Performance Analysis of Optimistic Concurrency Control Schemes with Different Rerun Policies

Philip S. Yu and Daniel M. Dias
IBM T. J. Watson Research Center, Yorktown Heights, NY 10598

Abstract
Concurrency Control (CC) is critical to the performance of transaction processing systems. Optimistic CC (OCC) offers an alternative to locking but is vulnerable to resource contention due to transaction aborts. With sufficient buffer, data blocks referenced by aborted transactions can be kept in memory and be available for access during rerun, thus greatly reducing the abort probability during rerun. Consequently, the pure OCC scheme, which only aborts a transaction at its commit time, can do better than aborting transactions as soon as conflict is detected (broadcast OCC). To further exploit this phenomenon, hybrid CC schemes are devised which employ a different CC scheme to handle rerun transactions. These include switching to static or dynamic locking (static and dynamic hybrid OCC schemes, respectively) or to broadcast OCC during rerun, while doing pure OCC for the first run. In the high data contention environment, where locking is inferior to OCC, we find that the static and dynamic hybrid OCC can do better than OCC. The elimination of IO during rerun reduces the data contention faced by the rerun transactions and makes locking preferable to OCC for rerun transactions. Substantial performance improvement can thus be achieved.

1. Introduction
Concurrency control is critical to the performance of transaction processing systems. Optimistic concurrency control (OCC) offers an alternative to locking. The two approaches produce different effects on hardware resource contention and data contention. Compared with locking, the OCC scheme is vulnerable to hardware resource contention due to transaction aborts, but provides more throughput when resource is unlimited and data contention is the limiting factor [AGRA87]. In IBM IMS Fast Path, to support high volume transaction processing the OCC approach is provided as an alternative to conventional locking to avoid lock waits and blocking [DATE82].

The pure OCC scheme only aborts a transaction at its commit time when the validity of its data accesses is checked. A variation of the pure OCC scheme has been suggested with the intent to save CPU processing. When a transaction is successfully committed, all conflicting transactions are detected and immediately aborted in the middle of their executions [ROBIN82]. This is referred to as broadcast OCC. Previous studies on pure or broadcast OCC schemes like [THOM85, DAN88, FRAN85] have not considered the effect of buffering on rerun transactions. All data blocks referenced during the previous run are implicitly assumed to be flushed out before the next run of an aborted transaction. Thus the buffer hit ratios of rerun and first run transactions are taken to be the same in these studies. A new approach to memory management can be adopted so that data blocks referenced by aborted transactions continue to be retained in memory and be available for access during rerun. Similarly, data blocks updated by a transaction, that cause an abort of another transaction, are also kept in memory and can be accessed without IO by the aborted transaction. The minimum buffer requirement is increased by a factor equal to the ratio of the average response time to first run execution time. (As the reruns do not include any read IO, this increase in minimum buffer requirement is expected to be moderate [DAN89] and should not be a concern given the trend of cheaper memory and larger buffer size.) The IO reduction during rerun can have a significant effect on the performance [YU88,90]. By capturing this buffer retention effect, we thus have a somewhat surprising results that the pure OCC scheme which brings in all required data blocks into memory during the first run can outperform the broadcast OCC for environments with high data contention by reducing the rerun abort probability.

