Performance Analysis of Concurrency Control Using Locking with Conditional Blocking

ABSTRACT

There is growing evidence that for a wide variety of database workloads and system configurations the two-phase locking (2PL) outperforms other types of concurrency control schemes. However, in the presence of long-lived transactions (LLTs), 2PL surrenders to a problem of long delay suspension because LLTs are qualified to lock data until they commit. To alleviate this problem we propose an extension to 2PL, named *conditional blocking* (CBL), that reduces the waiting probability on the basis of cycle detection to determine if conflicting lock modes could be held on the same data. Unlike the traditional serialization graph testing (SGT) that is also on the basis of cycle detection, CBL could delay a conflicting lock request even though the resulting graph does not contain cycles and thus it can reduce the number of transaction aborts. By properly balancing the waiting effect and the abort effect, CBL can lead to better performance than 2PL and SGT. Using a centralized database simulation model, we demonstrate that CBL exhibits substantial performance improvement over 2PL and SGT when LLTs are present.

Keywords: transaction processing, long-lived applications, concurrency control, conditional blocking, performance evaluation

1 Introduction

A *long-lived transaction* (LLT) has a long duration compared to most other transactions because it accesses many database objects, it has lengthy computations, it pauses for inputs from users, or because of a combination of these factors [1,2]. In many cases LLTs present serious performance problems. Two-phase locking (2PL) is the *de facto* standard concurrency control scheme utilized by transaction processing systems. When transactions are of long duration, 2PL requires that locks be retained for long periods. (Commonly, locks are held until the transaction finishes.) Other transactions wishing to access the locked data could face substantial delays. The problem is exacerbated if LLTs access a lot of data. LLTs are also more likely to be involved in deadlocks, resulting in transaction failures.

A number of 2PL extensions have been proposed to support LLTs efficiently [3-8]. The common approach used is to incorporate *application-specific semantics* into the transaction management. The previous 2PL extensions can be classified into three groups according to the types of application-specific semantics they use. The first group requires to predeclare the data that an LLT intends to read or write, and makes use of this information to release locks as early as possible. Altruistic locking [3] and five color concurrency control [4] are included in this group. The second group restructures LLTs into a collection of subtransactions that could be interleaved in any way with other transactions. This group includes sagas [5] and cooperating CAD transaction [6]. In the last group, users are responsible for preserving the consistency of LLTs. Notification-based transaction model [7] and group-oriented CAD transaction model [8] are included in this group.

The major problem of the previous 2PL extensions is that they can be applied to only specialized applications where their semantics are well understood. In other words, each extension is well suited for a special class of applications, but none of them seems to be suitable and applicable for all kinds of long-lived applications by itself. For example, predeclaration approaches revert back to the standard 2PL, if actions of LLTs are unforeseeable at the beginning, that may be usual in design applications. Transaction restructuring approaches also have a limitation that it could behave problematically in performance sense for applications where subtransactions are still long-lived. Note that in most design applications even a simple task tends to take days or weeks, not seconds.

Concurrency control schemes which incorporate user interactions assume that the user understands his/her applications perfectly. However, human beings tend to be error-prone, which could result in inconsistent data. Furthermore, in some design applications like a complex VLSI design project, the structure of a design activity is usually too complex for a user to understand the details of his/her design activity.

In this paper we propose a new locking scheme, named *conditional blocking* (CBL), that improves 2PL when LLTs are present. Unlike the previous 2PL extensions, CBL does not depend on any application-specific semantics. The basic idea of CBL is to allow conflicting lock modes to be held on the same data on the basis of cycle detection. For example, two transactions could hold write locks on a data, if cycles are not introduced. In order to preserve dependencies between transactions, CBL maintains a *transaction dependency graph* (TDG). TDG is a directed graph whose nodes are transactions and whose labeled edges represent *commit* dependency, *abort* dependency, or *wait-for* dependency between transactions. CBL attains serializable executions by ensuring the TDG always remains acyclic.

Unlike the traditional serialization graph testing (SGT) that is also on the basis of cycle detection, CBL may delay a conflicting lock request even though the resulting TDG does not contain cycles. This is because (1) unconstrained accesses to the already locked data might result in cascading aborts and (2) the frequency of cascading aborts increases as the graph contains more edges. Note that SGT accepts every conflicting lock request as far as the resulting graph does not contain cycles. In order to determine whether a conflicting lock request could be accepted, CBL uses the notion of *dependency depth* that represents a path length of TDG. If accepting a lock request would make the dependency depth exceed to some predefined nonnegative integer $\boldsymbol{d}$, CBL delays the lock request; otherwise, CBL accepts the request. Note that $\boldsymbol{d}$ is 0 in case of the standard 2PL, that is, a transaction encountering a lock conflict is blocked and thus cascading aborts are never introduced. On the other hand, $\boldsymbol{d}$ is ∞ for SGT, that is, a transaction is allowed to execute without being blocked; this could raise the probability of cascading aborts significantly. By properly restricting the value of $\boldsymbol{d}$, CBL can lead to better performance than 2PL and SGT.

In addition to the advantage of application independence, CBL has the following salient properties. First, CBL is *complementary* to some previous 2PL extensions. For example, in saga or cooperating CAD transactions, CBL can be applied to coordinate the

subtransactions. This is particularly desirable when the subtransactions are of long duration, too. Next, as we will show, CBL exhibits substantial performance improvements over 2PL even when every transaction is of short duration. In particular, the performance of CBL improves as resources (CPU and disks) become more plentiful. This is because CBL is more permissive and thus introducing additional resources improves the performance of CBL while it does not benefit 2PL significantly.

The rest of this paper is organized as follows. In Section 2, we present the database model with the notion of TDG. In Section 3, we give a detailed description of CBL. Section 4 describes the simulation models used to evaluate the performance of CBL. The simulation results are discussed in Section 5. Section 6 has concluding remarks.

