady state be n. Then, by Little's Law [4], we have $n = r\lambda$. Since the steady state lock contention level is assumed to be very small, almost all transactions present in the system are in active mode. An active transaction possesses l exclusive locks since we assume that a transaction acquires all its locks before executing and all locks are acquired in exclusive mode. (To model a system in which transactions acquire locks as they execute, one may assume that a transaction possesses $\frac{l}{2}$ locks on average. A system with shared and exclusive locks may be modeled by treating all lost locks as exclusive, since no knowledge about lost locks is available prior to lock recovery, and subsequently deriving an ODE for the rate of increase of unavailable exclusive locks, in much the same manner as in this section). Thus the total number of locks held during steady state is $lr\lambda$. By symmetry, since nodes are identical, the number of locks held by any single node in the steady state is $\frac{lr\lambda}{N}$. This is also equal to the number of lost locks immediately following a single node failure. Thus the fraction of the lock space constituted by lost locks immediately following a single node failure is given by $\frac{lr\lambda}{NL}$. Since the initial fraction of unavailable locks consists of just the lost locks, we have as the initial condition for Eq. (4)

$$u_0 = \frac{lr\lambda}{NL} \tag{5}$$

Now in order to compute the expected system throughput at time t, one needs to compute the probability that an incoming transaction is able to acquire all its locks, since this also gives the probability that the transaction will be able to complete execution instead of having to wait for an unavailable lock. The probability that a transaction arriving at time t is able to acquire all its locks is given by $(1-u_t)^l$ so that the throughput at time t (T_t) is given by

$$T_t = (1-u_t)^l\lambda \tag{6}$$

where T_t denotes the throughput at time t. We may use Eq. (6) to predict the throughput of the system following a single node failure.

Eq. (4) indicates that the rate of increase of unavailable locks following single node failure is directly proportional to the arrival rate of transactions and inversely proportional to the lock space size. In order to obtain more insight into the factors influencing this rate of growth we consider the period just following the failure, during which we may assume that u_t is very small. Under such an assumption, we may expand E_t in the right hand side of Eq. (4) in a Taylor expansion and simplify by discarding all second and higher order terms in u_t to obtain the following expression

$$\dot{u}_t \approx \frac{\lambda}{L}\frac{l(l-1)}{2}u_t \tag{7}$$

We observe that the number of locks per transaction has a quadratic effect on the initial rate of increase of unavailable locks. Thus if two database systems have the same level of lock contention in the steady state but one has a larger lock space as well as a larger number of locks per transaction, its throughput will start to drop off faster following a failure.

We note also that the number of nodes, N, enters into the analytical model of the system behavior only in the initial condition and that for larger N, the initial fraction constituted by lost locks is smaller. Thus for two systems of identical processing capability but

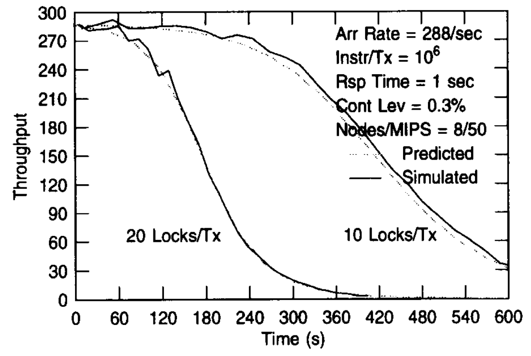


Figure 2: Throughput following failure for systems with different locks/tx

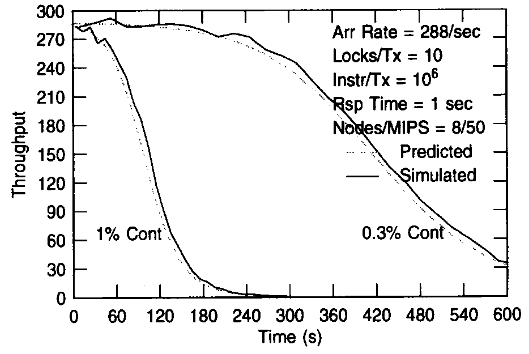


Figure 3: Throughput following failure for systems with different contention levels

different number of nodes, we expect the one with the larger number of nodes to be more tolerant of lost locks following a failure.

