Performance of Transaction Scheduling Policies for Parallel Database Systems

S. Dandamudi and Chun-Yip Chow

School of Computer Science, Carleton University
Ottawa, Ontario K1S 5B6, Canada

ABSTRACT - Transaction scheduling is an important performance issue in parallel database systems. Here we study the impact of transaction scheduling in the shared-nothing type of architecture. We do not model a particular database system in our study. Rather, we use an abstract model of the shared-nothing type of architecture. We consider four transaction scheduling policies. These policies can be broadly divided into two classes: policies that work independent of the current system state and policies that use the current system state information. We consider two policies belonging to each category - one policy works independent of the transaction characteristics and the other policy requires transaction size information. Our results, obtained via simulation, indicate that system state dependent policies provide substantial performance advantages over the other group of policies.

1. INTRODUCTION

In recent years, the demand for high transaction processing rates has continued to increase substantially. It is estimated that there will be need for systems that can process on the order of 1000 transactions per second [3]. At the same time, the amount data involved in processing these transactions is increasing. One approach to meet this demand for increased transaction processing power is to parallelize transaction processing by utilizing multiple processors. In such parallel database systems, transactions are divided into several subtransactions, all of which can be run in parallel.

A parallel database system can be built by connecting together many processors, memory units, and disk devices through a global interconnection network. Depending on how these components are interconnected there can be a variety of possible architectures. In this paper we consider the shared-nothing architecture in which each processor has its own (private) disk and memory units. Shared-nothing database systems are attractive from the standpoint of scalability and reliability [14]. Several commercial and experimental systems such as the Non-Stop SQL from Tandem, the DBC/1012 from Teradata, the Gamma machine at the University of Wisconsin [6], and the Bubba at MCC [9] use lock-based concurrency control mechanisms.

Database systems that use physical locks support the notion of granule [12]. A granule is a lockable unit in the entire database and typically all granules in the database have the same size. Before accessing the database entities, a transaction must lock the required granules. However, if the required granules are locked by another transaction in the system, the transaction will be blocked. The granularity of a column of a relation, a single record, a disk block, a relation or the entire database. Locking granularity refers to the size of lockable granule. Fine granularity improves system performance by increasing concurrency level but it also increases lock management cost. Furthermore, more lock and unlock operations will be issued, causing a higher overhead. In addition, more storage space is required for the locks. For example, if the granule size is a record, transactions can run in parallel as long as they use different records. But the system must maintain a lock table that has the same number of entries as the number of records in the database. Such lock tables are necessarily stored on secondary storage devices such as disk. Coarse granularity, on the other hand, sacrifices system performance but lowers the lock management cost. For example, if the granule size is the entire database, then transactions are forced to run in a serial order. However, the system handles only one lock, which can be kept in main memory. Locking granularity in parallel database systems has been studied in [4]. Here we study the interaction between locking granularity and transaction scheduling policies.

We use simulation as the principal means of performance evaluation. The remainder of the paper is organized as follows. A description of the simulation model is given in Section 2. Section 3 describes the scheduling algorithms considered. The results are discussed in Section 4. Finally, conclusions are given in Section 5.

CH2996-7/91/0000/0116$01.00 © 1991 IEEE
2. THE SIMULATION MODEL

As stated in Section 1, we assume the shared-nothing architecture in our simulation model. All relations are assumed to be horizontally partitioned across all disk units in the system using the round robin strategy [7]. Thus, any given relation is equally partitioned among all the disk drives (the relations are assumed to be large enough to allow this). For example, if a relation consists of 10,000 tuples (i.e., rows) and if there are 20 processors (and disks), then each processor/disk gets 500 tuples of the relation.

An overview of the simulation model used in this study is shown in Figure 1. This is a closed model with a fixed number of transactions \( n_{trans} \) in the system at any instance. As shown in Figure 1, the transactions cycle continuously through the system. In our simulation model, every time a transaction completes execution, a new transaction with a new set of parameter values is created so that the system always sees \( n_{trans} \) transactions. Our simulation model is an extension of the model used by Ries and Stonebraker [12,13] for uniprocessor database systems.

