Stochastic Analysis of Computer and Communication Systems
Hideaki Takagi (Editor)
Elsevier Science Publishers B.V. (North-Holland)

ISSUES IN MODELING LOCKING PERFORMANCE

Y.C. Tay

Department of Mathematics, National University of Singapore, Kent Ridge 0511, Republic of Singapore.

This expository article examines the issues that arise when modeling the effect of locking on the performance of transactions in a database system. Perhaps it will interest some performance analysis to see how issues similar to those they have encountered are addressed by others for a different problem; perhaps it will help some graduate student who is about to embark on a modeling project.

The issues discussed include the decoupling of data contention and resource contention, the role of simulation, the reliability of intuition, and the techniques for handling stochastic dependencies.

The article concludes with a suggestion that performance models be divided into two classes. The models in one class concentrate on studying the fundamental aspects, while those in the other concentrate on answering engineering questions. Such a differentiation in the role of performance models could remove some confusion over what to expect from these models, and how to use their results.

Contents

Y.C. Tay received a BSc from University of Singapore in 1980, and a PhD in Applied Mathematics from Harvard University in 1984.

1. Introduction.

A *database* is a collection of interrelated data. A *database management system* is a database together with a suite of programs for organizing, updating and querying the database. Typically, users access a database through application programs that run on top of the database management system [KS]. Examples of such applications are airline reservation and automatic teller systems.

Database management systems have three important features [B2]: *persistence* − if a program modifies some data, the changes remain after the program has terminated; *sharing* − more than one program can concurrently access the data; and *reliability* − the data must remain correct despite hardware and software failures.

In manipulating data, a program may cause the database to be temporarily incorrect. For instance, in transferring an amount of money from an account A to another account B, the database would be incorrect in the interval after the amount is deducted from A and before it is added to B (since that amount is missing from the total of the two accounts for the duration). The persistence of changes and this possibility of a temporary inconsistency lead to the requirement that either all changes made by the program up to its *commitment* (i.e. successful termination) is reflected in the database, or none at all. This is the concept of the *transaction*.

Since a database system allows several transactions to be active at the same time, the actions of transactions on shared data are interleaved. To prevent this interleaving from producing inconsistent data, there must be a *concurrency control* protocol to coordinate the transactions' actions. The concurrency control ensures that the effect of the interleaved actions on the database is *serializable*, i.e. it is equivalent to running only one transaction at a time. There is also a *recovery* protocal which guarantees that in the event of a failure, the data can be restored to a correct state, if it is corrupted.

The prevalent technique for concurrency control is *locking*: before a transaction can read or write on a piece of data, the transaction must set a readlock or a writelock (accordingly) on it. If another transaction already holds a lock on that data, and one of the two locks is a writelock, then there is a *conflict*, which must be resolved in some way. Conflict resolution is central to the problem of modeling how locking affects the performance of a database system.

There are two ways of resolving a conflict: (1) One of the two conflicting

transactions is *aborted* − all its changes to the database are undone, its locks are released, and it has to start all over. (2) The transaction that is trying to set the lock is *blocked* − it joints a queue of transactions waiting for the lock to be released. This blocking may lead to a *deadlock*, where several transactions wait for one another, unable to proceed. Such a situation must be detected and removed by aborting one of the transactions.

Concurrency control is just one factor among a myriad of other factors that influence system performance [S], but the complexity in modeling this one factor is already evident from the brief description above (more details can be found in [BHG]). The dependencies involved and the sheer size of the problem (gigabytes of data and dozens of concurrent transactions) are mind-boggling. An exact stochastic analysis would be completely intractable − one could not begin to write down the state space.

2. Details.

Performance related questions arise naturally once we look into the details of locking. For example, the division of secondary memory into pages makes it most convenient to lock a page at a time, even if a transaction's reads and writes refer to records, and a page contains more than one record. This brings up the question of *granularity*: How much data should be locked at a time? Intuitively, locking more would reduce the level of concurrency. On the other hand, it would reduce the number of locks a transaction needs, and thereby reduce the locking overhead. (Setting a lock requires a few hundred machine instructions.) How does granularity affect performance?

Moving to a more abstract standpoint, which is a better way of resolving a conflict: blocking the transaction requesting the lock, or aborting one of the conflicting transactions? In aborting a transaction, all the work that is already done by the transaction would be wasted, whereas in blocking a transaction, the work that is done by that transaction is conserved. It would thus seem that we should always block the requesting transaction, rather than abort one of the two transactions. Is this true?

Blocking can cause deadlocks. In the event of a deadlock, which transaction should we abort to break the cycle? One could, for instance, abort the transaction in the cycle that has consumed the least amount of resources. In general, a transaction in a deadlock may be blocking a few transactions, so one could also argue for aborting the transaction that is blocking the largest

number of transactions. There are several other possible criteria for choosing the victim. Which is optimum?

It is possible to avoid deadlocks altogether. Suppose a transaction is not allowed to begin execution if any of the locks it needs is already granted to some other transaction. It must wait till all those locks are available, get them, then begin execution; thereafter, it is not allowed to ask for more locks. This is called *static locking*, as against *dynamic locking*, where transactions get their locks as they are needed. Intuitively, static locking has the disadvantage that transactions tend to hold on to locks for a longer period than if they acquire the locks only when necessary. The advantage is that once a transaction gets its locks, it will not be blocked (so there will be no deadlocks) because it will not ask for more locks. Which is the better policy?

