Performance of an Adaptable Synchronization Method

Randi Karlsen and Mohamed A. Hassen

Department of Computer Science,
University of Tromsø, 9037 Tromsø, Norway
randi@cs.uit.no, mohamedh@stud.cs.uit.no

Abstract

This paper presents a performance study of an adaptable synchronization method, called Two-dimensional locking. Two-dimensional locking supports adaptable correctness where different correctness requirements are allowed both between transactions and within a single transaction. This synchronization method is designed for environments executing a mixture of long lasting and short transactions where high degree of concurrency is needed. The performance study compares Two-dimensional locking to Two-phase locking, and shows that Two-dimensional locking represents a vast improvement of both response time and throughput for long lasting transactions. Performance is also improved for short transactions especially when executing under the assumption of infinite resources.

1   Introduction

It is widely recognized that advanced database applications stresses the limits of functionality and performance of traditional transaction management, and that advanced transaction management is needed for these applications [1, 2]. Many advanced applications execute long-lasting transactions, and while Two-phase locking (2PL) is well suited for controlling execution of short transactions, more concurrency is needed when long-lasting transactions are introduced.

To increase concurrency, we use adaptable correctness and a supporting synchronization method called Two-dimensional locking (2Dlock). 2Dlock allows different correctness requirements both between transactions and within a single transaction. A correctness requirement for a transaction T may specify that isolation is required for some parts of T, while a weaker requirement is used for other parts. 2Dlock guarantees correctness for each transaction according to the transaction's specific correctness requirement. Since 2Dlock have the ability to execute transactions under a weak correctness requirement, causing fewer transaction conflicts, non-serializable transaction executions are allowed.

Related research, described in [3, 4, 5, 6, 7, 8], use transaction semantics to decompose transactions into atomic steps for which legal interleaving are explicitly specified. In contrast to the referred work, our approach does not assume any decomposition of transactions. We do on the contrary use transaction semantics to provide early release of locks in flat transactions. This makes it possible to release a lock on an object without delaying this until commit of some step. Another important difference is that we do not assume any detailed knowledge of concurrent transactions. The correctness requirement of a transaction T is determined by considering the needs of T only.

This paper describes 2Dlock and presents a performance study where we compare 2Dlock to 2PL. This study shows that 2Dlock improves performance of transaction executions, especially for long transactions executing under a weak correctness requirement.

The paper is organized as follows. Section 2 describes adaptable correctness, while section 3 presents Two-dimensional locking. Section 4 describes the performance study comparing 2Dlock to 2PL. Section 5 concludes.

2 Adaptable correctness

Transaction executions have traditionally followed the ACID properties, where isolation is the only correctness requirement guaranteed through serializable executions of transactions. However, there are in many applications a need for more concurrency and thereby non-serializable executions. We find that more concurrency is gained by using adaptable correctness, where we allow different correctness requirements both between transactions and within a single transaction. The following two examples show situations where adaptable correctness is useful.

Example 1: Different correctness requirements between transactions.

Assume the following history including operations from transaction T_1 and T_2.

$$H : write_1(x) \prec write_2(x) \prec write_2(y) \prec write_1(y)$$

In history H, transaction T_2 sees a database state which reflects a partial update of x and y. If traditional serializability is required, this history is considered inconsistent and therefore not allowed. Inconsistencies will typically occur if e.g. x and y are both modules of the same program P, and P can not be compiled and run unless both modules are updated. However, if T_2 does not compile and run the program modules, a partial update of x and y need not be considered inconsistent. This is also the situation if x and y are modules of two different programs. If we allow T_2 to succeed after seeing a partial update of x and y, the non-serializable history H represents a consistent execution of T_1 and T_2.

Example 2: Different correctness requirements within a transaction.

Assume a transaction updating a number of program modules.

$$T_3 : Update(A_1), Update(A_2), Update(C_1), Update(D_1)$$

Transaction T_3 updates modules from three different programs, i.e. modules A_1 and A_2 from program A, and modules C_1 and D_1 from program C and D respectively. The correctness requirement for T_3 determines that T_3 must access each program in isolation. This means that modules A_1 and A_2 must be used in isolation, while there are no such requirements between any other modules accessed by T_3.

The need for different correctness requirements is to some extent recognized in SQL-92, where the user may choose between 4 different isolation levels [9]. By using an appropriate isolation level, a transaction may read a partially updated database. However, the flexibility is limited in that SQL does not allow different correctness requirements within a single transaction, and that correctness requirements can not be relaxed for update transactions. Also, a weak isolation level may in SQL cause unwanted side effects such as 'dirty read', and 'unrepeatable read' [10]. These problems are avoided in our approach.