3.2 Simulation Results

A simulator was written for the system described in Section 2. The system behavior was observed for various realistic workloads. Transaction path lengths were chosen to be either 100000 or a million instructions while the number of locks per million instructions processed (locks per MIP) was varied between 10 to 40. Response times were chosen to be 1 second for all workloads and thus determined the degree of multiprogramming, or number of active transactions in the steady state. The maximum number of active transactions was fixed to be twice the steady state multiprogramming level. Arrival rates were chosen so that the steady state system utilization due to transaction execution was approximately 80%. Lock space sizes were fixed so that steady state contention levels varied between 0.3 to 1%. Most of these figures are typical for workloads of actual transaction processing systems [5],[6]. The aggregate system processing capability was kept constant at 400 MIPS but configurations of 8,16 and 32 nodes were simulated. We ran the simulations for 30 seconds prior to injecting a failure in a randomly chosen node. This was done to allow the system to reach steady state before the occurrence of a failure.

We now look at figures for throughput dropoff following system failure for various workloads. From Fig. 2 we observe that as locks per transaction goes up, the system post-failure behavior undergoes a rather drastic degradation. This is despite the fact that steady state contention levels are kept constant, which implies that the lock space size for the system executing transactions with more locks per transaction is

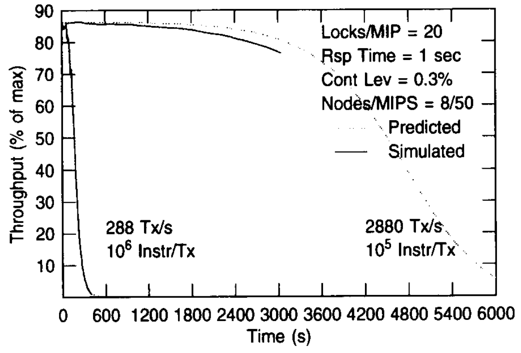


Figure 4: Throughput following failure for systems with different arrival rates

actually larger. However, this behavior is not unexpected, since Eq. (7) indicates that the negative effect of increased number of locks per transaction dominates over the positive effect of having a larger lock space. The effect of an increased steady state contention level is shown in Fig. 3. There is a twofold effect which comes into play here - increased initial contention levels, besides causing the initial throughput to be lower than a system with lower contention levels, also implies a smaller lock space, so that the rate of dropoff of throughput following a failure is also greater. From Fig. 4 we note that if contention levels and locks per MIP are kept constant, a system with a higher transaction arrival rate has a much better post-failure behavior than one with a lower transaction arrival rate. Since we are interested in evaluating systems with equivalent computing power, transaction path length is decreased for workloads with high arrival rates. This, and the fact that locks per MIP are constant can be applied to Eq. (5) to conclude

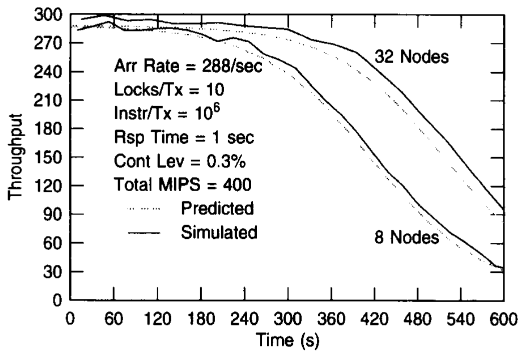


Figure 5: Throughput following failure for systems with different number of nodes

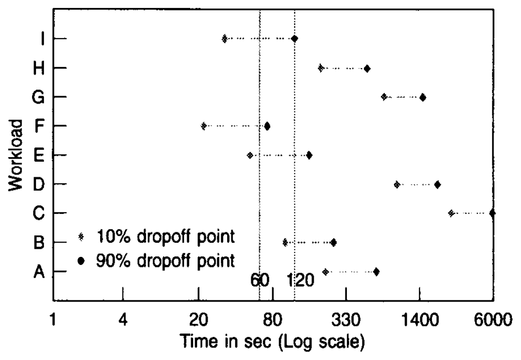


Figure 6: Throughput dropoff times for various workloads

Table 1: Characteristics of workloads in Fig. 6

Workload	Arr Rate	Locks/MIP	Cont Lev
A	288	10	0.3
B	288	20	0.3
C	2880	20	0.3
D	2880	40	0.3
E	288	10	1.0
F	288	20	1.0
G	2880	20	1.0
H	2880	40	1.0
I	288	40	0.3

