Experimental Evaluation of Real-Time Optimistic Concurency Control Schemes

J. Huang, J.A. Stankovic,
K. Ramamritham, D. Towsley
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

COINS Technical Report 91-16
January 1991

Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes *

Jiandong Huang
Department of Electrical and Computer Engineering

John A. Stankovic
Krithi Ramamritham
Don Towsley
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

February, 1991

Abstract

Owing to its potential for a high degree of parallelism, optimistic concurrency control is expected to perform better than two-phase locking when integrated with priority-driven CPU scheduling in real-time database systems. In this paper, we examine the overall effects and the impact of the overheads involved in implementing real-time optimistic concurrency control. Using a locking mechanism to ensure the correctness of the implementation, we develop a set of optimistic concurrency control protocols which possess the properties of deadlock freedom and a high degree of parallelism. Through experiments, we investigate, in depth, the effect of the locking mechanism on the performance of optimistic concurrency control protocols. We show that due to blocking, the performance of the protocols is sensitive to priority inversions but not to resource utilization. Further, in contrast to recent simulation studies, our experimental results show that with respect to meeting transaction deadlines, the optimistic approach may not always outperform the two-phase locking scheme which aborts the lower priority transaction to resolve a conflict. We also show that integrated with a weighted priority scheduling algorithm, optimistic concurrency control exhibits greater flexibility in coping with the starvation problem (for longer transactions) than two-phase locking. Our performance studies indicate that the physical implementation has a significant impact on the performance of real-time concurrency control protocols and is hence an important aspect in the study of concurrency control.

Keywords - real-time database systems, real-time transaction processing, deadlines, concurrency control, performance evaluation, system implementation.

*This work was supported by the National Science Foundation under Grant IRI-8908693 and Grant DCR-8500332, and by the U.S. Office of Naval Research under Grant N00014-85-K0398.

1 Introduction

Concurrency control is one of the main issues in the studies of real-time database systems. With a strict consistency requirement as defined by the notion of serializability, most real-time concurrency control schemes considered in the literature are based on two-phase locking (2PL) [18,2,9,17]. This is not surprising since 2PL has been well studied in traditional database systems and is being widely used in commercial databases. But 2PL, on the other hand, has some inherent problems such as the possibility of deadlocks and long and unpredictable blocking times. These appear to be serious problems for real-time transaction processing, since in a real-time environment, transactions need to meet their time constraints as well as consistency requirements.

Recently, some alternatives of two-phase locking have been proposed and studied [16,8,6,10,7,13]. Among them is a class of concurrency control schemes based on the well-know optimistic approach [12]. Ideally, optimistic concurrency control (OCC) has the properties of non-blocking and deadlock freedom. These properties make the scheme especially attractive to real-time transaction processing. In real-time database systems, OCC may be in a better position to be integrated with priority-driven CPU scheduling. Previous performance studies [6,7] have shown that under a policy that discards transactions which have missed their deadlines, OCC outperforms 2PL over a wide range of system utilizations.

The results in [6,7] are based on simulation, where optimistic concurrency control is carried out at the logical level and detailed implementation issues at the physical level are ignored. Particularly, a *broadcast* mechanism is used in the validation phase of the OCC protocol and no overhead is taken into account for the broadcast operation. In practice, the implementation of the broadcast and the corresponding overheads may affect the performance of a protocol.

In this study, we examine the overall effects and the impact of the overheads involved in implementing real-time optimistic concurrency control. Using a locking mechanism to ensure the correctness of the OCC implementation, we develop a set of optimistic concurrency control protocols. The protocols possess the property of deadlock freedom and have the potential for a high degree of parallelism. Integrated with priority-driven preemptive scheduling, the blocking time under the proposed protocol is limited and is predictable compared with 2PL. Our performance studies conducted on a real-time database testbed show that the blocking effect caused by the locking mechanism adopted in the implementation scheme has a major impact on the performance of the optimistic concurrency control protocol. In particular, the protocols are sensitive to priority inversion, but not to resource utilization. Furthermore, in contrast to the simulation results from [6,7], our experimental results show that OCC may not always outperform a 2PL protocol which aborts the lower priority transaction when conflict occurs. The "degraded" performance of the optimistic approach becomes apparent only because we considered the implementation details and since ours is a testbed, the overheads of the implementation manifest themselves in the performance figures. The experimental results indicate that the physical implementation schemes have a significant impact on the performance of real-time optimistic concurrency control.

In this work, we also investigate optimistic concurrency control in the context of the starvation problem. Because of their higher probability to conflict with other transactions, long transactions are likely to be repeatedly restarted and thus have less chance to meet their deadline than short transactions. Instead of limiting the number of transaction restarts, as is often proposed for traditional database systems, we use length and deadline sensitive priority assignment to address the problem. We show that integrated with the proposed weighted priority scheduling policy the optimistic concurrency control approach is more flexible in coping with the starvation problem than the two-phase locking scheme.

This paper is organized as follows. In Section 2, we present our locking-based OCC protocols and discuss implications of the implementation. In Section 3, we describe the real-time database testbed that was used for the performance studies. The experimental results are presented and discussed in detail in Section 4. Finally, we give concluding remarks and point out future research directions in Section 5.

2 Optimistic Concurrency Control for Real-Time Transactions

In this section, we first discuss the principles underlying optimistic concurrency control for real-time transactions, particularly regarding its validation. Then we propose a set of locking-based optimistic concurrent control protocols and discuss their implications in comparison with a two-phase locking approach. At the end, we present some conflict resolution policies used in conjunction with the proposed protocols.

2.1 Principle of Optimistic Concurrency Control

With the original OCC [12], the execution of a transaction consists of three phases: read, validation, and write. The key component in OCC is the validation phase where a transaction's destiny is decided. To ensure the serializability criterion, it must be satisfied that if transaction T_i is serialized before transaction T_j, then

Condition 1: the writes of T_i should not affect the read phase of T_j; and

Condition 2: T_i's writes should not overwrite T_j's writes.

Basically, the validation process can be carried out in either of the following two ways[1]:

- **Backward validation**: validating against committed transactions. When a conflict is detected, the validating transaction will abort itself.
- **Forward validation**: validating against active transactions. When a conflict is detected, either the validating transaction in validation phase or the conflicting transactions in read phase can be aborted. Futhermore, the validating transaction may defer validation till later on, thus avoiding any unnecessary abort.

[1]For detailed discussion, the reader is referred to [5].

In real-time database systems, conflicts should be resolved according to the priority associated with real-time transactions. As a result, either the validating transaction or the conflicting transaction(s) may be chosen for abort or possible delay. Clearly, to provide flexibility for conflict resolution, a transaction should be validated against active transactions instead of committed ones, i.e. forward validation is preferable.

Let T_j be the validating transaction and T_i (i = 1,2,...,n, $i \neq j$) be the transactions in their read phase. Let RS(T) and WS(T) denote the *read set* and *write set* of transaction T, respectively. Then, the validation operation can be described by the following procedure.

```
VALID := true
for Ti (i = 1,2, ..., n) do
    if WS(Tj) ∩ RS(Ti) ≠ {}
      then VALID := false
if VALID
    then execute write phase
    else invoke real-time conflict resolution
```

The condition WS(T_j) $\cap$ RS(T_i) $\neq$ {} guarantees that data read by the T_i's have not been written by T_j. To ensure Condition 2, I/O operations in the write phase must be done sequentially in validation order.

2.2 Optimistic concurrency control using locking (OCCL)

Validating against active transactions is straightforward at the logical level. For example, a *broadcast* mechanism can be used for the validation, where the validating transaction "notifies" other currently running transactions of data access conflict [6]. In the following, we describe a physical implementation of a set of validation protocols.

The proposed protocols are based on a locking mechanism [5], thus being named as OCCL[2]. In the system, each transaction T_i maintains its own *read set*, RS(T_i), and *write set*, WS(T_i). In addition, a systemwide lock table, LT, is shared by all concurrently executing transactions. We define two lock modes - *read-phase lock* (R-lock) and *validation-phase lock* (V-lock), where an R-lock for a data item is set in LT by transaction in its read phase while a V-lock on a data item is set only by a transaction in its validation phase. The two lock modes are not compatible. Briefly, the validation operation is carried out by checking the lock compatibility with the lock table.

As we mentioned above, to satisfy the requirement of *serializability*, two conditions must hold. Here Condition 1 is ensured by the locking mechanism. We give two protocols for ensuring Condition 2. Again, we use T_i (i = 1,2,...,n) to denote transactions in read phase and T_j as the transaction in validation phase. We bracket a critical section by "<" and ">".

[2]It was called *pseudo-locking* in our earlier work [10].

2.2.1 Serial validation-write: OCCL-SVW

A simple way to guarantee Condition 2 is to embed the validation phase and write phase in one critical section. We call this scheme *serial validation-write* in the sense that the validation phase and the write phase are indivisible.

During the read phase, each transaction T_i works on its own local copy and sets an R-lock in LT for every data object in its RS(T_i). During the validation phase, the validating transaction T_j checks its WS(T_j) against RS(T_i) by setting the V-lock for all data objects in its WS(T_j) in LT. If all the V-locks can be obtained, that means that the write set of the validating transaction does not intersect with the read set of any other active transactions. At this point, the validating transaction deletes its R-locks in LT and proceeds to its write phase. A failure to set a V-lock, on the other hand, indicates that Condition 1 has been violated, since this implies that a data object in WS(T_j) also falls in RS(T_i) of other transaction(s). In this case, the real-time conflict resolution policy is invoked to resolve the conflict (see Section 2.4). We give the protocol via the following pseudo code.