We describe adaptable correctness through *partial isolation*, where the isolation requirement applies to a set of objects called a *unit of isolation*. Specifying an isolation unit U for transaction T, means that T is not allowed to see a partial update of objects in U. A correct execution requires that it appears to T that other transactions executed on U either before or after T.

We distinguish between two different types of isolation units, singleton isolation units and group isolation units. *A singleton isolation unit* includes only one object, and indicates that this object must be used in isolation. We call this correctness requirement *object isolation*. Concurrency control problems like 'lost update', 'dirty read' and 'unrepeatable read' may violate object isolation. Object isolation is a basic and compulsory requirement for each object in the database. Every transaction will therefore automatically require isolation for each object it accesses. *A group isolation unit* includes two or more objects for which mutual consistency requirements exist and

therefore isolation is required. This requirement is called *group isolation*. Group isolation units are, in contrast to singleton isolation units, not compulsory and must be explicitly specified.

An isolation unit is always associated with a transaction. A transaction T may have a number of isolation units, which together describe the correctness requirement for T. Two isolation unit, specified either in different transactions or in the same transaction, may well intersect. As long as isolation is guaranteed for each unit, correctness is also guaranteed. A more detailed description of adaptable correctness and Two-dimensional locking is found in [11, 12].

3   Two-dimensional locking

Two-dimensional locking (2Dlock) is an adaptable synchronization method that allows different correctness requirements both between transactions and within a single transaction. 2Dlock adapts to the set of currently executed transactions by guaranteeing correctness for each transaction according to the transaction's specific correctness requirement. The user can either choose 'isolation' for the whole transaction, or she can specify a weaker, customized correctness requirement for the whole transaction or parts of the transaction. A weaker correctness requirement increases concurrency.

2Dlock presents some novel features which are all described below. Firstly, it supports partial isolation. Secondly, each lock consist of two separate *lock components*, which describes different transaction properties. Both properties are used to determine whether transactions conflict or not. Thirdly, 2Dlock allows early release of locks through the use of *lock switching* and unlock operations.

The costs of using 2Dlock is, at least compared to traditional Two-phase locking (2PL), more complex locking and recovery methods. Also, by allowing the users to specify correctness requirements for the transaction, we make the user (partly) responsible for database consistency and correctness.

Correctness specification
During specification of a transaction T, the user may also specify correctness requirements for T by explicitly determining group isolation units. Singleton isolation units are automatically determined for each object used by T. 2Dlock guarantees isolation for each isolation unit.

We use a new type of construct, $BEGIN\_ISOLATE \ldots END\_ISOLATE$, for specifying group isolation units. This construct encloses an *isolate region*, and every object accessed within this region belong to the same isolation unit. A specification of transaction T_3 from example 2 may be as follows:

$BEGIN\_TRANSACTION,$
$BEGIN\_ISOLATE, Update(A_1), Update(A_2), END\_ISOLATE,$
$Update(C_1), Update(D_1), END\_TRANSACTION$

For this transaction, four singleton units ($\{A_1\}$, $\{A_2\}$, $\{C_1\}$, $\{D_1\}$) and one group isolation unit ($\{A_1, A_2\}$) is identified.

A transaction is characterized with respect to each object it accesses. We say that T is an *Isolate*-transaction with respect to object o if o is included in one of $T's$ group isolation units, otherwise T is an *Interleave*-transaction with respect to o. Transaction T may well be an *Interleave*-transaction on an object that T updates. If two objects, o and o', are not included in the same group isolation unit, the transaction is allowed to execute on a partially updated database with respect to o and o'. This means that inconsistent retrieval is not considered a problem when executing on o and o'.

Lock components and lock stages

A lock set on object o by transaction T, has the form $lock^T(req\_lock, exec\_lock, o)$. This two-dimensional lock includes two separate lock components, i.e. a requirement-lock (*req_lock*) and an execution-lock (*exec_lock*) component, where each component is associated with a transaction property. The req_lock *component* reflects $T's$ correctness requirement on o, by identifying T as either an *isolate-* or an *interleave*-transaction with respect to o. The $exec\_lock$ *component* describes the effect T has on o. As in 2PL, we distinguish between *read-* and *write*-transactions. Both properties are needed to determine whether two operations conflict or not.