We analyze different enhancements to the pure OCC scheme to exploit the buffer retention effect. With buffer retention, a rerun transaction behaves differently from its first run and hence, different CC protocols can be used for rerun transactions to enhance the overall system performance in a high data contention environment. Three different hybrid CC schemes are examined. All three hybrid schemes differ from pure OCC in their rerun policies. First, we consider the static and dynamic hybrid OCC schemes which switch from OCC to static and dynamic locking after the first abort, respectively. Both schemes employ pure OCC for the initial run, but differ in the way locks are requested during rerun. Taking advantage of the knowledge of all locks required during the initial run, the static hybrid OCC obtains all locks before rerun. If all locks cannot be obtained simultaneously, the transaction will hold no locks and will retry later. Thus transactions never hold locks while waiting for locks under the hybrid static OCC scheme. The dynamic hybrid OCC gets locks on data items progressively during rerun. This is similar to the ordinary dynamic locking scheme. Both hybrid schemes guarantee that transactions will have at most one rerun. This is in contrast to the ordinary OCC schemes that can have any number of reruns. However, additional waits are required to obtain locks before or during the second run. The trade-off between aborts and lock waits during rerun are analyzed and compared. Even with the optimistic approach during rerun, a different type of OCC scheme may be used during rerun. In OCC with broadcast during rerun, a committing transaction detects conflicts with other transactions but only aborts the conflicting rerun transactions in the middle of their executions. The conflicting first run transactions...
would still run to completion to bring in all data blocks needed into memory. Note that there is no point running an aborted rerun transaction to its completion since all the data granules required are already in memory. Eliminating the database I/O's to reduce data contention has also been recognized in [REUT85], where it is suggested that a transaction be pre-executed so that the 'real' execution of the transaction gets very simple with all data required in memory once it has already been executed before. Nevertheless, the pre-execution approach tends to cost more CPU processing and may also complicate the transaction management than the buffer retention approach. The property that reruns of a transaction access the same granules as during the first run even though the runs are separated by those of conflicting transactions was called access invariance in [FRAN90], where the performance of other CC schemes that exploit access invariance was evaluated using simulations.

To model the performance of the transaction processing system under the different CC schemes, we need to capture the effect of both hardware resource (CPU) contention and data contention. Data contention results in transaction wait or abort, leading to resource consumption for transaction rerun, and consequently higher resource contention, leading in turn to higher data contention. We use an approximate analysis to model the effect of hardware resource contention and data contention. We use a decomposition approach to separate the analysis into two interacting components: hardware resource contention and data contention. Queueing models are used to estimate the resource contention effect. Data contention is estimated using a general mean value type analysis to model the different hybrid CC schemes. Interaction between the data and resource contention is captured using an iteration.

In Section 2, we present the models and the analysis for the different hybrid schemes. For lack of space, we describe the analysis for one hybrid scheme in detail and outline the others. The validation of the analysis with simulations and performance comparisons of the different schemes are provided in Section 3. Finally, summary remarks appear in Section 4.

2. The Model and Analysis

There have been numerous analytical studies of the performance of CC schemes. Most of the previous work has either ignored the interaction of data contention and resource contention or has not captured the effect of resource contention, e.g. [CHES83,GALL83,GRAY81,IRAN79,POTT80]. Analysis of locking in centralized databases can be found in [TAY85A,B], while [SEV83] examines analytical work on the performance of different CC schemes in distributed databases and simulation studies are reported in [CARE88]. Optimistic Concurrency Control (OCC) has been studied in [ROBI82,THOM85,DAN88]. Analytic studies to comparing locking with OCC for centralized databases can be found in [MEN82,MORR85,YU90], and detailed simulation studies are reported in [AGRA87]. In [YU85,YU87A,B,DIAS88] a decomposition position combined with iterations has been used for analyzing lock contention effects in coupled systems and also capturing the effect of hardware resource contention.

A slightly different approach is used in [DAN88] to capture the hardware resource contention in a centralized environment using a piecewise linear model. In general, the models for different CC schemes are very different and make different assumptions.

In this section we describe a uniform approximate analysis methodology, based on the decomposition approach similar to that in [YU87A,B], that can be used to analyze the different hybrid schemes. Detailed analysis of the static hybrid OCC scheme, is given in Section 2.1. The analysis method for the other hybrid schemes is outlined in the remainder of the section.

2.1 Static Hybrid OCC

This scheme uses pure OCC for the first run of a transaction and (static) locking for the second run of the transaction. In this scheme, the first run of a transaction is identical to the first run with the pure OCC. At the end of the first run, if there is a conflict with a committed transaction, or with a transaction in its second run, or with a committing transaction (writing the log), then the transaction is aborted. The aborted transaction waits till it can acquire locks on all the items it requires for the second run before the second run is started.