We assume a fork and join type of transaction structure. In this model, a transaction is assumed to be split into several sub-transactions, each working on fragments of the relations involved. For example, if the operation is the selection operation, then each processor will be working on a smaller relation to retrieve partial response to the selection operation. These partial responses are then combined to derive the final response. Thus, a transaction is not considered complete unless all the sub-transactions are finished. It is also assumed that each sub-transaction consists of two phases: an I/O phase and a CPU phase. While this model of transaction assumes that there is no communication between the sub-transactions, we believe that it is useful in obtaining insights into the influence of scheduling policies on performance of the type of database systems we are considering here. We intend to consider the impact of this communication in the near future.

We now briefly describe the simulation model (a more detailed description is given in [4]). The system is initialized to contain \( n_{trans} \) transactions. A transaction goes through the following stages.

1) The transaction at the head of the pending queue is removed and the required locks are requested. If the locks are denied, the transaction is placed at the end of the blocked queue. System will record the blocked transaction. This record is used to help release blocked transactions after blocking transactions finished their processing and released their locks. (The lock conflict computation and the blocked transaction release process are described at the end of this section.)

2) After a transaction receives all its locks, it is split into \( npros \) (the total number of processors in the system) sub-transactions. Then each sub-transaction is placed at the end of the I/O queue of its assigned processor. Notice that no two sub-transactions are assigned to the same processor. After completing the required I/O and CPU service, the sub-transaction waits in the waiting area for other sub-transactions of its parent transaction to complete.

3) A transaction is considered complete when all its sub-transactions are completed. A completed transaction releases all its locks and those transactions blocked by it. The releasing process of blocked transactions is described at the end of this section.

Note that the transactions request all needed locks before using the I/O and CPU resources. Thus deadlock is impossible. This basically simulates a conservative locking scheme [2]. In addition, it is assumed that the cost to request and set locks includes the cost of releasing those locks. The cost is assumed to be the same even if the locks are denied. In addition, we assume that processors share the work for locking mechanism. This is justified because relations are equally distributed among the system resources. Moreover, lock processing has priority over transaction execution for the CPU and I/O resources.

![Figure 1 Simulation model](image-url)
Now we review the input and output parameters of the simulation model.

**Input parameters**

- \( npros \) = the number of processors in the system.
- \( dbsize \) = the number of accessible entities in the entire database. An accessible entity is the unit moved by the operating system.
- \( ltot \) = the number of locks in the database system. If \( ltot = 1 \) then the granule size is the entire database. On the other hand, if \( ltot = dbsize \) then a granule corresponds to a database entity. Note that \( dbsize \) is the maximum value \( ltot \) can assume.
- \( ntrans \) = the number of transactions in the system. This can be interpreted as the number of terminal users who issue commands against the database.
- \( maxtransize \) = the maximum transaction size in the system. For the results reported here, the size of the transactions is uniformly distributed over the range 1 and \( \text{maxtransize} \).
- \( cputime \) = CPU time required by a transaction to process one entity of the database.
- \( iotime \) = I/O time required by a transaction to process one entity of the database.
- \( lcputime \) = CPU time required to request and set one lock.
- \( liotime \) = I/O time required to request and set one lock.
- \( tmax \) = the number of time units to run the simulation.

**Output parameters**

The following parameters are recorded for each simulation run.

- \( throughput \) = the number of transactions completed per unit time.
- \( RT \) = average response time (i.e., from the time a transaction enters the pending queue till the time it completes all processing and releases locks).
- \( BT \) = average blocking time (blocking time is the time spent by a transaction in the blocked queue awaiting the release of the needed locks).
- \( ST \) = average synchronization time (synchronization time is defined as the time between the completion of the first sub-transaction to the completion of the last sub-transaction of the same transaction).