2 Database Model

A database is a collection of *data*. Users interact with the database by invoking *transactions*. A transaction is an ordered sequence of read and write operations that are executed atomically on the data. A read (write) operation executed by a transaction T_i on data x is denoted as $R_i[x]$ ($W_i[x]$). The execution of transactions is modeled by a structure called a *history*. Formally, a history H is a partial order (Σ, $<_H$) where Σ is the set of all operations executed by transactions, and $<_H$ reflects the order in which the operations were executed. The *serialization graph* for H, denoted SG(H), is a directed graph whose nodes are transactions, and has edges $T_i \rightarrow T_j$ if one of T_i's operation precedes and conflicts with one of T_j's operations. A history H is serializable if and only if SG(H) is acyclic [9].

Transaction dependencies provide a convenient way of specifying and reasoning about the behaviour of concurrent transactions [1, 2]. In order to capture the various types of transaction dependency, we develop the notion of transaction dependency graph. A transaction dependency graph (TDG) differs from the serialization graph in three ways. First, the edges in TDG are labeled to represent the transaction dependency. Second, TDG may not include nodes corresponding to all committed transactions, in particular those that have committed long time ago. Last, TDG usually includes nodes for all active transactions, which by definition are not yet committed. A TDG is formally defined as follows:

Definition 1 (Transaction Dependency Graph): Given a history H, a *transaction dependency graph* TDG(H) = ($\boldsymbol{V}$, $\boldsymbol{E}$) is the directed graph defined by

- $\boldsymbol{V} = \{T_1, T_2, \ldots, T_n\}$ is a set of transactions in H
- $\boldsymbol{E}$ = {*Commit edges*} ∪ {*Abort edges*} ∪ {*Block edges*}

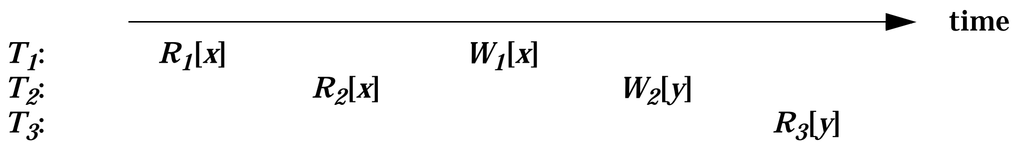
(a) *Commit edge* (‘$\rightarrow_C$’) represents *commit dependency* of two transactions. $T_i \rightarrow_C T_j$ means T_j cannot commit until T_i either commits or aborts. $T_i \rightarrow_C T_j$ iff (1) $R_i[x] <_H W_j[x]$ or $W_i[x] <_H W_j[x]$ and (2) $W_j[x]$ can be executed.

(b) *Abort edge* (‘$\rightarrow_A$’) represents *abort dependency* of two transactions. $T_i \rightarrow_A T_j$ means T_j should abort if T_i aborts. $T_i \rightarrow_A T_j$ iff $W_i[x] <_H R_j[x]$ and $R_j[x]$ can be executed.

(c) *Block edge* (‘$\rightarrow_B$’) represents *wait-for dependency* of two transactions. $T_i \rightarrow_B T_j$ means T_j should be blocked until T_i either commits or aborts. $T_i \rightarrow_B T_j$ iff there are two conflicting operations op_i and op_j such that $op_i <_H op_j$ and op_j cannot be executed. p

Commit edges and abort edges impose a commit order that prevents a transaction from prematurely committing, thereby preventing data inconsistencies. Abort edges additionally imply the transactions that have to be involved in cascading aborts due to reading uncommitted data. Block edges are used to detect deadlocks that happen to be occurred in the pessimistic concurrency control schemes like 2PL. The notion of the TDG can be applied to a number of concurrency control schemes. This is illustrated in Example 1.

Example 1 (Usage of TDG): Consider the following scenario of transaction executions.



A TDG produced by a 2PL scheduler is shown in Figure 1(a); in this case TDG includes only block edges. This is because every transaction encountering a lock conflict is blocked. For example, $W_1[x]$ is blocked due to $R_2[x]$, and $R_3[y]$ is blocked due to $W_2[y]$. In this case, TDG turns to the conventional wait-for graph. A TDG produced by an SGT scheduler is shown in Figure 1(b); in this case TDG does not include block edges, instead it includes only one commit edge and one abort edge. This is due to the fact that every transaction can execute without being blocked. n

Figure 1 TDGs in 2PL and serialization graph testing

With the notion of TDG, we are able to represent full spectrum of the degree of concurrency between 2PL and SGT. In other words, a block edge produced in 2PL may be transformed into a commit edge or abort edge in more concurrent schemes. This is achieved by allowing conflict lock modes to be held on the same data. For example, in Figure 1, a block edge $T_2 \rightarrow_B T_1$ of 2PL may be transformed into a commit edge $T_2 \rightarrow_C T_1$ if we allow the execution of $W_1[x]$ even though $R_2[x]$ has been executed. In this case, specifying when a block edge could be transformed into another type of edge is a key to determine the possible concurrency control schemes. We will use the notion of the dependency depth to determine the possible schemes and this is the major contents of CBL. In this section, we first define the notion of dependency depth.

Definition 2 (Dependency Depth): Given two transactions T_0 and T_k, a *dependency path* DP(T_0, T_k) in TDG is a set of finite non-null sequences, $T_0\ E_1\ T_1\ E_2\ T_2 \ldots E_k\ T_k$, each of which satisfies the following properties: (1) terms are alternately transactions and edges, such that, for $1 \leq i \leq k$, E_i implies $T_{i-1} \rightarrow_C T_i$, $T_{i-1} \rightarrow_A T_i$, or $T_{i-1} \rightarrow_B T_i$, (2) each edge is distinct, and (3) each transaction is distinct. The *length* of DP(T_0, T_k) is the maximum number of edges for every sequence in DP(T_0, T_k). A *dependency depth* DD(T_k) is the maximum length for every DP(T_i, T_k) that exists in TDG. p

For example, consider a TDG illustrated in Figure 2. DP(T_1, T_6) consists of three sequences $T_1 \rightarrow_A T_2 \rightarrow_B T_3 \rightarrow_A T_6$, $T_1 \rightarrow_C T_4 \rightarrow_B T_6$, and $T_1 \rightarrow_C T_4 \rightarrow_B T_5 \rightarrow_C T_6$. Then, the length of DP(T_1, T_6) is 3. Because the length of DP(T_1, T_6) is the largest one among those of other paths to T_6, DD(T_6) is 3.