The period in which o is locked by T is divided into two lock stages. The *first lock stage* starts when T first obtains a lock on o (before the first access to o), and ends when T completes its last operation on o. Locking o in this period guarantees object isolation. The *second lock stage* starts when the first lock stage ends, and lasts until T executes an unlock operation on o. Locking in the second lock stage guarantees group isolation. A transaction T locks an object during the second stage for two reasons: Firstly, if T is an isolate-transaction, to protect T from seeing a possibly inconsistent state produced by another transaction. Secondly, if T is a write-transactions, to shield the update(s) so that this does not cause inconsistency for other transactions. Since group isolation is optional, there are potentially fewer conflicting operations in the second lock stage. When going *from first to second lock stage*, objects are therefore released to non-conflicting transactions through lock switching and/or unlocking.

	req_lock modes			exec_lock modes	
lock mode	set by	set when	lock mode	set by	set when
int	Interleave-transaction	before use of o	read	Read-transaction	before read of o
isol	Isolate-transaction	before use of o	write	Write-transaction	before write of o
_	both types of transactions	in second lock stage	_	Write-transaction	in second lock stage

Figure 1: Lock modes for the requirement and execution lock component

	(int,read)	(isol,read)	(int,write)	(isol,write)	(isol,-)	(-,-)
(int,read)	yes	yes	no	no	yes	yes
(isol,read)	yes	yes	no	no	no	no
(int,write)	no	no	no	no	no	yes
(isol,write)	no	no	no	no	no	no

Figure 2: Compatibility matrix for Two-dimensional locking

Both the req_lock and the $exec\_lock$ component have three different lock modes (see figure 1). From figure 1 we see that mode *int* is a *req_lock* mode, set by an interleave-transaction before access to o, and it indicates that the transaction requires object isolation only. Mode *isol* is a

req_lock mode, set by an isolate-transaction before access to o, indicating that the transaction requires group isolation. The '-' lock modes are somewhat special. These locks are never used initially, but are used during the second lock stage to indicate that the transaction has completed its execution on o. The possible combinations of lock modes are shown in the compatibility matrix in figure 2. A description of how locks are used during the first and second lock stage is summarized in figure 3.

Locks used in the first lock stage	Changes at the start of second lock stage	Locks used in the second lock stage		
Lock(int,read,o)	——unlock(o)——▶			
Lock(isol,read,o)	(unchanged)	Lock(isol,read,o)		
Lock(int,write,o)	——switching——▶	Lock(-,-,o)		
Lock(isol,write,o)	——switching——▶	Lock(isol,-,o)	——switching——▶	Lock(-,-,o)

Figure 3: Locks used during the first and second lock stage

Lock switching and unlocking

In 2Dlock we seek to allow increased concurrency through early release of locks. Objects are released through both lock switching and unlock operations before transaction commit. *Lock switching* is used in situations where object isolation is already guaranteed, but where group isolation can still be violated. A lock on o can be switched after transaction T has finished the first lock stage on o. Figure 3 shows how locks are switched.

By studying the compatibility matrix in figure 2, we see that lock switching represents a downgrade for interleave-transactions only. Isolate-transactions handle the locks held on o before and after a switch as equivalent locks. An interleave-transaction may be given access to an object after a switch, while the same object can not be accessed by isolate-transactions.

To unlock an object, certain conditions must hold. For isolate-transactions and write-transactions there are different unlock conditions, which are described below. Assume here a transaction T using object o.

1. T is an isolate-transaction for o (i.e. T identifies group isolation unit U and $o \in U$). T can only unlock o after it has obtained a lock on every object in U. This guarantees isolation for U.

2. T is a write-transaction on o (i.e. T updates a set of objects G, where $o \in G$). T can only unlock o after it has obtained a lock on every object in G. This protects the updates so that it does not cause inconsistencies for other transactions.

3. T is neither an isolate-transaction nor a write-transaction on o. T can unlock o after all access to o is completed.

Allowing locks to be release before transaction commit, increases concurrency but can also cause problems in case of transaction abort. Consider history H in example 1 (see chapter 2). Assume a situation where T_2 has committed, while T_1 is still executing. If T_1 now aborts, the recovery mechanism must offer a way of undoing T_1 without affecting T_2. Traditional rollback can not be used in this situation. We investigate using *compensation* [13] as part of the recovery mechanism to avoid cascading abort.

4 Performance study

In this chapter we present the results of a simulation study comparing Two-dimensional locking (2Dlock) to Two-phase locking (2PL). Several interesting performance studies are found in the literature, e.g. in [14]. The most relevant to our work is [15]. Our study follow the same approach as used in [15], with a similar performance model and parameter settings.

4.1 Simulating 2Dlock and 2PL

This simulation study compares performance of 2Dlock to the well known Two-phase locking (2PL) method [16]. 2PL is a much used synchronization method, guaranteeing correct executions by producing serializable executions of transactions. 2PL follows a strong correctness criteria where the same correctness requirement (i.e. isolation) applies to every transaction.