**Variables for describing \( i \)th transaction**

- \( NUi \) = random \( \text{maxtransize} \) = number of database entities required by \( i \)th transaction \((1 \leq NUi \leq \text{maxtransize})\)
- \( LUi \) = \( \text{cei}l(NUi \times ltot / dbsize) \) = number of locks required by \( i \)th transaction

\[
\begin{align*}
\text{IOtime}_i & = NUi \times liotime = \text{I/O time required by } i\text{th transaction} \\
\text{CPUtime}_i & = NUi \times cputime = \text{CPU time required by } i\text{th transaction} \\
\text{LIotime}_i & = LUi \times liotime = \text{I/O time required for locking by } i\text{th transaction} \\
\text{LCPUtime}_i & = LUi \times lcputime = \text{CPU time required for locking by } i\text{th transaction}
\end{align*}
\]

**Decomposition of a Transaction**

As stated earlier, a transaction that receives all its locks is split into \( npros \) sub-transactions. The mean value of the CPU service time of a sub-transaction of transaction \( i \) is set to \( \text{CPUtime}_i / npros \). Since each processor is working on a fragment of a relation of equal size, we do not expect high variation among the CPU service times of the sub-transactions for operations such as selection. However, to account for the minor variations due to, for instance, conditional execution of code, we vary actual sub-transaction processor service time between 0.95 and 1.05 of the mean value. For other operations, the service time variance could be large. However, there is indication that processor service time variations as well as sub-transaction scheduling policies play small role in overall system performance [5].

Similarly, we set the mean value of I/O service time for a sub-transaction of transaction \( i \) to \( \text{IOtime}_i / npros \). However, there can be variations due to, for instance, seek time differences. Therefore, for the results reported here, we use exponential distribution to model this. The CPU and I/O lock processing times are set similarly.

**Computation of lock conflicts**

To compute lock conflicts, it is assumed that enough locks are unlocked for the requesting transaction to potentially proceed. Let \( T_1, T_2, ..., T_k \) denote the \( k \) active transactions currently using the I/O and CPU resources. Suppose that each transaction \( T_i \) has \( L_i \) locks. Then divide the interval \((0,1]\) into \( k+1 \) partitions.

\[
P_1 = (0, L_1 / \text{liot}) \\
P_2 = (L_1 / \text{liot} , (L_1 + L_2) / \text{liot}) \\
.. \cdot \\
P_k = ((L_1 + L_2 + .. + L_{k-1}) / \text{liot} , ((L_1 + L_2 + .. + L_k) / \text{liot} \\
P_{k+1} = ((L_1 + L_2 + .. + L_k) / \text{liot} , [1)
\]

To determine if a given transaction must be blocked, choose a random number \( p \), uniformly distributed on \((0,1)\). If \( p \) is in \( P_p \), for some \( j \leq k \), then the transaction is blocked by \( T_j \). Otherwise, the transaction may proceed. Hence, if the transaction is blocked, \( T_j \) will be the one to release it. Note, however, that the released transaction is not guaranteed to receive all its locks. Instead, the released transaction is placed in the pending queue and then goes through the lock request process. If locks are denied, it will be placed in the blocked queue again. This reflects the situation that there may be other transactions that are still holding a subset of the locks required by the released transaction. Thus the lock conflict computation models the fact that an arriving transaction in general requires locks currently held by more than one transaction.
3. TRANSACTION SCHEDULING POLICIES

As shown in Figure 1 transactions wait in the pending queue to be scheduled. The scheduling policy effectively determines which of these waiting transactions should be scheduled next. There are several scheduling policies that are applicable to transaction scheduling. These policies can be broadly divided into two classes: policies that work independent of the system state and policies that use the current system state information. We consider two policies belonging to each category - one policy works independent of the transaction characteristics and the other policy requires transaction size information.

System state independent policies

- First-In-First-Out (FIFO): The transactions are placed in the order of their arrival to the pending queue. The transaction at the head of the pending queue makes a lock request, the processing of which requires both the processor and I/O resources. If this request is granted, the transaction proceeds to the processing stage; otherwise, it is placed in the blocked queue as explained in Section 2. In either case, the transaction is removed from the pending queue and the transaction behind this transaction makes its lock request next.