OCCL_SVW:

Read phase:

< **for** every data object in RS(T_i) **do**
    set an R-lock in LT >

Validation and Write Phase:
VALID := true

< **for** every data object in WS(T_i) **do**
    set a V-lock in LT
    if an R-lock of that object exists
      then VALID := false
release T_j's R-locks in LT
if VALID
    then execute write phase
    else invoke real-time conflict resolution
release T_j's V-locks in LT >

OCCL_SVW is simple and is easy to implement. It can be applied both to main memory resident database systems where the write phase is done in main memory and to disk resident databases. The latter will produce good performance if most of the transactions are queries.

On the other hand, *serial validation-write* may not be necessary if conflicts occur rarely between update transactions. Also, since the write phase and the validation phase are embedded together in a critical section, the critical section can easily become a bottleneck. This is especially true of disk resident databases. To separate the write phase from the critical section, we now give another protocol, called *parallel validation-write.*

2.2.2 Parallel validation-write: OCCL-PVW

In order to separate the write phase from the critical section and at the same time to guarantee Condition 2, transactions in read phase need to set R-locks based on their *write set* as well as their *read set*. Let T_{RH} be the set of transactions which hold an R-lock on the same data object in LT. We give the protocol for parallel validation-write in the following.

OCCL_PVW:

Read phase:

< **for** every data object in RS(T_i) **and** WS(T_i) **do**
  set an R-lock in LT
  if a V-lock of that object already exists in LT >
    then wait for V-lock release

Validation and Write Phase:
VALID := true

< **for** every data object in WS(T_j) **do**
  set a V-lock in LT
  if an R-lock of that object exists **and** T_j is not the only one in T_{RH}
    then VALID := false
release T_j's R-locks in LT
if not VALID
  then invoke real-time conflict resolution >
  else execute write phase
< release T_j's V-locks in LT>

In this protocol, a *write-write* conflict between a transaction in its validation/write phase and a transaction in its read phase can be detected and the two write operations on the database are serialized by the locking scheme. Thus Condition 2 is guaranteed.

Compared with OCCL_SVW, OCCL_PVW provides greater concurrency by separating the validation phase and the write phase from one critical section. On the other hand, OCCL_PVW may cause a higher conflict rate, since there exist not only read-write conflicts but also write-write conflicts now. Which protocol is chosen depends on the type of database system (memory or disk-resident) and the kind of workload (write/read ratio).

2.3 Some Implications

Since a locking scheme is used in OCCL, it is necessary to compare OCCL with two-phase locking (2PL) approach in terms of mechanism and implementation.

2.3.1 Locking mechanism

With respect to locking mechanism, OCCL presented here is different from 2PL. Note that a V-lock is issued at the end of a transaction and the locking period is the duration of validation phase plus write phase. With 2PL, however, the write lock is issued whenever the update transaction accesses a data object for update and the locking period may be as long as the transaction lifetime. Furthermore, the R-lock used here will not block any concurrent transactions in read phase, while under 2PL any conflict between read/write locks will block the conflicting transactions.

In addition, OCCL is deadlock-free, even though R-lock and V-lock are used. This is guaranteed by letting the validating transaction set V-locks in the critical section. Since a transaction that has been granted all of its V-locks will not request any lock after leaving its critical section, it is not possible for it to wait for any other (lock-holding) transactions. Thus, a wait-for cycle cannot be formed.

Because both OCCL and 2PL use locks, *priority inversion* may occur.[3] With 2PL, priority inversion can be avoided by forcing the high priority transaction to abort the low priority transaction so that a higher priority transaction is never blocked. The problem is more complicated in OCCL than 2PL. Priority inversion may occur in two places with OCCL. One is in the validation phase, where V-lock setting fails. This problem is addressed by various conflict resolution policies (see Section 2.4). Priority inversion may also happen in the read phase when a transaction attempts to set an R-lock for the data object to be accessed. In this case, it is preferable to let the higher priority transaction in the read phase wait for the low priority transaction. This is because the low priority transaction is already in its validation stage or perhaps even in its write phase. Aborting a transaction near completion may cost more, on average, than blocking a higher priority transaction for a limited period of time. To shorten the blocking period, a priority inheritance scheduling scheme can be applied during the validation phase and write phase [11]. For instance, the CPU scheduler may raise the process priority of the validating transaction to the highest among the concurrent transactions, thus reducing the time for validation processing. In addition, we may use transaction priority to manage access to the critical section. When more than one transaction is waiting for the critical section, then the one with the highest priority will get access first. Therefore, the worst case blocking time for the higher priority transaction is limited to the delay involved in transaction validation (under OCCL_PVW).

2.3.2 Starvation problem

Another problem that 2PL and OCCL may encounter is *starvation*. In this context, starvation occurs when transactions are restarted again and again until they miss their deadline. Long transactions have a higher probability of being starved because of their higher probability of access conflict. This results in a lower deadline guarantee ratio for long transactions than for short transactions. In traditional database systems, OCCL may result in more severe starvation because of its high degree of parallelism. Many solutions to the starvation

[3] *Priority inversion* [16] refers to the situation where a high priority transaction is blocked by a low priority transaction due to access conflict.

problem have been proposed (e.g., [14,15,19]). These schemes basically rely on limiting the number of transaction restarts. Given the timing constraints in real-time database systems, we use CPU scheduling to address the starvation problem. Based on our earlier studies on transactions with different characteristics [9], here we group transactions into classes by transaction length and assign a weight to each class. The weighting factor is incorporated in the CPU scheduling such that long transactions may have higher priority over short transactions. Using transaction deadline information, the weighted transaction priority is calculated by

$$p = (d - t)/w, \quad d > 0,\ t > 0,\ w \geq 1.$$

where d is the transaction deadline, t is the time when CPU scheduling takes place, and w is the length weighting factor. The smaller the p value, the higher the transaction priority. The specific weights used are discussed in Section 3. Note that for transactions with the same length, this corresponds to the *earliest-deadline-first* scheduling strategy.

2.3.3 Implementation overhead

In terms of physical implementation, both OCCL and 2PL require a central lock table. For the sake of comparison, here we list the lock table operations required by the two schemes.

OCCL
1. insert a data object ID with R-lock into the lock table during read phase;
2. search for a data object ID and convert the corresponding R-lock into V-lock (if the object has been updated) during validation phase;
3. delete a data object ID when an R-lock is released during validation phase or when a V-lock is released at the end of write phase.

2PL
1. search for a data object ID and check its lock compatibility against the lock mode of lock holder(s);
2. insert a data object ID with *read* or *write* lock into the lock table;
3. delete a data object ID when a lock is release at the end of the transaction.

It is clear that the physical operations on the lock table are the same for the two protocols. Despite the similarity, there are some differences between OCCL and 2PL. For example, 2PL needs to detect potential deadlock before a lock request is queued while OCCL does not. The implementation overhead of the two concurrency control protocols has been examined through experiments and the results are presented in Section 4.

2.4 Conflict Resolution

With OCCL presented above, an algorithm is needed to resolve the access conflicts during the validation phase. This conflict resolution should consider transaction priority based on

transaction deadlines and length as discussed above. In other words, the resolution policy should aim at improving the performance of real-time transactions in terms of meeting transaction deadlines. Here are some basic resolution policies:

1. Commit: CMT
   Always let the validating transaction commit and abort all the conflicting transactions. This strategy guarantees that as long as a transaction reaches its validation phase, it will always finish. The advantage of this strategy is that the resources (CPU, I/O, etc.) consumed by a finishing (validating) transaction are never wasted. Applying CPU scheduling, we expect that transactions with higher priority have a higher probability of reaching the validation phase and in turn have a higher probability of committing.

2. Priority abort: PA
   Abort the validating transaction only if its priority is less than that of all the conflicting transactions. This strategy takes transaction priority into account, but still favors the validating transaction, aiming at reducing the resources wasted due to aborted transactions.

3. Priority wait: PW
   If the priority of the validating transaction is not the highest among the conflicting transactions, wait for the conflicting transactions with higher priority to complete. In some cases, the strategy of aborting conflicting transactions appears too conservative, causing unnecessary transaction abort. Consider the situation where the validating transaction conflicts with transactions which have only read operations. If the validating transaction has a lower priority compared with other conflicting ones, instead of being aborted, it may be deferred. In other words, this transaction is "preempted" from its validation phase and is placed in a waiting queue to wait until all of the conflicting transactions with higher priority finish their validation. One variation of the priority wait strategy is WAIT-50 proposed in [7], where a validating transaction will wait if at least 50% of the conflicting transactions have a higher priority over the validating transaction. The protocol aims at balancing the wait factor and the priority cognizance.

There can be other variations of the conflict resolution strategy. Since in this study we emphasize the fundamental analysis of OCC performance with respect to its implementation, we only examine the three simple conflict resolution policies discussed above.

3 Test Environment

The proposed locking-based optimistic concurrency control protocol, together with several real-time conflict resolution schemes, have been implemented and evaluated on our real-time database testbed RT-CARAT [9]. In this section, we briefly introduce the testbed organization and describe the system and workload parameter settings.

3.1 Testbed organization

Currently, RT-CARAT is a centralized, secondary storage real-time database testbed built on top of the VAX/VMS operating system. It contains all of the major functional components of a transaction processing system, such as transaction management, data management, log management, and communication management. The testbed is implemented as a set of cooperating server processes which communicate via efficient message passing mechanisms. A pool of transaction processes (TR's) simulate the users of the real-time database. Accordingly, there is a pool of data managers (DM's) which service transaction requests from the user processes (the TR's). There is one transaction manager, called the TM server, acting as the inter-process communication agent between TR and DM processes. The communications between TR, TM and DM processes are carried out through mailboxes, a facility provided by VAX/VMS. To be more efficient, TM and DM processes also share some information, such as transaction deadline and priority, through a common memory space, called the global section in VAX/VMS.