One could also prevent deadlocks by resolving every conflict according to some priority, such as the age of the transaction. We say a transaction is *younger* than another transaction if the former enters the system after the latter. Which of two transactions is younger could be determined if each transaction is given a *timestamp* upon entering the system. This timestamp is different for different transactions, and larger for younger transactions. There are two possible policies for using timestamps in deadlock prevention. In the *wound-wait* policy, if a transaction requests a lock that conflicts with one that is held by a younger transaction, the latter is aborted; otherwise, it waits. The other possibility is the *wait-die* policy: in this case, if a transaction requests a lock that conflicts with one that is held by another transaction, it waits for the lock if the other transaction is younger, and aborts otherwise. Both policies do not cause deadlocks, but each has a weakness: in wound-wait, a transaction wounds every younger transaction it conflicts with, while in wait-die, it must wait for every such transaction. Which is the better option?

Both wait-die and wound-wait base their decision between blocking and aborting on the timestamp. One could imagine other policies in which the decision uses a different basis. Suppose we agree not to disturb the transaction already holding the lock, and decide only to block or abort the requesting transaction. There is a whole spectrum of policies that balances blocking the requesting transaction against aborting it. We could, for instance, toss a biased coin (i.e. conduct a Bernoulli trial) to make the decision. By adjusting the bias in the coin, we can vary from consistently aborting the requesting transaction to consistently blocking it (unless that leads to a deadlock). Where is the optimum in this spectrum?

The wait-die and would-wait policies are basically locking policies, but they are at the intersection between locking policies and a whole class of policies that do not use locks, but manage concurrency control solely with timestamps. There are also policies that use neither timestamps nor locks; these policies keep track of the actions of each transaction on the data, and abort the transactions whose actions are interleaved with others in a way that my corrupt the data. In practice, however, only locking is used. Perhaps system designers intuitively believe that locking policies are superior to the others. Is this intuition correct?

Fundamental to the problem of locking performance is the fact that locking is necessary only because transactions are run in a multiprogramming fashion. With multiprogramming, there are two kinds of contention. There is the usual *resource contention* − e.g. queueing for the use of the processor and for I/O, and in the case of multiprocessor systems, contention over memories and buses. Added to this is *data contention* − conflicts over data that result in lock queues and transaction abortion. Each form of contention degrades system performance. What is the effect of each, and how do they interact?

The questions listed above are not exhaustive. There are many others concerning interaction with recovery [AD], multiple versions of data [CM], replicated data [GS], etc., but perhaps the ones listed suffice to illustrate the richness of the problem from the performance perspective.

3. Issues.

It is not the purpose of this article to answer the questions raised in the previous section − many are still open, anyway. Rather, the purpose is to examiner some of the issue that arise when we look for the answers. It follows that we do not attempt to survey the literature, or summarize the known results; such an attempt has already been made in [T3]. (For a survey of performance studies that includes timestamping and other forms of concurrency control, see [JS].) In particular, papers and results are mentioned only if they are relevant to the discussion at hand.

For simplicity, the discussion is restricted to exclusive locks, that is, there is a conflict whenever a transaction requests a lock that is already granted to another transaction. Where a claim is made here without substantiation, the underlying argument can be found in [T3].

Scope.

On the scope of a model, it would be nice to say that we first sit down and examine the issues, weed out those that are unimportant or intractable, then design a model that can handle what's left of the problem. In reality, one often begins with the simplest nontrivial approximate model that is tractable, and the scope is just what that model can handle. In the case of locking, this model is static locking. But static locking is not used in practice because a transaction may not know in advance which locks it needs (circumventing this may require the integration of static locking with another concurrency control algorithm [FRT]). Therefore, the performance analyst who presents a study of static locking immediately faces the possibility of losing the interest of the database system engineer.

(One should also bear the engineer's interest in mind when imposing assumptions on a model. Some models assume, in addition to static locking, that an arriving transaction that runs into a conflict is lost − it disappears from the system − rather than queue for the locks it needs. This assumption makes the model tractable, but also reduces its creditability: such an assumption may be acceptable and common for modeling telephone calls, but it is unreasonable for modeling transactions such as cash deposits at automatic tellers.)

Restrictions imposed by what exists technologically and commercially cut both ways: they make some models unrealistic, but they also naturally limit the scope of a model. We could, for instance, ignore the concurrency control protocols that are not based on locking, since they are not used commercially. With this restriction in scope, one can then design a model specifically for locking. However, as in any science, the performance analyst should transcend ephemeral restrictions and probe what is nonexistent [H1]. We should thus not only examine timestamping and other unimplemented protocols, but strive even for a model that can help use undestand concurrency control algorithms in general.

We see here a need to strike a balance between having a model that is close to what exists in practice, and accepting any model that improves our understanding. In the case of static locking, one could justify the model on the ground that it helps clear our intuition about locking behavior (see *Intuition*), but it would be hard to justify an n-th analysis of static locking, for any large n.

Type of Model.

What sort of model is appropriate? Here, we note that a major effect of locking is the introduction of queues for locks. Couple this with the success of using queueing networks to model computer systems, and the obvious candidate is a queueing network which has a queue for each lock, in addition to the usual queues for disks and processors. This seems like a natural extension of a proven model.

Unfortunately, one quickly encounters the problem of simultaneous resource possession − a transaction typically holds more than one lock. It is thus being 'served' at more than one queue, and may be waiting in some other queue. Such situations occur in non-database applications too; e.g. during I/O, a process is in possession of both the channel and the disk controller. However, the effect in such cases is minor enough to be approximated in some way [LZGS], while in the case of locking, the effect is pervasive and dominating. The results from such a model, even for static locking, are discouraging [GB].

This is not to say that queueing networks cannot be used in this context. There are successful models [H2, T5] for locking that are based on queueing networks, but the lock queues are trivial in that they are delay servers with service times calculated through an analysis of the data contention.