- Smallest Transaction First (STF): This policy requires transaction size information and orders the transactions in the pending queue in increasing order of transaction size. Thus at any time, all the new transactions arrive at this queue, only the transaction with the smallest size is allowed to make its lock request. Other than this difference, this policy is similar to the FIFO policy described above. It should be noted that an estimate of transaction size can be obtained. For example, if the transaction involves a select operation, then selectivity characteristics of the relation involved (along with the information on available index structures etc.) can be used to obtain a good estimate of the complexity involved in processing the transaction.

System state dependent policies

Note that a transaction making its lock request for a set of locks takes away processors and I/O servers from transaction processing to lock processing, which is an overhead activity. Since a given database can support only a certain number of transactions depending on the average transaction size, total number of locks in the system etc. when there is large number of transactions in the system this lock processing overhead may cause serious degradation in overall system performance. This degradation increases with increasing number of locks and increasing transaction size and the number of transactions in the system. In order to reduce the adverse effect of this overhead on overall system performance, we propose two adaptive policies that are in spirit close to the back-off strategy used in Ethernet. These two policies are adaptive versions of the above two policies in that the transactions waiting in the pending queue are not allowed to make their lock request unless the number of active transactions (i.e., transactions currently holding locks) is less than a specified threshold value. Thus the transactions in the pending queue wait until the system load drops below this value before making their lock requests. This reduces lock processing overhead when there is little chance of having their request granted. Similar strategies have successfully been used in multistage interconnection networks to reduce hot-spot contention [8] and for thread management in shared-memory multiprocessors systems [1].

- Adaptive First-In-First-Out (A-FIFO): This is the adaptive version of the FIFO policy described above. The transaction at the head of the pending queue is allowed to make its lock request only if the system load (measured in terms of number of active transactions that are currently holding locks) below the threshold level $n_t$.

- Adaptive Smallest Transaction First (A-STF): This is the adaptive version of the STF policy described above.

The above two policies depend on a threshold value $n_t$, which is computed as follows:

$$n_t = \alpha \cdot \frac{\text{dbsize}}{\text{avgTransize}}$$

where dbsize is the total database size and avgTransize is the average transaction size and $\alpha$ is a parameter that allows $n_t$ to be a fraction of the maximum number of transactions that can be supported by the database.

4. RESULTS AND DISCUSSION

This section presents the results obtained from the simulation experiments. To reduce transient effects, we discard information associated with the first 100 completed transactions. We use independent replication strategy to obtain confidence intervals (at least 30 runs were used for the results reported here). This strategy provided 95% confidence intervals that are less than 5% of the mean values (in most cases) reported in the following sections. Unless otherwise indicated, the results correspond to the input parameter values given in Table 1. The sub-transactions, waiting for either processor or I/O resources, are scheduled using FIFO scheduling policy. Also the transaction size is assumed to be uniformly distributed as described in Section 2.

Table 1 Input parameters used in simulation experiments

<table>
<thead>
<tr>
<th>parameters</th>
<th>values</th>
<th>parameters</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbsize</td>
<td>5000</td>
<td>npros</td>
<td>20</td>
</tr>
<tr>
<td>maxtransize</td>
<td>500</td>
<td>liotime</td>
<td>0.20</td>
</tr>
<tr>
<td>mmax</td>
<td>12000</td>
<td>lcpertime</td>
<td>0.05</td>
</tr>
</tbody>
</table>

4.1. Impact of Number of Transactions

In this section we discuss the effect of number of transactions (ntrans) in the system. By fixing the input parameters as in Table 1, several experiments were conducted by varying the number of transactions from 1 through 300. For these experiments, the number of locks (liot) is fixed at 100 and $\alpha$ is set at 0.5. For the parameters considered here, $\alpha$=0.5 results in the best performance. The impact of parameter $\alpha$ is discussed in Section 4.3.