that lock space sizes for both systems are the same. Then the throughput degradation for the system with the higher arrival rate can be expected to be less since transactions also possess fewer locks, which is the dominating effect as indicated by Eq. (7). Fig. 5 shows that in two systems with different number of nodes but identical total computing power and workload characteristics, the system with a larger number of nodes shows better post-failure behavior. This is in keeping with the conclusion drawn at the end of the previous subsection, since the initial fraction of unavailable locks is smaller for the system with more nodes. However, since all other parameters are the same for both systems, we expect that for time points where the fraction of unavailable locks is identical for both systems, the rate of throughput dropoff will also be identical. In other words, the throughput curve for the system with fewer nodes can be translated to the right to obtain the throughput curve for the system with the larger number of nodes.

Finally, Fig. 6 summarizes the post-failure behavior of various workloads, which are detailed in Table 1. The response times for each workload was fixed at 1 second and the workloads were run on 8 nodes each with a processing capability of 50 MIPS. The times for the throughput to drop off by 10% and 90% of the steady state value for each workload are plotted in the figure. It can be seen that workloads with low arrival rates (which imply high transaction path length) and large numbers of locks per MIP reveal potential problems in the form of rapid throughput dropoff following a failure.

Figs. 2 through 5 all indicate that the predictions of the analytic model and the results of the simulations match very well, thus validating the model. We may therefore dispose of the simulations and instead use the analytic model to predict the post-failure system behavior with confidence.

4 Post-Failure Behavior of System with Hot Locks

4.1 Analytic Model

Upto this point we have assumed a uniform lock access pattern which implies uniform access to all parts of the database. However, if accesses to various parts of the database are non-uniform, this implies that some locks are acquired more frequently than others on an average. We were interested in observing post-failure system behavior when a small fraction of the total number of locks was assumed to be hot, or accessed much more frequently than most locks. In order to distinguish hot locks from locks which were accessed relatively less frequently, we shall refer to locks belonging to the latter class as cold locks. We studied several different strategies governing the acquisition of hot locks. We assume that there are a fixed number of hot locks per transaction and that these are acquired after all cold locks are acquired. We outline the steps of developing the analytic model for the post-failure behavior for this strategy in order to provide an illustrative example. We have similarly derived analytic models for two other hot lock acquisition strategies - one in which hot locks are acquired before cold locks and one in which every lock has the same probability of being a hot lock. The derivations for the latter two strategies follow the same steps indicated here for the first strategy, and are not presented here in the interest of brevity.

We need to introduce the following new notation in addition to the notation introduced in Section 3

Cold locks per transaction = c
Hot locks per transaction = h
Cold lock space size = C
Hot lock space size = H

As before, we make the assumptions that $c \ll C$ and $h \ll H$. Let us now assume that at t time units following a single node failure, the fraction of unavailable cold locks is given by $u_{c,t}$ and the fraction of unavailable hot locks is given by $u_{h,t}$. The terms $u_{c,t}$ and $u_{h,t}$ then have meanings analogous to u_t in Section 3. By proceeding in a manner similar to the derivation of Eq. (4) we may establish the following system of ODEs governing the rate of growth of unavailable cold and hot locks following a failure

$$\dot{u}_{c,t} = \frac{E_{c,t}\lambda}{C} \quad (8)$$

$$\dot{u}_{h,t} = \frac{E_{h,t}\lambda}{H} \quad (9)$$

Here, $E_{c,t}$ and $E_{h,t}$ refer to the expected increase in the number of unavailable cold and hot locks due to a single transaction at time t. Now since we assume that cold locks are acquired before hot locks, we see that a single transaction can cause an increase in the number of cold locks by anywhere between 1 and $c-1$ by attempting to acquire an unavailable cold lock. The number of unavailable cold locks can increase by c if it subsequently attempts to acquire an unavailable hot lock. The expected increase, then, is given by

$$E_{c,t} = \sum_{k=1}^{c-1} k(1-u_{c,t})^k u_{c,t} + c(1-u_{c,t})^c\{1-(1-u_{h,t})^h\} \quad (10)$$

which simplifies to

$$E_{c,t} = (1-u_{c,t}) - c(1-u_{c,t})^c + \frac{(1-u_{c,t})^2\{1-(1-u_{c,t})^{c-1}\}}{u_{c,t}} + c(1-u_{c,t})^c\{1-(1-u_{h,t})^h\} \quad (11)$$

