Comparison of Distributed Concurrency Control Protocols on a Distributed Database Testbed*

Chia-Shiang Shih and Asit Dan

ECE Department, University of Massachusetts
Amherst, MA 01003

Walter H. Kohler†

Digital Equipment Corporation
200 Forest Street, MRO1-1/A65
Marlboro, MA 01752-9101

John A. Stankovic and Don Towsley

COINS Department, University of Massachusetts
Amherst, MA 01003

*This work was supported by the National Science Foundation, grant number SDB-8418216 and by a grant from Digital Equipment Corporation.

†Walter H. Kohler is manager of High Performance Transaction Processing System group at Digital Equipment Corporation.

Abstract

In this paper, we compare the performance of several concurrency control protocols by executing these protocols under various workloads in a common distributed database testbed environment. We study the effect of implementation overhead on the performance of three classes of distributed concurrency control protocols: two-phase locking, optimistic approach with backward validation, and optimistic approach with forward validation. We develop and evaluate several optimizations for optimistic concurrency control protocols with backward validation. To minimize the non-essential variations in implementation, we use a common hash table implementation for both the lock table as well as the validation table. The main emphasis here is to contrast the difference in protocol overhead based on the number of high level operations required by different protocols while taking the common low level implementation overhead into consideration. We believe that this is one of the first comprehensive, *experimental* study of distributed concurrency control protocols.

In our experimental environment, two-phase locking performed significantly better than the backward validation optimistic approaches on most of the workloads (except for read-only workloads). For query-intensive (read-only) workloads, optimistic protocols with backward validation can simplify their validation check and outperform both two-phase locking and the optimistic with forward validation. Two-phase locking also performed better than optimistic with forward validation (except for the long response time situations). We observed that the locking protocol usually required less system resources (CPU and Disk I/O) than the optimistic approaches and supported higher throughput. On some of the workloads, the most important performance degradation factor for the two-phase locking protocol was a long blocking time on locks whereas it was the high transaction abort rate for the optimistic protocols, especially for the ones with backward validation.

1 Introduction

During the past decade, numerous studies have been made comparing the throughput of various concurrency control protocols in both centralized and distributed database systems[1, 4, 6, 7, 9, 10, 15, 17, 18]. All of these works are based on simulation or analytical modeling. In general these studies either ignore the effect of protocol overhead or assume equal overhead for all protocols. The primary arguments given for these assumptions are as follows:

- If the resource requirement (CPU, I/O) of executing any transaction is large compared to the protocol overhead, the variation of protocol overhead introduces very little error.
- It is hard to compare protocol overhead because of implementation details. Any given protocol can be implemented in several ways.

We report on one of the first comprehensive experimental comparison of various distributed concurrency control protocols by executing these protocols under various workloads in a common distributed database testbed environment known as CARAT (Concurrency And Recovery Algorithm Testbed)[12, 16]. We study the effect of implementation overhead on the throughput under various distributed concurrency control protocols including two-phase locking[5], four variations of optimistic approaches with backward validation[13], and an optimistic approach with forward validation[4]. To minimize the non-essential variations in implementation, we use a common implementation of hash table both for the lock table as well as validation table. The main emphasis here is to contrast the difference in protocol overhead based on the number of high level operations required by different protocols while taking the common low level implementation overhead into consideration.

In our experimental environment, two-phase locking performed significantly better (higher throughput and lower response time) than the backward validation optimistic approaches on most of the workloads (except read-only workloads). The throughput of two-phase locking is also higher than that of optimistic protocol with forward validation. However, optimistic protocol with forward validation outperforms two-phase locking protocol for transactions with longer response time (due to longer transaction size or distributed execution). For query-intensive workloads (read-only workloads), optimistic protocols with backward validation can simplify their validation check and outperform both two-phase locking and optimistic protocol with forward validation. Besides supporting higher throughput, we observed that the locking protocol usually required less system resources (CPU and Disk I/O) than the optimistic approaches. On some of the workloads, the

most important performance degradation factor for the two-phase locking protocol was due to long blocking time on locks whereas it was due to the high transaction abort rate for the optimistic protocols, especially for the ones with backward validation.

The details of the CARAT testbed environment will be described in Section 2. Section 3 outlines the implementation of six different protocols: two-phase locking, four variations of optimistic protocols with backward validation, and an optimistic concurrency control protocol with forward validation. Section 4 describes the performance metrics and the various workloads under which the experiments are performed. Section 5 compares the performance of these six protocols under the different workloads. Finally, we summarize the paper in Section 6.

2 Description of the CARAT Testbed

CARAT (Concurrency And Recovery Algorithm Testbed) is a distributed database testbed developed at the University of Massachusetts to study the performance of various concurrency and recovery algorithms through direct measurement[12]. It is implemented as a set of cooperating server processes which communicate via a uniform message passing mechanism. Figure 1 illustrates the processes and message structure of CARAT for any two nodes of the system. In each node there is a TM (Transaction Manager) server process and a pool of DM (Data Manager) server processes, which are created during system start up. TR (Transaction) processes are created by users to execute database transactions. To access any data granule, the TR process sends a TDO (read/write) message to its local TM. TM forwards this message to a DM server process for actual database access. For any data granule access from a remote site, the local TM forwards the TDO message to the remote TM which in turn forwards it to one of its local DM servers. The motivation behind this message passing architecture as well as the details of this architecture can be found in [12]. The TR process also sends TBEGIN and TEND messages to the TM process to mark the beginning and the end of a transaction. Depending on the concurrency control protocol, TM and DM processes take various actions to ensure transaction serializibility. For example, for a locking based protocol, the DM acquires a lock on behalf of the transaction at each TDO step, and the TM executes a two phase commit protocol after receiving a TEND message.

For the purpose of performance measurement, transactions are created automatically by the TD (Test Driver) process according to a specified workload. The DC (Data Collector) process is responsible for collecting performance statistics. The SPY process in each node can monitor and

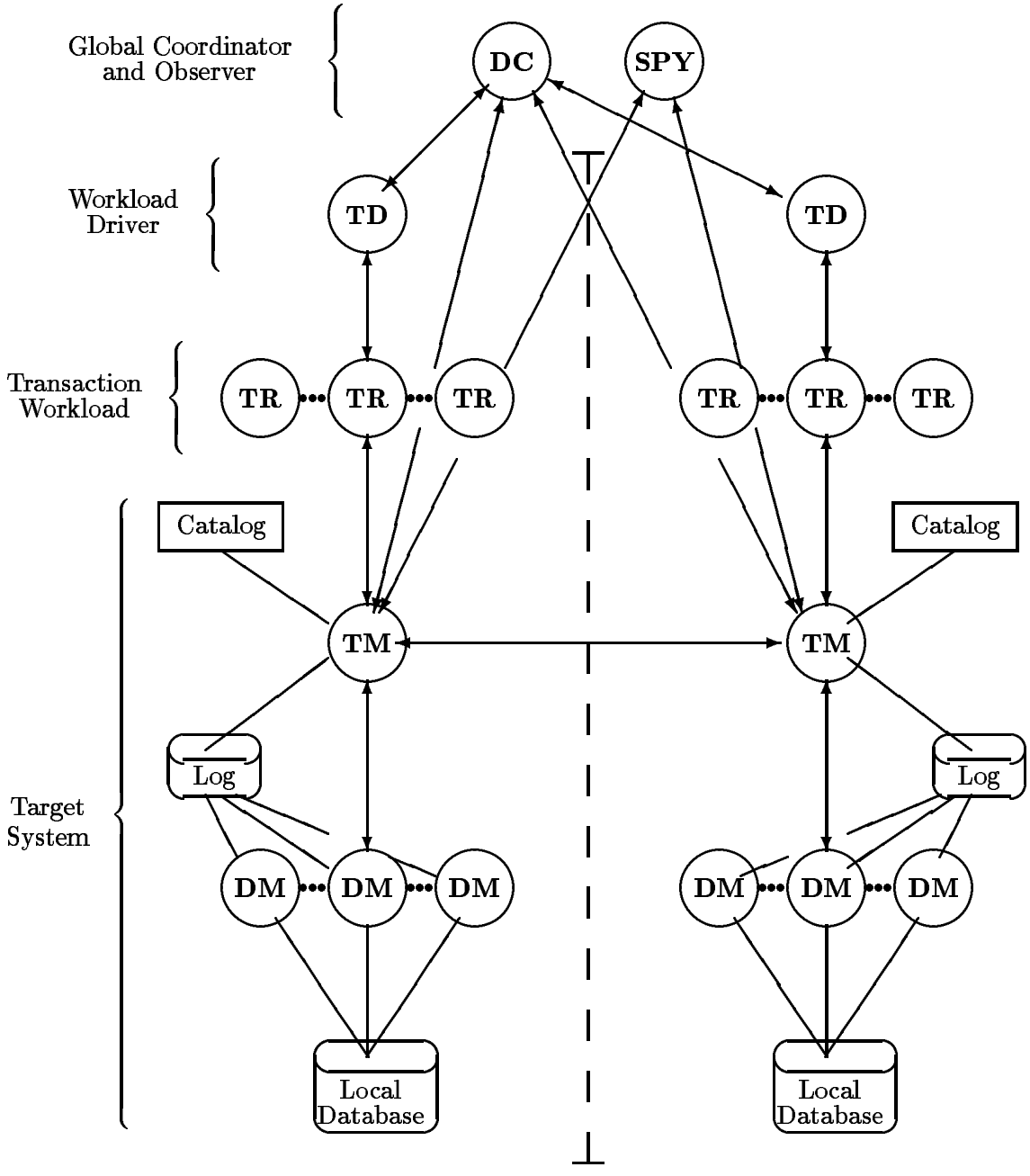


Figure 1: CARAT processes and message structure

display the system status while the transactions are executing and it is used primarily for debugging purposes.

The physical environment for the experiments includes a non-replicated database that is partitioned and distributed among five nodes where each node consists of a microVAX II and two RD53 disks — one database disk and one log disk. The nodes are connected by a local area DECnet.

3 Description of the Protocols

We have implemented three classes of protocols: two-phase locking (2PL), and optimistic with backward validation (OCC-BV) and optimistic with forward validation (OCC-FV). For the optimistic with backward validation, we have implemented four versions, differing only in how they perform their validation. To minimize the non-essential differences in the implementation of the different classes of protocols, we have tried to use standard data structures, such as a hash table for the implementation of both the lock table as well as the validation table. Other than the implementation of the protocols themselves, the remainder of the database system modules such as the Transaction manager, the Transaction process, the Buffer manager etc. are the same in all versions of the protocols. In the remainder of this section, we briefly describe all of these protocols. We note that these protocols rely only on the information about the read-sets and write-sets of a transaction. Optimization based on semantic information, such as the database and transaction structure, is not covered in this paper.

3.1 Two-Phase Locking with Distributed Deadlock Detection

Two-phase locking[5] protocols consist of two phases: a growing phase where locks are acquired, followed by a shrinking phase where locks are released. The protocol is implemented in the Lock Manager module at each node[1], that uses a hashing technique to manage lock tables containing Lock-Grant queues and Resource-Wait queues. The Buffer manager module that manages the private copy of each transaction issues a read or write lock request before reading a page into the buffer. A lock request is queued for a resource if the requested lock mode is not compatible with the locks held by other transactions or with other queued lock requests for the resource. All locks are held until a transaction is committed or rolled back.

The lock manager uses a hash table for the following operations on a data granule:

[1]The lock manager at each site is responsible for its local data items. Recall that the database is not replicated.

1. search and check its lock compatibility against the lock mode of other transactions which are currently holding or waiting for the lock of this data granule in the hash table;

2. insert it into the hash table if a lock is granted;

3. delete it from the hash table whenever the lock is released.

Both local deadlocks and global deadlocks are resolved by detection. Local deadlocks are detected by searching cycles in the Transaction-Wait-For-Graph that is encoded in a two dimensional array. Distributed deadlock detection is currently implemented with probe algorithm developed by Chandy and Misra[3, 2] based on probes. A complete description of the distributed deadlock detection implementation in CARAT can be found in [11] and [12].