The results are reported in Figure 2. These results show that adaptive policies yield substantial performance benefits compared to FIFO and STF policies. The reason, as discussed
in Section 3, is that the lock processing overhead is reduced by restricting the transactions waiting in the pending queue from making lock requests when the system load is high. Since there is no such control with FIFO and STF policies, the throughput actually drops as the number of transactions in the system increases. The adaptive policies, however, maintain the peak throughput independent of the actual number of transactions in the system. As can be seen from Figure 2b, the throughput difference between FIFO and STF policies, and between A-FIFO and A-STF policies is marginal. However, the corresponding response times, depicted in Figure 2a, show considerable performance advantages with STF and A-STF policies. This is because, for the same throughput, these policies are processing smaller transactions compared to the situation with FIFO and A-FIFO policies.

Surprisingly, FIFO policy gives marginally better throughput than the STF policy. One possible explanation for this behaviour is that STF policy grants locks to smaller transactions and leaves all larger transactions in the pending queue. These larger transactions, however, take processor and I/O resources away from useful transaction processing to lock request processing involving larger number of locks (compared to the situation with the FIFO policy).

FIFO and STF policies are more sensitive to the number of transactions in the system. The reason is that a sub-transaction requiring large I/O service time could potentially keep the server busy and hence introduce inordinate delays for other smaller sub-transactions. This increases lock hold time resulting in more transactions being placed in the blocked queue. As can be seen from Figure 2c the blocking time is the main component that causes the increase in total response time. The blocking time associated with the adaptive policies is negligible compared to that of the FIFO and STF policies. This is because of the "choking" strategy of these adaptive policies. Similar observations can be made for the synchronization time. However, as shown in Figure 2d, the synchronization time is substantially smaller (less than 10%) than the blocking time for FIFO and STF policies.

4.2. Effect of Number of Locks

This section discusses the effect of number of locks. With the input parameters set as in Table 1, several simulation experiments were run by varying the number of locks \( t_{\text{tot}} \) from 1 to \( d_{\text{size}} = 5000 \) for each policy. The number of transactions is fixed at \( n_{\text{trans}} = 200 \) for this set of simulation experiments. As in the previous section, \( \alpha \) is set at 0.5. The results are presented in Figure 3.

The convex nature of the response time curves in Figure 3 arises from the trade-off between the locking overhead and the allowed degree of concurrency. That is, increasing the number of locks initially decreases the response time by allowing concurrent execution of more transactions until a certain point after which the overhead for lock processing predominates without increasing the concurrency level; as a result the response time increases. Note that as the number of locks increases the associated overhead to process a transaction lock request increases proportionally.

![Figure 2 Performance of scheduling algorithms as a function of number of transactions](image-url)
The concave nature of the throughput curves is due to the tradeoff discussed above. As discussed in Section 4.1, the difference in throughput between the FIFO and STF policies, and between the two adaptive policies is marginal. From Figures 3b and 2b we note that, for FIFO and STF policies, large number of locks and large number of transactions tend to decrease system throughput (correspondingly results in an increase in response time). The combined effect of large number of transactions with large number of locks will result in substantial increase in lock processing overhead because of the multiplicative effect (i.e., the overhead is proportional to the product of the number of transactions and the number of locks). The two adaptive policies exhibit a more robust behaviour.

An important observation is that the type of scheduling policy has only a minor impact on the desired locking granularity (desired size of lockable entities). While all the policies support fairly coarse granularity (dividing the entire database of 5000 entities into 10 to 100 lockable entities is sufficient to ensure best throughput), the penalty for not maintaining the optimum number of locks is more with FIFO and STF policies compared to the other two policies. As shown in Figure 3b the throughput is fairly flat from 10 to 100 locks for A-FIFO and A-STF policies.

As discussed in Section 4.1, for FIFO and STF policies, it is the blocking time that is the major component of the response time. The curves of blocking time follow the shape of the response time curves given in Figure 3a. When the number of locks is less than 10, the blocking time associated with A-FIFO and A-STF policies is the same as that of the FIFO and STF policies, respectively. This is because the threshold value for the two adaptive policies is fixed at 10 for the parameters used here. When the number of locks is increased above 10, the blocking time drops and becomes negligible, as explained in Section 4.1.

