An Experimental Comparison of Locking Policies in a Testbed Database System

by

Walter H. Kohler
and
Kenneth C. Wilner *
Digital Equipment Corporation and
University of Massachusetts

John A. Stankovic
University of Massachusetts

ABSTRACT

Multiuser transaction processing and database systems commonly use well-formed, two-phase locking to maintain data consistency. The preliminary results presented here represent the first step of an experimental investigation of the impact of different locking schemes on transaction performance. Tests were performed using a simplified but functionally complete "testbed" system. The transaction throughput rates for three different locking policies (file level locking, page level locking, and mixed level locking) are compared as a function of the lock mode (exclusive or share) and the file size for a small centralized database application. The experimental results for this environment show that the choice of locking policy and lock mode can have a significant impact on transaction throughput performance. The tests also demonstrate the sensitivity of the conclusions to the choice of workload and system characteristics.

1.0 INTRODUCTION

Multiuser transaction processing and database systems commonly use well-formed, two-phase locking to maintain data consistency [GRAY79, KOHL81]. An important consideration in the design of a locking protocol is the choice of the lock granularity. For example, some database systems lock individual areas, files, pages, or records. (IBM's IMS/VS and System R and UNIVAC's DMS 1100 are examples.) Simulation studies by Ries and Stonebraker [RIES77, RIES79] and queueing studies by Irani and Lin [IRAN79] have indicated that the size of the lockable unit can have a significant impact on the performance of the database system. Their work suggests that the lockable unit should be small enough so that when it is locked it does not critically restrict access to unwanted data items it also contains, but large enough to avoid the overhead of setting many individual locks.

In the preliminary experiments presented here, we have chosen to compare two levels, file and page locking. In all cases the database is assumed to be centralized and consist of a single file of fixed size pages. If a transaction only needs a few pages in the file, it should lock on a page by page basis (page level locking) instead of locking the whole file. The unwanted pages remain available to other transactions. We will call this type of transaction a small transaction. On the other hand, if the transaction actually needs a large number or significant percentage of all the pages within a file, then it should be able to lock the whole file with one lock. This will eliminate the overhead that would be introduced by locking each page individually. We will call this a large transaction. Our distinction between small and large transactions is not precise, but it will be useful to classify transactions into different classes depending on their behavior.

Hierarchical locking provides a mechanism by which transactions can optimize system performance by locking objects of different granularity. By explicitly locking a large data item, all component data items contained within are implicitly locked. For example, a large transaction could explicitly lock the entire file (large granule) with a single lock, and thus implicitly lock all the pages (small granules). (See the work of Gray et al. for additional information on lock modes and granularity of locks [GRAY76].)

* Present Address: ISACOMM, Atlanta, GA.

The objective of hierarchical locking is to enable transactions to use the lockable unit which maximizes system performance. For example, in an environment that contains only small transactions, small lockable units (pages) should probably be used to enable many transactions to run concurrently and thus overlap CPU and input/output activity. Locking overhead will be small because only a few data items are desired. In an environment that contains only large transactions, large lockable units (whole files) may be necessary to avoid excessive locking overhead. Concurrency will be low in any case because many common data items will be accessed by transactions. When a mix of large and small transactions are present, it may be desirable for each type of transaction to use a different lock granularity.

The results presented here are the first step of an experimental investigation of the impact of different locking policies and system characteristics on transaction performance. The experiments were run on a prototype transaction processing system known as the Concurrency and Recovery Algorithm Testbed (CARAT) [GARC83]. In order to compare different policies, we implemented a general purpose lock manager which supports multiple lock modes (exclusive and share) and hierarchical locking. Transaction throughput data was then gathered for variations of three basic locking policies (file level locking, page level locking, and mixed level locking) as a function of the lock mode and the file size. The transaction workload consisted of a mix of small and large transactions.

The potential advantage of our testbed approach, as opposed to simulation, is that the experiments are performed using a "real" system. The goal of a testbed system is to be simple and flexible enough to permit a variety of algorithms and design strategies to be implemented, measured, and understood, but realistic enough to behave, with proper scaling, like an authentic system. Since it is necessary to build a testbed to see if this balance can be achieved, we built CARAT. The initial experiments were chosen to explore the behavior of the testbed system and to identify deficiencies in the design. As a result of these tests, the CARAT architecture and implementation will be improved before additional experimentation is undertaken.

Section 2 provides an overview of the experimental environment. We present the transaction processing model that CARAT follows and briefly describe how this model was implemented as a collection of cooperating processes. We also describe the locking policies which will be compared. Section 3 presents the experiments that were conducted in order to explore the behavior of CARAT and different locking policies in a centralized environment. Experiments on a distributed system will be undertaken after changes in CARAT have been completed. Finally, Section 4 contains our conclusions.

2.0 EXPERIMENTAL ENVIRONMENT

The most important performance measure for a transaction processing system is transaction throughput, i.e., the number of successfully completed transactions per unit time. There are basically three methods by which we can obtain this performance information [CHEN77]: (1) queueing models and statistical analysis; (2) simulation models; and (3) monitoring the real system. Each of these methods are suitable under certain circumstances, depending on the type and accuracy of the information desired and the size of the investment in time and equipment which can be justified. Our approach is to monitor a testbed system, CARAT, under controlled test conditions. Monitoring may be carried out at the macroscopic level, e.g., by measurements of throughput rates, response times, queue lengths and utilizations, or at the microscopic level, e.g., by analysis of machine instruction traces. For the initial experiments presented here, the macroscopic approach was chosen. The long term goal is to use experimental studies to develop and validate queueing and simulation models which can then be more confidently used to compare other variations of the algorithms and architecture.

2.1 Logical Structure And Transaction Model

For a centralized database where all transactions run at the site of the database, the logical structure of CARAT can be illustrated by Figure 1. This is derived from the Bernstein and Goodman distributed database management system architecture model [BERN81]. There are four logical components: transactions, a transaction manager (TM), a data managers (DM), and stored data. Transactions communicate with the TM, and TM communicates with the DM. The DM manages data and functions similar to a back-end database processor.