The transaction model for this case has \( L + 2 \) states, where a constant number \( L \) of data items are accessed. The transaction has an initial setup phase (including program fetch, and message processing). This is denoted as state 0, and is modeled as contributing time \( R_{inp} \) to the transaction response time, corresponding to an average of \( P_{inp} \) instructions and \( I_{inp} \) I/Os per transaction for setup. Following this, a transaction progresses to states 1, 2, ... , \( L \), in that order. At the start of each state \( i > 0 \) the transaction begins to access a new data item and moves to state \( i + 1 \) when a new data item is required. In state \( i \), the transaction has informed the concurrency control manager of access to \( i \) data items. After state \( L \), if the transaction is aborted, it returns to state 1, and switches to static locking mode. Otherwise, it enters commit processing in state \( L + 1 \). In the following analysis, we assume that exclusive access is required on each of these data items. In the first run of a transaction, the average time in state \( i \) is modeled as \( R_i \), corresponding to execution of an average of \( P_i \) instructions, and an average of \( I_i \) I/Os. In a rerun of a transaction due to an abort, the average time in state \( i \) is modeled as \( R_i' \), corresponding to an average of \( P_i' \) instructions, and no I/Os. (Recall that it is assumed that there is sufficient buffer memory so that no read I/Os are required for a rerun of a transaction.) At commit time, after accessing \( L \) data items, the concurrency control manager is so informed. After state \( L \), if aborted for the first run, the transaction waits for time \( T_{lockoff} \) before restarting. Otherwise, the transaction enters commit processing, and any conflicting transactions are marked for abort. It then writes commit records to log, modeled as an average time of \( T_{commit} \). During commit processing, exclusive access on the data items accessed is retained, and any ongoing transactions that access conflicting data with the committing transaction are marked for abort. We use an open queueing model, and assume...
Poisson transaction arrivals at rate \( \lambda \) transactions per second.

Since all locks are acquired before the second run is started and since we assume that all data reads for the transaction have already been done (only the log writes need to be done), the second run takes time \( \sum R' \), excluding commit. Hence, the probability of abort after the first run \( P_A \) is written as,

\[
P_A = 1 - \lambda L \left( T_{commit} + \sum_{i=1}^{L} R'_i \right)
\]

where \( L \) is the number of data items (granules) in the database. In this expression, \( 1 - \lambda L \) is the probability that a data item accessed by a transaction entering commit does not conflict with a transaction that has accessed \( i \) data items, and \( L \times \sum R'_i \) is the average number of data items accessed by transactions entering commit during the period \( R_i \), that the transaction is in state \( i \). The second term in the product accounts for the contention probability on access of the \( i \)-th data item with transactions holding exclusive access on data items during commit or rerun. The term \( \lambda L \times \sum R'_i \) is the average number of locks held by transactions in the second run and in commit. The ratio of this term to \( L \times \sum R'_i \) is the probability that the \( i \)-th data item accessed contends with an exclusive holding of a rerun transaction or committing transaction.

Assuming \( k \) tightly coupled processors. Since there are at most two runs of the transaction, the processor utilization is written as,

\[
p = \frac{\lambda (P_{INPL} + \sum_{i=1}^{L} P_i + \sum_{i=1}^{L} P'_i)}{k \times MIPS}
\]

where MIPS is the processor speed. In the above, the probabilities of abort are estimated in terms of the times in each state \( R_i \) and \( R'_i \), and the utilization \( p \) is expressed in terms of the abort probabilities. Now, \( R_i \) and \( R'_i \) can be estimated from \( p \) based on M/M/K results [LAVE83] as follows. Define,

\[
\alpha = \frac{(kp)^{k}}{k! (1 - \rho)} ; \quad \beta = \sum_{j=0}^{k-1} \frac{(kp)^{j}}{j!}.
\]

Then, the CPU time for a state with execution of \( PL_s \) instructions is approximated as,

\[
R_x = \gamma \times \frac{PL_s}{MIPS}
\]

where

\[
\gamma = 1 + \frac{\alpha}{(\alpha + \beta)(1 - \rho)}
\]

Then the times in states \( i \) are estimated as,

\[
R_{INPL} = \gamma \times \frac{P_{INPL}}{MIPS} + I_{INPL} \times IOTIME
\]

\[
R_i = \gamma \times \frac{P_i}{MIPS} + I \times IOTIME
\]

\[
R'_i = \gamma \times \frac{P'_i}{MIPS}
\]

(2.5)

All that remains to be estimated is the wait time that an aborted transaction waits before it acquires all the locks it needs for the second run of the transaction. The probability that the transaction entering the second run cannot acquire all the locks it needs is estimated as,

\[
P_{RETRY} = 1 - \left( 1 - \frac{\lambda L \times \sum_{i=1}^{L} R'_i}{Lspace} \right)^L
\]