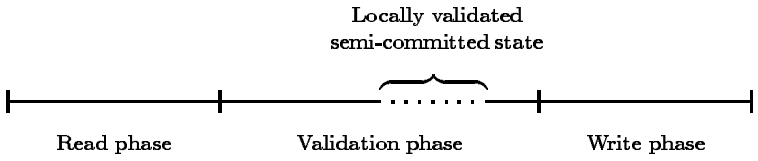
3.2 Optimistic Protocol with Backward Validation

Optimistic protocols using backward validation were first proposed by Kung and Robinson[13]. Here, transactions are executed in three phases, read, validation, and write. During the read phase, a transaction reads from the database and any updates are done on local copies of the data. At the end of the transaction, the transaction enters a validation phase to check to see if there is a serialization conflict with any other transaction. If the validation test is not passed, the transaction is aborted (and restarted). Otherwise, it enters the write phase to make the transaction updates permanent to the database. For read-only transactions, the write phase is not required.

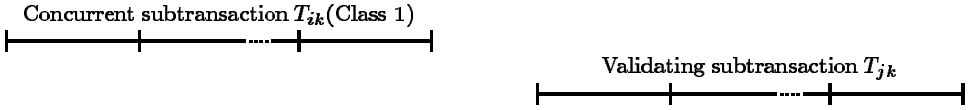
Figure 2-(i) shows the three-phase structure of a subtransaction T_{jk} (the part of transaction T_j executed at site k). For distributed transactions, validation is done using the following two-phase commit protocol:

1. In the first phase, local validation is carried out on behalf of the subtransactions of a transaction at each participating node independently and in parallel. At each node local validation is done within a node-wide critical section, but no system-wide global critical section is needed. If the subtransaction T_{jk} is successfully validated locally, the write-sets and/or read-sets of validated transactions are kept in a hash table to be used for the validation of other concurrent transactions.

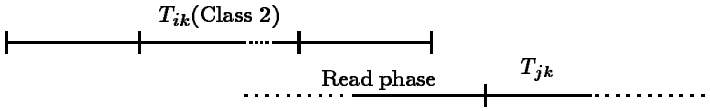
2. In the second phase, if any of the subtransactions fail to validate then the transaction is aborted (and restarted). Otherwise, the transaction T_j is globally validated and if T_{jk} is an update subtransaction, it enters its write phase.



(i) Structure of a subtransaction T_{jk}

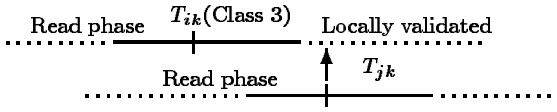


(ii) Condition 1



Write-set of T_{ik} $\cap$ Read-set of $T_{jk} = \phi$

(iii) Condition 2



Write-set of T_{ik} $\cap$ (Read-set of T_{jk} $\cup$ Write-set of T_{jk})= ϕ
AND
(a) Read-set of T_{ik} $\cap$ Write-set of $T_{jk} = \phi$
OR
(b) The validating subtransaction T_{jk} waits for the global outcome of all class 3 subtransactions T_{ik} before its local validation is completed

(iv) Condition 3

Figure 2: Local Validation Conditions for Distributed OCC

From the point of view of a validating subtransaction, T_{jk}, all other subtransactions T_{ik}, which executed at the same node and finished their read phase earlier, can be categorized into three classes. The three classes and their local validation conditions for a validating subtransaction T_{jk} are:

1. Class 1 consists of all the subtransactions T_{ik} which have completed their write phase before the validating subtransaction T_{jk} starts its read phase (Figure 2-(ii)). No conflict checking against this class of subtransactions is needed to assure the validation of T_{jk}.

2. Class 2 consists of all the subtransactions T_{ik} which have completed their write phase during the read phase of the validating subtransaction T_{jk} (Figure 2-(iii)). T_{jk} is locally validated if it does not conflict with any of this class of subtransactions T_{ik},

$$\text{Write-set of } T_{ik} \cap \text{Read-set of } T_{jk} = \phi.$$

3. Class 3 consists of all the subtransactions T_{ik} which have terminated their read phase during the read phase of the validating subtransaction T_{jk}, but have not yet terminated their executions (Figure 2-(iv)). T_{jk} is locally validated if it does not conflict with any of this class of subtransactions T_{ik},

$$\text{Write-set of } T_{ik} \cap (\text{Read-set of } T_{jk} \cup \text{Write-set of } T_{jk}) = \phi.$$

We now examine closely the problem of global serialization. Conflict between two concurrently validating transactions, T_i and T_j may not be detected if they are validated in a different order at two different nodes. For example, let transaction T_i read x at node 1 and write y at node 2, and transaction T_j write x at node 1 and read y at node 2. Using the above rules for class 3, both T_i and T_j may validate locally at both node 1 and 2, if T_{i1} is validated before T_{j1} at node 1 and T_{j2} is validated before T_{i2} at node 2. To avoid this anomaly, two different approaches can be taken:

(a) Ensure for all class 3 subtransactions T_{ik}, Read-set of $T_{ik} \cap$ Write-set of $T_{jk} = \phi$; or

(b) Require the validating subtransaction T_{jk} to wait for the global outcome of all class 3 subtransactions T_{ik} before completing its local validation.

Note that the latter strategy allows less concurrency but avoids the high overhead incurred by conflict checking.

By detecting a concurrent transaction of class 2 or 3 during validation, we may either check conflict against those transactions or we may simply abort the validating one. The implementation

of conflict checking, therefore, may involve one or more of these conditions. As indicated in Figure 2, each condition allows different levels of concurrency, and a more complicated conflict checking mechanism can result in a higher levels of concurrency. Consequently, there is a trade-off between the complexity of implementation and the level of concurrency allowed by the implementation. No conflict checking against class 2 and 3 corresponds to restricting transaction to execute serially. This is an option that we dismissed. Hence, conflict against class 2 subtransactions are checked in all variations of our implementations. Depending on the strategy of conflict checking against class 3 subtransactions various protocols emerge:

OCC-N: No conflict is checked against class 3 subtransactions.

OCC-A: Conflict is checked against class 3 using strategy (a).

OCC-B: Conflict is checked against class 3 using strategy (b).

OCC-AB: Conflict is checked against class 3 first using strategy (a). If a conflict is detected, it waits for the global outcome using strategy (b).

The OCC validator for each of these four variations of optimistic protocols, uses a hash table to check conflict by using following operations on a data granule:

1. search and check its serialization compatibility against other transactions which are currently registered with this data granule in the hash table;
2. insert it into the hash table if no conflicts are detected;
3. delete it from the hash table whenever it is no longer needed.

Under a low update rate situation, the performance of all the above variations of the protocols are further improved through a quick check on the combined number of entries in the hash table by subtransactions in locally validated state or in the write-phase. If there are no entries in the hash table, the validating subtransaction skips conflict checking, irrespective of the protocol used. More detailed descriptions of these implementations can be found in [16].

3.3 Optimistic Concurrency Control with Forward Validation

Under the forward validation approach, a transaction also consists of three phases, namely, read, validation and write. During the read phase the transaction enters its read-set in the common

hash table so that it can be immediately notified by other transactions about any of their updates conflicting with the read-set of this transaction. Upon receiving such notification, the transaction in its read phase is aborted and restarted. Once the transaction reaches its validation phase, it notifies all the other transactions in their read phase about any of its updates conflicting with their read-set. As the validation is done independently and in parallel at each of the participating nodes, a two phase commit is used to ensure the same serialization order at all nodes. Note, a locally validated subtransaction will notify other locally conflicting subtransactions in their read phase to be aborted. Also, the locally validated subtransaction may be aborted due to global validation failure. To avoid, any unnecessary abort, upon receiving any conflict notification the read subtransactions are actually suspended (to avoid any further wasted computation). Later, depending on the outcome of the notifier's global validation, the suspended transaction is either aborted or awakened. A time-out mechanism is used to avoid the vary rare case of deadlock where two transactions conflict at two nodes and start their validation at those nodes in reverse order.

The hash table operations required by this protocol are very similar to that of two-phase locking:

1. insert the weak lock into the hash table during read-phase;
2. search and notify all weak-lock holders about any conflict during validation phase;
3. delete it from the hash table whenever the lock is released.

3.4 Distributed Recovery

In this section we turn to the question of how to process transactions in a reliable manner. Specifically this issue involves building a recovery system which will make a database system behave as if it contains all of the effects of committed transactions and none of the effects of uncommitted ones. The basic technique for implementing durable transactions in presence of failures is the use of logs. A *log* contains information for undoing or redoing all operations performed by transactions. The two-phase commit protocol[8, 14], the simplest and most popular one of its kind, is adopted in the CARAT implementations.

The journalling (log) mechanisms currently implemented in CARAT are based on the *log write-ahead protocol.* Two journalling mechanisms are implemented: before-image journalling and after-image journalling. The TM and DM servers at each node share a single log file, which contains before-images or after-images of data objects and two-phase commit log records. In a commercial

system, journalling is processed at either a page or a record level; in CARAT, however, only page level journalling is implemented.

The before-images of the data pages are force-written to the disk by the DM server before the data pages are modified in the buffers. The after-images of the data pages are force-written to the disk before the DM server returns a PREPARED message to the TM server, i.e., during the first phase of the two-phase commit execution. We tested two locking protocols, one with before-image and a second with after-image journalling. For all optimistic concurrency control protocols, only the after-image journalling mechanism is implemented. Note that, according to the optimistic approach, all the modified data pages are maintained in the private memory and are flushed out to the database only when the COMMIT messages are received by the DM servers. Hence, the after-image journalling mechanism is preferred in the optimistic approach.

To safely record transaction state against system failures, the Coordinator TM server for a distributed transaction force-writes a "commit" or "abort" log record to the system log at the end of the first phase of the two-phase commit when the fate of the transaction is determined. The TM server also force-writes a "prepared" log record to the system log when a DM server reaches the *prepared* state. A "prepared" log record contains the coordinator node number that is used to resolve the transaction at system recovery time.

4 Experimental Setup

In this section, we first describe the workloads under which various sets of experiments were carried out. We then describe the various performance metrics measured, as well as the criterion used to evaluate the statistical significance of our results.

4.1 Transaction Types and Workloads

The transaction workload consists of a fixed number of concurrent users, each user running a particular type of transaction, during the entire run of the experiment. A transaction type is described by the following attributes

Local vs. Distributed Transactions — Local transactions perform no remote requests, while the data requested by distributed transactions are spread over multiple nodes. Distributed transactions are characterized by the number of participating nodes and how the requested data are distributed among them. In our experiments the data requests from each distributed

transaction were uniformly distributed (i.e., interleaved local and remote requests) among the involved nodes.

Read-only vs. Write Transactions — Read-only transactions perform no update operations. All requests made by write transactions are updates.

An workload is designated by two attributes according to the mix of various transaction types with different location and update attributes. The location attribute of the workload can be all local (L), all distributed (D), or mixed (M) environment with both local and distributed users. Similarly, its update attribute could be all read (R), all write (W), or both (B) read and write users. Based on the mix of various transaction types in the system, the following set of workloads were chosen for our experiments:

LR8 — 8 Local Read-only users per site.

LB8 — 8 Local users per site: 4 read users and 4 write users.

MB8 — 8 users per site: 2 local read users, 2 local write users, 2 distributed read users, and 2 distributed write users. Each distributed user accesses 1 remote node.

LWn — n Local write users per site.

DxWn — n Distributed Write users per site with each distributed transaction accessing x number of remote nodes.

Our database was partitioned so that each node consisted of 18,000 records stored in 3,000 pages (6 records per page) with 512 bytes per page. The data records accessed by a transaction in the experiments were chosen to be randomly and uniformly distributed across the whole database. The transaction size, specified as #TDO (no. of transaction DO steps) requests per transaction, was varied between 4, 8, 10, 12, 16, and 20. Here each request accesses four database records. The user think time between the end of a transaction and the beginning of the subsequent transaction is set to zero in all the experiments described here.

4.2 Performance Metrics

To compare the performance of different protocols, we use three different transaction performance metrics (throughput, response time, and abort ratio) and two system resource utilization metrics

(CPU utilization and Disk I/O rate). The system utilization metrics can reveal performance bottlenecks caused by system resources. To further understand the behaviour of specific protocols, data relevant only to that protocol was also collected. For instance, locking statistics were collected to help understand locking behavior.

1. **Record throughput** was measured in records accessed by successful transactions per second. This metric was chosen to normalize the effects of transaction sizes. We calculate the record throughput (records accessed per second) by multiplying the transaction throughput by the total number of data records accessed by a transaction, i.e., 4·#TDO.

2. **Transaction response time** was measured by the user (TR) process which initiated it. It is the time elapsed since the first TBEGIN message is sent to the TM until the last message TEND_K is received from TM of a transaction by the user process. The mean response time was then calculated by averaging the transaction response times of those committed transactions.