A transaction arriving at time t can cause the number of unavailable hot locks to increase by anywhere between 1 and $h-1$ if it acquires all its cold locks successfully (i.e, they are all available) and then acquires some available hot locks before attempting to acquire an unavailable hot lock. A transaction cannot cause the number of unavailable hot locks to increase by more than $h-1$ since this would then imply that it had acquired all the locks it needed, so that it could complete execution and release all acquired locks. The expected increase in the number of unavailable hot locks caused by a transaction at time t is then given by

$$E_{h,t} = (1-u_{c,t})^c \sum_{k=1}^{h-1} k(1-u_{h,t})^k u_{h,t} \quad (12)$$

which may be simplified to

$$E_{h,t} = (1-u_{c,t})^c[(1-u_{h,t}) - h(1-u_{h,t})^h + \frac{(1-u_{h,t})^2\{1-(1-u_{h,t})^{h-1}\}}{u_{h,t}}] \quad (13)$$

We may then substitute the expressions for $E_{c,t}$ and $E_{h,t}$ from Eqs. (11) and (13) in the system of Eq. (9) to obtain a system of ODEs which does not have a closed form solution, but may be iteratively solved by a suitable numerical method for solving systems of ODEs when initial conditions are provided. The expressions for the initial fractions of lost hot and cold locks may be obtained by reasoning in a manner similar to Section 3 and are as follows

$$u_{c,0} = \frac{cr\lambda}{CN} \quad (14)$$

$$u_{h,0} = \frac{hr\lambda}{HN} \quad (15)$$

Eq. (9) along with the the initial conditions provided by Eq. (15) may then be used in conjunction with an iterative solution technique for ODEs to predict the post-failure behavior of a system with hot locks. The throughput at time t ($T_{h,t}$) can then be computed as

$$T_{h,t} = (1-u_{c,t})^c(1-u_{h,t})^h\lambda \quad (16)$$

4.2 Simulation Results

The effect of hot locks was studied in two cases. Fig. 7 shows the post failure system degradation of two systems with identical cold lock contention levels. One system has no hot locks while 20% of the locks on the other are hot locks with a steady state contention level 10 times as great as the steady state contention level

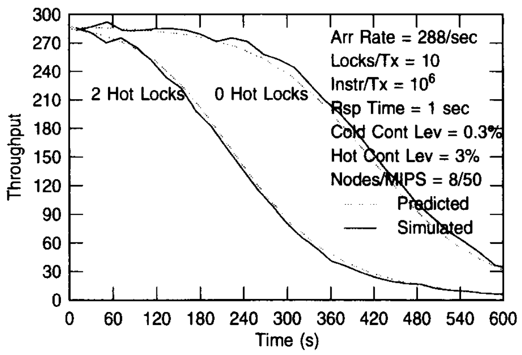


Figure 7: Effect of hot locks with identical cold lock contention levels

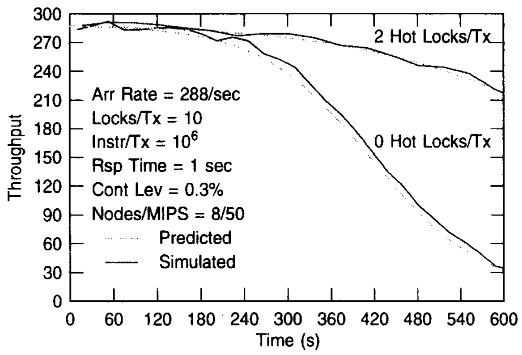


Figure 8: Effect of hot locks with identical overall contention levels

for cold locks. As expected, the system with hot locks performs significantly worse than the system without hot locks following a failure. In Fig. 8, the overall contention level has been kept constant for both the system with and without hot locks. For the system with hot locks, 20% of the locks are assumed to be hot locks with a contention level 10 times as great as the contention level for cold locks. For this system, the contention level for cold locks has been fixed at 0.11%, leading to an overall contention level of approximately 0.3%, the same as for the system without hot locks. We now observe that the throughput of the system with hot locks degrades more slowly. For the system with hot locks, only 20% of the locks are hot, and these have a contention level only 3 times that of the system without hot locks. Also, for this system, 80% of the locks are cold, with a contention level only a third that of the system without cold locks. The effect of low contention for the cold locks, which form the vast majority, dominates the behavior of the system with