Unlike in Section 4.1, the synchronization time increases substantially when the number of locks is large. While all the scheduling policies exhibit increased sensitivities to larger number of locks, synchronization time of FIFO and STF policies increases much more quickly. The reason is that the overhead of lock processing increases in direct proportion to the number of locks. Since lock processing has priority over regular transaction processing, the overall response time gets inflated substantially by not only increasing the blocking time but also substantially increasing the synchronization time. For example, when the number of locks is 1000, the synchronization time of FIFO policy is 20% of the corresponding blocking time.

4.3. Impact of Threshold Value
In the last two sections we have used an \( \alpha \) value of 0.5. This section presents the impact of this parameter on system performance as measured by the system throughput. Note that \( \alpha=1 \) allows the maximum number of transactions into processing stage and this is possible only when these transactions have a non-overlapping lock requests. In general, achieving this level of concurrency is not possible.
owing to lock conflicts. So the optimum $\alpha$ value would be less than one and depends on various factors such as the average transaction size, distribution of transaction size, number of locks, etc.

The sensitivity of system throughput to parameter $\alpha$ is shown in Figure 4. Figures 4a and 4b show the results when the number of locks is set at 100 and 500, respectively. All other parameters are set as shown in Table 1. The following observations can be made from Figure 4.

- The optimum $\alpha$ decreases with increasing number of locks. For example, from Figures 4a and 4b it can be seen that increasing the number of locks from 100 to 500 decreases the optimum $\alpha$ value from 0.5 to 0.3 for the A-STF policy. The corresponding values for the A-FIFO policy are 0.6 and 0.4. The reason is that the lock processing overhead per transaction increases with increasing number of locks. Thus balance between the lock processing overhead and the level of concurrency can be obtained at a lower threshold value.

- For small $\alpha$ values, A-STF policy gives marginally better throughput than A-FIFO policy. But as the threshold level increases A-FIFO policy provides better throughput for reasons discussed in Section 4.1. However, for large $\alpha$ values, the throughput difference between the two policies becomes marginal.

The impact of transaction size is shown in Figure 5 when number of locks is set at 100. Figure 5a presents the results for small transactions with the average transaction size of 50 entities (i.e., 1% of the database) and Figure 5b for large transactions with the average transaction size of 1250 entities (i.e., 25% of the database).

- It can be seen from Figures 4a and 5a that reducing the transaction size decreases the optimum $\alpha$ value. For example, when the average transaction size is changed from 250 entities (i.e., 5% of the database) to 50 entities (1% of the database) the optimum $\alpha$ values decreases from 0.5 to 0.2 for the A-STF policy. The corresponding values for the A-FIFO policy are 0.6 and 0.2, respectively.

- Notice that, in Figure 5a, the throughput is flat for $\alpha \geq 0.6$. Note that when $\alpha = 0.6$ the threshold value is set at 60 transactions. However, in order to get these 60 transactions to the processing stage, many more transactions would have to try for the locks. If this includes all the transactions within the system (here we have 200 transactions in the system) then no additional change in system throughput can be observed by increasing the threshold value. This observation is supported by simulat-
ion results (not presented due to space limitations), which show that the blocking time (as well as the number of active transactions holding locks) remains constant for this range of \( \alpha \) values. Note that all blocked transactions are placed in the blocked queue and they will be removed from this queue only when a transaction finishes and leaves the system. Similar behaviour can be seen in Figure 4a for \( \alpha \) values greater than 1.

- Both the adaptive policies exhibit similar sensitivity to threshold value when the transaction sizes are small as shown in Figure 5a. With increasing transaction size, A-STF provides better throughput because this policy chooses to process only the smaller transactions. Thus some larger transactions may experience live lock. In contrast A-FIFO is a fair policy and avoids live locks.

- The step-wise behaviour with large transaction sizes (in Figure 5b) is because of the fact that the actual threshold value does not change as smoothly as it does with smaller transaction sizes. For example, when the average transaction size is 1250 the threshold value \( n_\alpha = 4\alpha \). Thus, for instance, the threshold value is \( n_\alpha = 2 \) for \( \alpha \) values 0.3, 0.4, and 0.5.