3. **Transaction abort ratio** is the probability that a transaction is aborted due to deadlock or unsuccessful validation.

4. **CPU utilization** is the percentage of the time CPU is busy in processing CARAT transactions (operating system overhead such as process scheduling is excluded).

5. **Disk I/O rate** is measured as the number of I/O per second in processing CARAT transaction data.

Note that, Disk I/O includes only database I/O in our measurement. Two major Disk I/O operations are involved in processing CARAT transactions: (i) Disk I/O for database accesses (DB I/O) and (ii) Disk I/O for journalling (log I/O). In order to isolate the impact of the concurrency control protocol from the recovery scheme we made an assumption that the journalling overhead was relatively insignificant since logging was performed on a separate fast disk at each node. In other experiments (not reported here), we used a separate disk to log I/O, and we found that while the transaction response time increases slightly due to physical delay of log I/O, the qualitative performance of various protocols in comparison with each other remains the same. Without any loss of generality, we report only the results of those experiments in which the physical access to log disk was turned off. Because there was little difference in the performance of 2PL/AI and 2PL/BI, we also report solely on the performance of 2PL/AI and the five OCC/AI implementations. In the remainder of the paper, we refer to each of the protocols solely by the associated concurrency control protocol, e.g. 2PL instead of 2PL/AI.

4.3 Measurement

A CARAT test consists of three phases: start up phase, the steady state phase, and the termination phase. Measurements are taken over the entire test. Consequently, it was necessary to choose the run length such that the testbed behavior during the start up and termination phases have little effect on measurement results. For several of the workloads, we ran the test for 10-60 minutes and observed that the results stabilized at 30 minutes. Hence, all measurements are reported for test lengths of 30 minutes. In addition, we duplicated several tests 3-5 times in order to determine the accuracy of our measurements. We found that there was little difference from one run to another.

Five nodes were involved in our experiments, and assigned the same workload. Very similar performance data was obtained on each node due to the homogeneity among these nodes. Since the minor differences among nodes are negligible for our purposes, only the mean values of the data over all nodes are analyzed and displayed in the figures in the following sections. The system utilization (CPU, I/O) data was collected by SPM[2].

5 Performance Comparison

The following five sets of experiments were performed.

Experiment 1 — Protocol overhead: This experiment was carried out for the LR8 workloads to compare the overhead of all protocols without data contention. In this experiment, we varied the size of transactions (TDO steps).

Experiment 2 — Data contention: LB8 workload was chosen for this experiment to study the effect of data contention on the performances of all protocols, by varying the size of transactions (TDO steps).

Experiment 3 — Distributed execution: MB8 workload was chosen to study the overhead of distributed execution on the performance of all protocols together with varying the size of transactions (TDO steps).

Experiment 4 — Multi-programming level: The effect of the multi-programming level was studied for three kind of workloads: LWn, D1Wn, and DnW2.

[2]DEC's VAX Software Performance Monitor package is capable of collecting the overall system performance data, e.g. CPU utilization and disk I/O rate, in a specified period.

Experiment 5 — Query or update intensive workloads: The mix of read and write users was varied to study the effects on the performance of the different protocols. their relative numbers.

5.1 Experiment 1 — Protocol Overhead

It is clear from Figures 3–6 that the protocols divide into three groups according to their performance: (i) 2PL and OCC-FV, (ii) OCC-AB and OCC-A, and (iii) OCC-B and OCC-N. Protocols within each group are similar in their implementation details. The protocols in group (i) are based on locking techniques whereas, the protocols in group (ii) and (iii) are based on the validation techniques. The group (iii) showed an overall significantly better performance in terms of record throughput (Figure 3) and response time (Figure 6). Also, group (iii) showed the lowest CPU utilization (Figure 4) and the highest Disk I/O rate (Figure 5). On the other hand, the performance of the group (ii) and (iii) protocols are relatively similar to each other regardless of their difference in the high level implementation techniques (locking vs. validation). These observations lead to the following points:

1. The low level implementation techniques are the same for all six protocols. The performance differences under the no data contention read-only workload, therefore, can be attributed to the number of high level hash operations required for each transaction.

2. Under local-only workloads, the implementation of strategy (b) or its absence in the OCC validator does not affect the performance of OCC-BV protocols (see Section 3.2) as this is required only for the global serialization of distributed transactions. Hence, it is expected that {OCC-AB, OCC-A} will behave similarly as will {OCC-B, OCC-N}.

3. Strategy (a) implemented in the OCC validator requires that the read-set of a transaction be registered into the hash table for later transactions to check against. For local read-only transactions, each data granule which has been accessed during the read-phase should go through all three OCC validator hash operations described in Section 3.2. The OCC protocols without strategy (a) (OCC-B and OCC-N), therefore, performed much better than the ones with strategy (a) (OCC-AB and OCC-A).

4. The locking based protocols (group (i)) require that all three hash operations as described in Sections 3.1 and 3.3 be used for each data granule accessed by a transaction. Since these hash operations are similar to those implemented in the OCC validators, the performance of group (i) and (ii), are found to be similar.

5.2 Experiment 2 — Data Contention

As can be observed from Figure 7, 2PL performed better than all the OCC protocols for all transaction sizes. The poor performance of the latter protocols was mainly due to the high transaction abort ratios. The abort ratios of OCC-BV protocols are much higher (by at least 20% greater) than that of 2PL (see Figure 8). The percentage of the aborted transactions increases for all the protocols as the transaction size is increased. As shown in Figure 8, the abort ratios for OCC protocols were very sensitive to data contention. Between 10%-40% of transactions were aborted even for the shortest transaction workload (4 TDO steps per transaction). However, the abort ratio is less sensitive to the increase in transaction size. In the case of 2PL, there is very little abort due to deadlock for TDO steps less than 8, and it then increases sharply with the increase in transaction size. The CPU and disk utilizations were comparable for protocols. As the throughput was very much determined by the high abort ratio, the special implementation optimizations used with OCC-BV protocols provided no real benefit under this workload.

Figure 9 and Figure 10 present the CPU utilization and the Disk I/O rate. We observed that both the CPU utilization and the Disk I/O rate for 2PL decrease slightly as the transaction size increases. This can be explained by the fact that the lock blocking time increased as the transactions became large. This is also consistent with the lowered record throughput shown in Figure 7. On the other hand, for OCC protocols, the CPU utilization remained relatively constant and the Disk I/O rate increased as the #TDOs became large. This occurs even though the record throughput decreases as a function of # TDO's and is consistent with the wasted computation due to the high abort ratios.

5.3 Experiment 3 — Distributed Execution

The preceding discussion only deals with workloads in which no distributed transactions were involved. To investigate the performance characteristics in a distributed environment the MB8 workload is used which includes both local and distributed transactions. Both 2PL and OCC-FV performed better than OCC-BV protocols (Figure 12). This is becuse OCC-BV protocols suffered from high abort rate (Figure 13) and because various implementation optimizations made little difference in the performance of the protocols. We also observed that OCC-FV provides slightly higher throughput than 2PL for workloads containing long transactions (#TDO = 16, 20). Two plausible factors are: (i) the distributed commit protocol overhead and (ii) the distributed deadlock detection overhead. These overheads elongate the mean transaction response time (compare

Table 1: Mean Lock Blocking Time and Probability of Deadlock:
MBT: Mean Lock Blocking Time (in seconds)
Pdl: Probability of Deadlock (per transaction)

	LB8		MB8	
#TDO	MBT	Pdl	MBT	Pdl
4	1.870	0.0014	2.267	0.0017
8	3.911	0.0427	5.568	0.0410
12	5.358	0.2055	10.325	0.2121
16	6.266	0.4078	11.566	0.4694
20	6.663	0.5791	11.842	0.6495

Figure 11 and Figure 16) and hence, the mean lock blocking time (see Table 1). This increases the number of blocked transactions as well as the number of aborted transactions due to increased probability of deadlock (see Table 1). The throughput of OCC-FV is less affected due to the above two factors and hence, OCC-FV even slightly outperformed 2PL for longer transactions.

5.4 Experiment 4 — Effects of Multi-Programming Level

In this experiment, the workloads consisted of update users only and the transaction size was fixed to 8 TDO steps per transaction. We used three different approaches to increase the multi-programming level. In the first case, we increased the number of local write users from 1 to 12 (LWn workloads). In the second case, the number of distributed users was varied from 1 to 4, where each distributed transaction accessed only one other remote site (D1Wn workloads). In the last case, we kept the number of users per site unchanged (2), but varied the number of remote sites accessed by a transaction from 1 to 4 (DnW2 workloads).

Figures 17–21 show the performance for local write users. As can be observed from the figures, 2PL has an overall better performance than any of the OCC protocols. As the number of users increased, both the abort ratio and response time increased proportionally for 2PL. The abort ratios were fairly low, less than 13% for all the workloads, whereas the mean response time increased linearly from 3 seconds for single user workload up to 20 seconds for the twelve users workload. Due to the I/O device being a bottleneck, the throughput, the CPU utilization, and the Disk I/O rate became constant when the number of users increased beyond four.

For all OCC protocols, the abort ratio increased with the number of users, but the rate of increase leveled off for large numbers of users. This result follows the theoretical predictions of [4].

The abort probability depends on the database update rate. The abort ratio becomes less sensitive to high multi-programming level since the throughput does not increase with multi-programming level. The OCC-N protocol suffered the highest abort ratio and it was high even for two users. The abort ratio for OCC-FV protocol is less than that of OCC-BV protocols. Both CPU utilizations and Disk I/O rates reached a maximum rate for a higher number of users, except for the OCC-B protocol due to the blocking effect caused by the wait-timeout mechanism.

The second set of workloads raises the multi-programming level by increasing the number of distributed update users (Figures 22 and 23). As can be observed in the figures, the highest throughput was exhibited by the 2PL, followed by the OCC-FV, and then by the OCC-BV protocols.

For the third set of workloads, both the number of users and the transaction size were fixed, only the number of remote sites per transaction were increased (Figures 24 and 25). The throughput decreased by a small factor with the increase in number of remote nodes per transaction, due to the increased overhead of distributed execution.

5.5 Experiment 5 — Query or Update Intensive Workloads

Many database systems are query intensive (read only transaction) and are characterized by a very small update rate. During the execution of a transaction, if no database update is made by other transactions, no entries are made in the hash table. A transaction can avoid checking conflict under OCC-BV protocol by detecting such a special case. (Details of some other special optimizations implemented by maintaining various counters for low update rate can be found in [16].) This experiment was designed to compare the performance under a various mix of read and write (query and update) transactions. The number of users was fixed at 8, and the mix of R/W users was varied as 8/0, 7/1,..., 1/7, 0/8. As can be seen from the Figure 26, the throughputs of OCC-B and OCC-N were higher than for all other protocols in the absence of any update transaction (R/W mix: 8/0). The validation overhead was required by the above two protocols, in the presence of even a small database update rate (R/W mix: 7/1), and the protocols subsequently lost their advantages over the others. This may be an artifact of our experiment design, because we maintained a fixed mix of query/update users. One can envision an alternative scenario: on the average there is one read and seven write transactions, but there may be a long period of time when all transactions are read-only. In such a scenario, OCC-B and OCC-N can be expected to perform well, because of simplified conflict checking. The performance under such time-varying workloads will be the subject of our future study.

6 Summary

Although the complexity of a distributed database system makes experimental comparisons of concurrency control protocols difficult, we have succeeded in experimentally comparing three classes of concurrency control protocols, two-phase locking and the optimistic approaches with forward and backward validation. For the optimistic concurrency control protocol with backward validation we have developed, implemented, and evaluated several optimizations. Except for very small update rate, none of these optimizations proved as effective as two-phase locking or optimistic with forward validation protocols. We believe that this is the first comprehensive experimental study of distributed concurrency control protocols. In order to minimize the non-essential variations in implementation, we have used a common implementation of a hash table for both the lock table as well as the validation table. The main emphasis here is to contrast the difference in protocol overhead based on the number of high level operations required by different protocols while taking the common low level implementation overhead into consideration. From the measured values of basic performance metrics (throughput, response time, abort ratio, CPU and I/O utilization,etc.), we presented qualitative explanations for the observed performance characteristics of various concurrency control protocols.