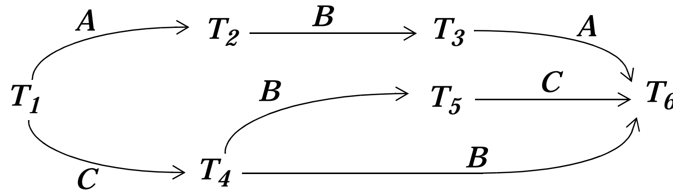


Figure 2 A sample TDG

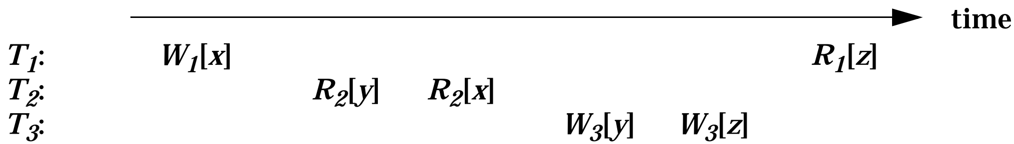
3 Conditional Blocking

3.1 Basic idea

As we have noted in Example 1, TDGs produced by 2PL include only block edges. This implies that long waiting may be incurred due to the interferences of LLTs. On the other hand, TDGs produced by SGT include only commit edges and abort edges. So SGT is able to prevent long waiting. However, in SGT, LLTs increase the likelihood of cascading aborts through abort dependency.

The basic idea of CBL is to properly balance the waiting effect and the abort effect for the purpose of alleviating the problem of long waiting with fewer aborts of LLTs. This is achieved by limiting the dependency depth of every transaction in TDG into a some nonnegative integer. Suppose that the dependency depth is limited to d. A transaction T_i that requests a conflicting lock is blocked if accepting the lock request would make DD(T_i) > d. Otherwise, the lock request is accepted and T_i is allowed to execute. Therefore, depending on the value of d, the waiting effect and the abort effect can be balanced. This is illustrated in Example 2.

Example 2 (Effects of Dependency Depth): Consider the following scenario of transaction executions.



A TDG in which d is limited to 0 is shown in Figure 3(a). This corresponds to the standard 2PL. In this case, T_2 and T_3 must wait until T_1 terminates. Figure 3(b) shows the case when d is limited to 1. Note that T_2 is allowed to execute $R_2[x]$ because in this case DD(T_2) becomes 1, which is the value of d. However, T_3 again is not allowed to execute $W_3[y]$ because accepting $W_3[y]$ makes DD(T_3) to 2, which is greater than d. Figure 3(c) shows the case when d is limited to 2. T_3 is now allowed to execute $W_3[y]$; however, in this case, a dependency cycle is created and some transaction must be aborted to resolve the cycle. n

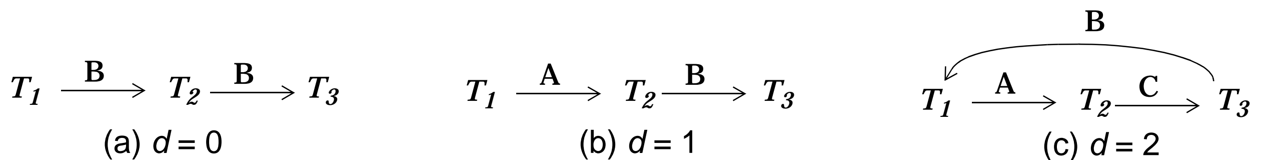


Figure 3 TDGs for different dependency depths

3.2 The locking protocol

We present a class of concurrency control schemes CBL(d) in which the dependency depth is limited to d, where d is a nonnegative integer. CBL(d) is an extension to 2PL, hence it follows the same rules for acquisition and relinquishing of locks, i.e., a transaction may not acquire a lock on any data once it has relinquished at least one lock. For each active transaction T_x, CBL(d) scheduler maintains DD(T_x) that represents the current dependency depth of T_x. It also maintains a TDG to represent transaction dependencies.

When a CBL(d) scheduler receives an operation $p_i[x]$ from a transaction manager, it first adds a node for T_i in its TDG, if such a node does not exist. Then it adds an edge from T_j to T_i for previously scheduled operations $q_j[x]$ that conflicts with $p_i[x]$, if such T_j exists. At this time, if DD(T_j) is equal to or greater than DD(T_i), DD(T_i) is set to DD(T_j) + 1. The label of each edge can be determined as follows.

- If DD(T_i) becomes larger than d, then every edge due to $p_i[x]$ becomes block edge.
- Otherwise the edge becomes an abort edge if $q_j[x]$ is write operation and $p_i[x]$ is read operation. On the other cases, the edge becomes a commit edge.

When every edge is added to TDG, we have two cases:

1. The resulting TDG contains cycles. This means that scheduling $p_i[x]$ now would make the resulting execution be non-serializable. Therefore, one or more transactions that involved in the cycles should be aborted. The victim selection strategies will be described in Section 3.3. Suppose a transaction T_V is selected as a victim. Then, the scheduler deletes T_V and all edges incident to T_V from TDG. Since TDG is acyclic now, the execution produced by the scheduler is serializable.
2. The resulting TDG is still acyclic. In this case, if $p_i[x]$ does not introduce block edges, the scheduler accepts $p_i[x]$ and registers the lock acquired by $p_i[x]$ to the lock table. Otherwise, T_i is blocked until every block edge due to $p_i[x]$ is deleted from TDG.