Using the underlying VAX/VMS operating system real-time priorities, the priority-driven preemptive scheduling is done by a CPU scheduler embedded in the TM. Upon the arrival of a new transaction, the scheduler assigns a priority to the transaction according to the CPU scheduling policy. The scheduling operation is done by mapping the assigned transaction priority to the real-time priority of the DM process which carries out the transaction execution. At this point, an executing DM will be preempted if it is not the highest priority DM process at the moment, otherwise it will continue to run until it completes or until it needs to wait for an I/O. Concurrency control is part of the DM process. It incorporates the CPU scheduler of the TM process in its real-time conflict resolution.

RT-CARAT is a system that contains a fixed number of users that submit transaction requests one after another, with a certain think time (τ) in-between. This model captures many applications in the real world, although certainly not all applications (e.g., an open system model is more appropriate for a process control system). For example, in an airline reservation system, there is a fixed number of computer terminals. The airline clerk at each terminal may check a flight, reserve a seat, or cancel a reservation for customers. After submitting a request to the system, the clerk waits for a result. He may submit another request after getting a response from the previous one.

A transaction is characterized by its *length* and *deadline*. The length is specified by $T(x, y)$, where x is the number of steps that a transaction needs to execute, and y is the number of records accessed in each step. Transaction deadline is randomly generated from a uniform distribution within a deadline window, $[d\_base, \alpha \times d\_base]$, where d_base is the window baseline and α is a variable determining the upper bound of the deadline window. For each workload in the experiments, d_base is specified first by the formula:

$$d\_base = avg\_rsp - stnd\_dvi$$

where avg_rsp is the average response time of the *read-only* transactions with the same length when executed in a non real-time database environment, and $stnd\_dvi$ is the standard deviation of the response time.

Table 1: System Parameters

Parameter	Settings
Disks	disk1: database; disk2: log.
MPL (multi-programming level)	10, 8, 6, 4
DB (database size)	$MPL * 100$ blocks (6 records/block)

A transaction terminates upon completion or a termination abort. The latter refers to the situation where a transaction has missed its deadline and it is thus aborted by the system. A transaction aborted due to deadlock or data access conflict will be restarted as long as it has not passed its deadline. Hence a transaction may make multiple *runs* before it eventually terminates. Note that a restarted transaction will access the same set of records as it did in its first run.

3.2 Parameter settings

Table 1 summarizes the system parameter settings. The experiments were conducted on a VAXstation 3100/M38 with two RZ55 disks, one for the database and the other for the log. Given the physical machine, in order to examine the degree of resource contention (CPU and I/O), the system multi-programming level (MPL) is varied from 10 to 4. While this is a low degree of multiprogramming, compared to what we would find in practice, the database size in the experiments (1000 - 400 blocks) is also smaller than we would find in practice. With a proper system scaling, many factors, such as the level of data access conflict, can model practical situations. Thus, the performance results obtained from the smaller system can reflect the performance of a larger system. In our experiments, in order to isolate the effect of resource contention from that of data contention, the database size is set proportional to MPL.

Table 2 describes the workload parameters and their settings in the experiments. We consider two workloads; one where all transactions consist of 6 steps, $P[x = 6] = 1$, and the other where half have 4 steps and the other half 8 steps, $P[x = 4] = P[x = 8] = 1/2$. The latter workload is used particularly for analyzing the starvation problem. The number of records to be accessed per transaction step, y, is fixed at 4. The deadline window factor, α, is a timing-related parameter which specifies the deadline distribution of real-time transactions. The smaller the value of α, the tighter the transaction deadlines and vice versa. In RT-CARAT, a transaction is either *read* (where each *step* is a sequence of FIND and GET operations) or *write* (where each *step* is a sequence of FIND, GET and MODIFY operations).[4] The probability that a transaction is a write transaction, P_w, is another parameter we are interested in, since it directly affects transaction conflict rate. The transaction external think time, τ, is set at 0 in the experiments.

[4]FIND, GET, MODIFY etc. are the statements of Data Manipulation Language in VAX DBMS. The corresponding operations are fully implemented on RT-CARAT.

Table 2: Workload Parameters

Parameter	Settings
x (steps per transaction)	4, 6, 8 steps
y (records accessed per trans. step)	4 records
α (deadline window factor)	2.0 - 6.0
P_w (prob. of write transactions)	0.0 - 1.0
τ (external think time)	0.0 seconds

4 Experimental Results

In this section, we present experimental results from our performance studies.

Table 3 lists the schemes examined in the experiments. We consider two basic concurrency control protocols, 2PL and OCCL, in combination with different conflict resolution policies.[5] 2PL_NRT and OCCL_NRT are two baselines for the purpose of performance comparisons. They correspond to 2PL and OCCL schemes in non real-time (NRT) database systems, where a multi-level feedback queue algorithm is used for CPU scheduling. In case of access conflict, under 2PL_NRT, the lock-requesting transaction is put into a *wait* queue; under OCCL_NRT, the validating transaction *always commits*. 2PL_WAIT and OCCL_CMT employ priority-driven, preemptive scheduling. Transaction priority is assigned according to *earlier-deadline-first* policy. Still, the two schemes do not take transaction timing constraints into account for resolving access conflict. 2PL_PA and OCCL_PW consider transaction priority for both CPU scheduling and conflict resolution. Note that here the conflict resolution scheme PW refers to WAIT-50[7].

Table 3: Schemes Examined

Scheme	Conflict resolution	CPU scheduling
2PL_NRT	wait	Multi-level feedback queue
2PL_WAIT	wait	Earliest deadline first
2PL_PA	priority abort	Earliest deadline first
OCCL_NRT	commit	Multi-level feedback queue
OCCL_CMT	commit	Earliest deadline first
OCCL_PW	priority wait	Earliest deadline first

Besides the above schemes, we also examined the conflict resolution policy PA for OCCL (i.e., OCCL_PA). The results show that PA performs no better than CMT due to

[5]The optimistic concurrency control protocol implemented on RT-CARAT is OCCL_PVW. This is because the testbed is a disk-resident real-time database and cannot afford the long waits (for writing) inherent in OCCL_SVW. In the rest of the paper, we refer to it as OCCL.

aborts of the (validating) transaction near its completion. To save space, we do not include these experimental results here.

The basic metric used for performance evaluation is *deadline guarantee ratio*, which is the percentage of transactions that complete by their deadline. We also collect statistics on transaction abort ratio, blocking time, wasted operations, and CPU and I/O utilizations so as to provide insights into the protocol performance.

The data collection in the experiments is based on the method of replication. The statistical data has 95% confidence intervals whose end points are within 2% of the point estimate for deadline guarantee ratio. In the following graphs, we only plot the mean values of the performance measures.

4..1 Experiment 1: Protocol overhead

In our performance studies, we first compare the implementation overhead of the two types of concurrency control protocols, 2PL and OCCL.

The overhead is measured by the average CPU processing time spent on concurrency control per page. To capture the overhead under all the execution paths, we vary the write probability P_w. At this point, other parameter settings are irrelevant. Figure 1 indicates that the implementation overheads of the two protocols are quite close. This is due to the fact that even though the two protocols differ at the logical level (two-phase locking vs. optimistic approach), the underlying physical implementations are very similar. Both protocols rely on a locking technique for data access control, and they both involve hashing operation and lock table management. Despite the similarities, 2PL employs deadlock detection while OCCL does not. However, our previous studies [9] have shown that the deadlock detection on RT-CARAT does not incur significant overhead. On the other hand, the implementation of OCCL costs more to maintain read/write sets for each individual transaction. This may be the reason why OCCL has slightly larger overhead than 2PL.

Knowing that the two logically different protocols have similar overhead with respect to their physical implementations, we now analyze how the implementation schemes affect the performance of the two protocols.

4.1 Experiment 2: Data contention

In this experiment, we examine the protocol performance under different data contention levels by varying the write probability, Pw. We fix the multi-programming level at 8 with $x = 6$ and $\alpha = 5$.

Figure 2 shows the *transaction deadline guarantee ratio* for six schemes. As one would expect, the deadline guarantee ratio drops as data contention increases. The performance of two baselines, 2PL_NRT and OCCL_NRT, is consistent with the results from previous studies (e.g., [4,1]), i.e., non real-time two-phase locking outperforms non real-time optimistic approach under data and resource contention. Here an interesting observation is that combined with priority-driven preemptive scheduling, the optimistic approach (OCCL_CMT)

performs better than two-phase locking (2PL_WAIT). Furthermore, as we incorporate transaction priority into conflict resolution for the two types of protocols, 2PL_PA further increases the deadline guarantee ratio, with respect to 2PL_WAIT, by as much as 17% for P_w = 0.6, while OCCL_PW performs only slightly better than OCCL_CMT.

The performance of these schemes may be affected by several factors, such as transaction blocking time, priority inversion and abort ratio. Based on the implementation details, we explain the results shown in Figure 2 in the following.