### 4.4. Effects of Transaction Size

In this section, the impact of average transaction size is presented. Note that the number of entities required by a transaction is determined by the input parameter maxtransize (maximum transaction size). The transaction size is distributed uniformly from 1 to maxtransize. Thus the average transaction is approximately \( \text{maxtransize}/2 \). In this set of experiments, all the input parameters are fixed as in Table 1 except for maxtransize which is set at 100 and 2500. This corresponds to an average transaction size of 1% and 25% of the database, respectively. For the smaller transaction size workload, \( \alpha \) is set at 0.2 and for the larger transaction size workload, \( \alpha \) is set at 0.7. Note that, from Section 4.3, these \( \alpha \) values provide the peak throughput.

![Figure 6](image6.png)

**Figure 6** Performance of scheduling algorithms as a function of number of transactions (a) Small transactions (average transaction size = 1% of database) (b) Large transactions (average transaction size = 25% of database)

![Figure 7](image7.png)

**Figure 7** Performance of scheduling algorithms as a function of number of locks (a) Small transactions (average transaction size = 1% of database) (b) Large transactions (average transaction size = 25% of database)
Figure 6 shows the system throughput as a function of number of transactions. These results were obtained with the number of locks set at $\ell = 100$. We make two important observations from this figure. First, the two oblivious policies - FIFO and STF policies - are less sensitive to the number of transactions when the transaction sizes are small. This is a direct consequence of reduced lock processing overhead. Note that, for a fixed number of locks, the lock processing overhead is directly proportional to the transactions size. Second, the performance difference between FIFO and STF policies and between A-FIFO and A-STF policies is marginal for small transactions. However, for large transactions, (for instance, when the average transaction "-touches" 25% of the database), the difference between the two adaptive strategies increases with the number of transactions. This is mainly due to the fact that A-STF strategy selects to process only small transactions and the number of such transactions increases with the total number of transactions in the system. This, however, leads to live lock situation discussed in Section 4.3. In contrast, A-FIFO strategy is a fair policy and maintains a constant throughput.

Figure 7 shows the throughput as a function of number of locks. The number of transactions is fixed at $n_{trans} = 200$ for this set of experiments. Several observations can be made from the data presented in Figure 7. First, the observations made about the data in Figure 6 are generally valid even when the number of locks is varied. That is, for small transactions, the difference between FIFO and STF policies and between A-FIFO and A-STF policies is marginal. For small transactions, there is substantial improvement in system throughput associated with the two adaptive strategies. Note that, from Figure 6a, if the number of transactions is reduced to about 25, the difference would be marginal. Second, the optimal point (at which the throughput peaks) shifts toward the right (i.e., toward larger number of locks or finer granularity) with decreasing transaction size. For systems with small transaction sizes, finer granularity reduces unnecessary lock request failures. For systems with large transaction sizes, fine granularity does not improve the degree of parallelism because more database entities are required by each transaction; but fine granularity increases lock overhead substantially. However, all these optimal points have a number of locks that is less than 100. This suggests that coarse granularity is sufficient.

5. CONCLUSIONS

We have compared the performance of four transaction scheduling policies in a shared-nothing parallel database system. These include two adaptive policies that use system load information. We have shown by means of simulation that the two adaptive policies outperform the two oblivious policies. Our results also indicate that coarse locking granularity is sufficient and this is not seriously influenced by the type of transaction scheduling policy used in the system.

The results presented here show that the threshold value in adaptive scheduling policies is a function of various system and transaction characteristics such as the number of locks in the system, average transaction size etc. To implement these adaptive policies we need some type of learning mechanism that automatically adjusts the threshold value to the optimum depending on system and workload characteristics. For example, system throughput can be used as an indicator to adjust the threshold value. We are planning to investigate these issues in detail in the near future.

ACKNOWLEDGEMENT

We gratefully acknowledge the financial support provided by the Natural Sciences and Engineering Research Council of Canada and Carleton University.

REFERENCES