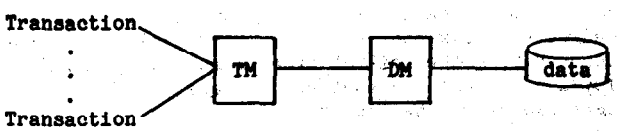


Figure 1. Centralized transaction processing system architecture.

In the CARAT system, transactions consist of a sequence of local computations, terminal input/output, and database requests bracketed by TBEGIN and TEND operations. This model of a transaction is shown in Figure 2. The database requests are performed by "steps" which are similar to subroutines. However, the step is executed by another cohort process, with parameter and return values passed via messages. Steps are written in FORTRAN and make data manipulation (DML) calls to a simple network database system called WAND [GERR76]. For example, if the database system supported a part-inventory database, steps would exist to add a part, delete a part, look up a part, modify a part, etc.

A complete transaction is composed of a sequence of step executions. A typical transaction might want to execute a step to look up a part, and then execute a step to modify some of the part data. A two-phase commit protocol is used to guarantee that the effects of a transaction's steps are atomic with respect to the stored database [GRAY79]. The TEND operation signals the end of a transaction and initiates the two-phase commit protocol.

```
TBEGIN
    local computation and i/o
    call step 1
    local computation and i/o
    call step 2
      .
      .
      .
    local computation and i/o
    call step n
    local computation and i/o
TEND
```

Figure 2. CARAT transaction model.

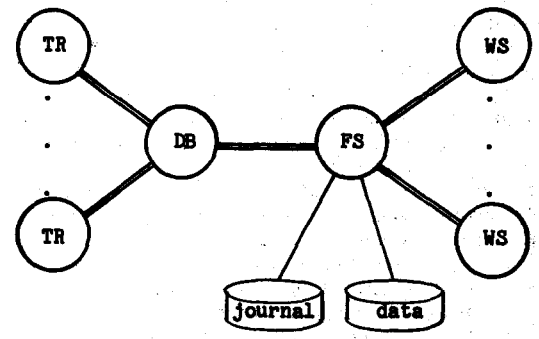
2.2 CARAT Implementation

CARAT is implemented as a collection of cooperating processes that communicate via messages. The CARAT processes belong to one of four types: user transaction process (TR), database server process (DB), WAND server process (WS), and file server process (FS). The processes at one node and their communication paths are shown in Figure 3.

This four process model was chosen for ease of implementation and generalization to other environments. For example, a simple back-end database system could be implemented by putting the FS and WS processes with the data on a separate dedicated back-end computer. However, we have learned from the initial experiments that this four level process structure severely limits the performance of the system.

Each process type has a few specified functions and may communicate with other process types using a well-defined set of message types. Additional details can be found in [GARC83]. The functions of each process type are discussed below.

Transaction Process (TR). TR is a user process which performs local computation and input/output, and requests cohort processes to execute WAND steps. (In our model, the WAND server (WS) processes described below serve as transaction cohorts.) In Figure 2 the step requests were indicated by calls. The calls are actually implemented by a message send/receive mechanism. An arbitrary number of TR's can exist at a node. Each TR sends its TBEGIN, TEND, and step requests to the database server (DB) which functions as the transaction manager (local controller) for the transaction. Upon receiving a TBEGIN message, the DB process generates and returns a unique timestamp which is then used to identify the transaction.



Key:

Process to Process Communication

Process

File

Figure 3. CARAT processes at one node.

Database Server Process (DB). The database server coordinates the execution of the transaction. It accepts requests to execute steps from user transactions and monitors their completion. The requests are forwarded to the file server (FS) which assigns a WAND server (WS) to execute the step. When a TEND message is received from a TR process, the database server process assumes the role of coordinator in the two-phase commit protocol.

WAND Server Process (WS). WAND server processes function as transaction cohorts. Each WAND server executes its own copy of the step and WAND database code. Each WAND server is a serially reusable resource which gets assigned to execute a specific step for a specific transaction. It is released when the step is done. To allow for increased concurrency among transactions, there are usually multiple WAND servers. If there are fewer WAND servers than transactions, concurrency control must be performed to synchronize their assignment. The file server (FS) performs this task.

When a WAND data manipulation command is executed by a step within a WAND server, the high level record request is converted into a request to lock and then read or write pages of the WAND database file. (Unlock requests are sent by the database server because it coordinates the two-phase commit). In the present implementation, requests for locks and database pages are serialized by the file server process (FS), which implements locking and is the only process allowed to read from or write into the stored database. When there are many concurrent transactions which are input/output intensive, the throughput of the file server will limit the overall performance. The number of pages in the database file and the size of a page are parameters which are set when the file is created.

The file server will reject a read or write request if the page has not been locked on behalf of the transaction.

During the execution of a step, the WAND server maintains a local buffer (workspace) of the four most recently used pages. When a page is received (read) from the file server, it becomes the most recently used, and the least recently used page is swapped out if the buffer is full. If the swapped out page has been modified, it is sent back to the file server which actually does the write to the stored database. However, before this is done the file server writes the original page on a before image journal maintained on disk (stable) storage. At the end of a step, any modified pages still in the buffer are also written out and journaled. The journal is used by the concurrency control and recovery mechanisms to roll back the changes made by aborted transactions.

The choice of four pages for each WAND server buffer seemed reasonable as a starting point, since most of the tested transactions require five pages on the average and the test database is small. Other sizes will be used in later experiments.

File Server Process (FS). In addition to managing the database and journal files as described above, the file server maintains the lock tables and executes the locking and concurrency control policies. This process services all queued message requests in FIFO order.

Measurements taken on CARAT, but not presented here due to space limitations, show that for our workload FS is the bottleneck in the system. This limits much of the potential concurrency and helps to explain some of the experimental results we have obtained. We have also observed CARAT to be CPU bound. A very large number of process-to-process messages are required per transaction and message processing is CPU intensive. We believe that an unreasonably large part of the CPU time is spent on message passing overhead. Further experiments are needed to quantify our suspicion. Now that we better understand the limitations of the current CARAT design, we intend to modify the architecture and implementation to enable CARAT to perform more like commercial systems.