The most common model is the Markov chain, with the state space aggregated in some way to give an approximate but tractable analysis. (Data contention is analyzed while evaluating the transition rates between states.) There are several ways of making this aggregation. In the case of static locking, a state may specify only how many transactions are active and how many are blocked [PL]; this state space could be refined further to indicate how many transactions would be unblocked in the even of a commit [MW]. In the case of dynamic locking, a state may specify how many locks a transaction is holding, as well as the status of the transaction − active or blocked, and the residual time if blocked [CGM1]. Or, the blocked transactions could be split into those that are blocked by an active transaction and those that are blocked by a blocked transaction [SS]. Sometimes, the Markov chain is combined with a queueing network that models the resource contention [PL, RT].

There are also ad hoc models that are designed together with the analytic technique one has in mind. For example, time may be slotted [FR], or the transactions likened to be in a fluid [TSG].

The favored option in the literature is for a closed model. This is probably because one could then use the model to solve for the open case by using the usual decomposition argument [C2]. Such a decomposition is justifiable if the performance of the system is determined primarily by the interaction among the transactions, and secondarily by fluctuations in the input stream. It is not clear how true this is.

Some accounts of analytic models are prefaced with the usual arguments favoring such models over simulation models. Yet, the methodology adopted is often the same, in the following sense: The problem is considered solved once the model is reduced to a set of equations, and some computational procedure given for solving this set of equations; in many cases, the models' predictions are compared to simulation results to verify that they are reasonably accurate.

But what does the model tell us about the behavior of the system? The assumption seems to be that, if the engineer would like to know something about her system, she can feed in the parameters into that set of equations, solve them, and find out. If she wants to know the effect of a change in a parameter, for instance, she should then solve the system for various values of that parameter. This methodology is just like simulation, except that the simulator is a set of equations instead of a discrete event system. One might call it an *analytic simulation.*

(Such analytic models are common, not just in the area of locking performance, but in other areas as well − e.g. polling systems, multiple access protocols, finite buffer queueing networks [T1, T2, O1,P]. For polling models, how does response time depend on the order of the queues, assuming equal switchover times? For multiple access protocols, is there a characterization of the point where throughput begins to drop? For finite buffer networks, under what conditions would adding a buffer space be better than speeding up the server? These questions are not addressed by the analytic models proposed for those systems. Similarly, for the well-established separable networks, under what conditions would adding a server to a queue be better than speeding up the serves at that queue? For these networks, although there is an exact solution, this solution does not help us answer questions like that last one on the behavior of a network [T4].)

For an analytic model to be worthy of its name, its solution should itself yield to analysis. We should be able to deduce through the model the behavior of the system without having to compute a single numerical solution.

Parameters.

If the model is open, one of its parameters would be the input rate of transactions − which would also be the throughput − and the model must predict the response time. If the model is closed, then one parameter would be the number of concurrent transactions, N, and the model must predict the throughput. (Little's Law [K] notwithstanding, the response time does not follow immediately from N and the throughput, because of the complication caused by restarts; see *Tricks*.)

There are at least two parameters necessary for specifying transaction characteristics. One of them is the number of locks k needed by the transaction. This is taken to be a constant in some models. In reality, k may, say, depend on the values read by the transaction, and is thus not a constant. One could therefore specify some distribution on k; an obvious one is the geometric distribution, i.e. after acquiring each lock, the transaction decides with a fixed probability to commit. Unfortunately, this model behaves poorly if data contention is heavy. (We should be careful about adopting the usual exponential and geometric distributions in unfamiliar circumstances.)

The other parameter for the transactions is the time T between two lock requests, *if the transaction is run by itself*. The qualification is necessary since T depends on N, because of resource contention. The model should also specify a distribution for T; this distribution could conceivably depend on the number of locks the transaction is holding.

The last parameter necessary to complete the model is the number of *granules* D, where each granule corresponds to a lock. For example, if data is locked one page at a time, then a granule is a page. Strictly speaking, one must also specify the *access distribution*, that is the probability that a transaction will access a particular granule. If we assume the distribution is uniform, then this probability would be $1/D$. Thus, D is only a minimal specification for the access distribution.

Performance Measures.

As in any performance model, three important performance measures are throughput, response time, in the case of open models, degree of concurrency N. In the context of concurrency control, there are two other obvious measures, namely probability of conflict and probability of deadlock.

The probability of conflict could be per lock request, or per transaction.

The latter refers to the probability that a transaction will encounter a conflict during its execution. Both are uninteresting measures.

The probability of conflict per request does not adequately measure the level of conflict among the transactions, since the latter depends on k (the number of locks per transaction). For instance, long transactions may suffer a high probability of running into a conflict during its execution, yet have a very low probability of conflict per request. A better measure of the level of conflict would be the ratio of N_a, the number of active (i.e. non-blocked) transactions, to N. Moreover, a small error in predicting the probability of conflict per request could lead to a large error in predicting the throughput, since that small error is repeated for each lock acquired by the transaction. The probability of conflict per transaction, on the other hand, does not indicate how often a transaction encounters a conflict.

Similar remarks apply to the probability of deadlock. One added difficulty is that this probability is very small, thus making it difficult to measure and predict accurately. A better measure regarding deadlocks would be the ratio of deadlock rate to throughput, i.e. the number of restarts per completion.

There are other appropriate means of measuring the performance of a concurrency control protocol. Some are specifically for comparing alternative protocols. For example, one could compare two protocols by considering the set of possible interleavings of transactions that each protocol permits [KP] – this is a highly theoretical measure. In general, the performance analyst has the prerogative to propose whatever measure she thinks is of interest, but her model should also provide the commonly used measures, so that others can make comparisons. Otherwise, we may be left with contradictory claims based on different measures, with no common measure to help resolve the contradiction.

The most important unresolved issue concerning performance measures is the computational requirements of a protocol. If a protocol provides a higher throughput than another protocol, but requires more computing power, is it 'better'? As yet, there are no proposed measures for the resource requirements of a given concurrency control algorithm.