(2.6)

2.2 Dynamic Hybrid OCC

This scheme also used pure OCC for the first run of a transaction, and locking for the second run of the transaction. The difference between this scheme and the static hybrid OCC scheme is that here locks in the second run are obtained dynamically during the second run, rather than obtaining all at the start of the second run as done in the static scheme. As before, at the end of the first run, if there is a conflict with a committed transaction, with a transaction in the second run (holding locks), or with a committing transaction (writing the log), then the transaction finishing the first run is aborted. The aborted transaction backs off for a time \( T_{backoff} \) and then begins the second run, progressively acquiring locks for data accessed. The average time for the second run, \( \{R'_i\} \), includes both the CPU time, and lock waiting time (as before I/O read time in the second phase assumed to be nil). The average number of locks held by all transactions in the second run, excluding commit period, are \( \lambda P_A \sum_{i=1}^{L} R'_i \). The probability of abort after the first run can be derived in an analogous manner to equation (2.1), with the difference that the average number of locks held by transactions in the second run is now \( \lambda P_A \sum_{i=1}^{L} R'_i \) instead of \( \lambda P_A L \sum R'_i \). Now the processor utilization \( p \) is estimated by equation (2.2) of the previous section, and constants \( \alpha, \beta, \gamma \), by equations (2.3) and (2.4), and \( R_{INPL} \).
and \( R \), by equation (2.5). All that remains to be estimated is \( R' \), since it now includes waiting times to acquire locks.

We refer the reader to [YU90] for the method we use to estimate the lock waiting time for dynamic locking that can be applied to this hybrid scheme with minor differences.

### 2.3. OCC with Broadcast During Rerun

In this scheme, the transaction uses pure OCC for the first run of the transaction, and uses the early abort of broadcast OCC for any subsequent reruns. In the first run, it behaves similar to the static and dynamic hybrid schemes.

At the start of each state \( i > 0 \) the transaction begins to access a new data item and moves to state \( i + 1 \) when a new data item is required. In state \( i \), the transaction has informed the concurrency control manager of access to \( i \) data items. After state \( L \), if the transaction is aborted, it returns to state 1. During reruns, upon entering state \( i \), the transaction not only informs the concurrency control manager of its access to the \( i \)-th data item, but also checks to see if it has already been marked for abort. If already marked for abort, the transaction is immediately aborted and returns to state 1. The probability of abort \( P_A \) after the first run of a transaction is given by

\[
P_A = 1 - \prod_{i=1}^{L} \left( 1 - \frac{1}{L_{space}} \right) \left( 1 - \frac{L T_{Comm}}{L_{space}} \right).
\]

The second term in the product accounts for the contention probability on access of the \( i \)-th data item with transactions holding exclusive access on data items during commit processing. Again, for lack of space, we refer the reader to [YU90] for the method of analyzing the broadcast OCC of rerun transactions, that can be applied here with minor changes.

### 3. Performance Comparison

In this section we compare the performance of the different OCC schemes described in the previous section. The estimates from the approximate analyses of Section 2 are also compared with simulation results. To study the robustness of the schemes, we first examine the case where the system is hardware resource limited, and then relax the resource constraint to see the effect of higher data contention.

We use the following parameters for the comparison. The transaction parameters are similar to those reported in [YU87A] for a measured IMS workload on a parts database from a heavy equipment manufacturer. The initial instructions executed, \( P_{Inst} \), of 150K, initial I/O (1/100L) of 5, instructions executed in state \( i \) of 20K \( V_i \), average I/Os (\( I \)) in state \( i \) of 0.73 \( V_i \), an average I/O time (\( I O T I M E \)) of 0.035 seconds, states per transaction (\( L \)) of 15, commit time (\( T_{Comm} \)) of 0.025 seconds, and a backoff time \( T_{Backoff} = T_{Comm} \) Note that the probability of repeated I/O during a rerun is assumed to be zero, i.e., \( P_i = 0 \). We vary the transaction rate \( \lambda \), the lockspace \( L_{space} \), and the number of tightly coupled processors \( K \).