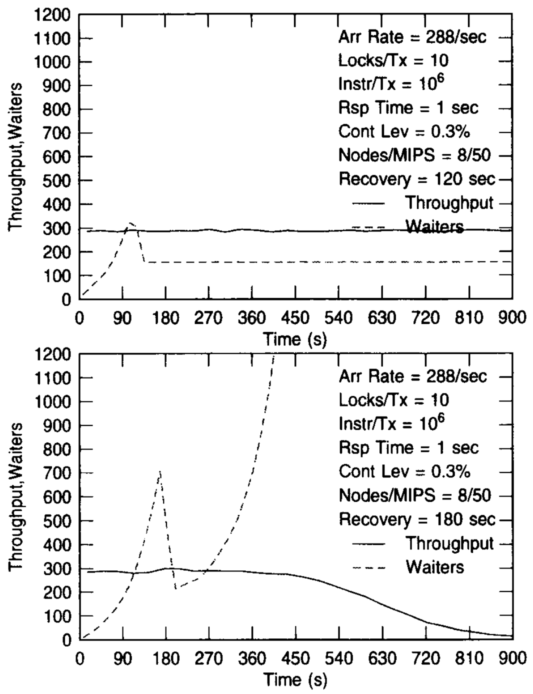


Figure 9: Throughput following failure and lock recovery for 0.3% contention

hot locks.

5 Behavior Following Lock Recovery

5.1 Simulation results

We felt that it would be interesting to study the post lock recovery phase of a database system with a single node failure. We were interested in observing how soon the system would be able to achieve steady state behavior by reducing the number of waiting transactions to steady state levels. We expected the time to reach steady state to be adversely influenced by large lock recovery times, since this would cause a large buildup of backlogged transactions. From Figs. 9 and 10 we observe that if the time to lock recovery is too large, enough transactions get backlogged to prevent the system from ever reaching steady state, i.e., the large number of backlogged transactions cause the system to thrash. At the time of lock recovery, if the fraction of the lock being held by waiting transactions is large, then any incoming transaction will almost certainly attempt to acquire a lock which is held by another transaction, causing it to wait until the lock has been released. The transaction, having caused the number of unavailable locks to increase further, will

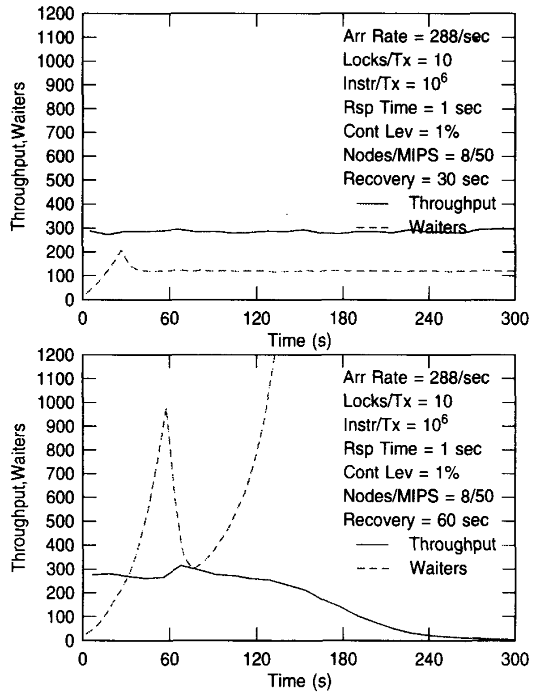


Figure 10: Throughput following failure and lock recovery for 1% contention

now have to join a queue of waiters. Another transaction will then be executed in its place with much the same effect. The number of unavailable locks will then continue to grow so that much of the CPU time will be spent in executing code to move transactions from active to waiter queues instead of actually executing transaction code, a classic example of thrashing. This phenomenon has also been observed to occur in a fault-free system when the degree of multiprogramming becomes too large, so that an incoming transaction has a high probability of experiencing lock contention [7].