2.3 Locking Policies And Concurrency Control

WAND supports a simple two-level data unit hierarchy consisting of a single database file which contains many pages. This allows for two locking granularities: a single large granule (the whole file) and many small granules (individual pages). WAND converts requests for database records to requests for pages by a hashing algorithm. Before a file or page is accessed, the proper lock must be granted by FS.

When locking is used as the concurrency control scheme, some mechanism must be used to avoid, prevent, or detect deadlock among transactions. For the experiments presented here, the wait-for-graph method is used for deadlock detection [RYPK79]. If a lock is available, the request is granted immediately. If the lock is not available, the request is put on a wait queue and the wait-for-graph is constructed to see if a cycle has been generated. If a cycle exists, the youngest transaction (one with the largest timestamp) is aborted. Since multiple cycles may have been created, this process is repeated until all cycles have been removed.

The three locking policies that are the basis for our experimentation and analysis will be called file level locking, page level locking, and mixed level locking. We will investigate two variations: the locks may be set in exclusive (EX) or share (S) mode. The policies are now defined.

File Level Locking Policy.

In the file level policy, an entire database file is treated as a single granule, which is locked by one lock. This policy is accomplished by having a transaction set an exclusive (EX) or share (S) lock on the file. This sets an implicit EX or S lock on each page.

We call this the file level locking policy to denote the fact that the lowest level data item that is explicitly locked is the file. No locks are explicitly set on the pages. From a performance point of view, this policy has the advantage of minimal locking overhead, but the disadvantage of minimal concurrency if the lock is exclusive. With file level exclusive locking, transactions that want to access any part of the file will execute sequentially.

Page Level Locking Policy.

In the page level policy, transactions contend for access to pages by setting individual locks on pages. Transactions are allowed to lock the file concurrently, but must wait or abort if they try to lock a page granted to another transaction in an incompatible mode. This policy is accomplished by having each update transaction set an intention exclusive (IX) lock on the file, and an exclusive (EX) lock on pages. Read-only transactions set an intention share (IS) lock on the file and share (S) locks on the pages. See [GRAY76, GRAY79] for more information on these and other lock modes.

We call this the page level locking policy to denote the fact that explicit locks are set on the pages. This policy has the advantage of higher concurrency since multiple transactions may use the database concurrently. But there is higher locking overhead because a transaction must obtain a lock on each page that it wishes to access. Furthermore, a transaction can be aborted due to deadlock.

Mixed Level Locking Policy.

The mixed level policy takes advantage of hierarchical locking. Those transactions that wish to access many pages (large transactions) lock at the file level to minimize the locking overhead. Transactions that only need a few pages (small transactions) lock at the page level to minimize conflict. This is accomplished by having large update transactions set an exclusive (EX) lock on the database file, while small update transactions set an intention exclusive (IX) lock on the file

and exclusive (EX) locks on individual pages. Similarly, large read-only transactions set a share (S) lock on the database file, while small read-only transactions set an intention share (IS) lock on the file and share (S) locks on individual pages.

We call this the mixed level locking policy because it is a mixture of the other two policies. With this policy only the small transactions can be aborted due to deadlock. Since the large transactions use the file level locking policy, they request a single lock and will never be aborted.

2.4 Experimental Setup

The Testbed Performance Monitor (TBPM) was used to control the experiments and compute transaction throughput rates [CHUN81]. One of the functions of TBPM is to act as a terminal emulator or transaction driver. It oversees the execution of multiple concurrent transaction processes. Each transaction process reads input from a script file and performs output to a log file. CARAT cannot distinguish between actual terminal users and user transaction processes controlled by TBPM.

The first step in performing an experiment is to define and create a centralized CARAT configuration. For example, the number of WAND servers, the database files, etc. must be specified. The second step is to define a workload and create the script files. The workload is defined by parameters which include the number of concurrent transactions, the type of transactions, the data to be included in the script, etc.

In our implementation, each transaction process is executed cyclically by the transaction driver. New script data, which is randomly generated according to a known distribution, is used for each cycle. The transaction process is responsible for keeping track of its own elapsed time. Upon completion, the elapsed time is sent back to the transaction driver in a message. The message contains the status of the completion, .i.e., completed successfully, aborted due to conflict with another transaction, or aborted due to "user" error. The transaction driver stores this information for later statistical processing.

For the experiments presented here, eight transaction processes were used. Each transaction waits for some external think time, runs, completes, and then waits again before restarting. The external think time is a random sample from an exponential distribution whose mean is specified as part of the workload characterization. The purpose of the external think time is to model the behavior of a terminal user who "thinks" for some period of time after completing one transaction before submitting another request. It represents the time between the completion of one transaction and initiation of the next from the same terminal. For the workload and locking policies used in our tests, the small transaction response times were measured to be approximately twice the external think time and the large transaction response times varied from one to eight times its external think time. This means that each transaction is in the system approximately two-thirds of the time and the degree of multiprogramming is about five.

The transactions enter the system by requesting a TBEGIN. When a step computation is requested, one of eight WAND servers is assigned to process the request. (Since the system has eight WAND servers and eight transactions, a WAND server is always available to process a step.) The WAND server executes the WAND step and returns the results. This is repeated until the transaction issues a TEND to terminate the transaction and commit any updates. The transaction then cycles around, and waits again. The duration of the experiment is defined by a run count parameter, where run count is the total number of transactions which must be completed before the experiment is stopped. As discussed in Section 3.1 below, a run count of 300 was found to be adequate for our tests.

The transactions used for the experiments execute one WAND step. We chose to design the transaction this way so that all overhead not related to locking, unlocking, and accessing pages is kept to a minimum and is held constant for all transactions. Each transaction performs the same function. It simply requests that the WAND server read a random set of records from the database file. The database was constructed so that each database page contained one database record. Pages were 256 bytes long. Runs were made with the database file containing up to 1024 pages.

Since practical database systems contain many orders of magnitude more pages, we realize that, for this and other reasons, the preliminary tests presented here are inadequate for drawing conclusions about practical systems. Additional tests on larger databases are required to determine how these results will scale up. However, tests on a small database are more manageable as a first step in exploring the testbed's behavior.