Different design decisions can be made when implementing a synchronization method. For both 2Dlock and 2PL, we have chosen an implementation using incremental locking, immediate restart in case of conflicts, and simultaneous unlocking of objects at transaction commit time. This means that for both 2Dlock and 2PL, transactions read-lock an object before a read operation and may later upgrade to a write-lock before an update. If a lock request is denied, the requesting transaction is aborted and restarted after a restart delay. By not allowing conflicting transactions to wait, deadlock is avoided. For 2Dlock we implement early release of locks by allowing lock switching before transaction commit. Lock switching is executed according to the rules in section 3.

While working with 2Dlock we have distinguish between two types of adaptable correctness. Single transaction correctness where one correctness requirement is used for the whole transaction, and mixed correctness where different correctness requirements may be used within a single transaction. 2Dlock has evolved from being a synchronization method supporting single transaction adaptable correctness to supporting mixed adaptable correctness. The following performance study uses a version of 2Dlock that supports single transaction adaptable correctness.

4.2 Parameters and performance metrics

Table 1 presents the simulation parameters that applies to all experiments. Parameters that vary from experiment to experiment are given with the description of the relevant experiment.

Parameter	Meaning	Values
DB_SIZE	Number of objects in the database	1000 granules
SHORT_TRAN_SIZE	Mean size of a short transaction	8 granules
MIN_SIZE_SHORT	Size of smallest short transaction	4 granules
MAX_SIZE_SHORT	Size of largest short transaction	12 granules
LONG_TRAN_SIZE	Mean size of a long transaction	31 granules
MIN_SIZE_LONG	Size of smallest long transaction	13 granules
MAX_SIZE_LONG	Size of largest long transaction	50 granules
WRITE_PROB	Probability of a write operation	0.25
NUM_OF_TERMINALS	Number of terminals	200 terminals
MPL	Multiprogramming level	5, 10, 25, 50, 75, 100, and 200
RESTART_DELAY	Mean transaction restart delay	Adaptive
EXT_THINK_TIME	Mean time between transactions	1 second
OBJ_IO	I/O time for accessing an object	35 milliseconds
OBJ_CPU	CPU time for accessing an object	15 milliseconds

Table 1: Simulator parameter settings.

The unit of the database locked by the concurrency control mechanism is called a "granule", where the size of the granule can vary from system to system. To avoid unnecessary long simulation

time we have chosen a database size of 1000 granules to create an environment where conflict probability is very high.

The first experiment simulates the execution of short transaction only, while the next two experiments mixes long and short transactions. We model transaction size according to the number of items that a transaction reads and writes. A short transaction will normally access less than 2% of the database (4 - 12 granules), while long transactions will access up to 5% (13 - 50 granules). These experiments are motivated by the desire to investigate how transaction throughput and response time is effected as one runs long and short transactions in the same environment.

The two main performance metrics used throughout this paper are transaction throughput rate and transaction response time. *Transaction throughput* is the number of transactions completed per simulation time (one second is assumed to be one simulation time). *Transaction response time* is the elapsed time (measured in seconds) between transaction initiation and completion, including time spent waiting in the ready queue and restart delay. One additional performance related metric which are used in analyzing the results of our experiments is restart ratio. *Restart ratio* gives the average number of times a transaction has to restart per commit.

In the first two experiments we assume a system with one hardware resource (containing one CPU and two disks). In these experiments both data contention (conflict probabilities) and resource contention (waiting for CPU and disk) will limit the transaction throughput and response time. To investigate the level of concurrency of the two algorithms, we assume a system with infinite resources in our third experiment. In this experiment the resource contention problem is ignored and the only factor limiting transaction throughput and response time is data contention.

In the first experiment, only short transactions are executed. For 2Dlock, 100 terminals generate short interleave-transactions, while the remaining 100 terminals generate short isolate-transactions. In the next two experiments, 100 terminals generate long transactions while the other 100 terminals generate short transactions. The long transactions are in 2Dlock always executed as interleave-transactions, while short transactions are executed as isolate-transactions. For 2PL, all transactions are executed with the strong isolation requirement. In all experiments, the number of concurrent transactions, called multiprogramming level, is varied from 5 to 200.