Resource Contention.

There is a close interaction between data and resource contention. For instance, intense data contention can cause many transactions to be blocked,

thereby leading to more context-switching and swapping, both of which add to the resource contention. On the other hand, looking at the interaction from a higher level, if more transactions are blocked, then there is less demand on processing time and disk I/O, so resource contention is reduced. How should these two forms of contention be modeled?

One could point to the complexity of their interaction as a justification for a simulation or experimental model. This is a solid justification, and there is a clear case for such studies. Nonetheless, one must also attempt to analyze individually each form of contention, as well as their interaction, and this calls for modeling them separately.

Some models simplify the issue by considering only the data contention problem. One could argue that the modeling effort is aimed at studying the performance effect of the concurrency control, regardless of the level of resource contention; but this argument is weak, considering the feedback effect of resource contention on data contention. Other models incorporate both by having a model for each, and combining them in some way. Typically, resource contention is modeled through specifying a function for the inter-request time T. This function could be evaluated through a queueing network model of the computer system [PL, T5], or specified more abstractly as a function of N_a, the number of active transactions, with N_a either as a random variable [MW], or as an average. There is an important difference between the latter two approaches.

By using the average value of N_a, it is possible to solve the data contention and resource contention aspects independently, then integrate the two. We call this *decoupling*. This may not be possible if N_a is a random variable: in [MW], solving the data contention at a given value of N involves a recursion that incorporates both data and resource contention for other values of N, so data contention cannot be solved independently of resource contention.

One advantage of decoupling is that it is a powerful technique for performance analysis. Some of the conflicting results in the literature, for example, were first reconciled by decoupling data and resource contention, and analyzing their interaction; the simulation models that handle both simultaneously failed to discover the interaction between the two, until later. Decoupling also provides a framework for reasoning informally to an engineer about system behavior. What one has to worry about is whether separate modeling in general, and decoupling in particular, removes some important synergistic effects from the model.

Deadlocks.

Deadlock detection and resolution is a fascinating problem, as evidenced by the fact that papers on this problem continue to appear regularly. In the context of locking, one's intuition is that it is also a serious performance problem, since deadlocks lead to restarts that mean wasted and additional work. In fact, one of the earliest work on locking performance concerns deadlock resolution.

The truth is, the effect of deadlocks on transaction performance is minimal. Both empirical and simulated data show that deadlocks are rare for database locking, and that they have a second order effect on performance compared to blocking.

This fact points out a need to be hard-headed. Deadlocks may fascinate, but if performance is dominated by blocking, then they should be of secondary concern. Issues like when to check for deadlocks and which transaction to abort in a deadlock are, from the perspective of overall performance, non-problems. Worse, studying such problems can lead one to pitfalls.

Consider the problem of which transaction to abort in breaking a deadlock. It is known that most deadlocks involve only two transactions. To generate big enough cycles in a simulation study of the problem would require driving the system into a thrashing state, that is where throughput drops as the number N of transactions increases.

Although the impact of deadlocks may be small, that does not absolve one from estimating its magnitude. Experience shows that getting a good estimate of the probability of conflict is not too difficult, but doing the same for the probability of deadlock is much harder. Estimating the latter involves analyzing he way transactions block one another. Consider: A transaction may hold more than one lock; there may be a few transactions waiting for each of these locks; these blocked transactions may in turn block others in a similar manner. A deadlock occurs when the transaction at the top of this hierarchy requests a lock held by some transaction lower down in the hierarchy. (A detailed analysis for even just two transactions is difficult [R].)

The accuracy of approximation methods in the literature for the deadlock rate is unimpressive. Perhaps estimating the probability of deadlock is inherently difficult, since the observed variance in the deadlock rate is usually large.

Nonuniform Access.

Almost all performance models assume that a transaction's lock requests are uniformly distributed over the database. This is an unrealistic assumption. The large volume of data in a database usually requires indices to guide a transaction to the data it needs, and these indices are more heavily accessed than the rest of the database. (Such heavily accessed areas are called *hot spots.*) Also, transactions sometimes scan some section of the database in a sequential manner, looking for some piece of data. Access is clearly not uniform in that case.

From the beginning, it was recognized that nonuniformity in the access distribution should be modeled in some way. An early model [MK] of hot spots assumes, say, 80% of the accesses are directed at 20% of the database. A regression analysis [LN] of simulation data under this model showed that the performance is similar to that under uniform access over a smaller database. This result probably provided an excuse for staying with the uniformity assumption; intuition indicates that an arbitrary distribution would be intractable, and there are enough problems for the model already. But this intuition may be wrong.

It has been shown that, surprisingly, one can assume an arbitrary distribution in the case of static locking, and still calculate the probability of conflict in polynomial time [ZBS,SY]. (The trick used is similar to Buzen's convolution method [B3].) In the case of dynamic locking, arbitrary distributions can be tackled with a novel technique called *data flow balance* [HZ]. These results suggest that nonuniform distributions may not be as hard to handle as one might expect.

Distributed Systems.

By a distributed system, we mean the database is spread over several sites, each with its own processor(s). Data could be replicated, i.e. multiple copies of data may exist at different sites, and a transaction may have to access data from more than one site, or even migrate from site to site.

Parallel systems, where several processors share memory, are excluded from the above category. From the point of view of concurrency control, one could model parallel systems as a centralized system by incorporating, say, multiple servers, semaphores and hardware spin locks in the resource contention model.

There have been several simulation studies on the performance of distributed locking, but few analytical models. Perhaps this is due to the complications brought by new issues. One of them is communication cost. It is possible that this could, in some way, be separated from data contention, just as resource contention could. Another issue is recovery. Since a transaction involves multiple sites, a transaction must be aborted if one of these sites fail. The commitment process therefore requires rounds of message exchange among the sites. Meanwhile, the locks cannot be released. Modeling the commitment process may require a structural change to any performance model designed for centralized locking.