Seven of the eight transactions are considered small. They access on the average five records (or .5% of the 1024 page database) and have an external think time of 5 seconds. The eighth transaction is considered large and accesses an average of 102 records (or 10% of the 1024 page database). The large transaction has an external think time of 10 seconds. The number of records accessed by a particular small or large transaction is a random sample from a geometric distribution with a mean of 5 or 102 respectively. Because of the way the transactions are defined and the database is constructed, the actual records selected by a transaction are chosen uniformly with replacement from among all those in the database. Although access patterns of real systems have been observed to be highly non-uniform, uniform access was chosen for the initial tests since it would be easier to understand and model statistically.

The experiments were performed on a Digital Equipment Corporation VAX 11/780 running under the VMS operating system with four megabytes of main memory and two RM03 disk drives. No other "users" were on the system when the experiments were run. Table 1 summarizes the test environment.

Table 1. Summary of test environment.

Run Count	Size of database file
o 300 transactions	o 16 to 1024 pages

One Large Transaction	Seven Small Transactions
o average of 102 records o 10 second average external think time	o average of 5 records o 5 second average external think time

3.0 EXPERIMENTAL RESULTS

The four experiments presented in this section were performed to investigate the impact of different locking policies and system characteristics on throughput and response time. The test system is composed of one performance monitor (TBPM), eight transactions (TR's), one database server (DB), one file server (FS), eight WAND servers (one WS for each transaction), and one database file.

Since the large and small transactions are contending for the same resources, some locking policies may favor one class over the other. Consequently, we present the overall throughput, the throughput of the large transactions, and the total throughput of the seven small transactions. By definition, the sum of the large throughput plus the small throughput equals the overall throughput.

Experiment 1 shows that the results are repeatable given identical controllable test conditions. Experiments 2 and 3 identify conditions for which each of the file level, mixed level, and page level locking is best. Experiment 4 attempts to quantify the impact of two different factors on throughput.

3.1 Experiment 1: Confidence Intervals On Throughput

The purpose of this experiment was to obtain an estimate of the variability of our measured results on repeated "identical" test runs. The variability can be attributed to two factors: random events that are part of the underlying VAX/VMS system on which CARAT is running and random events that are part of the CARAT workload.

We selected a few cases from the experiments to be presented later, ran each case several times to collect independent data, and then computed the confidence intervals on throughput and response time. The significance of the statistics themselves are treated in the later sections. Here we simply investigate the variability of the results. The length of the run was another factor we had to consider. Through experimentation we found that increasing the run count beyond 300 did not significantly decrease the measured variance, so all experiments used a run count of 300.

Table 2 shows the results for three typical cases. Each case is identified by the number of pages in the file and the locking policy that was used. For the first case, the mixed level exclusive locking policy was used with a file of 1024 pages. In the second case the mixed level exclusive locking policy was used again, but this time the file contained only 64 pages. For the third case, the page level exclusive locking policy was used and the file contained 1024 pages. All throughput values are measured in transactions per second. Response times are measured in seconds. The sample mean and 90% confidence interval were computed and are shown beneath the data. Even based on three runs per case, the computed confidence intervals are very small. Because of these observations and the fact that we are only looking for trends in our initial tests, we decided that it would be adequate to collect just one datum value for each workload and locking policy to reduce the time required to perform the tests.

Table 2. 90% Confidence Intervals for Several Input Cases

Mixed Level Exclusive Locking Policy
No. of pages = 1024

	Throughput (trans./sec.)			Response Time (seconds)		
Run	Overall	Large	Small	Overall	Large	Small
1	.468	.050	.418	11.25	12.47	11.10
2	.466	.048	.418	11.22	12.80	11.04
3	.470	.049	.421	11.19	12.65	11.10
Mean	.468	.049	.419	11.22	12.64	11.05
+/-	(.002)	(.001)	(.002)	(.03)	(.16)	(.04)

Mixed Level Exclusive Locking Policy
No. of pages = 64

	Throughput (trans./sec.)			Response Time (seconds)		
Run	Overall	Large	Small	Overall	Large	Small
1	.444	.050	.393	11.24	12.04	11.13
2	.428	.049	.380	11.67	12.69	11.54
3	.458	.051	.407	10.92	12.43	10.74
Mean	.443	.050	.393	11.28	12.39	11.14
+/-	(.025)	(.001)	(.024)	(.36)	(.31)	(.38)

Page Level Exclusive Locking Policy
No. of pages = 1024

	Throughput (trans./sec.)			Response Time (seconds)		
Run	Overall	Large	Small	Overall	Large	Small
1	.444	.021	.423	11.16	38.91	9.92
2	.480	.020	.460	10.21	42.61	8.96
3	.467	.020	.447	10.48	39.60	9.21
Mean	.464	.020	.443	10.62	40.37	9.36
+/-	(.017)	(.001)	(.018)	(.46)	(1.9)	(.47)

3.2 Experiment 2: Baseline Comparisons

The purpose of this experiment was to compare the three locking policies: file level, page level, and mixed level locking under conditions prevalent in the testbed as described in Section 2. For this experiment, we ran each policy with exclusive locks (EX or IX) while varying the number of pages from 16 to 1024. Figures 4, 5, and 6 show the overall transaction throughput, the large transaction throughput, and the small transaction throughput respectively, for each of the three locking policies. This information is also shown in Table 3 along with read counts and percentage of abort statistics. All data is based on a single run for each configuration.

3.2.1 Throughput Vs. File Size -

We will first consider how the file size (number of pages) affects the throughput for each policy. Simulation studies have considered the situation in which a fixed size database is partitioned into a variable number of uniform sized granules, where the granule is the lockable unit [RIES77, RIES79]. In our experiments, pages are the lockable unit and decreasing the number of pages in the database file is similar (but not identical) to decreasing the number of granules. In our setup the pages are of fixed size for all experiments and transactions access fewer pages as the file size is decreased. This has the effect of increasing contention for locks and making the transaction less I/O intensive.

Figure 4 shows that when using file level locking the overall throughput increases slightly (less than 9%) as the number of pages in the file is decreased from 1024 to 16. Mixed level locking and page level locking show a different trend. As the file size is reduced from 1024 to 16 pages, throughput goes down, over 12% for mixed level locking and over 31% for page level locking.