A transaction can be blocked due to access conflict. Under OCCL, this happens in the transaction read phase where an R-lock requesting transaction has to wait for the transaction holding the V-lock. In addition, under OCCL_PW, a validating transaction may be blocked when it conflicts with higher priority transactions in read phase. Under 2PL, blocking can occur at any point along the course of its execution whenever there is a *read-write* or *write-write* conflict. Figure 3 depicts the *average transaction waiting time* (in seconds) for each blocking instance. Overall, the waiting time under OCCL scheme is shorter than under 2PL. This is because even though both schemes rely on locking, OCCL shrinks the locking period (V-lock holding time) to the final stage of transaction execution, thus reducing the waiting time. Furthermore, as we discussed in Section 2.3.1, applying priority-driven CPU scheduling to OCCL further reduces the waiting time as much as 40% (comparing OCCL_NRT with OCCL_CMT and OCCL_PW). Compared with OCCL_CMT, the waiting time under OCCL_PW is increased by about 10%, from 0.59 to 0.65 (seconds), for P_w = 0.2. On the other hand, when P_w is high, the two schemes perform the same. This is a direct result of the implementation which avoids cyclic V-lock conflicts between two write transactions.

Figure 4 shows the *total waiting time* for each transaction run. The trend here is similar to what we observed in Figure 3.

As discussed in Section 2.3.1, *priority inversion*, a special case of transaction blocking, may occur under both 2PL and OCCL. Figure 5 plots the *average number of priority inversions* encountered per transaction run. In 2PL_PA, a high priority transaction will not wait for a low priority transaction when a conflict occurs. Hence 2PL_PA performs the best in terms of avoiding the problem of priority inversion. Note that the priority inversion under 2PL_PA is slightly greater than 0. This is because on RT-CARAT, the high priority transaction is forced to wait for a lower priority transaction if the low priority transaction has already completed its write operations on the database and is about to release its locks. Under OCCL, since a blocked higher priority transaction in read phase has to wait for a V-lock holder to complete its validation phase and write phase, the probability for priority inversion to occur is higher than 2PL_PA, especially when Pw is large. Again, combined with priority-driven CPU scheduling, OCCL_CMT and OCCL_PW perform much better than OCCL_NRT. Under OCCL_PW, a transaction in read phase has less chance to be blocked since the validating transaction might be in the validation-wait state. Thus, the probability of priority inversion under OCCL_PW is slightly lower than that under OCCL_CMT.

Transaction abort rate is another major factor that affects the protocol performance. Figure 6 illustrates the *average transaction abort ratio* (i.e., the percentage of transactions aborted due to deadlock or access conflict). Clearly, the wait-oriented schemes, 2PL_NRT

and 2PL_WAIT, result in a much lower abort ratio than the abort-oriented schemes - 2PL_PA and OCCL. With a high degree of parallelism and the shorter blocking time (see Figures 3 and 4), all the OCCL schemes have a lower abort ratio than 2PL_PA when the data contention is low, but a higher abort ratio when the data contention becomes high. The saturation behavior under 2PL is due to its increased blocking effect when the data contention becomes high. Among the three OCCL schemes, OCCL_PW has the lowest abort ratio (12% lower than OCCL_CMT for $P_w = 0.2$), since it incorporates a wait mechanism in the validation phase.

In the context of transaction abort, we also measure the *transaction abort length*, i.e., the number of steps that have been processed when a transaction is aborted. To reduce the wasted system resources, the abort length should be as short as possible. Figure 7 shows the abort length for the six schemes. Since 2PL_PA may detect access conflict and immediately apply priority abort at any point along the course of its execution, it results in the shortest abort length. 2PL_NRT and 2PL_WAIT allow aborts only when a deadlock is detected. This "pessimistic" wait strategy leads to longer abort lengths. For OCCL, because of its high degree of parallelism, a transaction may proceed until it is aborted by a validating transaction, thus resulting in the abort length longer than for 2PL_PA. Integrated with CPU scheduling, OCCL_CMT and OCCL_PW make the higher priority transaction proceed faster than other lower ones, which means that the abort length of the transactions in read phase becomes shorter than that under OCCL_NRT.

Figure 8 depicts the *wasted operations per transaction execution*, i.e., the number of steps wasted for each submitted transaction. This measurement reflects the combined effect of both transaction *abort ratio* and *abort length*.

With respect to *resource consumption*, CPU and I/O utilizations are plotted in Figure 9 and Figure 10, respectively. As one expects, the wait-oriented schemes, 2PL_NRT and 2PL_WAIT, consume less resources than the abort-oriented schemes - 2PL_PA and OCCL. Due to a high degree of parallelism and the shorter blocking time, OCCL results in higher CPU and I/O utilizations than 2PL_PA.

Having examined the protocol performance in detail, we come to the following points with respect to the performance results demonstrated in Figure 2.

- CPU scheduling algorithm that takes transaction deadlines into account plays an important role in improving the performance of concurrency control protocols, particularly for OCCL which provides a high degree of parallelism and short blocking period.

- The three schemes, 2PL_PA, OCCL_CMT and OCCL_PW, with the least priority inversions, perform the best. The difference between the three schemes depends on the amount of wasted operations. 2PL_PA performs the best when data contention is high, since it results in the least wasted operations.

- The wait strategy employed by OCCL_PW has no significant impact on improving OCCL performance. It is the effect of increased waiting time that overshadows the performance gain due to reducing wasted operations. In addition, the implementation

scheme for avoiding cyclic V-lock conflicts prevents the wait strategy from taking part in conflict resolution when the probability of write-write conflict is high.

Our results show that 2PL_PA, OCCL_CMT and OCCL_PW are superior to the other protocols. Moreover, the further experiments with various workload settings show that there is no significant performance difference between OCCL_CMT and OCCL_PW. To simplify the presentation and to save space, we only demonstrate and compare the performance of 2PL_PA and OCCL_CMT in the following sections.

4.2 Experiment 3: Deadline distribution

Deadline distribution may also affect protocol performance. Extending Experiment 2, we vary the tightness of transaction deadlines while fixing the write probability Pw.

We first examine the possible effect of the deadline distribution on performance when low data contention is low, $Pw = 0.2$. Figure 11 plots the deadline guarantee ratio versus deadline window factor α. As we have observed in Figure 3, when the deadline is loose ($\alpha = 5$), 2PL_PA and OCCL_CMT show similar performance, since transactions complete by their deadline most of the time. Now as α decreases, OCCL_CMT becomes superior to 2PL_PA. This can be explained as follows: When data contention is low, the two protocols have nearly the same probability of priority inversion (see Figure 5) and the same amount of *wasted operations* (see Figure 8). Under such a condition, the protocol with shorter blocking time (see Figure 4) wins.

Next we vary the deadline window factor α under high data contention with $Pw = 0.8$. Figure 12 shows the deadline guarantee ratio for 2PL_PA and OCCL_CMT, respectively. In contrast to the results shown in Figure 11, here 2PL_PA outperforms OCCL_CMT when the deadlines are relatively loose. This is mainly due to the fact that both the wasted operations and the probability of priority inversion under OCCL_CMT increase as data contention becomes high. Even though 2PL_PA has a longer blocking time, it works better as long as the transaction deadlines are long enough.

4.3 Experiment 4: I/O resource contention

All of the experiments presented above are carried out in a system with I/O resource contention, where the I/O utilization under 2PL_PA and OCCL_CMT was always above 93% with average queue length > 4 (see Figure 10). In this set of experiments, we examine the protocol performance in a system where there is no severe resource contention. To do so, we reduce MPL from 8 to 4. Note that the database size is also reduced correspondingly, from 800 to 400 (blocks), so that the level of data contention for $MPL = 4$ is comparable with that for $MPL = 8$.

We first exercise the two concurrency control schemes under low data contention. Figure 13 illustrates the deadline guarantee ratio versus deadline window factor α with $Pw = 0.2$. Under such workloads, the I/O utilization drops below 83%. Comparing Figure 13 with Figure 11, we observe again that the two protocols perform basically the same. We

have also observed (not shown here) that the two protocols perform the same with respect to priority inversion and wasted operations, but 2PL_PA results in longer waiting time than OCCL_CMT on locking. This is the main reason why reducing resource utilization does not affect the protocol performance.

The possible effect of resource contention is then examined under high data contention. Figure 14 shows the deadline guarantee ratio for $Pw = 0.8$. Comparing it with Figure 12, we see the similarity again, despite the drop of I/O utilization from 95% for $MPL = 8$ to 80% for $MPL = 4$ (under OCCL_CMT). Under high data contention, the high abort ratio and the long abort length of OCCL_CMT lead to a larger number of wasted operations, about 25% higher than 2PL_PA. Furthermore, the chance of priority inversion for OCCL_CMT becomes high (0.16), as compared to 2PL_PA (0.04). These two factors, particularly the priority inversion, degrade the OCCL_CMT performance.

Here we can see that reducing resource utilization does not improve OCCL performance. Under OCCL, due to the use of locking, the effect of priority inversion is sensitive to the duration of the write phase. Therefore, it is the I/O speed that needs to be improved.

4.4 Experiment 5: Transaction length

The transactions we have exercised thus far are equal in length ($x = 6$). We now look at workloads with a mix of different transaction lengths. To make data analyzable and yet comparable with previous results, we exercise the workload with two lengths of transactions, $x = 8$ (*long*) and $x = 4$ (*short*), with mean value 6 (i.e, $P[x = 4] = P[x = 8] = 1/2$).