4.3 Experiment 1: Short transactions, one resource unit

In this experiment we analyze the impact of limited resources on the performance of the two concurrency control algorithms. We assume a database system with one resource unit, executing short transactions only. In the case of 2Dlock, 100 terminals generate interleave-transactions, while the other 100 terminals generate isolate-transactions.
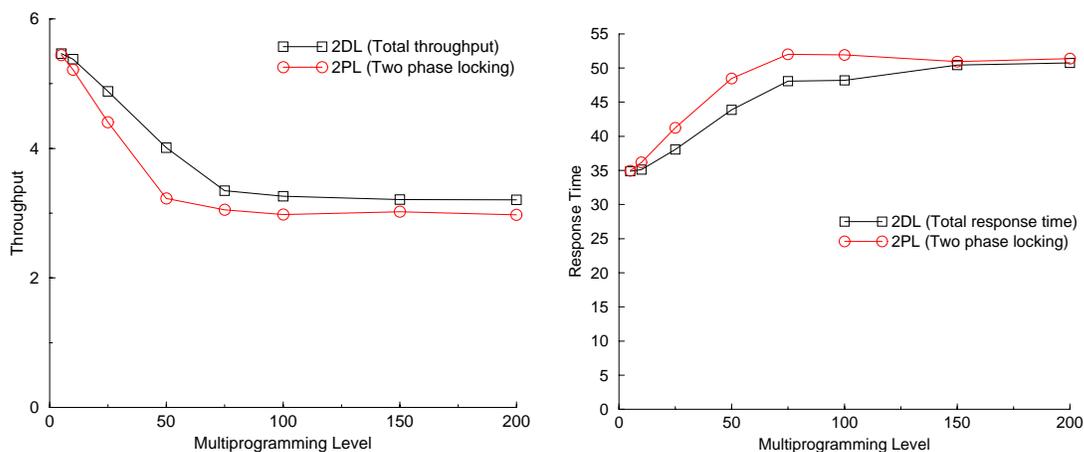

Figure 4: Transaction Throughput and Response Time (experiment 1)

The throughput and response time results are presented in Figure 4. Note that we for 2Dlock we do not distinguish between interleave- and isolate-transactions. For both throughput and response time, 2Dlock performs slightly better than 2PL under all multiprogramming levels.

Observe for both algorithms, as the multiprogramming level increase, the throughput first decreases and then remains roughly constant. Similarly, the response time for both algorithms increases and then remains roughly the same. This happens for the following reason: When the multiprogramming level increases data contention (conflict probabilities) increases as well. This has the effect of increasing the number of restarted and delayed transactions, which in turn increases the response time.

The results obtained for 2PL, i.e. both throughput and response time, corresponds well with results of a similar experiment presented in [15].

4.4 Experiment 2: Long and short transactions, one resource unit

This experiments is motivated by the desire to investigate how transaction throughput and response time are effected as one runs long and short transactions in the same environment. Now half of the transactions in both 2PL and 2Dlock are long transactions (i.e. 100 terminals are generating long transactions). All other parameters are the same as in our previous experiment. Throughput results for this experiment is presented in Figure 5, long transactions on the left and short transactions on the right.
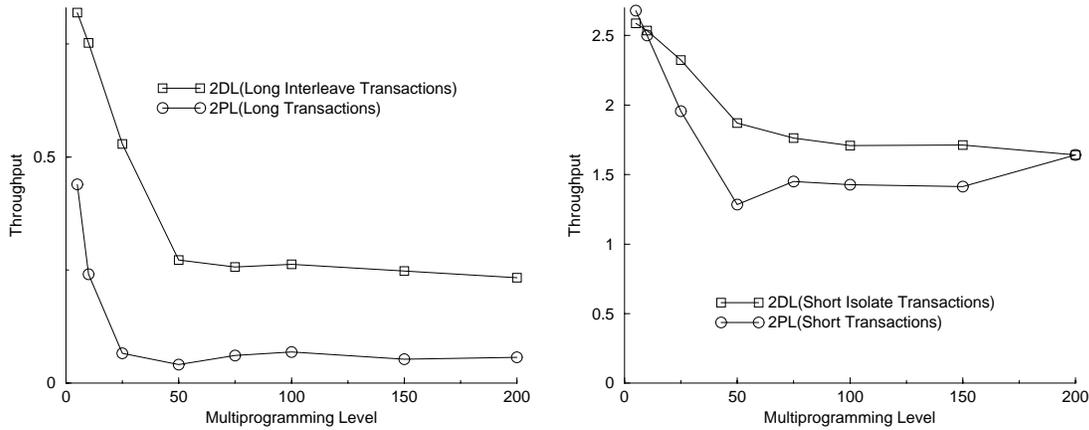

Figure 5: Transaction Throughput (experiment 2)