All of the protocols analyzed above were simulated using a discrete event simulation. The simulation keeps track of the data accessed by each transaction and explicitly simulates data contention, transaction aborts, waits for locked data items, queuing and processing at the CPU, I/O waits, and commit processing. The tightly coupled processors are simulated as having a common work queue. The CPU service times correspond to the CPU MIPS rating and the specific instruction pathlengths given above (and are not exponentially distributed). The CPU is released by a transaction for each I/O, when lock contention occurs and during backoff after an abort. Deadlocks cannot occur except for the dynamic hybrid OCC scheme. In the case of a contention that leads to a deadlock for the dynamic hybrid OCC scheme, the transaction is aborted and all locks held are released. The states of a transaction in the simulation are the same as described in Section 2 for the analysis.

Figure 3.1 shows the average transaction response time versus the transaction rate for a lockspace of 1000 granules and 5 tightly coupled processors of 10 MIPS each, assuming no repeated read I/O during the rerun of an aborted transaction. Note that for a relatively small lock space is chosen for this case so that we can better differentiate the robustness of the schemes and demonstrate the phenomena that cause the difference. Further, as described in [TAY84], non-uniform access to the lock space can be modeled by uniform access to an effective database size, that is smaller than the original lock space. Therefore, the occurrence of hot spots in the database leads to comparatively small database size. Sensitivity to the lock space parameter is considered later. In Figure 3.1, we consider the three hybrid schemes. Both analytical and simulation estimates are presented for the three schemes. Also shown are simulation estimates for the pure OCC, and broadcast OCC and locking schemes. As described in Section 2, an iteration is used for each of the analytical estimates, and the estimates were found to converge in a small number (usually less than ten) iterations. Notice the close agreement between the simulation and the analysis. The analytical estimate is almost identical to the simulation estimate before the knee of the curves, and is a little optimistic beyond the knee, but accurately projects the location of the knee. The principal limitation in the transaction rate that can be supported for this case is the CPU resource limited. This is evident from Figure 3.2, which shows the CPU utilization versus transaction rate for the same parameters as Figure 3.1. We note that the utilization predicted by the analyses are in close agreement with simulation estimates and are indistinguishable from the curves in Figure 3.2. Since the CPU utilization depends on a correct estimate of the probability of abort \( P_A \) and \( P_A \), these estimates in the analysis are also found to be in agreement with simulation. From Figures 3.1 and 3.2, it is observed that for this case, the limit in transaction rate occurs when the CPU utilization becomes very high. Except for static and dynamic hybrid OCC, the OCC schemes can have several aborts, leading to higher CPU consumption and hence they fare worse than the static and dynamic hybrid OCC schemes. Comparing the static and dynamic hybrid OCC schemes, the CPU utilizations of the two schemes are almost identical in Figure 3.2. However, the dynamic hybrid OCC scheme has a smaller response time than the static scheme. The reason is that, in the case of a lock contention, the static scheme backs off and tries again after the backoff time, while the dynamic scheme waits until the lock is released by the transaction holding the lock. Com-
paring the pure and broadcast OCC schemes, it is observed that using broadcast results in a larger CPU utilization. This is in spite of the fact that the transaction is aborted as soon as the abort is detected, rather than proceeding to the end of the transaction before aborting. The reason for this striking behavior is due to the assumption that there is sufficient buffer memory so that a data block read does not need to be reread from disk again during subsequent reruns of the transaction. Hence, the second and further runs are very short for the pure OCC, but can be long for the broadcast OCC since some I/Os are done in the reruns. This leads to a larger probability of second and subsequent aborts for the broadcast OCC, and hence to larger CPU utilization. OCC with broadcast during rerun can reduce the CPU utilization by aborting the rerun transactions earlier and thus improve the performance.