In CBL(d), when a transaction T_i requests a data that is found already locked by another transaction T_j, T_i checks to see if DD(T_j) ≥ d. If DD(T_j) is less than d, T_i is allowed to access the data. Thereafter T_i can continue its execution even if DD(T_j) later becomes equal to or greater than d. In other words, a later change of DD(T_j) is not propagated to

DD(T_i). This means that a transaction can only be blocked at the time it requests a lock; after a transaction enters an execution state, it will not transition directly to being blocked. The implication of this policy is to reduce the overhead of maintaining dependency depth. Note that the cost of propagating changes on dependency depths is overly expensive. We use the notion of dependency depth just as a *measure to categorize* possible schemes of which concurrency range from 2PL to SGT. Hence, the value of DD(T_i) that CBL(d) scheduler maintains does not need to reflect an exact value it really is on TDG.

Now, we describe the commit procedure and the abort procedure of CBL(d). A transaction can commit if it has no incoming edges in TDG. A transaction should abort if it is involved in a cycle of the TDG and is selected as a victim. When a transaction T_i commits or aborts, CBL(d) scheduler deletes its node and edges from TDG. At this time, for each transaction T_x where $T_i \rightarrow T_x$ exists, DD(T_x) is reset to 0. The new value of DD(T_x) may not be correct, since T_x could depend on other transactions. However setting DD(T_x) to exact value requires traversing TDG to find all the predecessors of T_x and to compare their dependency depths. As we described previously, since the value of DD(T_x) need not be exact, CBL(d) does not cost the additional overhead of traversing TDG. As a result, the transaction termination overhead of CBL(d) is about equal to the standard 2PL.

3.3 Reducing the abort effects

3.3.1 Victim selection strategies

In CBL(d), cascading aborts may happen. When T_i aborts, some transaction T_j that has an incoming abort edge $T_i \rightarrow_A T_j$ should also abort. Furthermore, if T_i is an LLT, more transactions could be involved in the cascading aborts because T_i has executed quite a long time and thus there are more chances to create the abort edges. This implies that we must be careful to select victims so that the abort effects become minimized. Example 3 shows the fatal effect due to the simple-minded victim selection strategy.

Example 3 (Simple-minded Victim Selection Strategy): Consider a TDG illustrated in Figure 4(a), in which T_{i+1} has abort dependency from T_i, where $1 \le i < 4$. Suppose that T_1 tries to read a data that has been updated by T_4. Then a dependency cycle is created (Figure 4(b)). Suppose T_1 (that is blamed for making the cycle) is selected as a victim. Note that this is a usual strategy adopted in 2PL and SGT, since all cycles include T_1. However, aborting T_1 makes T_2 abort again because of the abort dependency. This in turn makes T_3 abort and finally T_4 abort. Every transac-

Figure 4 TDGs for describing victim selection strategy

tion in a cycle should be aborted due to the simple-minded victim selection strategy. In this case, the best choice as a victim is T_4 because aborting T_4 does not result in any other transaction aborts. n

When appending a new edge $T_n \rightarrow T_1$ introduces a new cycle, CBL(*d*) selects a victim with the victim selection strategies shown in Figure 5. The victim selection strategies are *judicious* in the sense that it fixes the number of victims to ***1*** within a cycle. In other words, for every transaction in the cycle, there are no cascading aborts due to the abort dependencies. Note that Strategy 3 guarantees the best choice for a victim when every edge in the cycle is abort edge. This is because aborting any other transactions in the cycle eventually results in aborting the selected victim. Example 3 illustrated this.

LLTs should not be aborted as far as possible. This is why we continue the trace until we find an edge E_k that is not an abort edge and T_k is not an LLT. If there is such E_k, we select T_k as a victim (Strategy 1). Otherwise, one of the LLTs that are not involved in abort edges is selected as a victim; in fact, we choose the last one among them (Strategy 2). Strategy 2 can be optimized if transaction-specific information is available. In particular, let $\Phi(T)$ imply the number of locks held by T or the elapsed time since T began. Now the

```
/* Suppose the cycle is T1 → T2 → T3 → . . . → Tn-1 → Tn → T1 */
/* Suppose Ek represents Tk → Tk+1 */

candidate = T1; change_victim = TRUE;
for (k = 1; k < n; k++)                /* Trace every edge in the cycle from E1 */
    if (Ek is not an abort edge) {     /* Ek is either a commit edge or a block edge */
        change_victim = OPTIONAL;
        if (Tk is not an LLT)
            select Tk as a victim transaction and exit                  Strategy 1
        else    candidate = Tk;
    }
if (change_victim is OPTIONAL)         /* LLT is selected as victim */
        select candidate as a victim transaction and exit               Strategy 2

/* Every edge in the cycle is abort edge */
else    select Tn as a victim transaction and exit                      Strategy 3
```

Figure 5 Victim selection strategies

candidate is updated to T_k if $\Phi(T_k)$ is less than $\Phi(candidate)$. This guarantees that aborting candidate transaction selected in Strategy 2 minimizes the wasted work.

The victim selection strategies can be implemented without any extra traversal of TDG. The basic idea is to prepare candidate victims while we traverse TDG to detect cycles. When we encounter a new edge E_k to be searched, we decide whether T_k could be a victim. If E_k is not an abort edge and T_k is not an LLT, T_k is fixed to the candidate victim and will not be changed thereafter. Otherwise, T_k is set to the current candidate victim and it may be changed in the course of scanning the following edges. When a cycle is detected, we can resolve the cycle by aborting the candidate victim.

A lock request of T_i could create multiple cycles. In this case, we should resolve each cycle with the victim selection strategies. If T_i is selected as a victim for some cycle, all the cycles can be resolved by aborting T_i because they include T_i; otherwise all the victims of cycles should be aborted. Note that this policy of resolving multiple cycles may not be optimal, since the victim selection strategies of Figure 5 can be applied to only one cycle. A victim selected by Strategy 1 or Strategy 2 could have abort edges in other cycles. One way around this problem is to list all the cycles and transactions participating each cycle and to select the best victims. However, in the worst case, the complexity of listing all the cycles and its participating transactions is $O(3^{n/3})$, where n is the number of the transactions [10]. This is certainly too costly. So we exploit another solution that makes Strategy 1 more reasonable. Next section describes this issue.