Figure 5 shows that 2Dlock provide higher throughput for both transaction categories. In 2Dlock long transactions has weak interleaving constraint, and will therefore not be restarted as often as long transactions executing under 2PL (see Figure 6).
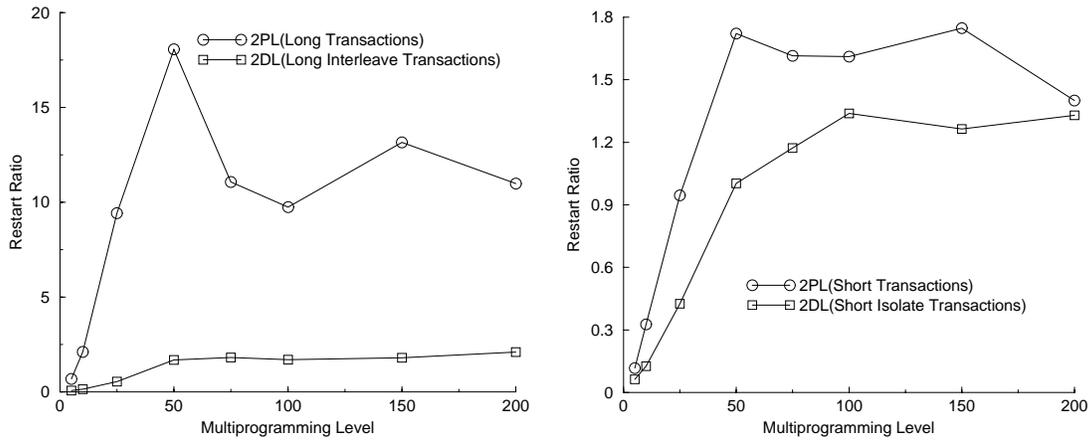
Figure 6: Restart Ratio (experiment 2)

Response time results for this experiment are presented in Figure 7, and shows that 2Dlock has lower response time than 2PL. For long transactions 2Dlock is a clear winner. Studying the response time for short transactions we see that the difference is very little and both algorithms perform roughly the same. Given the fact that both algorithms execute short transactions in a serializable fashion, we could make the general conclusion that 2Dlock will always perform the same as 2PL for the isolate (serializable) transactions, regardless of the number and size of concurrent transactions. However, we mean that this is not the case because according to Figure 6 there are more active transactions (competing for resource) in 2Dlock than 2PL at any multiprogramming level because of the frequent restarts in 2PL. This means that in 2Dlock each transaction has to wait for hardware resources for longer periods of time than their counter part in 2PL. This has the effect of increasing the transaction response time.
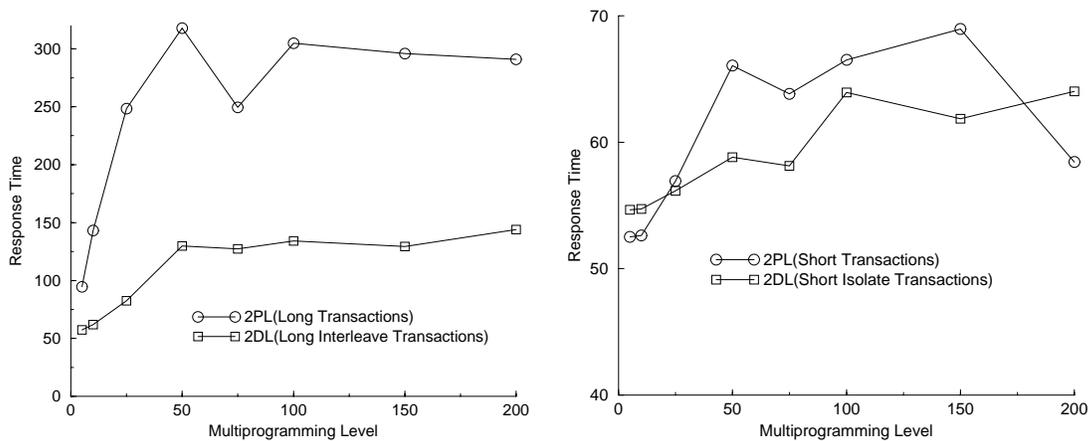

Figure 7: Transaction Response Time (experiment 2)

We conclude that 2Dlock provides higher concurrency among transactions than 2PL. But, that this higher concurrency creates a negative feedback (increase the response time and limit the throughput) because of the hardware resource contention problem. In our next experiment we have eliminated the resource contention problem to see how the performance are effected.