5.2 Strategies for Avoiding Thrashing

Many simple strategies suggest themselves for dealing with the observed thrashing behavior but none is without cost. One needs to ensure that the number of waiters does not grow out of hand, so that at some point waiting or incoming transactions need to be aborted. Note that in our simulations we assumed an infinite buffer capacity for storing waiting transactions, while a real system will have memory limitations which will eventually prevent new transactions from being buffered. However, the plots in this section indicate that thrashing can begin to occur when the number of waiters reaches fairly modest levels, so that if one does not adopt aggressive transaction abort strategies following a failure, instead relying on physical memory limitations to limit the number of waiting transactions, by the time the lock recovery is complete the system may well be prone to thrashing. We suggest three alternative strategies for avoiding the post lock recovery thrashing behavior. One is to limit the total number of transactions in the system, including waiters, to a modestly large number, say a small integer times the expected steady state number of transactions in the system. This has the disadvantage that following a failure, the number of waiters can exceed this limit rather quickly, preventing the entry of further transactions into the system, causing throughput to drop to 0. Another solution is to avoid imposing any limit on the number of waiters in the post-failure period, and instead abort all waiters at the time of lock recovery (as opposed to aborting only the transactions waiting on lost locks). This allows all transactions which can execute during the pre lock recovery phase to complete, thus ensuring a higher throughput than the previous strategy. However, this may cause a large buildup of waiting transactions, so that the system may spend a long time aborting transactions immediately following lock recovery. The third strategy is to use the wait depth limiting scheme proposed by Thomasian et. al. [8]. This scheme aborts transactions which issue a request for any lock for which the number of waiters exceeds a threshold. This can also prevent uncontrolled build up of waiting transactions so that the system has a better chance of recovering to nearly its normal throughput rate following lock recovery. The problem here is to choose the wait depth level to be as large as possible without causing enough waiter buildup to cause the system to thrash following lock recovery.

6 Conclusions and Future Work

We have studied the post-failure behavior of a distributed database system executing transactions in parallel. We have shown through simulations and analytic modeling that throughput can be expected to drop off with varying degrees of rapidity for various workloads following a single node failure. In particular, we have shown that workloads with long transaction path lengths, moderately high locks per transaction and contention levels of 1% or greater exhibit very rapid throughput dropoff behavior, of the order of two minutes or less. We have also observed post-failure behavior of systems with hot locks and conclude that hot locks do not cause worse failure behavior when overall contention levels are the same as a similar system without hot locks. We have also shown that unless lock recovery is completed while the system is still operating close to its fault-free level, enough waiters may build up to cause the system to thrash following lock recovery. For such systems with potential problems following a failure, either the lock recovery procedure has to be designed to be quick enough to forestall disaster, or one of the methods suggested in Section 5 for aborting transactions in the post-failure phase should be adopted.

We have so far been unable to extend the analytic model to predict system behavior following lock recovery. This remains an open and interesting question, since the availability of such a model would enable the recovery manager of the database system to take early decisions on aborting transactions by foreseeing potential problems. It would also be useful to provide a formula for the number of waiting transactions and the level of the unavailable fraction of the lock space which could be expected to cause the system to thrash since this information could also be used to aid the recovery manager to effect quicker system recovery. We have also not made a detailed study of the factors which might influence the lock recovery process to take more or less time. Empirical evidence indicates that this time is typically of the order of a couple of minutes, but can vary with system parameters such as lock space size and workload characteristics. A detailed study of the lock recovery process to uncover the influence of these factors on the time to lock recovery as well as the post-failure behavior of the system would enable us to avoid the thrashing behavior observed here and design systems with higher sustained throughput following a failure.

References

[1] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Computing Surveys*, December 1983.

[2] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *CACM*, November 1976.

[3] E. Rahm, "Recovery concepts for data sharing systems," *Proc. 21st Int. Symp. on Fault Tolerant Comput.*, June 1991.

[4] K. S. Trivedi, *Probability & Statistics with Reliability, Queuing and Computer Science Applications.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.

[5] P. S. Yu, D. M. Dias, J. T. Robinson, B. R. Iyer, and D. W. Cornell, "On coupling multi-systems through data sharing," *Proc. IEEE*, May 1987.

[6] P. S. Yu and A. Dan, "Performance evaluation of transaction processing coupling architectures for handling system dynamics," *IEEE Trans. Parallel Distr. Systems*, February 1994.

[7] A. Thomasian, "Two-phase locking performance and its thrashing behavior," *IBM Research Report*, August 1991.

[8] P. A. Franaszek, J. R. Haritsa, J. T. Robinson, and A. Thomasian, "Distributed concurrency control based on limited wait-depth," *IBM Research Report*, May 1991.