This phenomenon can be explained as follows. Although the probability of a transaction requesting a page already in the buffer goes up, the probability of a conflict over a page, and therefore the probability of an abort (due to deadlock) also goes up. For the mixed level locking policy, it is the small transactions that are contending for the pages, and they are the ones that get aborted. The large transaction cannot be aborted because it only sets an explicit exclusive (EX) lock on the file. For page level locking, both the large transaction and the small transactions explicitly lock pages, and both can be aborted if deadlock occurs. It appears that the advantage of accessing a page already in the buffer is outweighed by the disadvantage of aborting.

The percent of transactions aborted as a function of the number of pages in the database file for mixed and page locking policies is shown in Table 3. (The percent of transactions aborted, Abort (%), was computed as the total number of aborted transactions divided by the sum of the successful and aborted transactions.) As expected with mixed level locking, the large transactions are never aborted. The small transactions are not aborted either until the file size is reduced to 256 pages, at which time 2.5% get aborted. The percentage stays fairly low until the file size is reduced below 64 pages and then it increases rapidly as the page size is further reduced to 16 pages. With page level locking, large and small transactions can be aborted. Table 3 shows that, as the file size is decreased, the trend is for both types to get aborted much more frequently than with the mixed level locking policy.

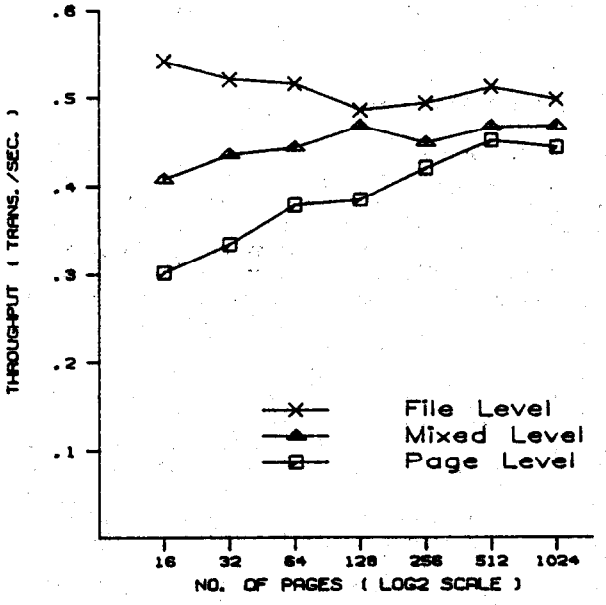


Figure 4. Baseline: Overall transaction throughput.

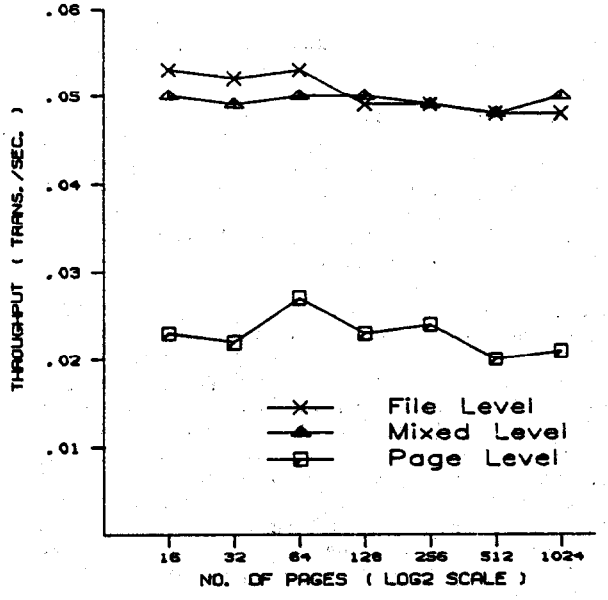


Figure 5. Baseline: Large transaction throughput.

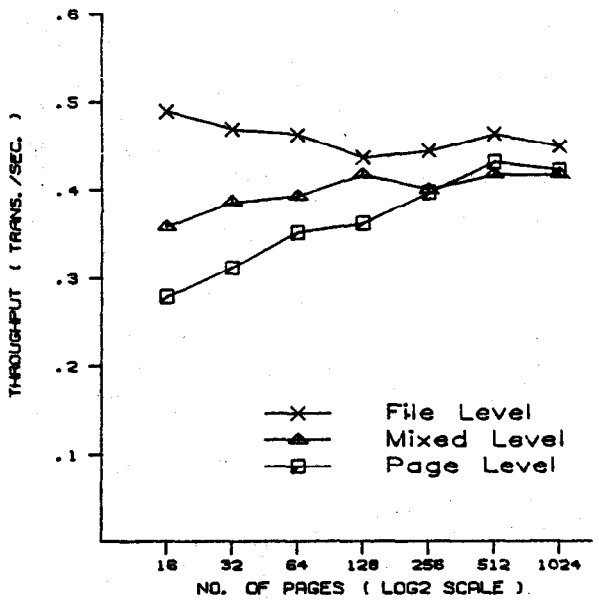


Figure 6. Baseline: Small transaction throughput.

Table 3. Baseline Data.

File Level Exclusive Locking Policy

Pages	Throughput (trans./sec.) Overall	Large	Small	Read Count
1024	.498	.048	.450	3767
512	.512	.048	.464	3761
256	.494	.049	.445	3788
128	.486	.049	.437	3744
64	.516	.053	.463	3663
32	.521	.052	.469	3440
16	.542	.053	.489	3059

Mixed Level Exclusive Locking Policy

Pages	Throughput (trans./sec.) Overall	Large	Small	Abort (%) Large	Small
1024	.468	.050	.418	0.0	0.0
512	.467	.048	.418	0.0	0.0
256	.449	.049	.400	0.0	2.5
128	.468	.050	.418	0.0	2.9
64	.444	.050	.393	0.0	6.3
32	.436	.049	.386	0.0	15.3
16	.408	.050	.358	0.0	27.1