One can get a feel for the complications involved by considering the case of replicated data. A read could be satisfied by any copy of the data, and a write must be installed in all copies of the data. The performance of a transaction is therefore affected by which copy of the data is chosen for a read, and how a write is propagated to all copies. Even if we ignore the requirement of enforcing correct interleaving of operations, that is a research problem in itself [C1].

In contrast to the situation with locking, there is a significant number of analytical models for distributed timestamping (e.g. [B1, CGM2, GS, KKM, L]), which is curious, since centralized timestamping is rarely studied and still not well understood.

Simulation.

There are two common mistakes in simulation studies of locking. One is in attributing some performance degradation to restarts when blocking is, under the given conditions, the dominating influence. Another is a failure to delimit the range of the parameters; such a limit exists because of thrashing (see *Back of an Envelope.*) Both mistakes are avoidable: measuring the effect of both blocking and restarts would prevent the first mistake, and a preliminary search for some constraint on the combination of parameter values would prevent the other.

A slip in making both checks could still lead to mistake. For example, the effect of restarts grows rapidly after thrashing begins. Thus, one may observe some phenomenon, measure the underlying effect of blocking and restarts separately, and correctly conclude that the phenomenon is caused by the restarts, without checking if thrashing has occurred. In this example, one may be studying a phenomenon that occurs only after thrashing begins

– an effort that is of questionable value. Enforcing and coordinating all such checks in a simulation of a complicated system could be tedious.

One of the strongest arguments for simulation models is that they allow us to incorporate more details than is possible with an analytic model, thus reducing the number of assumptions. One could make the simulations even more realistic by using trace data from a real system handling a stream of real transactions [PR]. However, the analyst would have to deal with any peculiar features in the system that may influence the results – for example, some real systems allow some transactions to run in a non-serializable manner. Also, the reference string in the trace data may have been perturbed by the scheduler in some ways (the timing and order of the data accesses may have been altered). If a simulation is run with the perturbed reference string on a scheduler that is different from the one in the real system, how would the results relate to a simulation run with the original stream of transactions?

In any case, the specificity of simulations (with or without trace data) contributes to their weakness. Simulation models (including those based on Petri nets [O2]) for a sophisticated problem like locking usually contain several 'magic numbers' that are not changed throughout the simulations; e.g. the number of disks and processors, the disk access time, the time-sharing quantum, etc. Even if these numbers are generally acceptable, are the results insensitive to changes in these numbers? Such an assumption would be ill-advised [TGS, ACL]. Moreover, even if the numbers used are based on the state-of-the-art technology, what do the results say about systems that may be built five years later?

For computer science to be a science, its results must withstand changes in technology. Such robustness is another motivation for decoupling resource and data contention. It allows an analysis of data contention that is independent of the software and hardware underlying the system, yet permits us to draw conclusions about how the protocol performs with different levels and modes of resource contention. This is analogous to the idea of portability of software among, say, variants of an operating system.

Despite the above remarks about simulation, it has a role in any major modeling effort, a role that is larger than the customary validation of model predictions, For example, a simulator can help the researcher decide on a choice of approximations for her analysis. It can also test whether some strategy for making an approximation will work. To give another example, the researcher may – with some effort – be able to remove some assumption underlying *both* the simulator and the analytic model; in this case, running

the simulator with and without the assumption can help her decide if the difference is going to be worth the effort of removing the assumption from the analysis. In short, a simulator can be a very useful guide for an analytic model.

Intuition.

Intuition is unreliable when dealing with systems that have complex interactions. We give three examples to illustrate this point.

One very simple conflict resolution policy would be to restart a transaction if a lock it needs is already held by another transaction. This is the so-called *no-waiting* policy. This policy is not used in practice, perhaps because intuition tells us that it would perform badly: why restart the transaction (thus losing what it has done so far) when it costs nothing to just block it? In fact, there *is* a cost in blocking. When a transaction is blocked, it is not 'using' the locks that it already holds, while it prevents other transactions that need those locks from getting them and making progress. Even if one could immediately see this antisocial effect of locking, it is still difficult to believe that it could be worse than the draconian no-waiting policy. But it could.

To give another example, recall that static locking is where a transaction does not begin execution until it gets all the locks it needs. In contrast, dynamic locking is where a transaction gets a lock only when it needs the lock. In both cases, the locks are held till the end of the transaction. Intuitively, static locking is a loser, since the transaction is getting locks before it needs them, thus reducing the amount of concurrency. This reasoning turns out to be superficial. There are conditions under which locks are held longer under dynamic locking than under static locking.

The third example concerns thrashing. Intuition may tell us that the drop in throughput is caused by deadlocks and restarts, i.e. the system has been driven to a point where it is spending too much of its time redoing deadlocked transactions, thus losing its throughput. Such an intuition would be consistent with what we have learnt about thrashing in operating systems [DKLPS], but is wrong.

These three examples are not hypothetical − they can be found in the literature. How could one's intuition go wrong?

In the example of the no-waiting policy, there are at least two reasons

why the intuition is partly correct. One of them is the confusion between resource and data contention, and the other involves looking at the appropriate performance measure. An excessive amount of restarts is had only because it causes intense contention for resources; as a means of resolving data contention, restarts are not worse than blocking in their effect on performance, if we factor out the resource contention. Also restarts are bad for response time, but not necessarily bad for throughput. In a sense, restarts allow the system to pick from the input stream those transactions that can get through with no hassle, and delay the rest till the conflicting transactions have left.