2PL performed better than the OCC protocols in terms of transaction throughput under most of the workloads in our experimental environment. The lock blocking effect was identified as an important performance degradation factor for the 2PL protocol. Both the CPU and disk became less utilized as the lock blocking rate or mean lock blocking time increased. Under a very high conflict situation, the abort rate due to deadlocks also increased very sharply. Optimistic protocols, on the other hand, suffer from wasting computational resources due to high transaction abort rate. The experiments were CPU bound for OCC protocols under most of the workloads since the CPU utilizations were almost saturated (around 90% utilized). The various implementations of the optimistic protocols with backward validation resulted from the trade-offs between the complexity of the mechanism required and the levels of the concurrency provided. Under low data contention situations, less complex implementations (OCC-N and OCC-B) outperformed the more complex ones (OCC-A and OCC-AB) in terms of higher record throughput and lower response time. However, when data contention was high, these performance differences were diminished. For a query-intensive system, where database update rate is low, the performance of optimistic protocols can be dramatically improved, by special implementation optimizations (maintaining the count of hash entries etc.).

The implementation of the optimistic protocol with forward validation is similar to that of 2PL (use of lock table) and hence, to its performance under many workloads. Under the workloads, where optimistic protocols with backward validation suffered high abort ratio, OCC-FV performed much better than OCC-BV protocols. Under workloads, where 2PL lost its advantage due to long lock holding times (due to longer transaction size or distributed execution), OCC-FV slightly outperformed 2PL as its throughput degraded at a lower rate.

References

[1] R. Agrawal, M. J. Carey, and M. Livny, "Concurrency Control Performance Modeling: Alternatives and Implications," *ACM Transactions on Database Systems*, Vol. 12, No. 4, pp. 609–654, December 1987.

[2] K. M. Chandy, J. Misra, and L. M. Hass, "Distributed Deadlock Detection," *ACM Transactions on Computer Systems*, Vol. 1, No. 2, pp. 144–156, May 1983.

[3] K. M. Chandy and J. Misra, "A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems," In *Proc. 1st ACM SIGACT-SIGOPS Symp. on the Principles of Distributed Computing*, pp. 157–164, Ottawa, August 1982.

[4] A. Dan, W. H. Kohler, and D. Towsley, "Modeling the Effects of Data and Resource Contention on the Performance of Optimistic Concurrency Control Protocols," In *Fourth International Conference on Data Engineering*, pp. 418–425, February 1988.

[5] K. P. Eswaren, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM*, Vol. 19, No. 11, November 1976.

[6] B. I. Galler and L. Bos, "A Model of Transaction Blocking in Databases," *Performance Evaluation*, Vol. 3, pp. 95–122, 1983.

[7] J. Gray, P. Homan, R. Obermack, and H. Korth, *A Straw Man Analysis of Probability of Waiting and Deadlock*, Research Report RJ 3066, IBM Reserach Laboratory, San Jose, CA, February 1981.

[8] J. N. Gray, "Notes on Database Operating Systems," In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, pp. 393–481, Springer–Verlag, Berlin, 1978.

[9] N. Griffeth and J. A. Miller, "Performance Modeling of Database Recovery Protocols," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 6, pp. 564–572, June 1985.

[10] K. B. Irani and H. Lin, "Queueing Network Models for Concurrent Transaction Processing in a Database System," In *Proc. of ACM SIGMOD Int'l Conf. on Management of Data*, pp. 134–142, Boston, MA, May 1979.

[11] B. P. Jenq, *Performance Measurement, Modelling, and Evaluation of Integrated Concurrency Control and Recovery Algorithms in Distributed Database Systems*, PhD thesis, University of Massachusetts, Amherst, February 1986.

[12] W. Kohler and B. P. Jenq, "CARAT: A Testbed for the Performance Evaluation of Distributed Database Systyems," In *Proc. of the Fall Joint Computer Conference*, pp. 1169–1178, IEEE Computer Society and ACM, Dallas, Texas, November 1986.

[13] H. T. Kung and J. T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems*, Vol. 6, No. 2, pp. 213–226, June 1981.

[14] B. G. Lindsay et al., *Notes on Distributed Databases*, Research Report RJ 2571, IBM Reserach Laboratory, San Jose, July 1979.

[15] D. Potier and P. Leblanc, "Analysis of Locking Policies in Database Management Systems," *Communications of the ACM*, Vol. 23, No. 10, pp. 584–593, October 1980.

[16] C. Shih, *Performance Measurement and Evaluation of Concurrency Control Algorithms in a Distributed Database Testbed System*, Master's thesis, University of Massachusetts, Amherst, February 1989.

[17] A. W. Shum and P. G. Spirakis, "Performance Analysis of Concurrency Control Methods in Database Systems," In F. J. Kylstra, editor, *Performance 81*, pp. 1–19, North-Holland, Amsterdam, 1981.

[18] A. Thomasian, "An Iterative Solution to the Queueing Network Model of a DBMS with Dynamic Locking," In *Proc. of 13th Conf. Computer Measurement Groups*, pp. 252–261, San Diego, CA, December 1982.