Page Level Exclusive Locking Policy

Pages	Throughput (trans./sec.) Overall	Large	Small	Abort (%) Large	Small
1024	.444	.021	.423	0.0	3.4
512	.452	.020	.432	7.1	8.0
256	.421	.024	.397	0.0	13.2
128	.385	.023	.362	35.7	21.5
64	.379	.027	.352	38.2	27.7
32	.334	.022	.312	62.3	39.7
16	.302	.023	.279	63.5	49.7

3.2.2 Throughput Vs. Locking Policy -

We now discuss the relative performance of the three locking policies for a fixed size file of 1024 pages. (Similar comparisons can be made for other file sizes studied here.) Table 3 and Figure 4 show that for our baseline sample workload, file level locking was the best in terms of overall throughput. The next best was mixed level locking, and then finally page level locking. For the large transactions, Figure 5 shows that file level locking, and mixed level locking were almost equal. Page level locking was by far the worst. For the small transactions shown in Figure 6, file level locking was the best, then mixed level locking, followed by page level locking. We stress that these results are true for the baseline workload and system characteristics but will not be true under other circumstances.

In order to understand why one policy is better than another for our sample workload, one must understand the basic differences in how they execute. We will start by comparing the two performance extremes: file level and page level locking. File level locking has the advantage of minimal locking overhead, but the disadvantage of minimal concurrency. On the other hand, page level locking has the advantage of maximum concurrency (which is low in our case because the CARAT testbed is CPU bound), but the disadvantage of maximum locking overhead. The best policy for a particular situation depends on whether the improvement in performance due to increased concurrency is outweighed by the loss due to increased locking overhead. For the particular workload and system characteristics used here, the benefits of concurrency (minimal) are outweighed by the overhead of setting locks (expensive). Other experiments, which we present in Section 3.3, demonstrate that file level locking is not best under all circumstances.

Figure 4 shows an improvement in throughput performance with the mixed level locking policy compared with page level locking, so there is an advantage to reducing the locking overhead. However, file level locking still performed better than the mixed level because the penalty of locking was greater than the benefit from concurrency, even for the small transactions. The small transactions run so quickly that they are better off running sequentially by setting one lock on the entire database file than running in parallel and setting multiple locks on individual pages. We believe this is a result of the current structure of CARAT which is both CPU bound and has a bottleneck in the file server (FS) process. Modifications to CARAT to increase concurrency and reduce lock overhead are now being made. Finally, note that the large transaction throughput is approximately the same for file level locking and mixed level locking, because in both cases only a single lock is set and contention from small transactions is about the same.

3.3 Experiment 3: Transaction Duration Vs. Lock Overhead

In the previous experiment, where the throughput

with file level locking was shown to be better than both page level and mixed level locking, transactions were short and the system was CPU bound due to message traffic. The purpose of Experiment 3 was to investigate the sensitivity of the baseline results to other workload and system characteristics. In particular, we chose to investigate how the duration of a transaction and the time required to set a lock can affect performance.

We added an internal think time to control the duration of a transaction without changing the amount of work that it performed. After the transaction has been granted all requested locks and has completed processing, it hibernates for an additional think time before committing and releasing its locks. This internal think time is chosen as a random sample from an exponential distribution with a specified mean. During the internal think time, the transaction is not using up CPU cycles. It is just holding its locks. It can be thought of as approximating a transaction which waits for the user to decide if the updates are to be committed or rolled back.

We also added a constant lock time wait to the processing of a transaction's lock request by the lock manager within the file server process. The lock time wait increases the time required to set a lock and can be used to approximate the effect of increased locking overhead that might exist in other systems, for example, due to paging lock tables and/or due to waiting incurred because of a very busy system.

The internal think time and the extra lock time wait are input parameters to the experiment. The results shown in Table 4 are for an average internal think time of 3.0 seconds, and a constant lock time wait of 0.3 seconds. While other "waits" were tested, this extremely high lock time is presented here to demonstrate an extreme condition. Three cases were run for each of the three exclusive mode locking policies of Experiment 2. For all the runs, the database file contained 1024 pages. Case 1 added internal processing time only, Case 2 added lock time wait only, and Case 3 added both. The results, which will be discussed below, give some insight into the impact of transaction duration and locking overhead on throughput performance.

3.3.1 Internal Think Time Overhead -

In Case 1, where an average internal think time of 3 seconds was added, page level locking produced the best overall throughput, followed by mixed level, and then file level locking. The overall throughputs were .387, .328, and .202 transactions/sec., respectively. This is the inverse of the baseline case, Experiment 2, where file locking was the best. The Experiment 3 results can be explained as follows. Since transaction duration has been increased, the benefit of concurrency is higher. For the small transactions the penalty of locking is relatively small since they only access a few pages. Therefore, for the small transactions the best policy is for all transactions to lock at the page level. The second best policy for the small transactions is mixed level locking, where the small transactions still lock at the page level but the large transactions lock at the file level. The small transactions do not perform as well with this policy because they now must wait for the large transaction to execute. The worst policy for the small transactions is the file level policy because the duration of transactions is long and all processing is effectively sequential.

Table 4. Effect of Additional Overhead.

Case 1: Avg. Internal Think Time = 3.0 Seconds
Lock Time Wait = 0.0 Seconds
Number of pages = 1024

Locking Policy	Throughput (transactions/sec.) Overall	Large	Small
File	.202	.023	.179
Mixed	.328	.035	.293
Page	.387	.019	.368

Case 2: Avg. Internal Think Time = 0.0 Seconds
Lock Time Wait = 0.3 Seconds
Number of pages = 1024

Locking Policy	Throughput (transactions/sec.) Overall	Large	Small
File	.448	.045	.403
Mixed	.285	.032	.253
Page	.229	.008	.221

Case 3: Avg. Internal Think Time = 3.0 Seconds
Lock Time Wait = 0.3 Seconds
Number of pages = 1024

Locking Policy	Throughput (transactions/sec.) Overall	Large	Small
File	.199	.023	.176
Mixed	.227	.026	.201
Page	.205	.007	.198

For the large transactions, it is better to run sequentially with respect to the small transactions, and thus reduce locking overhead for itself. This is shown by the fact that the best policy for the large transaction was the mixed level locking policy with a throughput of .035 transactions/sec. This is followed by the file level locking policy with a throughput of .023 transactions/sec. The large transactions did not perform as well for the file level locking policy because the large transactions must also wait for the small transactions to be executed sequentially. The worst policy for the large transactions is page level locking where the large transactions have to compete with the small transactions in setting locks on each page.