3.3.2 Restricting the abort dependencies of long-lived transactions

An LLT could abort not only when it is selected as a victim but also due to the cascading aborts through the abort dependencies. We can reduce the possibility of aborting LLTs by *preventing LLTs to read uncommitted data of short transactions* even though allowing such read operations does not exceed the limit of dependency depth, d. This means that we always prohibit abort dependencies from short transactions to LLTs. Since short transactions have brief period, blocking LLTs would not make the LLTs wait long. By prohibiting the abort dependencies, Strategy 1 of the victim selection strategies becomes reasonable. This is because aborting a short transaction that is selected as a victim in Strategy 1 never introduces aborts of other LLTs.

4 Simulation Model

We compare the performance of CBL(*d*) with that of 2PL and SGT using a simulation model. Much of our model is adapted from the model used in [11]. However, the transaction (workload) model has been modified to incorporate multiple transaction types, which has a significant effect on the performance of the protocols. The simulation model was implemented using the CSIM [12] discrete-event simulation package.

4.1 Transaction system model

Our simulations are based on the closed queuing model illustrated in Figure 6. The model consists of the following components: *transaction generator* (TG), *transaction manager* (TM), *transaction scheduler* (TSCH), and *data manager* (DM).

TG has a role to generate transactions, each of which is modeled as a sequence of database operations, i.e., each lock request is followed by a database access operation. The TG consists of a variable number of terminals. The actual control of transactions into the system is controlled by the *inter_arrival_delay* parameter, which is modeled as a Poisson distribution. Each terminal issues transactions, one at a time, waiting for a transaction to finish before issuing the next transaction.

For each transaction, the TM forwards lock requests to the TSCH by placing them in the *sch_queue*. If the lock request is granted, the TM receives a data access completion mes-

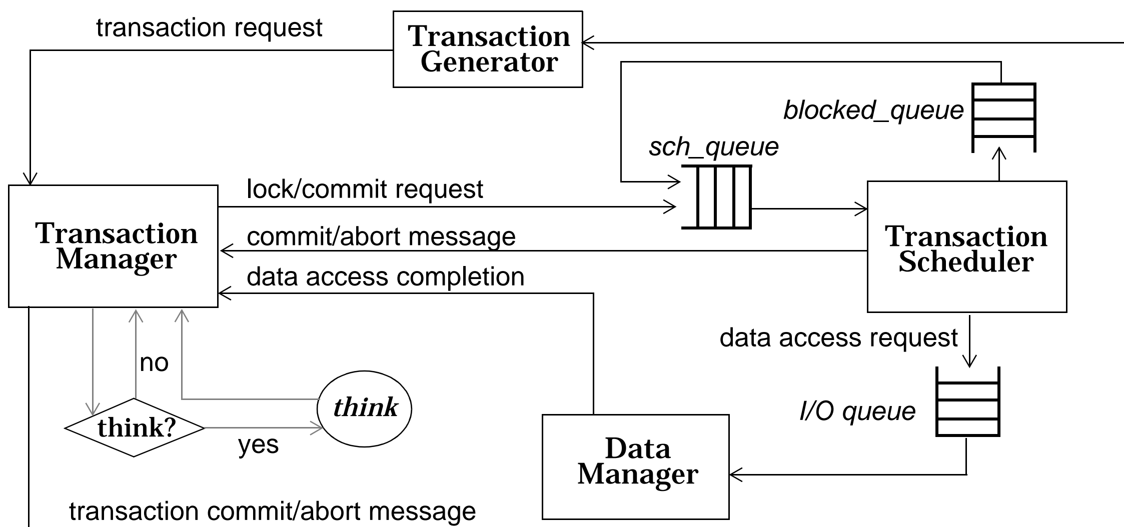


Figure 6 Transaction system model

sage from the DM. In one of the experiments, we examine the performance of concurrency control schemes under interactive workloads. The think path in the model provides an optimal random delay that follows data accesses for this purpose. When every data access is completed, the TM requests the transaction commit to the TSCH, and then it is informed of the result state of the transaction (*commit* or *abort*) from the TSCH. It then forwards the result state to the corresponding terminal.

Lock requests in *sch_queue* are taken out by the TSCH in a first-in-first-out order. The TSCH processes each lock request with the concurrency control scheme it adopts. If a lock request is granted, the TSCH forwards the database access operation to the DM. When the TSCH delays a lock request on data *x*, the request is placed in *blocked_queue*[*x*], where it remains until it can be re-scheduled. After the lock becomes available, the TSCH re-schedules the lock request by moving it from *blocked_queue*[*x*] back to the *sch_queue*, where it is processed again. Locks are released when transactions commit or abort.

DM has a variable number of disks, *num_disks*. For each disk, there are one I/O server and one I/O queue. The TSCH stores a database access request to the I/O queue of the corresponding disk. The I/O server takes out the database access request from its I/O queue in a first-in-first-out order, and then executes the request. When the request is completed, the DM sends a data access completion message to the TM. TM and TSCH utilize CPU servers to perform their jobs. The parameter, *num_cpus*, specify the number of CPU servers. The CPU servers are modeled as a pool of servers, all identical and serving a common CPU queue in a first-in-first-out order.

4.2 Experiment methodology

Table 1 describes the parameters that are used to specify the resources and overheads of the system and shows their values that all of our experiments have in common (except where otherwise noted). Parameters that vary from experiment to experiment are not listed in Table 1 and will instead be given with the description of the relevant experiments.