Figure 15 shows the transaction deadline guarantee ratio versus P_w for 2PL_PA and OCCL_CMT. Examining the average deadline guarantee ratio, we can see that the result is similar to what we have observed in Figure 2 for $x = 6$, i.e., 2PL_PA performs better than OCCL_CMT when data contention is high. However, as we examine the deadline guarantee ratio of long and short transactions, we see that under data contention, OCCL_CMT outperforms 2PL_PA for short transactions while 2PL_PA performs much better than OCCL_CMT for long transactions. In addition, under both schemes, the deadline guarantee ratio of short transactions is much higher than that of long transactions. This observation identifies the starvation problem. Clearly, both of the abort-oriented schemes result in transaction starvation. Due to its high degree of parallelism, OCCL_CMT leads to the more severe starvation than 2PL_PA.

We have proposed a *weighted priority scheduling* policy to cope with the starvation problem (see Section 2.3.2). Figure 16 shows the effect of such a CPU scheduling scheme on transaction starvation. Here we associate a weighting factor w to long transactions, varying from 1.0 to 2.6, while fixing w at 1.0 for short transactions. When w is equal to 1.0, the scheduling scheme follows the *earliest-deadline-first* policy. At this point, the average deadline guarantee ratio coincides with the previous results for $x = 6$ (see Figure 11). But, long transactions suffer from the starvation problem. As w increases, under OCCL_CMT, the deadline guarantee ratio of long transactions increases while the deadline guarantee ratio of short transactions decreases. Under 2PL_PA, however, the deadline guarantee ratio

of long and short transactions changes slowly. Note that the average deadline guarantee ratio under both schemes does not change with w.

The observation from the experiment indicates that OCCL_CMT is a more flexible scheme in that it can be integrated with an appropriate CPU scheduling policy in order to resolve transaction starvation. This is due to the fact that the transaction blocking time under OCCL_CMT is much shorter than that under 2PL_PA (see Figure 3), which gives the CPU scheduler more freedom to carry out priority scheduling. In addition, the weighted priority scheduling scheme follows the conservation law, i.e., the increase of the deadline guarantee ratio for long transactions leads to the decrease of the deadline guarantee ratio for short transactions, and the average deadline guarantee ratio is kept constant. This brings up the question of *fairness* on transaction scheduling. At this point, there is no criterion for a "fair scheduling". In practice, the system designer may choose the weighting factors for different groups of transactions such that their performance requirements can be met.

5 Conclusions

We have investigated real-time optimistic concurrency control at its level of physical implementation. We have developed a set of locking-based protocols for the optimistic approach. The protocols, together with several conflict resolution schemes, have been implemented and evaluated on a real-time database testbed. In contrast to the previous results from simulation[6,7], the results obtained from the testbed show that the optimistic concurrency control may not always outperform the two-phase locking which incorporates priority information in its conflict resolution. In particular, the performance difference between the two concurrency schemes is sensitive to the amount of data contention, but not to the amount of I/O resource contention (as measured by resource utilization). The optimistic scheme performs better than the two-phase locking scheme when data contention is low, and vice versa when data contention is high. The "degraded" performance of the optimistic approach (in comparison with the results shown in [6,7]) is due to the blocking effect (which in turn results in priority inversion as well as high abort rate) caused by the locking mechanism adopted in its implementation. The experimental results indicate that with respect to logical operations, the physical implementation schemes have a significant impact on the performance of real-time optimistic concurrency control.

In this paper, we have also explored the starvation problem with respect to the deadline guarantee ratio for transactions of different length. The performance studies show that both the abort-oriented two-phase locking and optimistic approaches result in starvation for long transactions. Integrated with the proposed *weighted priority scheduling*, the optimistic concurrency control scheme exhibits a greater flexibility in coping with the starvation problem.

Even though this study reveals some weakness of the optimistic approach with respect to its implementation, we believe that this approach is still a candidate for real-time concurrency control owing to its high degree of parallelism and its flexibility in handling conflict resolution and in integration with CPU scheduling. Since the effectiveness of the approach is closely related to its physical implementation scheme, its performance can be

further improved by adding certain processing components into the system. For example, regarding the locking-based scheme developed in this work, if a disk controller can perform the *write* operations in transaction validation order and it can also intelligently manage the order of *read* and *write* operations [3], the V-lock holding period can be largely reduced. The integration of concurrency control with I/O scheduling is an interesting topic for future work. Another example for improving the performance of the optimistic approach is the use of a database cache which can accommodate data pages to be accessed by restarted real-time transactions. The development of such a technique also remains part of future work.

Acknowledgements

The authors would like to thank Purimetla Bhaskar for helpful discussions and his assistance in implementing the OCCL_PW protocol on RT-CARAT.

References

[1] Agrawal, R., M.J. Carey and M. Livny, "Concurrency Control Performance Modeling: Alternatives and Implications," *ACM Transaction on Database Systems, Vol.12, No.4*, Dec. 1987.

[2] Abbott, R. and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," *Proceedings of the 14th VLDB Conference*, Aug. 1988.

[3] Abbott, R. and H. Garcia-Molina, "Scheduling I/O Requests with Deadlines: A Performance Evaluation," *Proceedings of the 11th Real-Time Systems Symposium*, Dec. 1990.

[4] Carey, M.J. and M.R. Stonebraker, "The Performance of Concurrency Control Algorithms for Database Management Systems," *Proceedings of the 10th VLDB Conference*, 1984.

[5] Harder, T. "Observations on Optimistic Concurrency Control Schemes," Information Systems, Vol. 9, No.2, 1984.

[6] Haritsa, J.R., M.J. Carey and M. Livny, "On Being Optimistic about Real-Time Constraints," PODS, 1990.

[7] Haritsa, J.R., M.J. Carey and M. Livny, "Dynamic Real-Time Optimistic Concurrency Control," *Proceedings of the 11th Real-Time Systems Symposium*, Dec. 1990.

[8] Huang, J., "Real-Time Transaction Processing," *Ph.D. Dissertation Prospectus*, Dept. of Electrical and Computer Engin., University of Massachusetts, June 1989.

[9] Huang, J., J.A. Stankovic, D. Towsley and K. Ramamritham, "Experimental Evaluation of Real-Time Transaction Processing," *Proceedings of the 10th Real-Time Systems Symposium*, Dec. 1989

[10] Huang, J. and J.A. Stankovic, "Concurrency Control in Real-Time Database Systems: Optimistic Scheme vs. Two-Phase Locking," *A Technical Report, COINS 90-66*, University of Massachusetts, July 1990.

[11] Huang, J., J.A. Stankovic, K. Ramamritham and D. Towsley, "On Using Priority Inheritance in Real-Time Databases," *A Technical Report, COINS 90-121*, University of Massachusetts, Nov. 1990.

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

[13] Lin Y. and S.H. Song, "Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order," *Proceedings of the 11th Real-Time Systems Symposium*, Dec. 1990.

[14] Peinl, P. and A. Reuter, "Empirical Comparison of Database Concurrency Control Schemes," *Proceedings of the 9th VLDB Conference*, Florence, Italy, 1983.

[15] Pradel, U., G. Schlageter and R. Unland, "Redesign of Optimistic Methods: Improving Performance and Applicability," *Proc. IEEE 2nd Int. Conf. on Data Engineering*, 1986.

[16] Sha, L., R. Rajkumar and J.P. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *ACM SIGMOD Record*, March 1988.

[17] Son, S.H. and C. Chang, "Performance Evaluation of Real-Time Locking Protocols using a Distributed Software Prototyping Environment," *Proceedings of the 10th International Conference on Distributed Computing Systems*, Paris, France, May 1990.

[18] Stankovic, J.A. and Wei Zhao, "On Real-Time Transactions," *ACM SIGMOD Record*, March 1988.

[19] Thomasian, A. and E. Rahm, "A New Distributed Optimistic Concurrency Control Method and a Comparison of its Performance with Two-Phase Locking," *Proceedings of the 10th International Conference on Distributed Computing Systems*, Paris, France, May 1990.