The policy producing the highest overall throughput (large plus small transactions) was page level

locking. The throughput of the small transactions was high enough to make up for the low throughput of the large transactions. This case points out the interaction between the two classes of transactions and shows that the best policy for one class may not be the best for another.

3.3.2 Lock Time Wait -

In Case 2, a large (0.3 second/lock) lock time wait was added. As expected under this condition, the file level locking policy supported the highest overall throughput, .448 transactions/sec., followed by the mixed level with a throughput of .285 transactions/sec., and then page level with a throughput of .229 transactions/sec. Since the total locking overhead is now extremely large, performance is best where the number of locks required per transaction is minimal. This occurs for the file locking policy where each transaction sets a single lock on the file.

Even the small transactions performed best with file level locking. Performance dropped significantly for the mixed level and page level policies where the small transactions set locks on every page they access. The large transaction throughput also went down sharply (from .045 transactions/sec. to .008 transactions/sec.) for page level locking due to the impact of locking overhead and contention for pages.

3.3.3 Internal Think Time And Lock Time Wait -

In Case 3, both the lock time wait and internal think time were included. The effect was a reduction in overall throughput for each locking policy. File level locking and page level locking show the lowest overall throughput with values of .199 and .205 transactions/sec., respectively. We conjecture that the throughput for the file level policy is low for the same reasons as in Case 1, but the difference when compared with the other policies is not as pronounced. As in Case 2, page level locking throughput is also low because the penalty of locking is very high. Finally, mixed level locking is the best overall with a throughput of .227 transactions/sec. It is best among the three policies for this particular workload because it reduces the penalty of locking for the large transaction, but maximizes the benefit of concurrency for the small transactions.

3.4 Experiment 4: Concurrency And Locking

Experiment 4 quantifies two things for the CARAT system: one, the difference between maximum throughput potential and actual throughput as limited by the lock policies, and two, the throughput overhead of asking for and setting locks. To do this five variations of the locking policy are compared for the baseline workload described in Table 1. Three of the policies are ones studied above, file, mixed and page level locking. The two new policies are:

No Locks - With this policy no locks are requested or set. While this could leave the database in an inconsistant state if updates were performed, it is useful for comparison purposes. Since no locks are requested, we have eliminated the overhead of waiting for and setting locks. Since throughput is not being limited by concurrency control or locking overhead, we are measuring the maximum throughput potential. Of course, actual throughput and concurrency is still limited by the system and the workload. Concurrent transactions must timeshare a single physical CPU and the file server sequentializes access to the stored database file.

Mixed Level Read - This is the mixed level locking policy with all locks set in intention share (IS) or share (S) mode as approporate. This policy incurs the same locking overhead as the mixed level exclusive locking policy, but transactions never wait due to concurrency control since all locks are set in share mode. There is maximum concurrency but the penalty of mixed level locking still exists.

Figure 7 shows the overall throughput. The individual throughputs for the large and small transactions are not shown due to space limitations. The results presented for file level, mixed level, and page level exclusive locking are taken from the baseline comparisons (Experiment 2).

3.4.1 Internal Think Time Overhead -

In Case 1, where an average internal think time of 3 seconds was added, page level locking produced the best overall throughput, followed by mixed level, and then file level locking. The overall throughputs were .387, .328, and .202 transactions/sec., respectively. This is the inverse of the baseline case, Experiment 2, where file locking was the best. The Experiment 3 results can be explained as follows. Since transaction duration has been increased, the benefit of concurrency is higher. For the small transactions the penalty of locking is relatively small since they only access a few pages. Therefore, for the small transactions the best policy is for all transactions to lock at the page level. The second best policy for the small transactions is mixed level locking, where the small transactions still lock at the page level but the large transactions lock at the file level. The small transactions do not perform as well with this policy because they now must wait for the large transaction to execute. The worst policy for the small transactions is the file level policy because the duration of transactions is long and all processing is effectively sequential.

Figure 7 has two different line styles. The dashed lines are for the no locks and mixed level read policies. These policies are called no waiting policies because the lock request will always be granted immediately, even if another transaction is accessing the page. There is maximum concurrency with these policies. The other three policies, file level exclusive, mixed level exclusive, and page level exclusive are plotted with solid lines to denote that lock requests may have to wait.

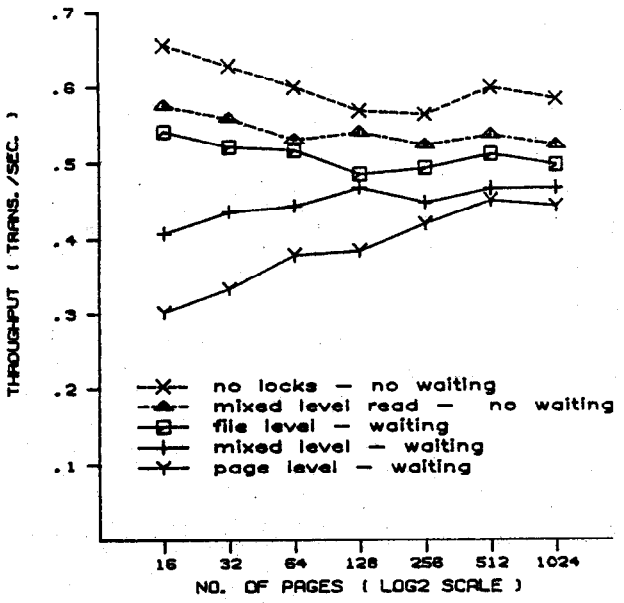


Figure 7. Factors - Overall transaction throughput.

For the no waiting policies, no transactions can be aborted because no transactions wait. This is also true for the file level exclusive locking policy. In that case, deadlock cannot occur because transactions are only contending for a single file and lock mode conversion is not allowed.