In the second example, there is a critical difference between static and dynamic locking: a transaction holds no locks when it is blocked under static locking, while a blocked transaction under dynamic locking may be holding locks. Each time a transaction is blocked under dynamic locking, the period over which it holds its locks is lengthened. it is thus possible that transactions end up holding on to locks longer than if they had used static locking, so it is not clear that static locking is really a loser.

In the third example, locking causes thrashing not through deadlocks, but when too many transactions are tied up waiting for locks that they need but cannot get. The number of active transactions then drops, bringing down with it the throughput. The picture of frenzied activity for thrashing in operating systems leads one to expect a similar picture for database systems, when in fact the appropriate one is of a large number of idling transactions waiting for a few transactions to finish, so they can proceed. (This picture is not entirely correct either: if half the transactions are blocked, the system is already thrashing.)

Intuition is based on accumulated knowledge, and is reliable as long as we stay within the domain where that knowledge is accumulated. When we move into a new area, where the mechanisms and interactions (such as the cascading effect of blocking) are unfamiliar, we should guard against our own intuition. Even within a familiar domain, intuition may be valid only over a limited range, and extrapolation is hazardous: In this first example above, an intuition that is correct under normal conditions fails when resource contention is sufficiently low, and a similar failure happens in the second example when data contention is high.

Tricks.

Perhaps the biggest issue in a performance model is how the dependencies

should be handled. Some of the tricks used − such as state space aggregation and decoupling − have already been mentioned. We now outline other techniques that have been used to tackle the dependencies.

One widely used technique is hierarchical decomposition, where the transaction processing system is analyzed as a closed subsystem, i.e. every committed transaction generates a new transaction. This subsystem can then be used as a load-dependent queue in a larger system that includes, say, the source of the transactions, or in an analysis of the stability of the system [C2]. Another common technique is to use an average value of a variable when the value of the variable is called for. For example, to compute the probability of a lock conflict, one might use N_L/D, where N_L is the *average* number of locks held by the transactions (assuming uniform access). Even when average values are used consistently throughout the model, the results are surprisingly accurate. The surprise stems from the fact that the performance measures are nonlinear with respect to the variables once conflict becomes significant.

One of the most difficult dependencies is caused by deadlocks. Let N_i be the average number of transactions holding i locks. Then N_i depends on i because the more locks a transaction holds, the more likely it is that the transaction must be aborted because it encounters a deadlock (because it is blocking more transactions), so N_i tends to decrease as i increases. This dependency of N_i on i is very hard to analyze. Fortunately, one could wave it away by pointing out the dominant effect of blocking, which can cause thrashing to occur even before the deadlock rate exceeds one percent of the throughput. With this handwaving, N_i becomes independent of i. Note that the difficulty here is not with handling the restart itself, but in analyzing the deadlock that causes the restart. In the no-waiting policy, every conflict causes a restart, so N_i depends critically on i. Yet, the policy can be accurately modeled.

A restarted transaction, in reality, would probably ask for the same sequence of locks that led to the restart. No model proposed so far has been able to handle this fact. All models assume, in one way or another, that a restarted transaction resamples the sequence of locks it needs. (This is similar to Kleinrock's "independence assumption" that the length of a message is resampled when the message leaves one node and arrive at another in a packet-switching network [K].) There is a way of justifying this.

Observe that if an aborted transaction is restarted immediately, it is likely that the transaction it conflicted with is still in the system, so the conflict could be repeated. There is therefore reason for delaying the restart of

an aborted transaction, so as to let the conflicting transaction leave the system first. In the meantime, a new transaction is introduced into the (closed) system in place of the aborted transaction, so it *looks* like the aborted transaction is restarted with a new sequence of lock requests. By the time the aborted transaction is restarted, the delay has weakened the dependency to an extend that it also looks like a new transaction, one hopes. This delay adds a complication to the response time, but it is easy to handle.

Having waved away deadlocks, the major difficulty left is in analyzing the waiting time for a lock. This involves looking at the way transactions block one another. Is the blocked transaction first in queue for the lock? Is the transaction holding the lock itself blocked? If so, the same questions can be iterated. One way [T5] of getting around this difficulty is to approximate the waiting time as half the response time of a transaction. Another way [CGM1] is to model the waiting time by having a blocked transaction repeat the request (after waiting for a random interval) for the lock, up to a specified maximum number of repetitions; the transaction restarts if it fails to get the lock after the maximum number of repeats. Unfortunately, this model in some sense reduces the locking policy to the no-waiting policy. This is an example of an approximation that unwittingly removes an important aspect of the system that is being modeled.

Back of an Envelope.

One could see from the preceding discussion the complexity in modeling locking. Yet, might there be some quick-and-dirty, 'back-of-an-envelope' calculation that works? Surprisingly, there is [GHOK]. The following is a variation of the argument.

Recall that each transaction requires k locks. Assuming deadlocks are rare, a transaction in the system would be holding an average of $k/2$ locks. There are N transactions in the system, so they hold $kN/2$ locks altogether. The probability of a conflict per lock request is therefore $kN/2D$, where D is the number of granules (assuming uniform access). There are k requests, so a transaction is expected to suffer $k^2N/2D$ conflicts. Let R be the response time of a transaction, i.e. the time between when a transaction begins and when it commits. Whenever a transaction is blocked, it must wait for a time of $R/2$ to get the lock. The total time a transaction spends waiting for locks is therefore $k^2NR/4D$, on average. Now, the average time between two lock requests is T, if there is no conflict. Assuming the first lock request is

preceded, and the last lock request is succeeded, by a similar period of T, the total time for a transaction is $(k+1)T$, if there are not conflicts. It follows that

$$R = (k+1)T + \frac{k^2 N\ R}{4D}.$$

Solving for R and using Little's law, we get

$$\text{throughput } = \frac{N}{(k+1)T}\left(1 - \frac{k^2 N}{4D}\right).$$

This is precisely the throughput approximation in [TGS] for the case where data contention is low.*

There are some hidden assumptions and approximations in this calculation. For example, there is some fudging with the waiting time of $R/2$: no consideration is given to whether the lock is held by a transaction that is itself blocked, nor to whether the blocked transaction is first on the queue for that lock. Even so, this simple calculation is remarkably accurate. Differentiating (with respect to N) the above expression for throughput, we even get a prediction that data contention will cause throughput to drop at the point where $k^2 N/D = 2$. There is a similarly simple and accurate calculation for static locking.