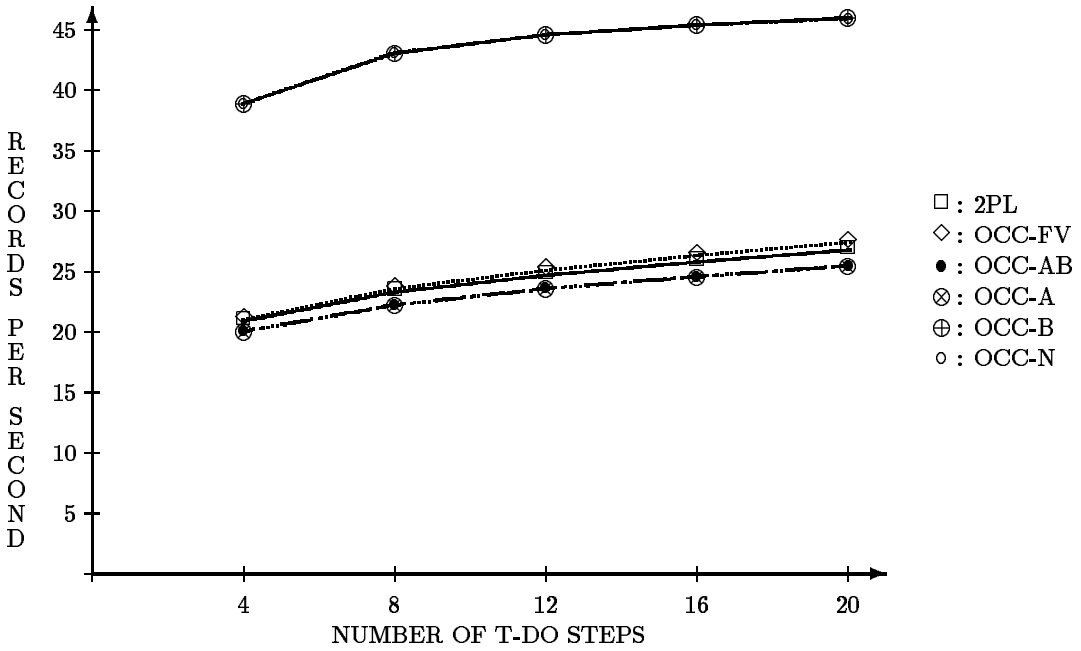


Figure 3: Record Throughput vs. Transaction Size for LR8 Work Load

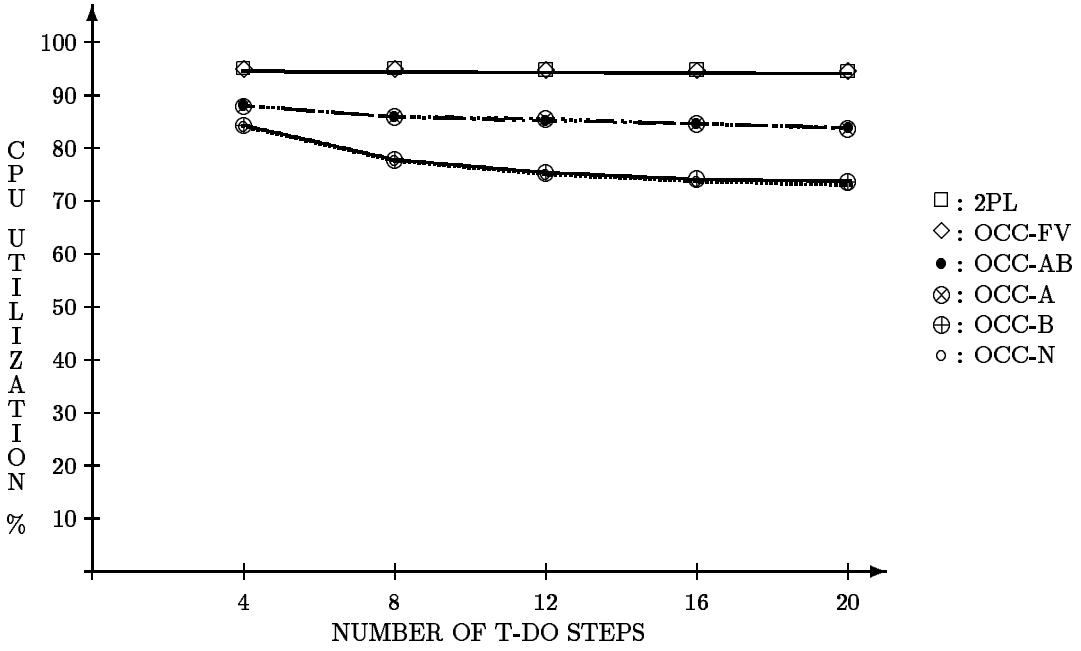


Figure 4: CPU Utilization vs. Transaction Size for LR8 Work Load

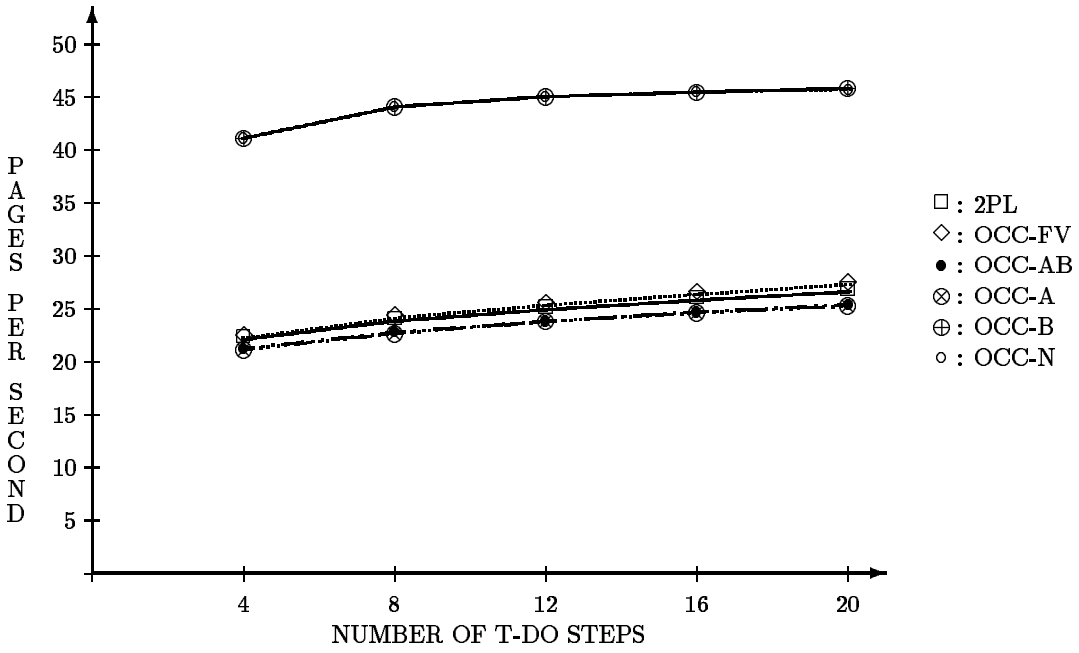


Figure 5: Disk I/O Rate vs. Transaction Size for LR8 Work Load

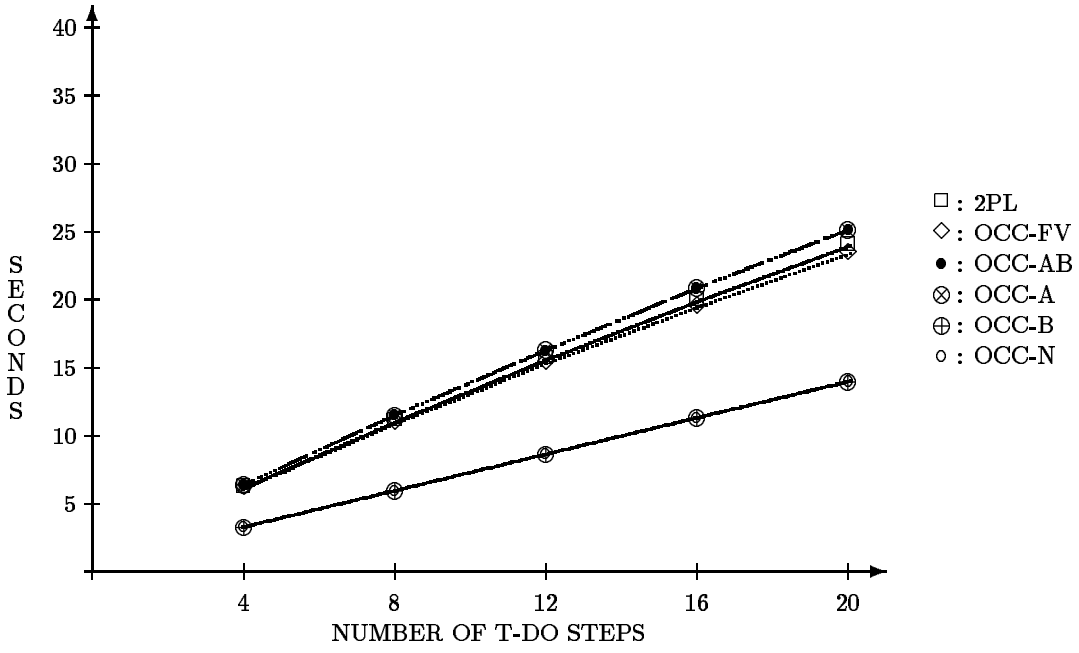


Figure 6: Transaction Response Time vs. Transaction Size for LR8 Work Load

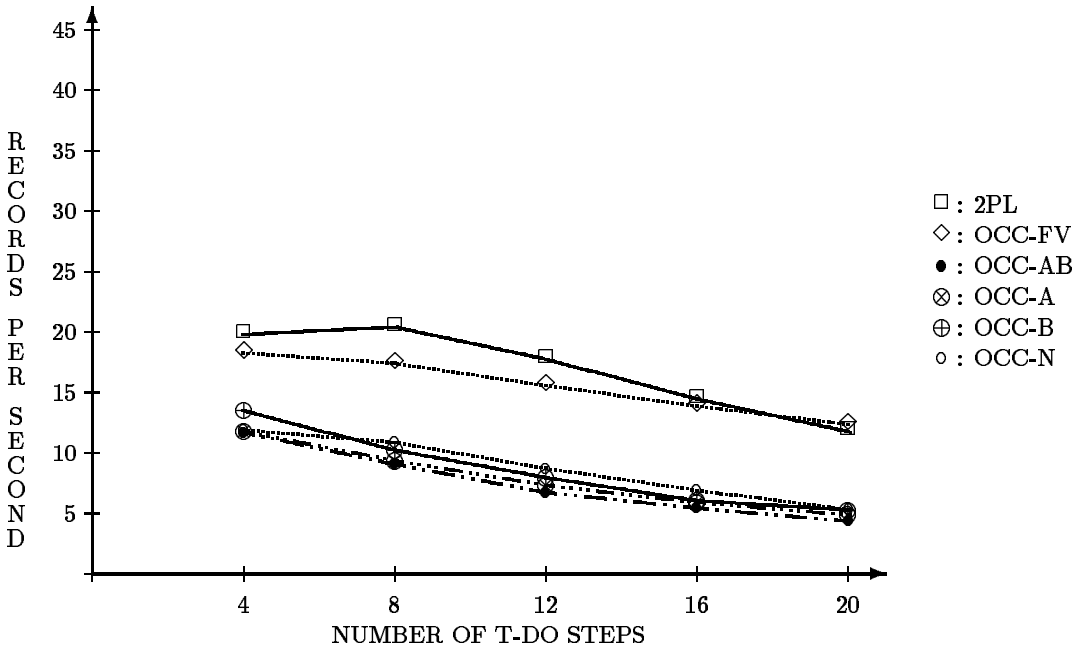


Figure 7: Record Throughput vs. Transaction Size for LB8 Work Load

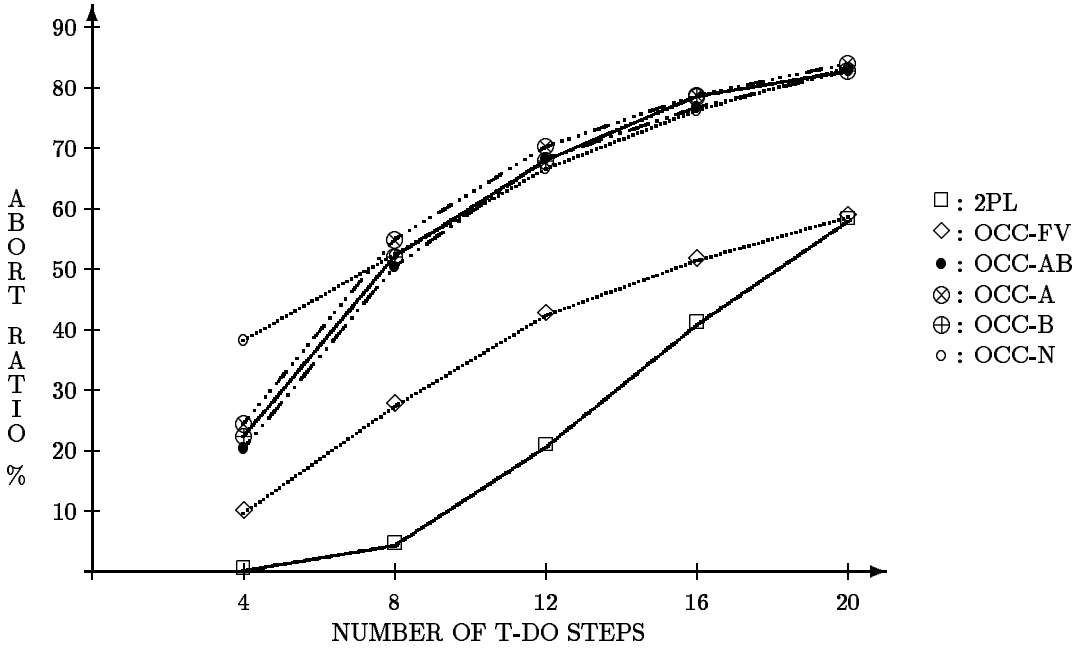


Figure 8: Abort Ratio vs. Transaction Size for LB8 Work Load

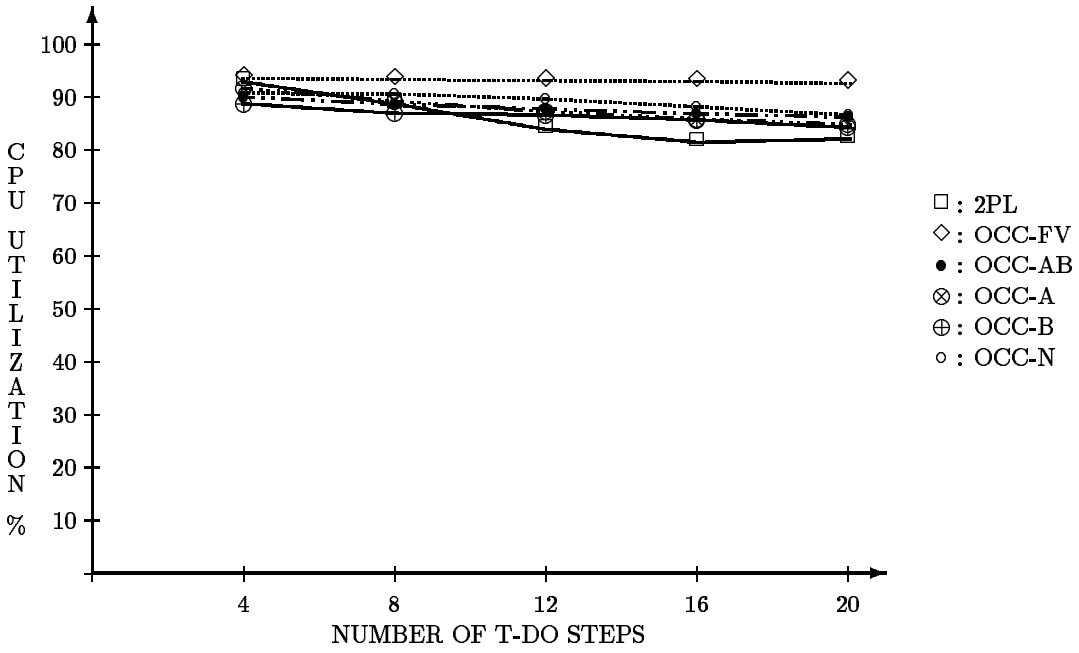