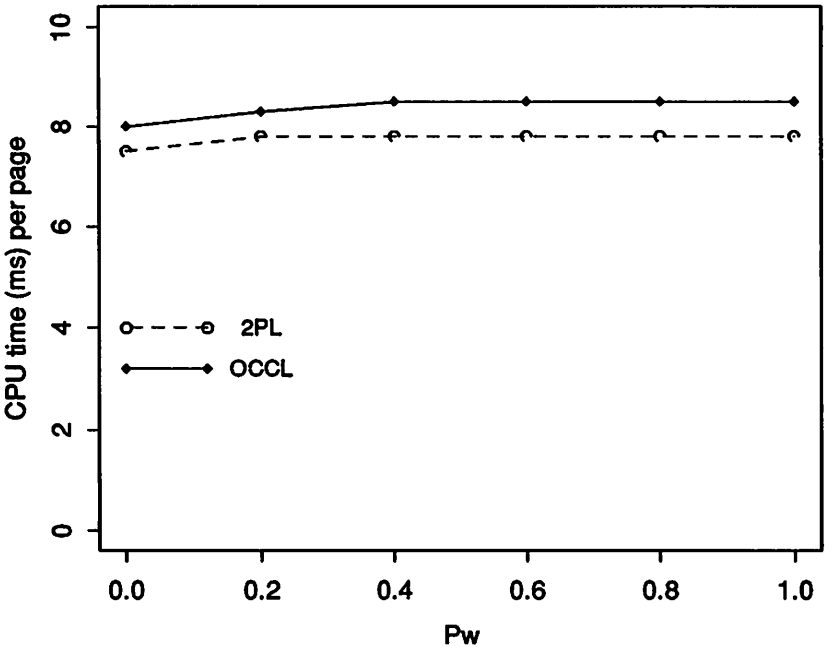

Fig.1: Concurrency control overhead

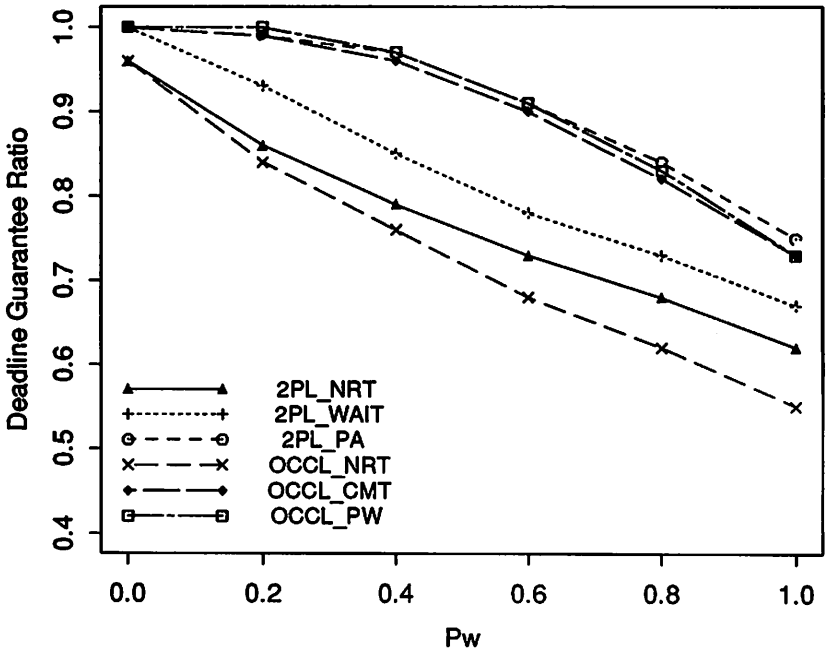

Fig.2: Data contention, MPL = 8, x = 6, a = 5

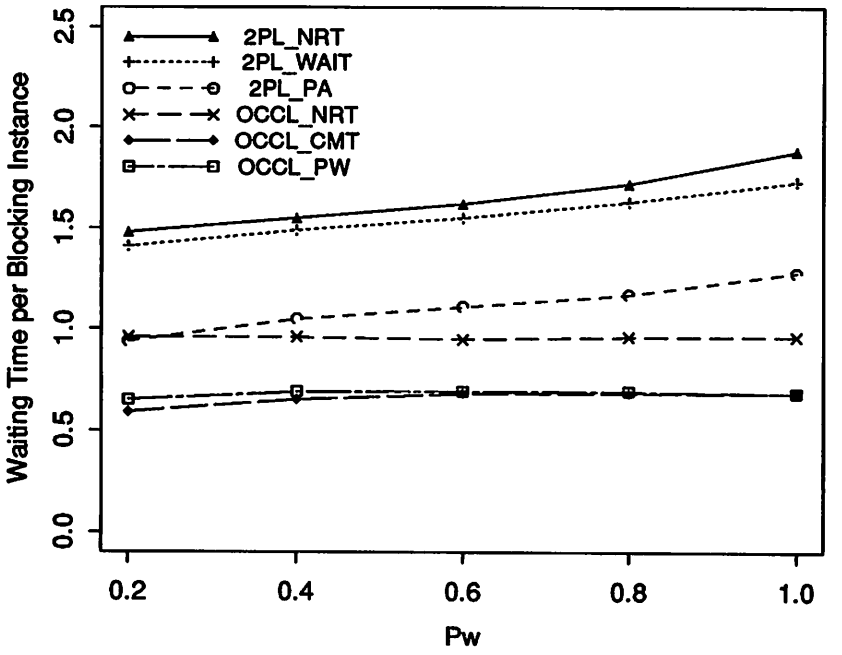

Fig.3: Data contention, MPL = 8, x = 6, a = 5

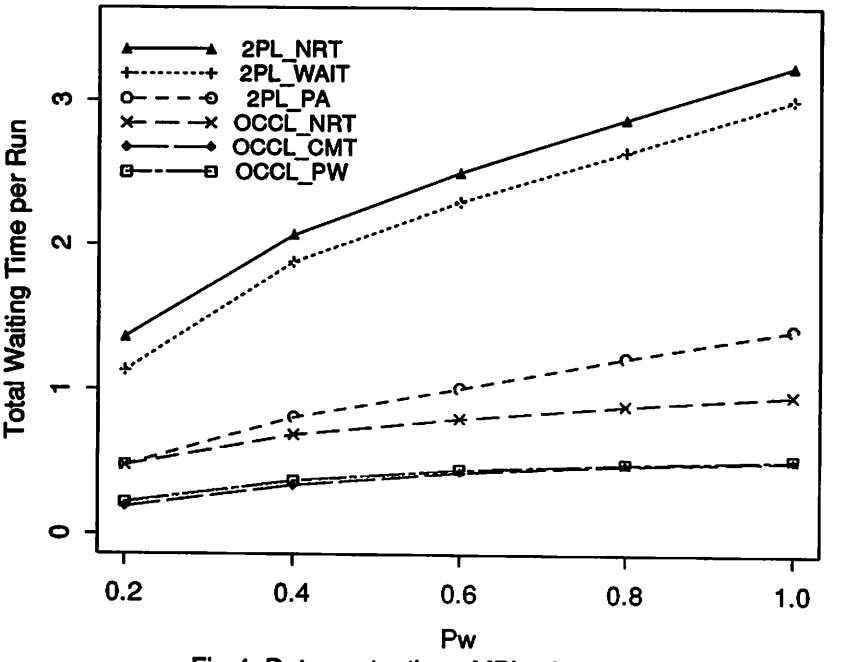

Fig.4: Data contention, MPL= 8, x = 6, a = 5

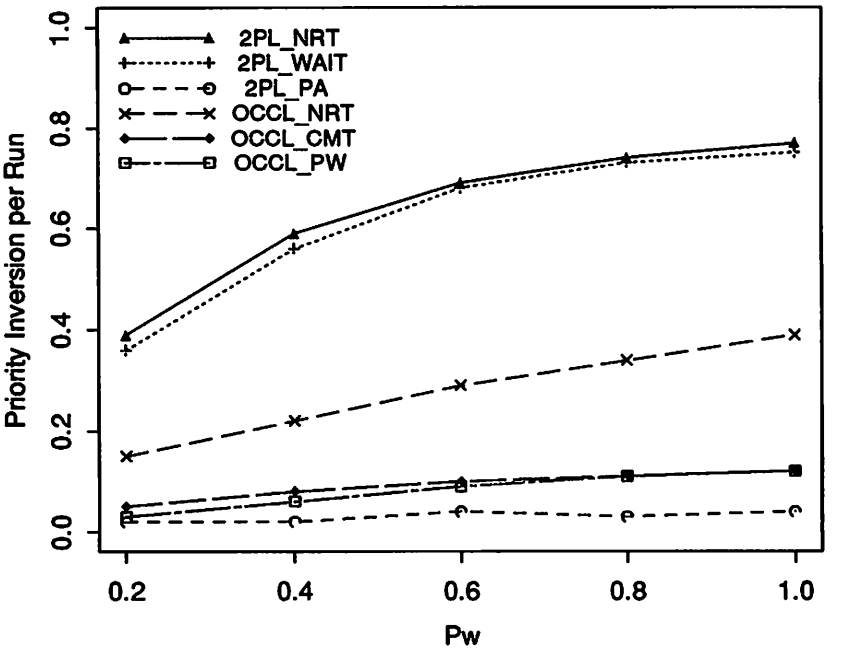

Fig.5: Data contention, MPL = 8, x = 6, a = 5

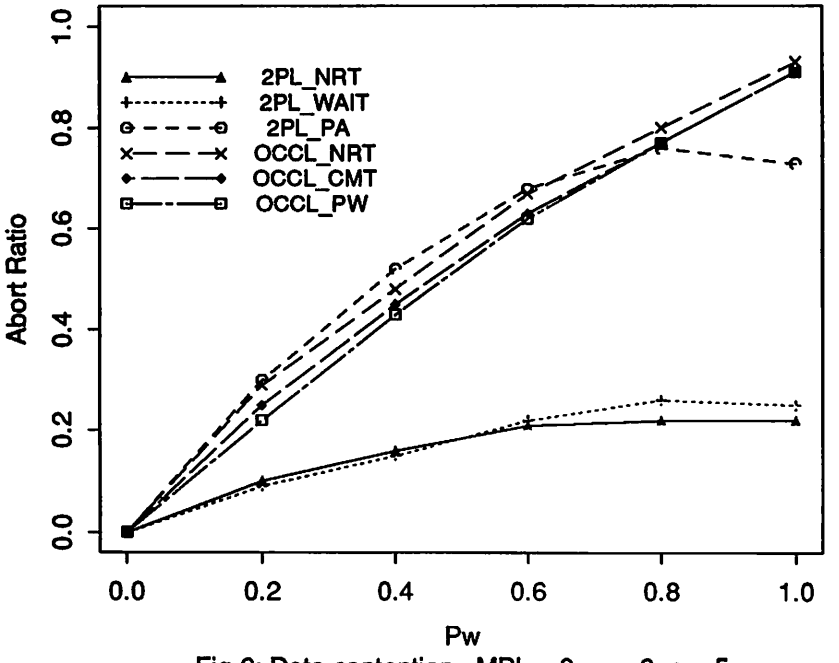