4.5 Experiment 3: Long and short transactions, Infinite resources

This experiment is motivated by the desire to investigate how transaction response time and throughput are effected when the resource contention problem is ignored and only data contention (conflict rate) is considered. To model a system with infinite number of resources, we have increased the number of hardware resource to equal the multiprogramming level. Each active transaction will now have one resource of it's own, and will therefore never have to wait for resources. All other parameters remain the same as in the previous experiment. Throughput results for this experiment are presented in Figure 8.
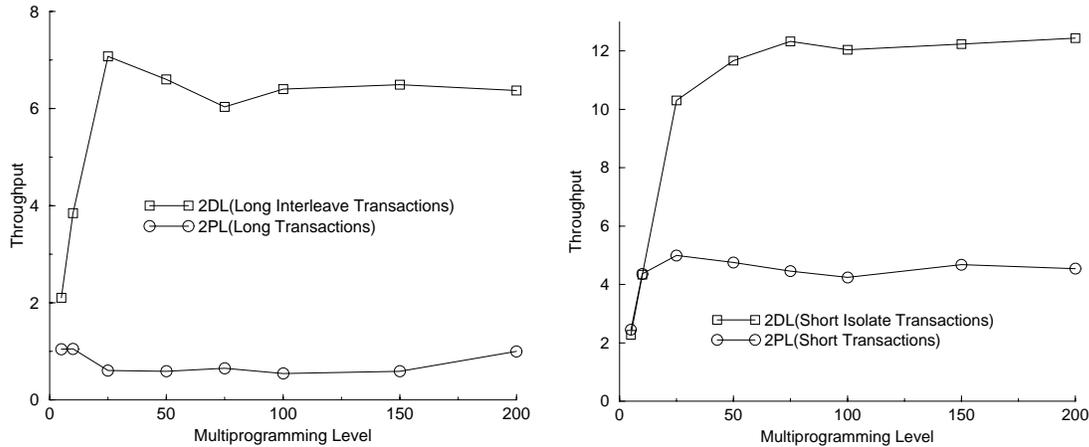

Figure 8: Throughput (experiment 3)

In this experiment, 2Dlock has higher throughput than 2PL at all multiprogramming level, and for both transaction categories. Figure 9 shows the transaction response time for this experiment, where 2Dlock has lower response time than 2PL for both transaction categories. The length of duration of long lasting transaction causes serious performance problems for the short transactions in 2PL. This happens because long transactions are allowed to lock objects until they commit (which increases the transaction restarts). In 2Dlock, long transactions release the locks as early as possible, and provide therefore higher degree of concurrency. Figure 10 present the restarts for this experiment, and it shows that 2Dlock has lower transaction restarts than 2PL for both long and short transactions.
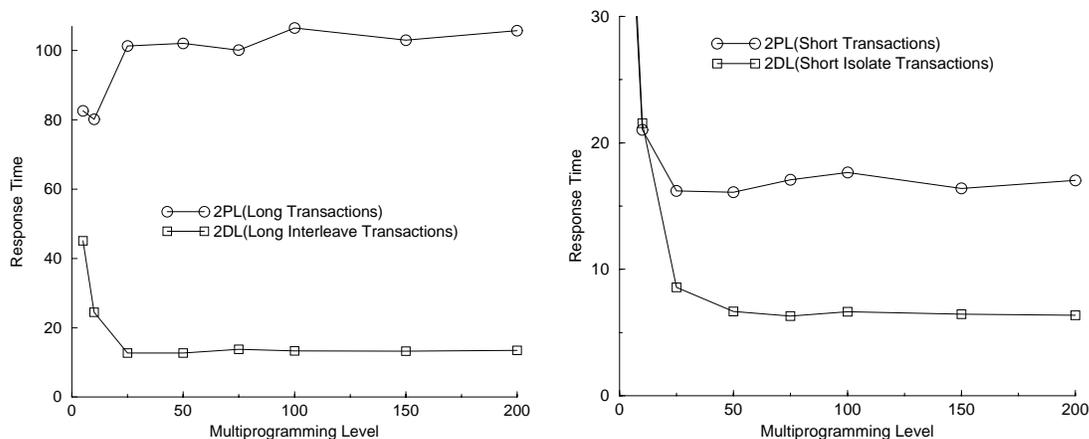

Figure 9: Transaction Response Time (experiment 3)