We use a relatively small database to investigate the differences of concurrency control schemes. For each transaction, the sequence of operations and data to be accessed are determined in a probabilistic manner. There are two types of transactions: *short-lived* trans-

Table 1 Simulation parameters

Parameters	Descriptions	Settings
db_size	number of data in database	1000 data
io_delay	I/O time for accessing a data	35 milliseconds
cpu_delay	CPU time for accessing a data	12 milliseconds
cc_cpu_delay	CPU time for servicing a concurrency control request	3 milliseconds
num_terms	number of terminals	20 - 130
short_tran_size	short-lived transaction (SLT) size	10 operations
long_tran_size	long-lived transaction (LLT) size	50 operations
tran_size_dev	transaction size deviation	0.1
llt_pct	LLT percentage	0.2
short_wr_op_pct	write operation percentage of SLT	0.25
long_wr_op_pct	write operation percentage of LLT	0.25
int_think_time	intratransaction think time	0, 10 seconds
inter_arrival_delay	transaction inter-arrival delay	10 seconds

actions (SLT) and *long-lived* transactions (LLT). The percentage of LLTs among total transactions is specified by *llt_pct*. The size of SLT is determined by a uniform distribution between *short_tran_size* ± *short_tran_size* × *tran_size_dev*. The size of an LLT is determined by a uniform distribution between *long_tran_size* ± *long_tran_size* × *tran_size_dev*. The write operation percentage determines what percentage of a transaction's operations will be writes. It is represented by two parameters: *short_wr_op_pct* for SLTs and *long_wr_op_pct* for LLTs. In our experiments, both parameters are set to 0.25. LLTs could have intratransaction think time, specified by *int_think_time* parameter. To model interactive workloads, an LLT can be made to undergo a thinking period whenever it starts a new write operation.

A database system, which uses locking to control access to data, is subject to *data contention*. In addition, any database system that has a limited number of resources, is subject to *resource contention*. As a result of data contention and resource contention, the amount of work performed by the system would not increase linearly with the number of terminals. This non-linear behavior is due to *thrashing* and the point at which thrashing begins to occur is called as *thrashing point*. To evaluate the effect of resources on our system, one CPU resource and two disk resources were chosen to represent one *resource unit*. The number of resource units in the system is represented by the model parameter *num_resource_units*. This balance of CPUs and disks makes the utilization of these resources about equal with

our parameter values, as opposed to being either strongly CPU bound or strongly I/O bound.

The two main performance metrics used in the experiments are the *throughput rate* and the *response time* (turnaround time). The throughput rate is measured as the number of transactions that successfully commit per second. The response time in seconds is measured as the difference between when a terminal submits a transaction and when a transaction successfully commits. The time includes any time spent in the queue and time spent due to restarts. Since we are simulating a closed queuing system, the response time is the inverse of throughput; hence, we do not explicitly show response time. The *average restart ratios* of transactions are also calculated in some cases. The restart ratio is defined as the average number of times a transaction is restarted per commit. It is particularly useful in determining the effects of victim selection strategies and cascading aborts.

A form of the batch mean method was used for the statistical analysis of simulation results. Each simulation was run for 20 batches with a batch time of 1000 seconds. The first 100 seconds of each batch (10 percent) were discarded in order to account for initial transient conditions. Using this method we were able to produce tight 90 percent confidence intervals for our simulation results. The confidence interval information has been intentionally omitted from the performance graphs for the sake of clarity.

In our experiments, we compare the performance of six concurrency control schemes (CCSs): 2PL, SGT, DD1, DD3, DD5, and DD100. In 2PL, transactions wait on any conflicting lock request, and are restarted only in case of deadlocks. Each deadlock is checked for each wait, and the transaction making the request is selected as the victim. Locks are released together at the end-of-transaction. In SGT, transactions never wait. Whenever a transaction requests a locked data, dependency edges are appended to TDG between conflicting transactions. Like 2PL, if the resulting TDG contains cycles, the transaction making the request is selected as a victim, and all edges adjacent to the transaction are deleted from TDG. Each DDk is an implementation of CBL(d), where the value of d is set to k.

Whenever a transaction is selected as a victim and aborts, the transaction should be restarted. The restarted transaction retains its old script. Hence if it begins execution again immediately, it may conflict with the transaction that causes it to abort. To prevent this, it is held back for some random length of time before restarting. While the aborted transaction is held back, a different transaction is admitted in its place. From the modeling point of view,

we may therefore consider an aborted transaction as restarting immediately after releasing all its locks, and with a new script. So in the experiments, we adopt *fake restart* assumption, in which a restarted transaction is replaced by a new, independent transaction, rather than running the same transaction again.

5 Experiment Results

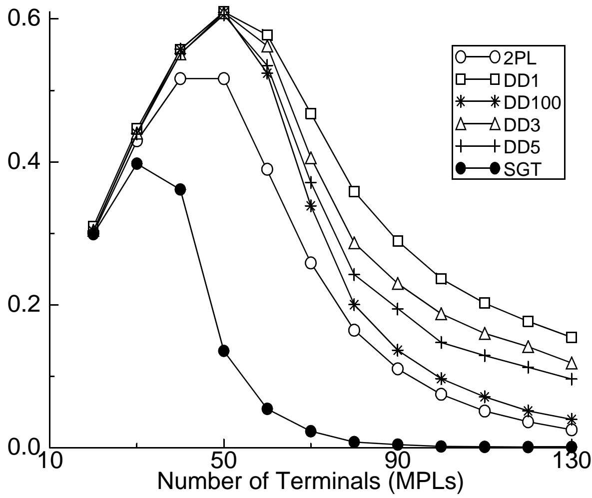
We compared the performance of six CCSs under the following conditions: (1) when LLTs access large number of data, (2) when LLTs execute long period due to user interaction while they access rather small number of data, and (3) when there are no LLTs in the system.