Fig.6: Data contention, MPL = 8, x = 6, a = 5

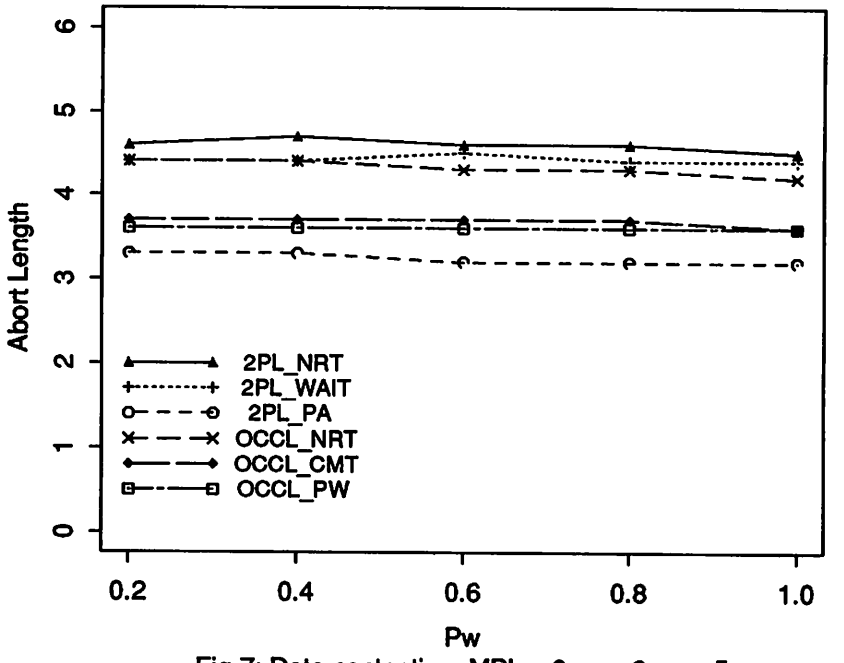

Fig.7: Data contention, MPL = 8, x = 6, a = 5

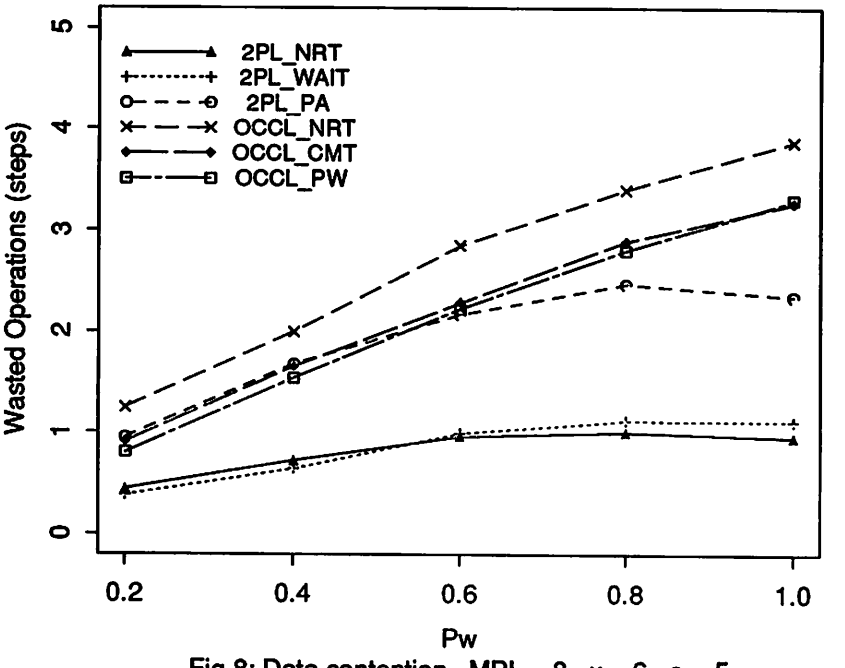

Fig.8: Data contention, MPL = 8, x = 6, a = 5

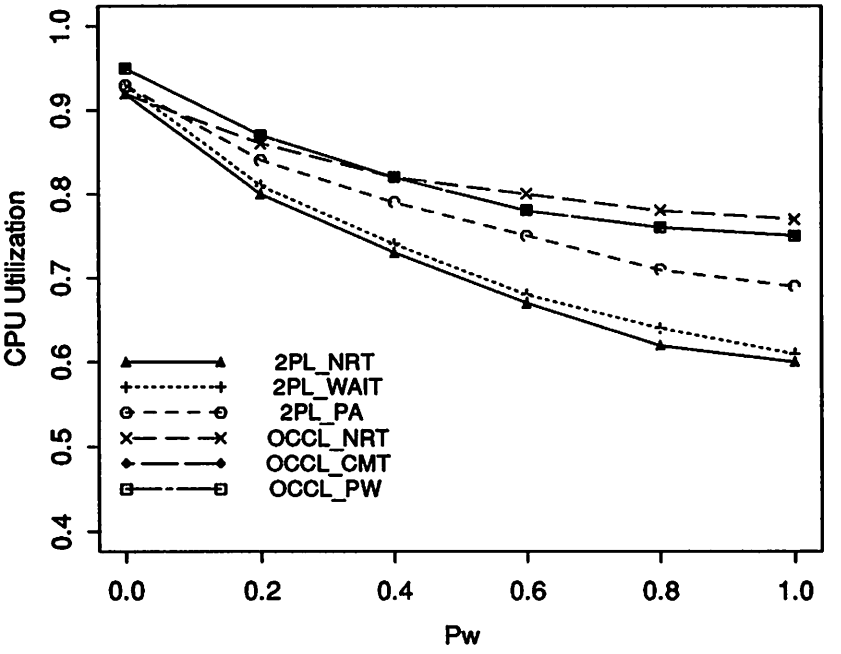

Fig.9: Data contention, MPL = 8, x = 6, a = 5

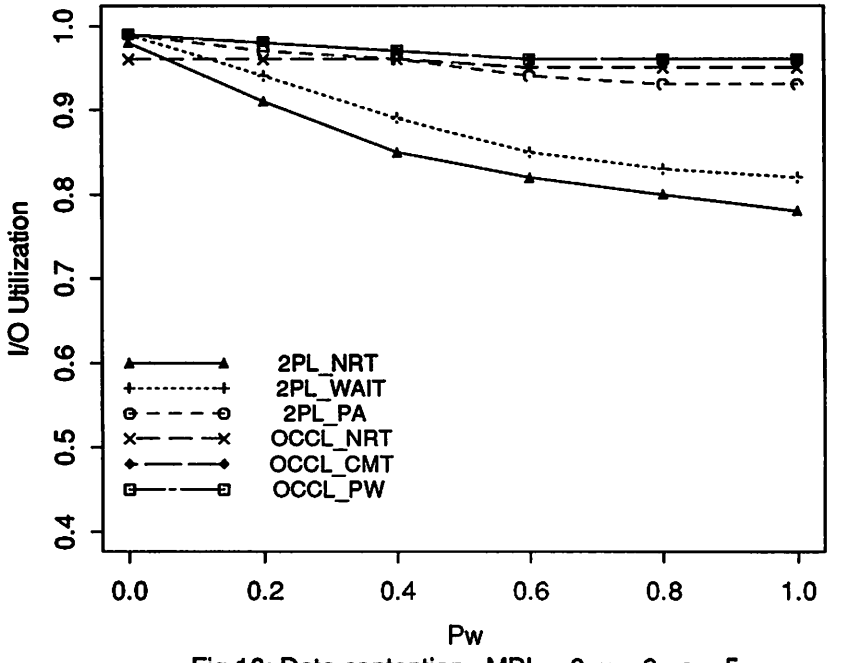

Fig.10: Data contention, MPL = 8, x = 6, a = 5

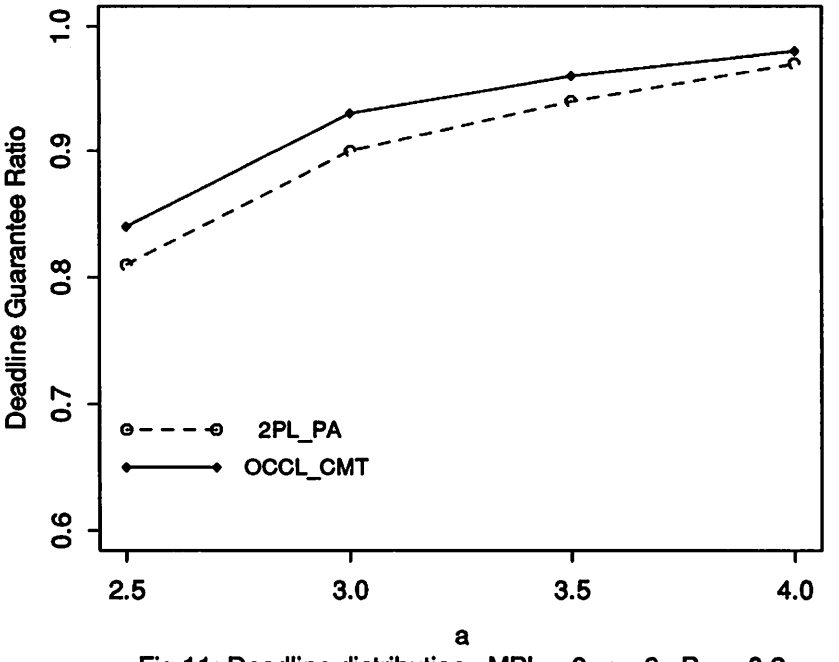

Fig.11: Deadline distribution, MPL = 8, x = 6, Pw = 0.2