According to throughput and response time results, restart ratio and also given the fact that conflict is the only factor that can limit the transaction throughput and response time, we can make the general conclusion that 2Dlock provides higher degree of concurrency than 2PL.
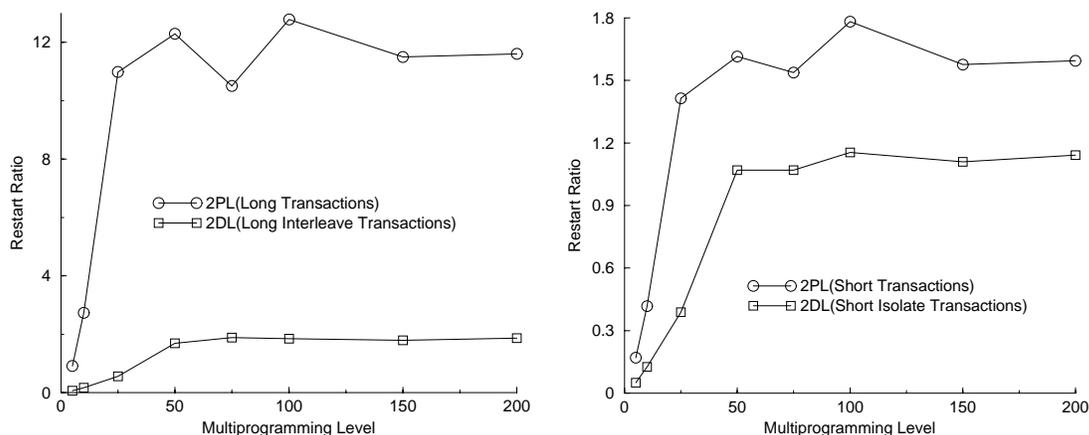

Figure 10: Transaction Restart Ratio (experiment 3)

5 Conclusions

We have in this paper presented a performance study of Two-dimensional locking. 2Dlock is an adaptable synchronization method allowing a mixture of weak and strong correctness requirements both between transactions and within a single transaction. The goal of 2Dlock is to increase concurrency by allowing the use of weak correctness requirements.

The performance study compare 2Dlock to 2PL. Performance of 2Dlock represents a vast improvement of both response time and throughput for long transactions. Performance is also improved for short transactions especially when executing under the assumption of infinite resources. The performance study shows that 2Dlock is well suited for environments executing long transactions if these transactions can be executed under a weak correctness requirement. The user will experience a considerable performance improvement for long transaction, while the performance of short transactions will be as good as or slightly better then 2PL.

References

[1] Elmagarmid, A.K. (ed.), Database Transaction Models for Advanced Applications, Morgan Kaufmann, 1992.

[2] Jajodia, R., Kerschberg, L. (eds.), Advanced Transaction Models and Architectures, Kluwer Academic Publishers, 1997.

[3] Bernstein, A.J., Lewis, P.M., Transaction Decomposition Using Transaction Semantics, Distributed and Parallel Databases, vol. 4, no. 1, 1996.

[4] Garcia-Molina, H., Using Semantic Knowledge for Transaction Processing in a Distributed Database, ACM Transactions on Database Systems, vol. 8, no. 2, June 1983.

[5] Lynch, N., Multilevel Atomicity - A New Correctness Criterion for Database Concurrency Control, ACM Transactions on Database Systems, Vol. 8, No. 4, 1983.

[6] Farrag, A.A., Ozsu, M.T., Using Semantic Knowledge of Transactions to Increase Concurrency, ACM Transactions on Database Systems, Vol. 14, No. 4, 1989.

[7] Shasha, D., Simon, E., Valduriez, P., Simple Rational Guidance for Chopping Up Transactions, Proc. of the 1992 ACM SIGMOD Int. Conf. on Management of Data, San Diego, California, June 2-5, 1992.

[8] Ammann, P., Jajodia, S., Ray, I., Semantic-Based Decomposition of Transactions, in Advanced Transaction Models and Architectures, R. Jajodia and L. Kerschberg (eds.), Kluwer Academic Publishers, 1997.

[9] Melton, J., Simon, A.R., Understanding the new SQL: A Complete Guide, Morgan Kaufmann, 1993.

[10] Gray, J., Reuter, A., Transaction Processing: Concepts and Techniques, Morgan Kaufmann Publishers, 1993.

[11] Karlsen, R., Adaptable Correctness through Two-dimensional locking, Proc. of the 8th Int. Workshop on Database and Expert Systems Application (DEXA), Toulouse, France, September 1-2, 1997.

[12] Karlsen, R., Supporting Partial Isolation in Flat Transactions, in Lecture Notes on Computer Science, LNCS 1677, Springer, 1999.

[13] Korth, H.F., Levy, E., Silberschatz, A., A Formal Approach to Recovery by Compensating Transactions, Proc. of the 16th VLDB Conf., Brisbane, Australia, 1990.

[14] Kumar, V. (ed.), Performance of Concurrency Control Mechanisms in Centralized Database Systems, Prentice Hall, 1996.

[15] Agrawal, R., Carey, M.J., Livny, M., Concurrency Control Performance Modeling: Alternatives and Implications, in Performance of Concurrency Control Mechanisms in Centralized Database Systems, V. Kumar (ed.), Prentice Hall, 1996.

[16] Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L., The Notion of Consistency and Predicate Locks in a Database System, Communication of the ACM, vol. 19, no. 11, Nov. 1976.