Figure 3.3, which is based on simulations, is similar to Figure 3.1 except that the number of tightly coupled 10-MIPS processors is increased to 10. For locking, data contention continues to limit the transaction rate that can be supported. (The curve for locking is again off the chart in Figure 3.3.) The larger resource allows a larger transaction rate to be supported. However, the comparison between the various schemes changes. At higher transaction rates, the contention between transactions increases since the size of the database is held fixed. Hence, the number of aborts for the optimistic schemes increases, with the effect being more pronounced for the broadcast OCC scheme. Notice the crossover in the response time for the static and dynamic hybrid OCC schemes. At high transaction rates, and consequently larger contention levels, the static hybrid OCC scheme has smaller response time, and can support a higher transaction rate. This is because the static scheme does not hold locks while it waits. That is, in the static scheme, if all the required locks are not available, the transaction backs off and tries later to obtain the locks again, but does not retain the locks during the back-off period. The dynamic scheme on the other hand, gets locks progressively, and retains the locks it has already acquired while it may wait for another lock. Notice that the difference between the pure OCC and the dynamic hybrid OCC scheme diminishes at higher contention levels, because the larger resource utilization of the pure OCC is offset by larger wait times for the dynamic hybrid OCC scheme. OCC with broadcast during rerun now performs better than the dynamic hybrid OCC. This trade-off is illustrated further in Figure 3.4, which is for the same parameters as above, except that the number of tightly coupled processors is increased to 15, providing larger CPU resource. Again the simulation results (at this high data contention level) are in close agreement with the estimates from the analysis. Now, there is a crossover in the response time of the dynamic hybrid OCC scheme and the pure OCC scheme, in addition to a crossover with that of the static hybrid OCC scheme. The limitation in transaction rate for the dynamic hybrid OCC scheme is due to lock contention waits. OCC with broadcast during rerun shows noticeable improvement over pure OCC as more repeated reruns occur during high data contention the early abort during rerun leads to more significant saving on CPU requirement. The static hybrid OCC scheme still supports a higher transaction rate than the pure OCC or OCC with broadcast during rerun because of its smaller resource utilization.

Figure 3.5 (based on simulations) shows the response time for the same case as in Figure 3.4 except that the lock space is increased to 5K (from the 1K of the previous charts). Comparing Figure 3.5 to Figures 3.4 it is observed that the larger lockspace leads to the comparison between the schemes reverting to being similar to that in Figure 3.1. That is, the larger lock space leads to lower contention levels and the limitation in transaction rate is due to resource contention rather than lock contention, leading to larger throughputs for the hybrid OCC schemes over the pure OCC and OCC with broadcast during rerun.

4. Summary

With sufficient memory, a buffer retention approach to memory management can be adopted so that data blocks referenced by aborted transactions continue to be retained in memory and be available for access during rerun. Under buffer retention, it becomes advantageous to complete all the reads during the first run, even if the transaction is destined to abort so as to reduce the run time of future reruns. It was found that the broadcast OCC, which is generally regarded to be superior to the pure OCC, can actually perform worse than the pure OCC under buffer retention, particularly at high level of data contention. (If all read I/O's are redone during rerun, abort upon conflict as in the broadcast OCC is the right strategy.)

Based on the pure OCC scheme, three hybrid OCC schemes are considered to further exploit the buffer retention effect on rerun transactions and improve the performance in a high data contention environment. Since rerun transactions have no (or few) database I/O's and face a lower level of data contention as compared to first run transactions, a different CC protocol can be used for rerun transactions to take advantage of this situation. We find that in the high contention environment where OCC outperforms locking, the use of locking (only) during transaction reruns is preferable to OCC. As the data contention level increases further, dynamic locking during transaction rerun can become contention limited. However, using static locking during transaction reruns continues to outperform pure OCC even at extremely high levels of data contention. We note that using static locking only during transaction reruns becomes feasible because the first run of a transaction can provide information on locks required, assuming that the same data is accessed during a rerun of a transaction. Furthermore, even staying with the optimistic approach during rerun, forcing early abort during rerun as in the OCC with broadcast during rerun can improve performance.

To analyze the performance under different CC protocols, approximate analytic models are developed based on a decomposition approach and using a mean value type analysis. The effect of buffer retention on rerun transactions is captured. The accuracy of the analysis is validated through simulations. The close agreement between the analysis and the simulation in predicting the throughput and response time shows the robustness and generality of the modelling approach. Since simulations for high data/resource contention environments can be very time-consuming.
consumption, an analytic approach can provide a much more efficient method to determine the throughput and response time of each CC scheme and select the most suitable CC scheme.

References


Fig. 3.1. Throughput vs. Response Time for Different Approaches.

Fig. 3.2. CPU Utilization for Different Approaches.

Fig. 3.3. Response Time with Increase Total MIPS.

Fig. 3.4. Response Time with Further Increase in MIPS.

Fig. 3.5. Response Time with Increase in Look Space.