We will first consider the general trends of the curves as a function of the size of the file. For overall throughput in Figure 7, there are two distinct classes of curves. The top three curves show an increase in throughput as the number of pages decreases. The bottom two curves show a decrease. These results can be explained as follows. The top three curves include the no waiting policies plus the file level exclusive locking policy. The three policies have one thing in common, there is no contention (or waiting) for pages. Since there is no contention for pages, the dominant performance factor is the number of messages sent and new pages read. As the the number of pages in the file decreases, the probability of having the desired page already in the WAND server buffer increases. Therefore, fewer messages are sent to lock and read pages. These meassages are costly in CPU time. By reducing the number of messages and file server page I/O, throughput is increased.

The bottom two curves include page level exclusive, and mixed level exclusive policies. Both of these are waiting policies, .i.e., transactions contend for exclusive access to the database file or its pages. As explained in the baseline experiment (Experiment 2), the throughput decreases as the number of pages decreases because there is a significant increase in the percent of transactions aborted. This outweighs the benefit of sending a reduced number of messages and reading fewer pages.

Comparing the no locks curve to each of the "waiting" curves provides an approximate quantification (in terms of throughput) of the difference between maximum potential concurrency and actual concurrency. Actual concurrency in the "waiting" policies is limited by high lock overhead, by the FS sequentializing access to the disk, and by concurrency control itself. For example, from Figure 7 it can be seen that for 1024 pages there is a 15% loss in throughput between the no locks policy and file level locking, a 21% loss for mixed level locking and a 25% loss for page level locking.

Next, comparing the mixed level read curve to the mixed level exclusive policy provides a quantification of the effect on throughput of lock conflict itself. Again, from Figure 7 it is seen that at 1024 pages the effect is a 10% reduction in throughput. As expected, the curves also show that the loss is greater for smaller file sizes.

Finally, comparing the no locks and mixed level read curves quantifies the difference in throughput as a result of locking overhead. This effect is approximately a 12% loss in throughput.

4.0 CONCLUSIONS

The experimental data we have presented for our testbed system indicates that the "best" locking policy depends on the transaction and system characteristics. For example, in the baseline tests file level locking is best; when internal think time is added page level locking is best; when lock wait time is added file level locking is best; and when both internal think time and lock wait time are added mixed level locking is best. We have also presented data that attempts to quantify the effect on throughput of both lock conflicts and locking overhead. We have not run enough different workloads and databases to be able to say what will happen under other circumstances. Our system results may also be sensitive to other uncontrolled parameters and scaling. However, by comparing different locking policies for specific workloads on a specific implementation of a testbed, we have a better understanding of the factors which might influence transaction throughput performance when well-formed, two-phase locking is used for concurrency control in a centralized database system.

Finally, some of the preliminary results have also identified needed modifications to the testbed itself. CARAT will be modified to remove the file server bottleneck and as much of the message overhead as possible. The four process structure will be reduced to three by incorporating the file server functions into the WAND servers. This should help make the testbed more realistic and yet still retain its flexibility.

Acknowledgments. We would like to thank the reviewers for their constructive criticisms and suggestions, which have helped to revise the presentation and improve the course of our future research. The CARAT project was started by Frank Germano, now at APOLLO Computer, and has been supported for the past two years by the Corporate Research Group and the RAD committee of Digital Equipment Corporation. It is now being supported by the National Science Foundation under Grant

ECS-8120931 to the University of Massachusetts. Other people who have contributed to the project as part-time consultants at Digital Equipment Corporation include Hector Garcia-Molina, Mike Chung, Bao-Chyuan Jenq, Jack Kent, Paul Selsky, and Shou-Pin Yu.

DISCLAIMER. This document describes a research prototype. No DEC product direction or intent should be assumed.

5.0 REFERENCES

[BERN81] Bernstein, P.A., N. Goodman, "Concurrency control in Distributed Database Systems," Computing Surveys, Vol. 13, No. 2, June 1981, pp. 185-221.

[CHEN77] Chen, P.P. and Yao, S.B., "Design and Performance Tools for Data Base Systems," Proceedings, 3rd International Conference on Very Large Data Bases, 1977.

[CHUN81] Chung, M., "Design and Implementation of the Testbed Performance Monitor," Technical Report, Corporate Research Group, Digital Equipment Corporation, September 1981, 19 pages.

[GARC83] Garcia-Molina, H., F. Germano, and W. H. Kohler, "Architectural Overview of a Distributed Software Testbed," Proceedings Sixteenth Hawaii International Conference on System Sciences, January 1983.

[GERR76] Gerritsen, R., et al., "WAND User's Guide," Department of Decision Science, The Wharton School, University of Pennsyvania, April 15, 1976.

[GRAY76] Gray, J.N., R.A. Lorie, G.R. Putzolu, and I.L. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base," in Modeling in Data Base Management Systems, Nijssen, editor, North Holland, 1976.

[GRAY79] Gray, J.N., "Notes on Data Base Operating Systems," in Operating Systems: An Advance Course, R. Bayer, R. H. Graham, and G. Seegmuller, Editors, Springer-Verlag, 1979, Sections 5.7.6 - 5.7.7.3, pages 438 - 450.

[IRAN79] Irani, K.B., and H.L. Lin, "Queueing Network Models for Concurrent Transaction Processing in a Database System," Proceedings SIGMOD International Conference on Management of Data, 1979, pp. 134-142.

[KOHL81] Kohler, W.H., "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," Computing Surveys, Vol. 13, No. 2, June 1981, pp. 149-183.

[RIES77] Ries, D.R., M.R. Stonebraker, "Effects of Locking Granularity in a Database Management System," ACM Transactions on Database Systems, Vol. 2, No. 3, September 1977, pp. 233-246.

[RIES79] Ries, D.R., M.R. Stonebraker, "Locking Granularity Revisted," ACM Transactions on Database Systems, Vol. 4, No. 2, June 1979, pp. 210-227.

[RYPK79] Rypka, D.J., and A.P. Lucido, "Deadlock Detection and Avoidance for Shared Logical Resources," IEEE Transactions on Software Engineering, Vol. SE-5, No. 5, September 1979, pp. 465-471.