5.1 Large transaction experiment (LXACT)

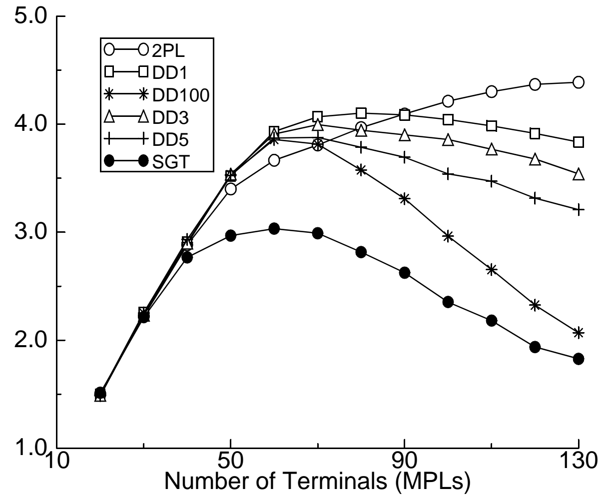
In this experiment, LLTs access an average of 50 data without any intratransaction think time, and the size of SLT is set to 10. The parameter, *num_resource_units* is set to 2.

Figure 7 shows the throughput results of LLTs and SLTs for CCSs. At low MPLs, all the CCSs exhibit similar behavior, whereas at high MPLs there are a number of differences. With regard to LLTs, DD*k* exhibit substantially superior throughput results than 2PL and SGT. This is primarily due to *the effect of victim selection strategies*. In 2PL and SGT, if cycles are created because of a lock request, the transaction making the request is selected as the victim. This implies that, between LLTs and SLTs, there is no difference on the probability of being selected as victims once they make a lock request. Furthermore, since the number of lock requests made by LLTs is five times of that made by SLTs, LLTs are more prone to be aborted than SLTs. Among DD*k*, DD1 outperforms the other CCSs. This is due to *the increased restart ratios of LLTs* in more concurrent CCSs, resulting from cascading aborts and the increased number of locks. Figure 8(a) shows the restart ratios of LLTs.

With regard to SLTs, DD*k* exhibit slightly higher throughput results than 2PL until MPL of 70, even though the restart ratios of DD*k* are larger than that of 2PL as illustrated in Figure 8(b). The reason is that, in low MPLs, the restart ratios of DD*k* and 2PL are low and thus *the effects of transaction restarts are not crucial*. So, CCSs with more concurrent schemes like DD*k* outperform 2PL. In high MPLs, however, 2PL outperforms DD*k* due to

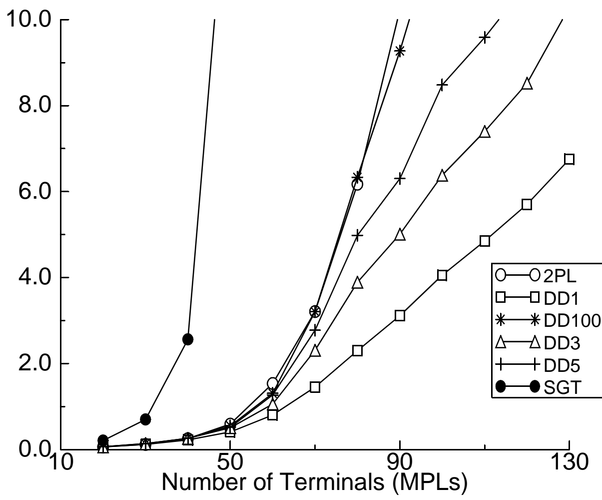


(a) LLTs

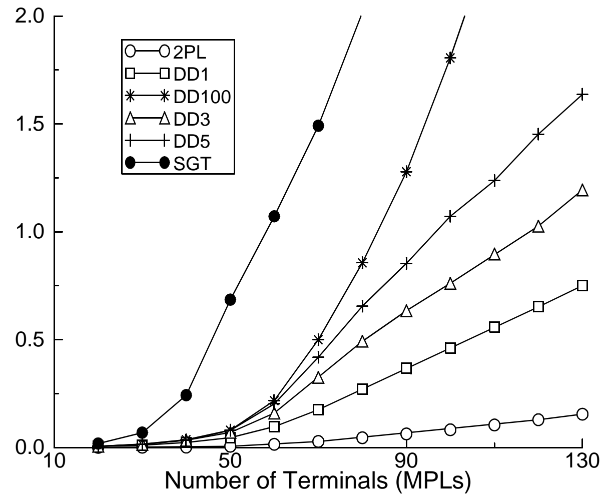


(b) SLTs

Figure 7 LXACT - Average Throughput



(a) LLTs



(b) SLTs

Figure 8 LXACT - Restart Ratios

the following reasons. *First*, DD*k* try to abort SLTs instead of LLTs when dependency cycles are occurred, and thus the restart ratios of SLTs are higher than those in 2PL. *Next*, LLTs are likely to be blocked in 2PL. This means that, in 2PL, the number of locked data in a given time must be small compared to DD*k*. So, the lock request made by an SLT could make fewer dependency cycles than DD*k*. This is one of the reasons that DD1 has the smallest restart ratios of LLTs among the other DD*k*. *Last*, in DD*k*, there are dangers of cascading aborts. The cascading aborts must be occurred more frequently when the value of *k* is large. This is why DD100 has the largest restart ratio among the other DD*k* schemes. Since 2PL does not result in cascading aborts, the restart ratios of SLTs can be further reduced. Note that SGT has the largest restart ratio in both LLTs and SLTs. This is due to the simple-minded victim

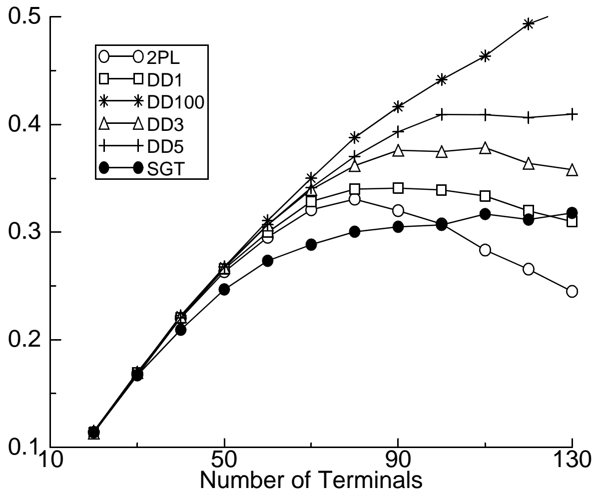
selection strategies, the highest degree of concurrency, and as a consequence, a number of cascading aborts.