Figure 9: CPU Utilization vs. Transaction Size for LB8 Work Load

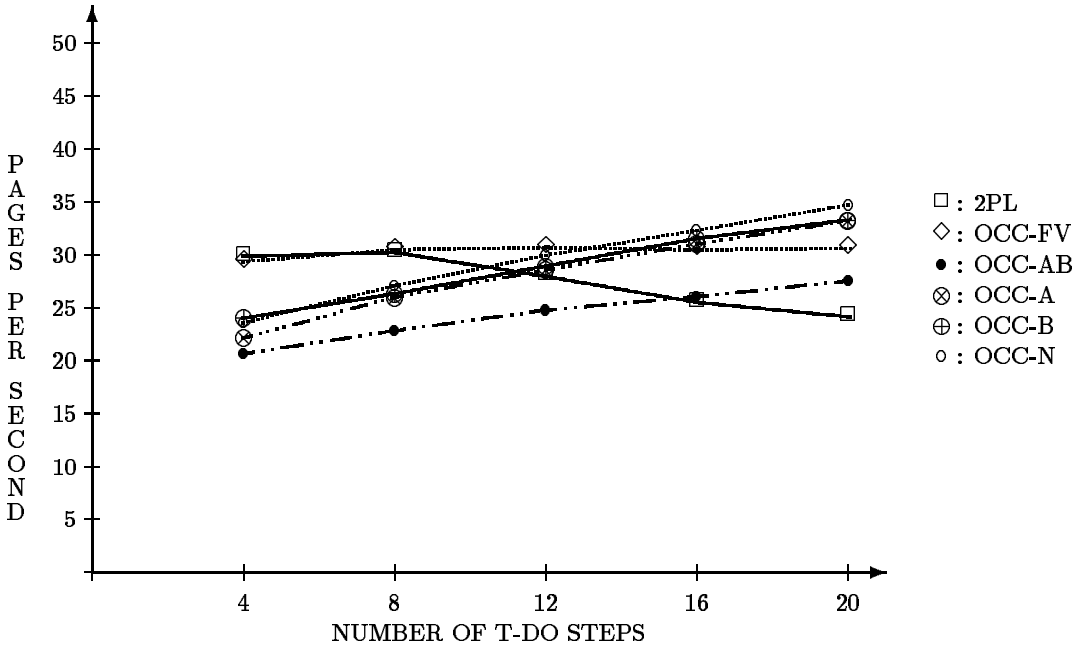


Figure 10: Disk I/O Rate vs. Transaction Size for LB8 Work Load

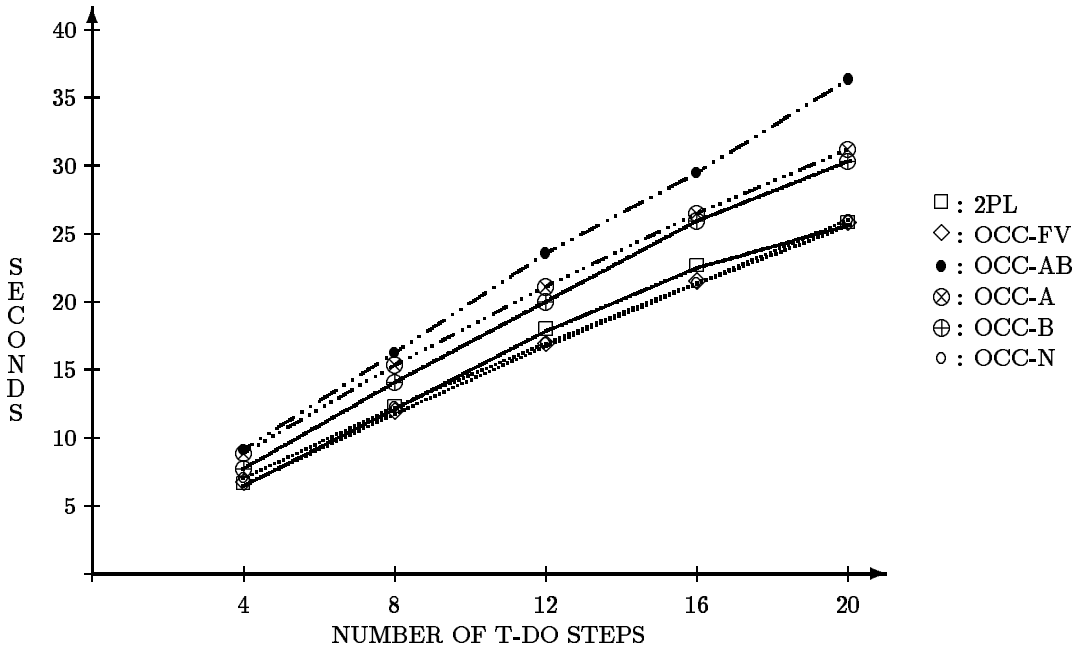


Figure 11: Transaction Response Time vs. Transaction Size for LB8 Work Load

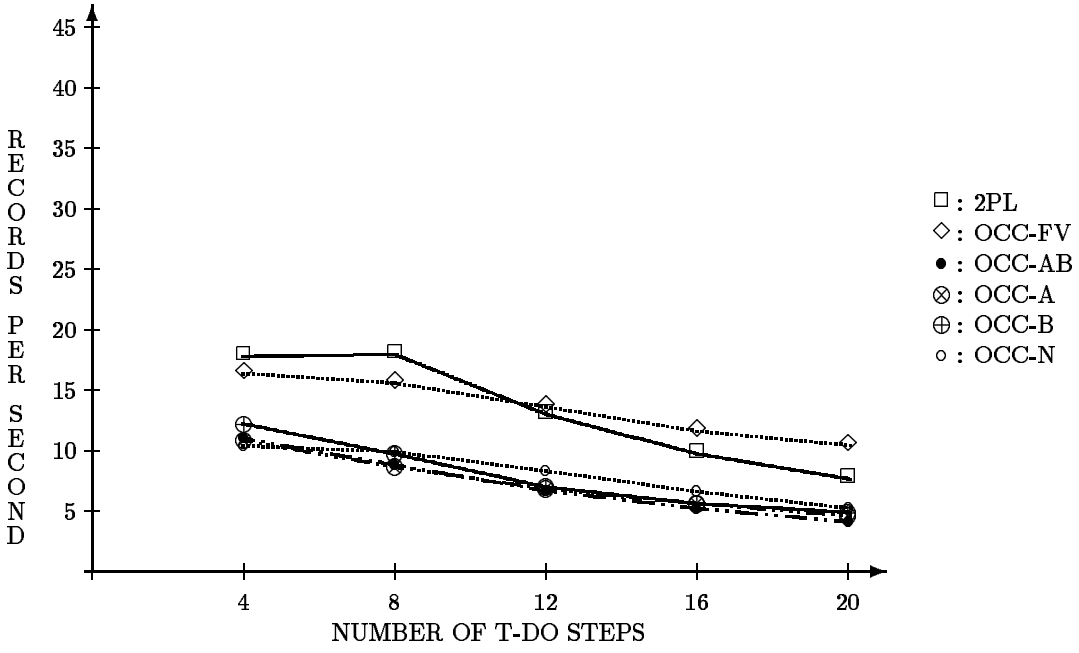


Figure 12: Record Throughput vs. Transaction Size for MB8 Work Load

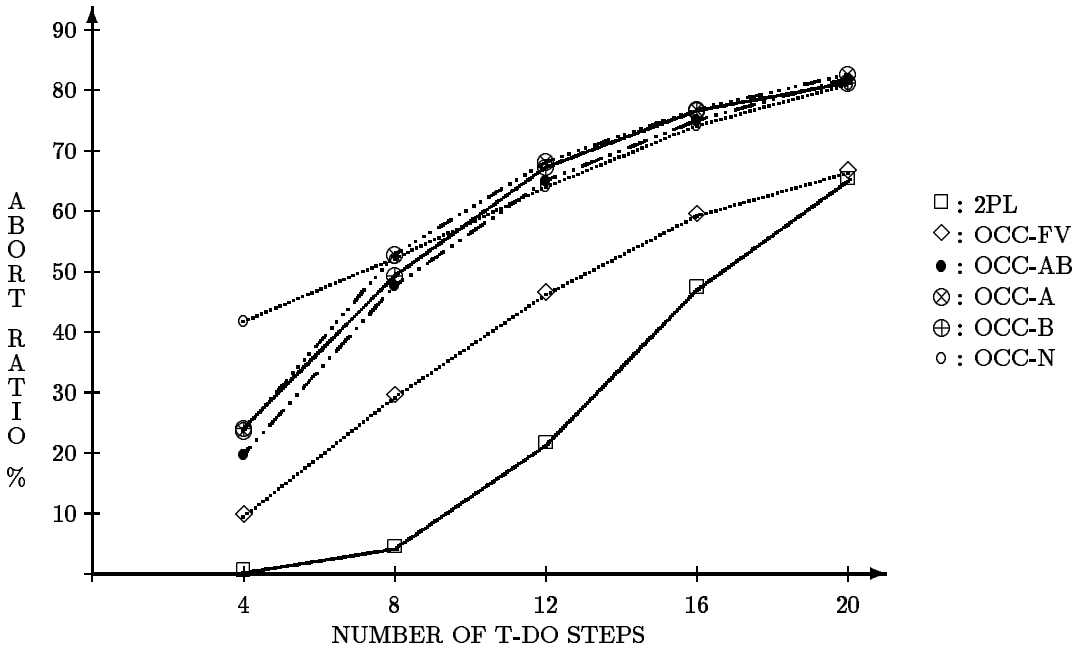


Figure 13: Abort Ratio vs. Transaction Size for MB8 Work Load

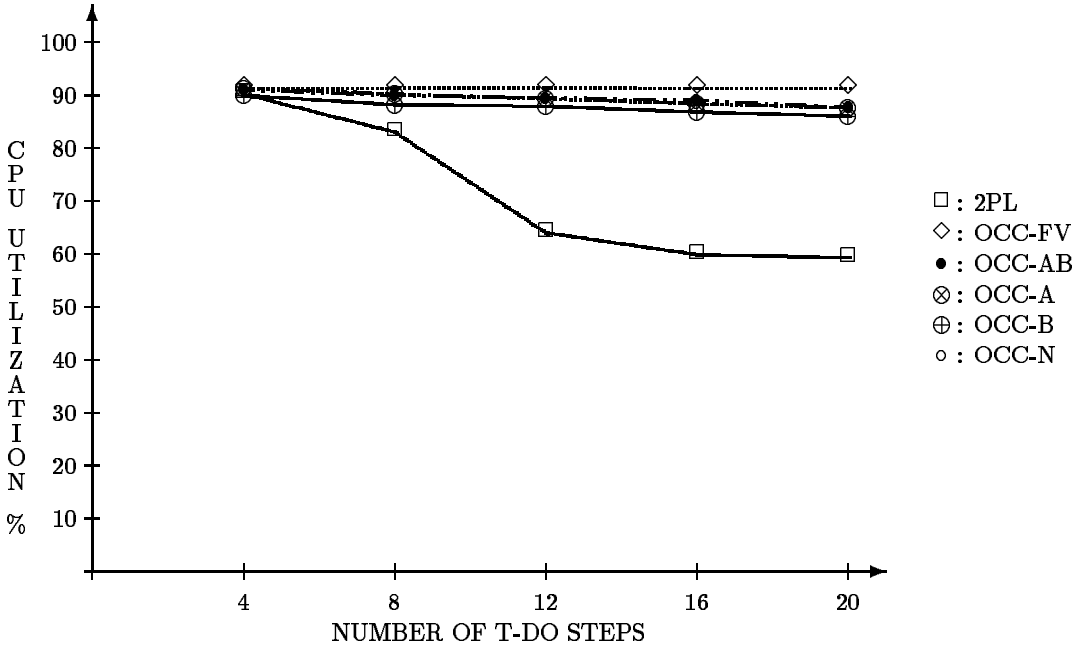


Figure 14: CPU Utilization vs. Transaction Size for MB8 Work Load

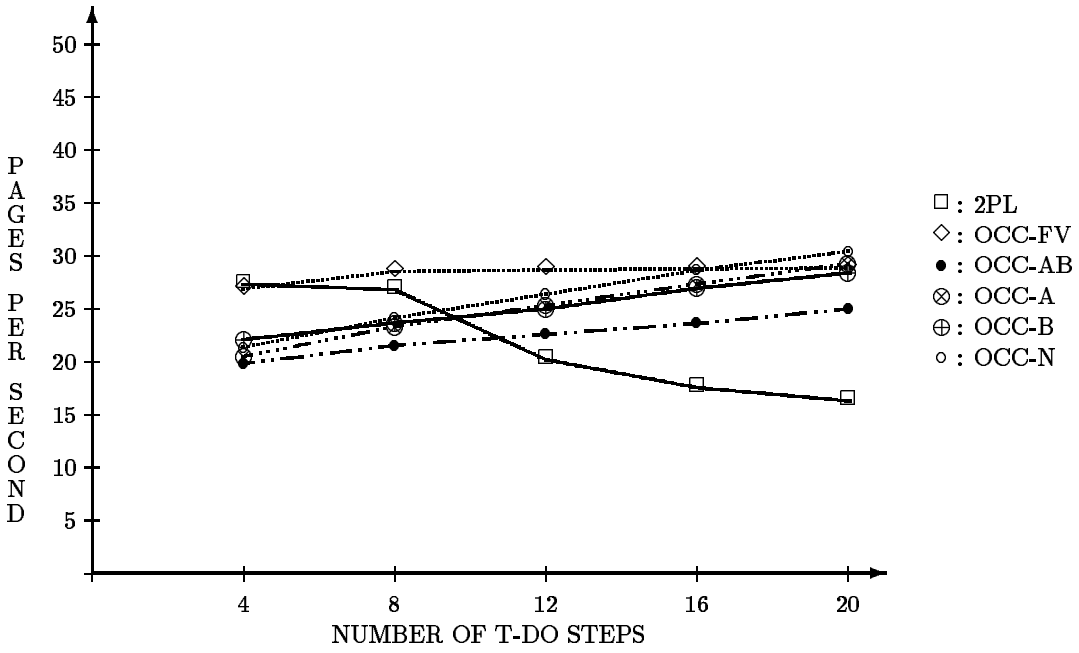


Figure 15: Disk I/O Rate vs. Transaction Size for MB8 Work Load