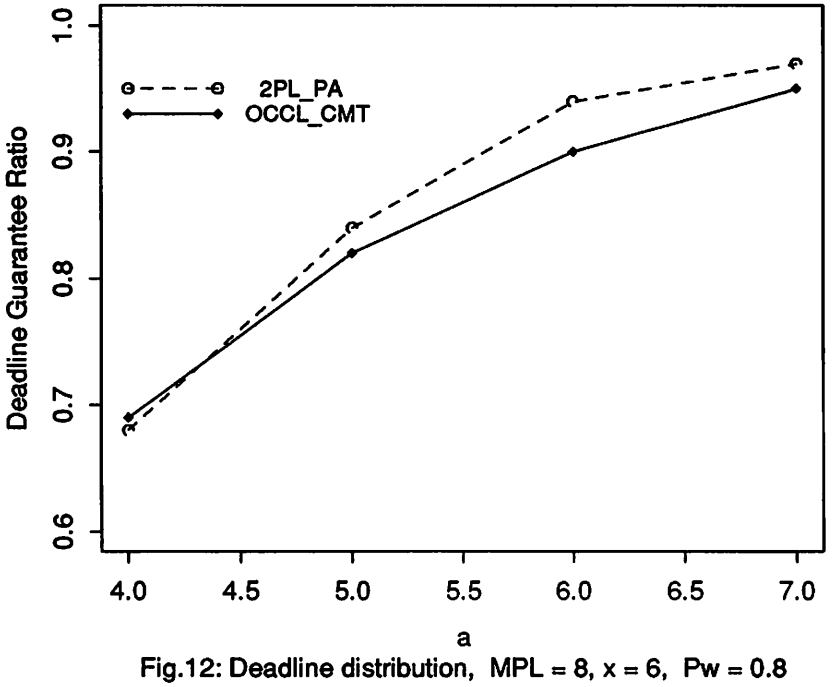

Fig.12: Deadline distribution, MPL = 8, x = 6, Pw = 0.8

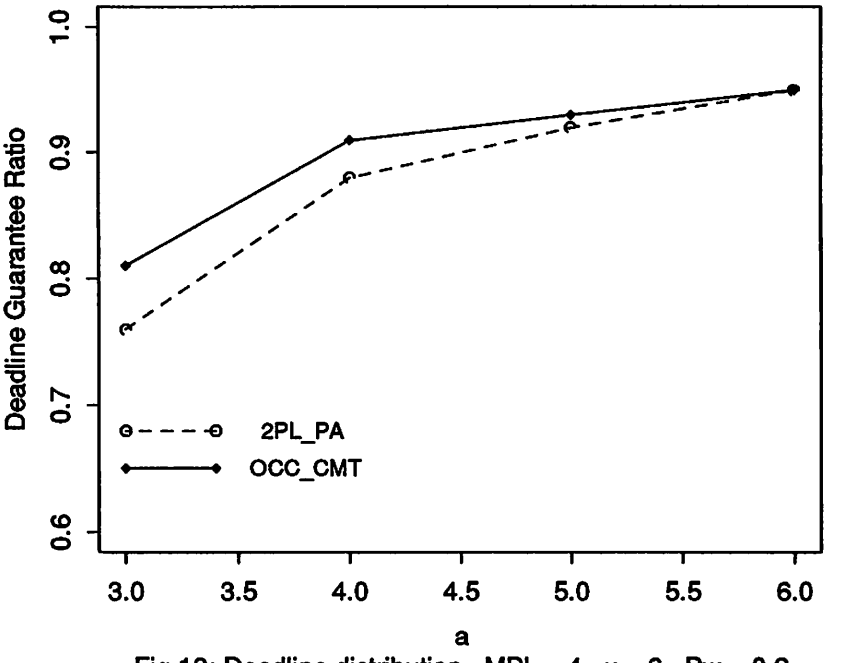

Fig.13: Deadline distribution, MPL = 4, x = 6, Pw = 0.2

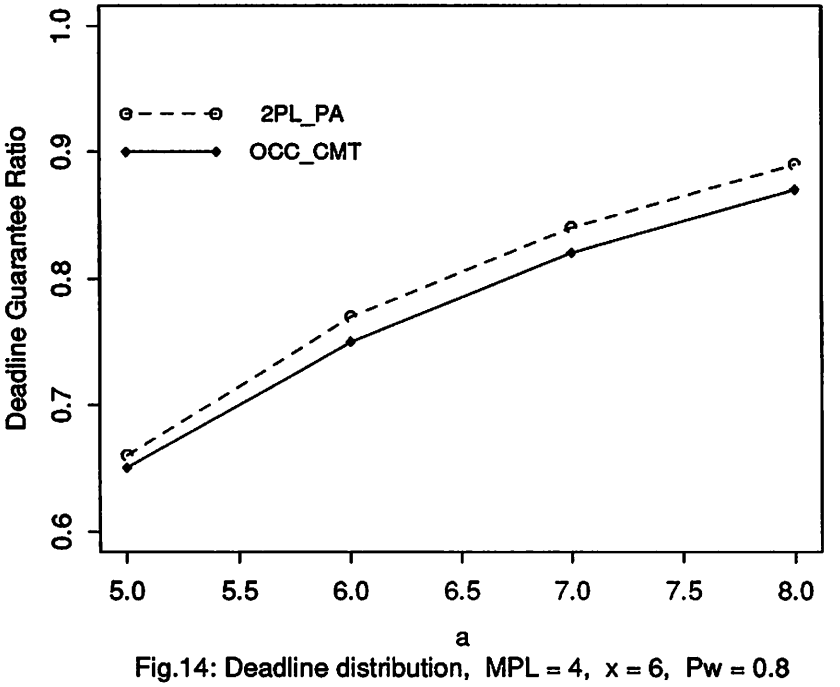

Fig.14: Deadline distribution, MPL = 4, x = 6, Pw = 0.8

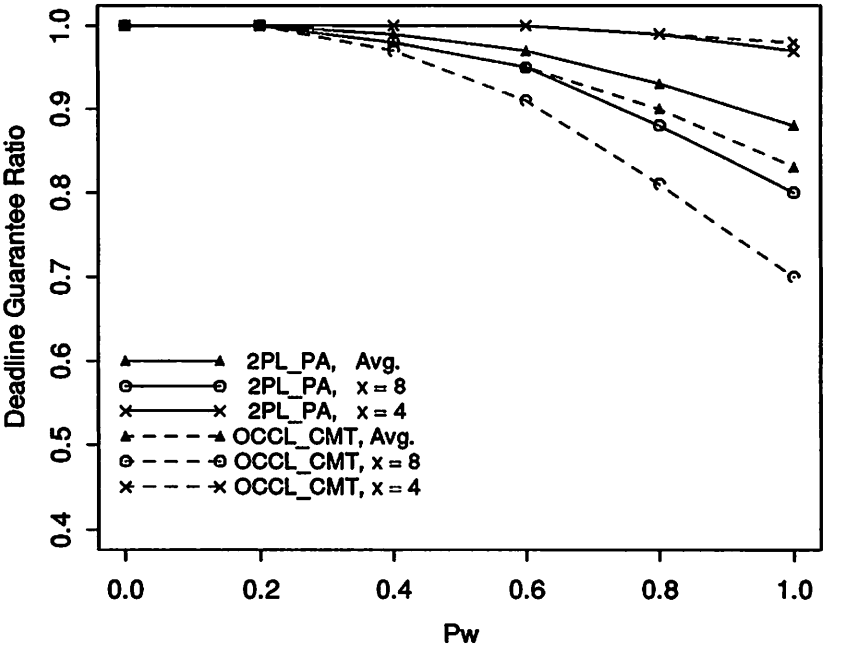

Fig.15: Mixed transactions, MPL = 8, x = [4,8], a = 5

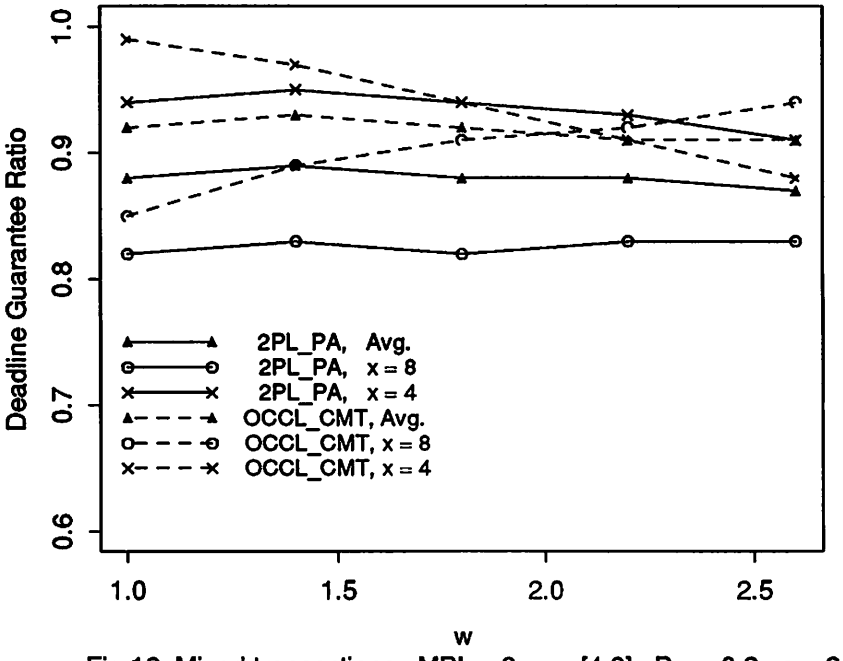

Fig.16: Mixed transactions, MPL = 8, x = [4,8], Pw = 0.2, a = 2