We should not be overly impressed by the difficulty of a problem. Before delving into the mess, it is useful to do some quick calculation to serve as a guide. The above analysis cannot go very far without elaborating on the model and investigating its assumptions, but it would be an excellent starting point.

4. Conclusion: Usefulness.

There is often a gap between what the performance analyst perceives to be his mission in a modeling effort, and what the system engineer expects. The analyst usually considers his job done when he has solved the model in some way, and verified that the solutions agree with simulation results. The engineer, on the other hand, may ask: But is the model *useful*? How can the model help in the engineering process?

As input, the engineer could probably specify the transaction rate the system must support, estimate the number of locks each transaction needs,

* There is an error in equation (12.4) of [TGS]; the average of 1/3 and 1/6 is 1/4, not 1/4.5.

and a great deal of other details. Given these, can the model predict if there will be serious contention for locks (never mind predicting the throughput) in the system? If so, would it help if she changes the granularity of the locks from page-level to record-level, bearing in mind that such a change would significantly increase the difficulty and the time needed to maintain reliability? If that indeed helps, how much further can the system be pushed with record-level locking? These are examples of some practical questions that a model might be expected to answer. None of the models in the literature can adequately cope with questions of this sort.

This gap between what a model offers and what is demanded of it corresponds to the gap between two classes of models. One class contains those models that help us understand in some general fashion the behavior of a system. Their usefulness lie in identifying the critical issues (e.g. establishing the dominant effect of blocking), improving the collective intuition (e.g. clarifying how restarts affect performance), and providing paradigms for discussing the issues (e.g. separating resource and data contention). All these contribute towards establishing a science for these systems. One might call them *academic* models. Results from academic models can help a system designer understand her system qualitatively, but they cannot help her quantitatively. The amount of details omitted and the number of assumptions made to render such a model tractable would make any quantitative claim suspicious.

On the question of granularity, for instance, academic models can only offer a plot of how performance varies as the granularity is continuously varied, and examine the cause behind the variation. But granularity does not, in reality, vary over a continuous spectrum. As the above questions illustrate, other consideration may restrict the choice to just, say, between two levels of granularity. To decide on such a choice would require a model from another class. This other class of models would be tailored specifically for the problem at hand, and incorporate the relevant magic numbers (what is the disk latency?) and details about the implementation (is there a B-tree?) and the computing environment (does processor scheduling use priorities?). Because the scope is more restricted, the model can incorporate such details without becoming intractable. Such models would be very useful for understanding a particular aspect (e.g. granularity) of the particular system being modeled, and help in the design decisions. One might call these *commercial* models. They are the ones that should be expected to be accurate in their quantitative assessment of the systems that they model. However, they should not be used as the basis for general claims because their results may be specific

to the systems they model so closely.

We should certainly try to close the gap between the two classes, wherever it is possible to do so. But a gap will always exist, and bearing this in mind should help us avoid any confusion over what to expect from a model, and how to use its results.

Acknowledgement.

Many thanks to Phil Bernstein, who posed the questions concerning granularity, and Hideaki Takagi, who offered me this opportunity to discuss these issue. Their comments on a draft of this article, as well as those from Alex Thomasian and Ted Johnson, also helped me in the revision.

References.

[ACL] R. Agrawal, M.J. Carey and M. Livny. Models for studying concurrency control performance: alternatives and implications. *ACM Transactions on Database Systems 12, 4* (Dec. 1987), 609-654.

[AD] R. Agrawal and D.J. DeWitt. Integrated concurrency control and recovery mechanisms: design and performance evaluation. *ACM Transactions on Database Systems 10, 4* (Dec. 1985), 529-564.

[B1] F. Baccelli. A queueing model of timestamp ordering in a distributed system. In *Performance '87*, P.J. Courtois and G. Latouche (eds.). North-Holland, Amsterdam, 1988, 413-431.

[B2] F. Bancilhorn. Obect-oriented database systems. In *Proc. ACM Symposium on Principles of Database Systems*, (Austin, Texas, March 1988), 152-162.

[B3] J.P. Buzen. Computational algorithms for closed queueing networks with exponential servers. *Commun. ACM 16, 9* (Sept. 1973), 527-531.

[BHG] P.A. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Massachusetts, 1987.

[C1] R.G. Casey. Allocation of copies of files in an information network. In *Proc. AFIPS Spring Joint Computer Conference* (1972), 617-625.

[C2] P.J. Courtois. Decomposability, instabilities, and saturation in multiprogramming systems. *Commun. ACM 18, 7* (July 1975), 371-377.

[CGM1] A. Chesnais, E. Gelenbe and I. Mitrani. On the modeling of parallel access to shared data. *Commun. ACM 26, 3* (Mar. 1983), 196-202.

[CGM2] W. Cellary, E. Gelenbe and T. Morzy. *Concurrency Control in Distributed Database Systems.* North-Holland, Amsterdam, 1988.

[CM] M.J. Carey and W.A. Muhanna. The performance of multi-version concurrency control algorithms. *ACM Transactions on Computer Systems 4, 4* (Nov. 1986), 338-378.

[DKLPS] P.J. Denning, K.C. Kahn, J. Leroudier, D. Potier and R. Suri, Optimal multiprogramming. *Acta Informatica 7, 2* (1976), 197-216.