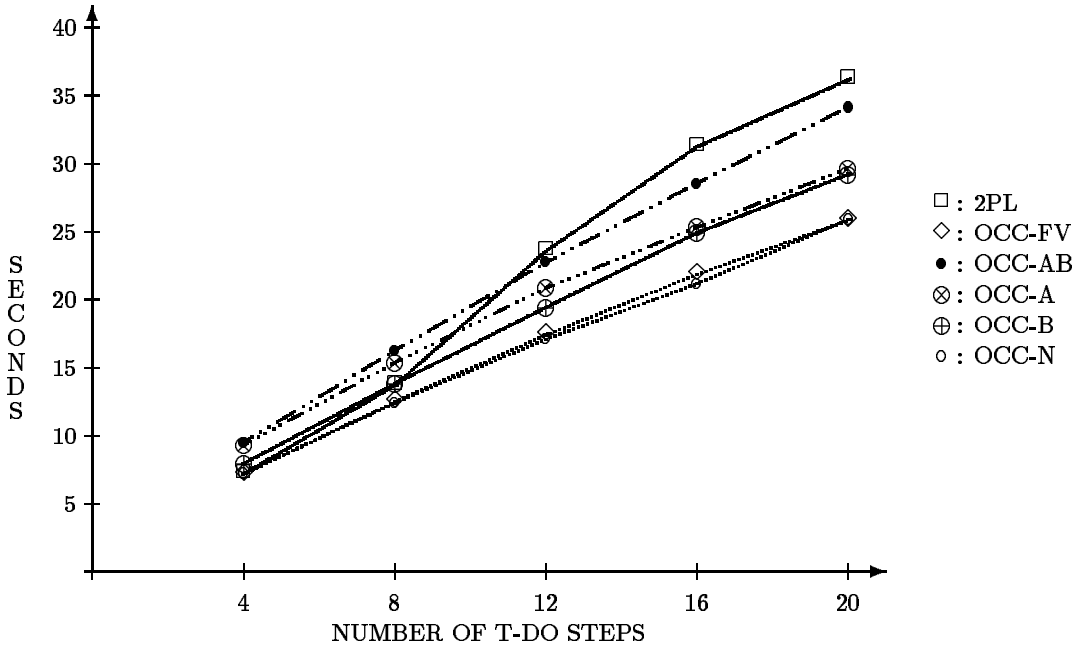


Figure 16: Transaction Response Time vs. Transaction Size for MB8 Work Load

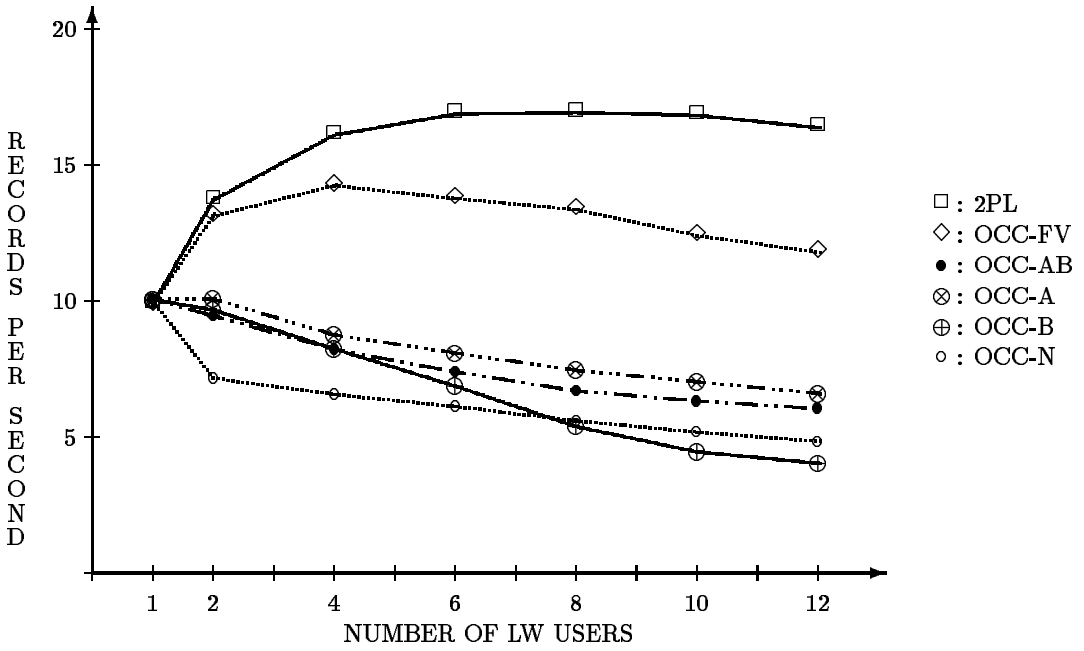


Figure 17: Record Throughput vs. LW Users/Site (8 T-DO steps per transaction)

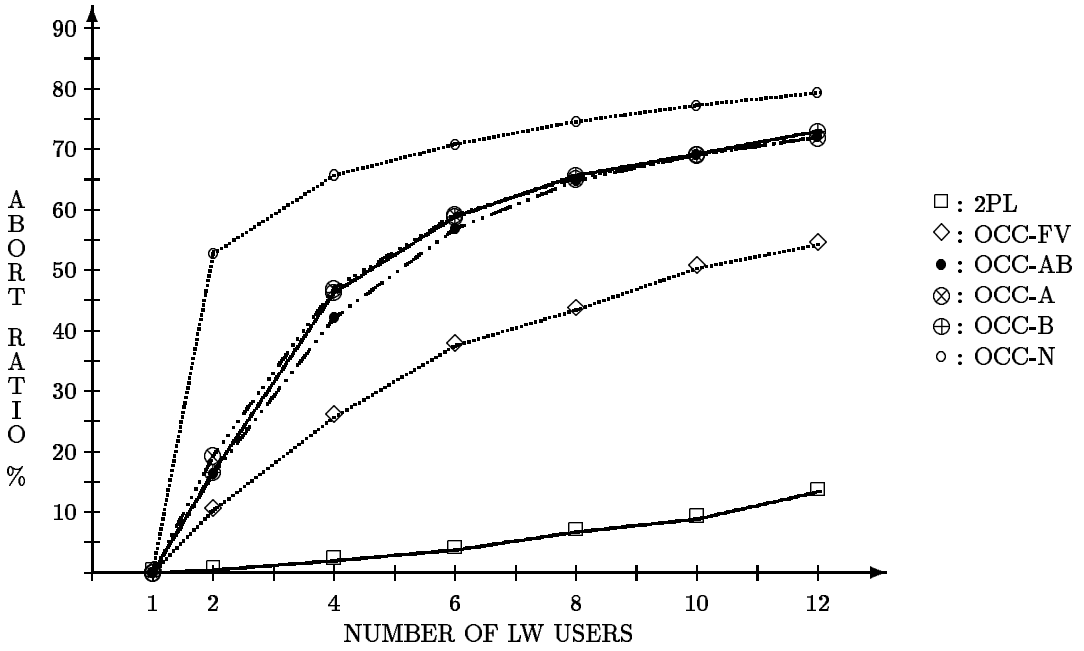


Figure 18: Abort Ratio vs. LW Users/Site (8 T-DO steps per transaction)

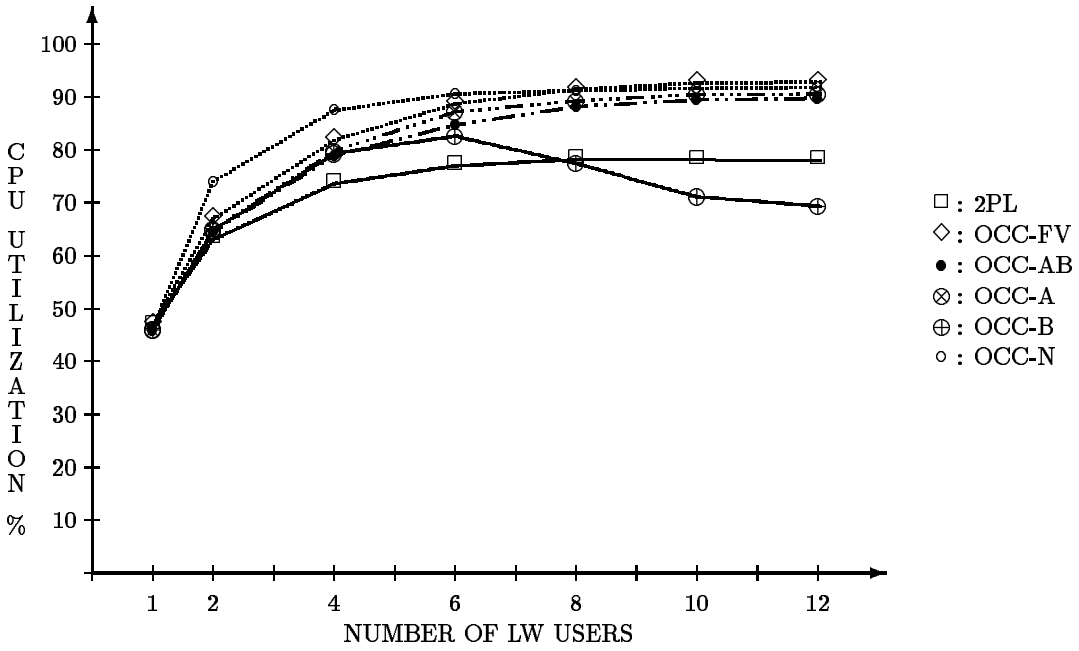


Figure 19: CPU Utilization vs. LW Users/Site (8 T-DO steps per transaction)

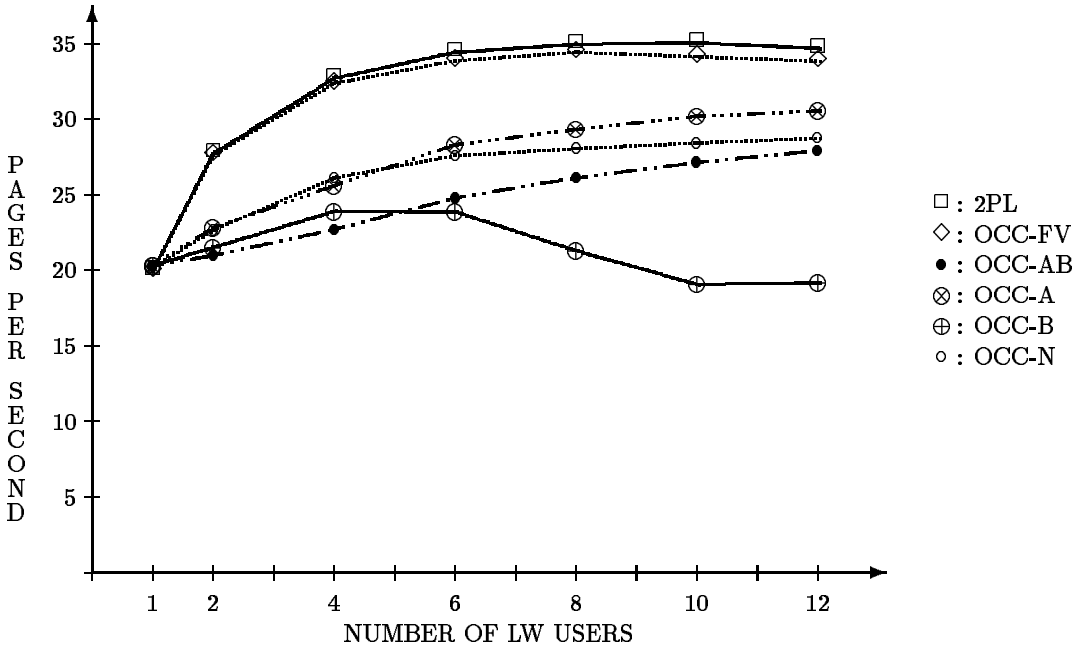


Figure 20: Disk I/O Rate vs. LW Users/Site (8 T-DO steps per transaction)

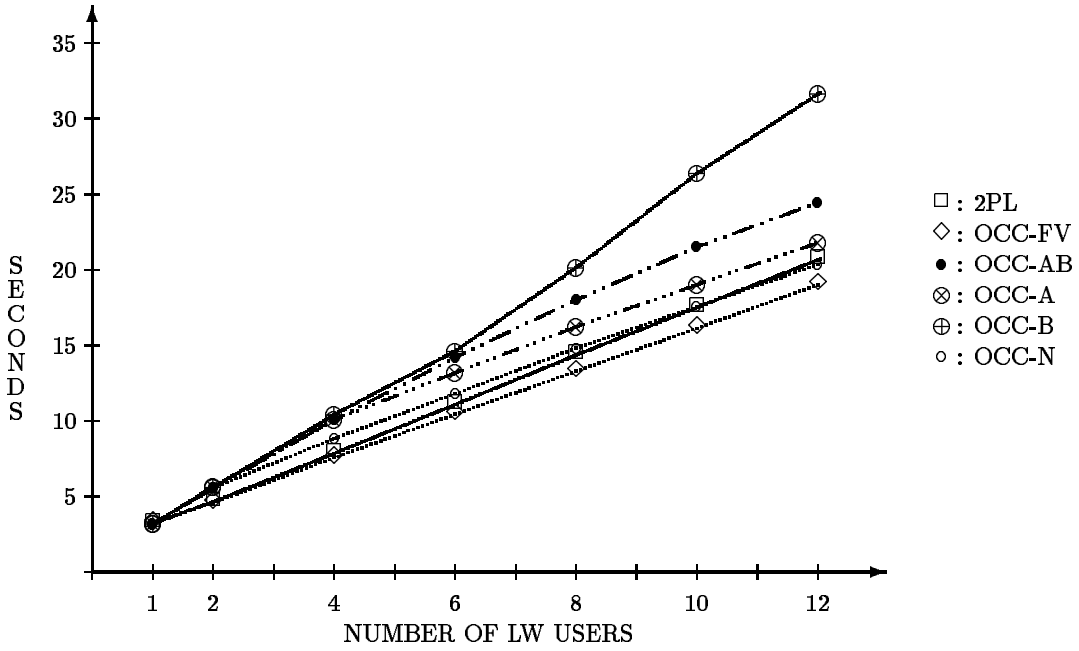


Figure 21: Transaction Response Time vs. LW Users/Site (8 T-DO steps per transaction)

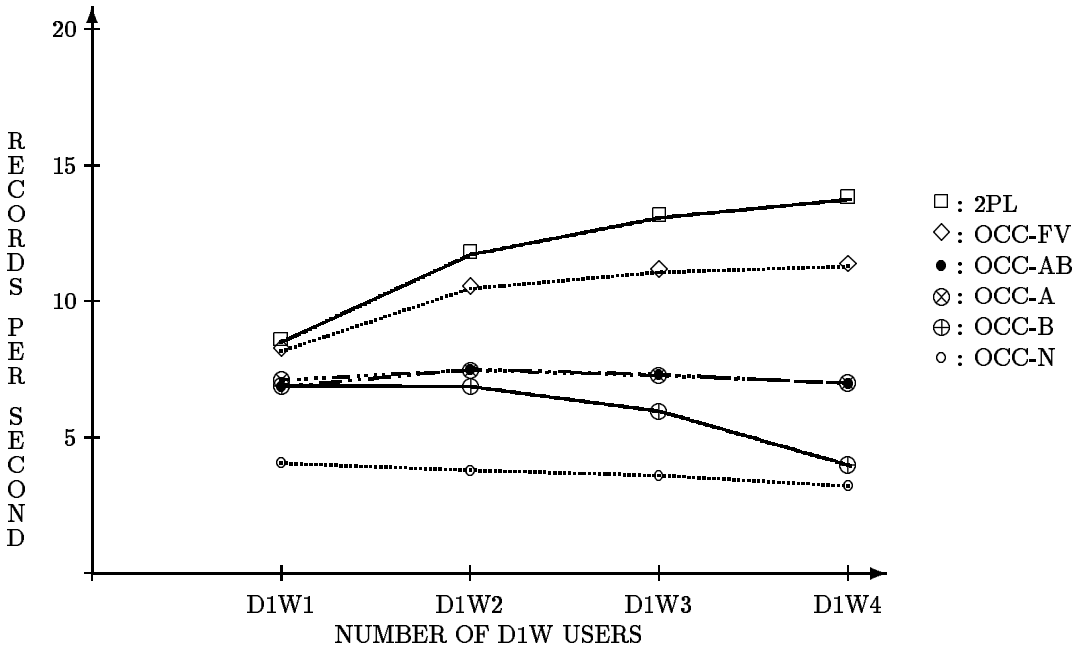


Figure 22: Record Throughput vs. D1W Users/Site (8 T-DO steps per transaction)

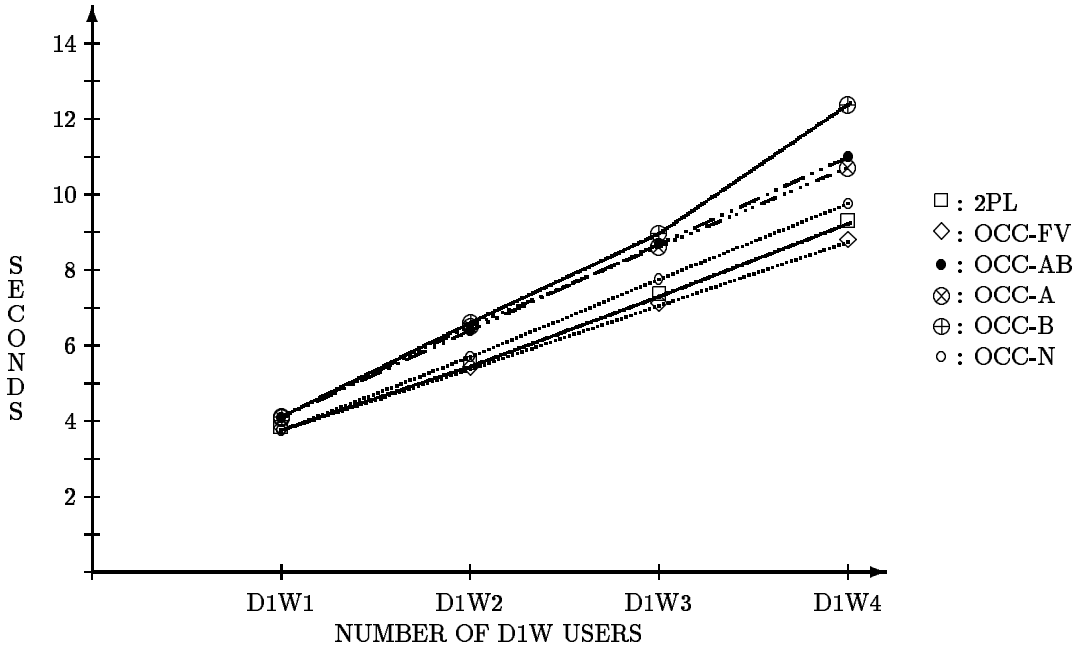


Figure 23: Transaction Response Time vs. D1W Users/Site (8 T-DO steps per transaction)

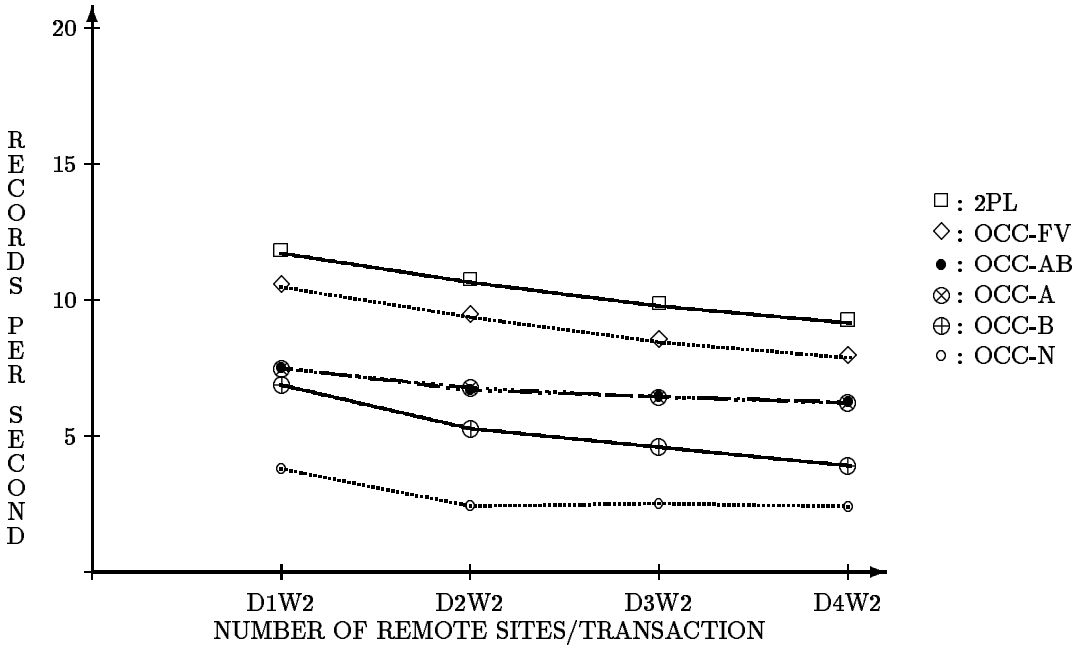


Figure 24: Record Throughput vs. Remote Sites/Transaction (2 users initiated at each site with 8 T-DO steps per transaction)

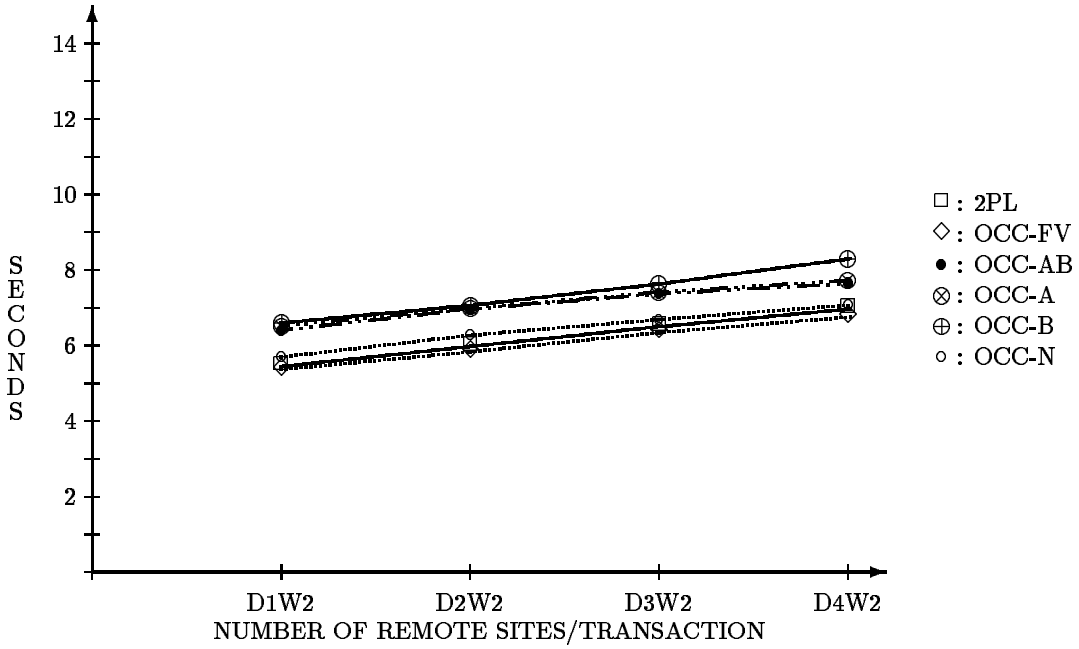


Figure 25: Transaction Response Time vs. Remote Sites/Transaction (2 users initiated at each site with 8 T-DO steps per transaction)

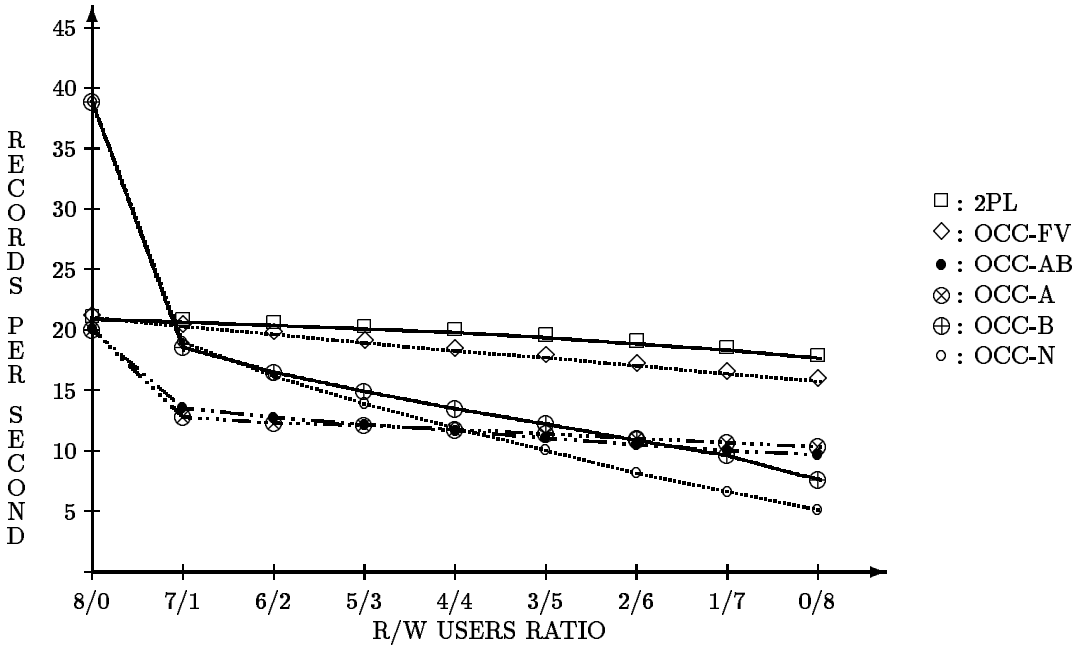


Figure 26: Record Throughput vs. R/W Transaction Mixing Ratio (8 local transaction users initiated at each site with 4 T-DO steps per transaction)