[FR] P.A. Franaszek and J.T. Robinson. Limitations of concurrency in transaction processing. *ACM Transactions on Database Systems 10, 1* (Mar. 1985), 1-28.

[FRT] P.A. Franaszek, J.T. Robinson and A. Thomasian. Access invariance and its use in high contention environments. IBM Research Report RC14665, Yorktown Heights (June 1989).

[GB] B.I. Galler and L. Bos. A model of transaction blocking in databases. *Performance Evaluation 3*(1983), 95-122.

[GHOK] J. Gray, P. Homan, R. Obermarck and H.F. Korth. A strawman analysis of the probability of waiting and deadlock in a database system. Tech. Rep. RJ3066, IBM Research, San Jose (Feb. 1981).

[GS] E. Gelenbe and K. Sevcik. Analysis of update synchronization for multiple copy data bases. *IEEE Transactions on Computers 28*, 10 (Oct. 1979), 737-747.

[H1] J. Hartmanis. Observations about the development of theoretical computer science. *Annals of the History of Computing 3*, 1 (Jan. 1981) 42-51.

[H2] C.S. Hartzman. The delay due to dynamic two-phase locking. *IEEE Transactions on Software Engineering 15*, 1 (Jan. 1989), 72-82.

[HZ] M. Hsu and B. Zhang. Modeling performance impact of hot spots. Technical Report TR-08-88, Aiken Computation Laboratory, Harvard University (April 1988).

[JS] T. Johnson and D. Shasha. A survey of performance analyses of database concurrency control algorithms. Manuscript (May 1988).

[K] L. Kleinrock. *Queueing Systems, Vol. 1*: Theory. John Wiley, New York, 1975.

[KKM] F. Kamoun, L. Kleinrock and R. Muntz. Queueing analysis of the ordering issue in a distributed database concurrency control mech-

anism. In *Proc. International Conference on Distrbuted Computing Systems* (Paris, France, April 1981), 13-23.

[KP] H.T. Kung and C.H. Papadimitriou. An optimality theory of concurrency control for databases. *Acta Informatica 19*, 1(1983), 1-11.

[KS] H.F. Korth and A. Silberschatz. *Database System Concepts.* McGraw-Hill, New York, 1986.

[L] V.O.K. Li. Performance models of timestamp-ordering synchronization algorithms in distributed databases. *IEEE Transactions on Computers 36*, 9(Sept. 1987), 1041-1051.

[LN] W.K. Lin and J. Nolte. Performance of two phase locking. In *Proc. Berkeley Workshop on Distributed Data Management and Computer Networks* (Berkeley, CA, Feb. 1982), 131-160.

[LZGS] E.D. Lazowska, J. Zahorjan, G.S. Graham and K.C. Sevcik. *Quantitative System Performance.* Pretice-Hall, New Jersey, 1984.

[MK] R. Munz and G. Krenz. Concurrency in database systems − a simulation study. In *Proc. ACM SIGMOD International Conference on Management of Data* (Toronto, Canada, Aug. 1977), 111-120.

[MW] R.J.T. Morris and W.S. Wong. Performance analysis of locking and optimistic concurrency control algorithms. *Performance Evaluation 5*, 2 (May 1985), 105-118.

[O1] R.O. Onvural. Closed queueing networks with finite queues. This volume.

[O2] M.T. Ozsu. Modeling and analysis of distributed database concurrency control algorithms using extended Petri net formalism. *IEEE Transactions on Software Engineering 11*, 10 (Jan. 1985), 1225-1240.

[P] H.G. Perros. Open queuing networks with blocking. This volume.

[PL] D. Potier and Ph. Leblanc. Analysis of locking policies in database management systems. *Commun. ACM 23*, 10 (Oct. 1980), 584-593.

[PR] P. Peinl and A. Reuter. Empirical comparison of database concurrency control scheme. In *Proc. International Conference on Very Large Data Bases* (Florence, Italy, Oct. 1983), 97-108.

[R] A. Reuter. An analytic model of transaction inteference in database systems. Research Rep. 68/83, University of Kaiserslautern, 1983.

[RT] I.K. Ryu and A.Thomasian. Analysis of database performance with dynamic locking. *J. ACM*, to appear.

[S] K.C. Sevcik. Data base system performance prediction using an analytic model. In *Proc. International Conference on Very Large Data Bases* (Cannes, France, Sept. 1981), 182-198.

[SS] A.W. Shum and P.G. Spirakis. Performance analysis of concurrency control methods in database systems. In *Performance '81*, F.J. Kylstra (ed.). North-Holland, Amsterdam, 1981, 1-19.

[SY] M. Singhal, and Y. Yesha. A polynomial algorithm for computation of the probability of conflicts in a database under arbitrary data access distribution. *Information Processing Letters* 27 (1988), 69-74.

[T1] H. Takagi. Queuing analysis of polling models. This volume.

[T2] Y. Takahashi. CSMA Protocols. This volume.

[T3] Y.C. Tay. *Locking Performance in Centralized Databases*. Academic Press, Florida, 1987.

[T4] Y.C. Tay, An approximate analysis of some product-form queueing networks. Manuscript, 1988.

[T5] A. Thomasian. An iterative solution to the queueing network model of a DBMS with dynamic locking. In *Proc. Computer Measurement Group Conference* (San Diego, CA, Dec. 1982), 252-261.

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

[TSG] Y.C. Tay, R. Suri and N. Goodman. A mean value performance model for locking in databases: The no-waiting case. *J. ACM 32*, 3 (July 1985), 618-651.

[ZBS] J. Zahorjan, B.J. Bell and K.C. Sevcik. Estimating block transfers when record access probabilities are non-uniform. *Information Processing Letters 16* (1983), 249-252.