5.2 Interactive transaction experiment (INTER)

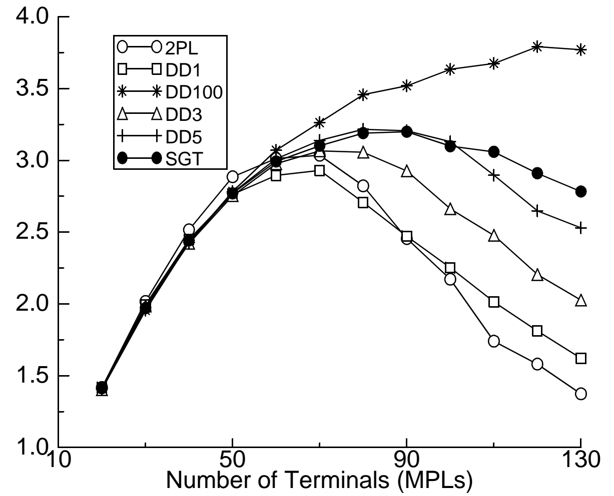
In this experiment, we studied the impact of interactive transactions on the performance of the six CCSs. We modeled interactive transactions that perform a number of reads, think for some period of time, and then perform their writes. Each LLT has intratransaction think time (*int_think_time*). The value of *int_think_time* is set to 10 seconds. At the same time, *long_tran_size* is set to 10 like *short_tran_size*, in order to investigate the effect of large intratransaction think times on the various CCSs, while conflicts between transactions are rare. Note that this notion of LLT is usual in design applications, where a designer's transaction works on a small number of design objects for a long period without wanting these objects to be released. In this experiment, the number of resource units is set to 2.

Figure 9 shows the throughput results of LLTs and SLTs for the six CCSs. In case of LLTs, DD100 outperforms the other CCSs. This is due to *the high degree of concurrency with infinitesimal restart ratio*. Since the transaction size of each LLT is decreased to 10, transaction conflicts become rare. So the probability of introducing new cycles turns low. Furthermore, even in case of introducing cycles, DD*k* seldom select LLTs as victims due to the victim selection strategies. As a result, the restart ratios of LLTs can be reduced. Figure 10(a) shows the reduced restart ratios of LLTs in every DD*k*. Because the restart ratios of LLTs are low, DD*k* with large *k* values perform better than the other DD*k* with small *k* values. This is why DD100 shows the best performance among the other DD*k*.

There are a number of interesting observations on this experiment. First, SGT still has high restart ratio due to cascading aborts. Note that SGT could restart LLTs if an SLT aborts, because the LLTs are allowed to read uncommitted data of the SLT. Second, even though SGT has higher restart ratio than 2PL, the throughput of SGT is better than 2PL in high MPLs. The reason is that the restart delay was set to 0 in our experiment. If a transaction aborts, the transaction restarts immediately. As a result, the repeated execution of a restarted transaction in SGT could terminate earlier than the first execution of a blocked transaction in 2PL. Last, in 2PL, the restart ratios of LLTs are higher than those of SLTs. One might think that the restart ratios of LLTs and SLTs must be equal, because they have

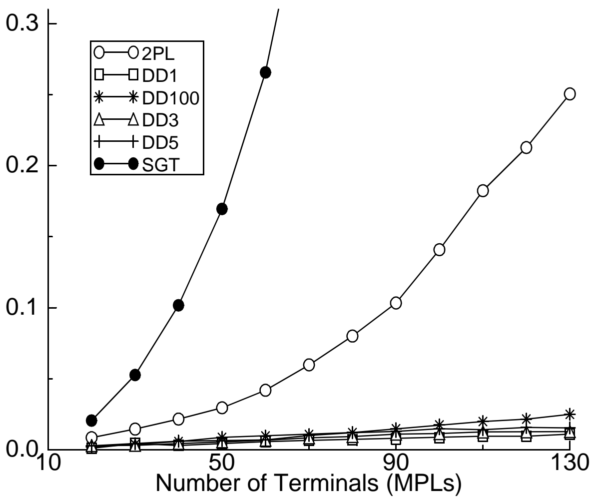


(a) LLTs

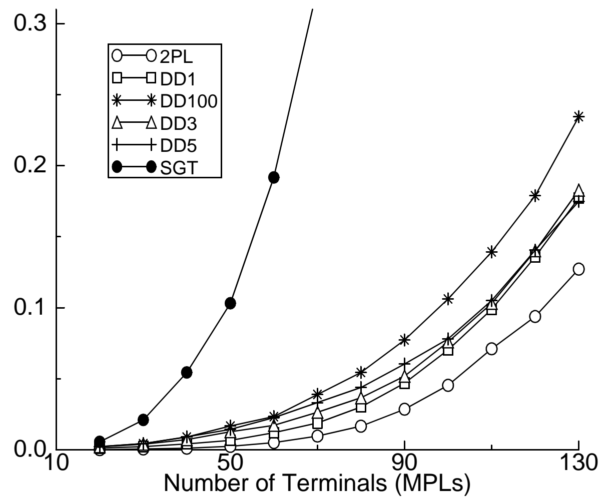


(b) SLTs

Figure 9 INTER - Average Throughput



(a) LLTs



(b) SLTs

Figure 10 INTER - Restart Ratios

the same write percentage and transaction size. However, this is not true due to very large intratransaction think time compared to the execution time of SLTs. So, a number of SLTs could be blocked while LLTs are in thinking stages, and after thinking, most of active transactions are now LLTs. At this time, most of dependency cycles are introduced by LLTs, and then the LLTs must be aborted due to the victim selection strategies of 2PL.

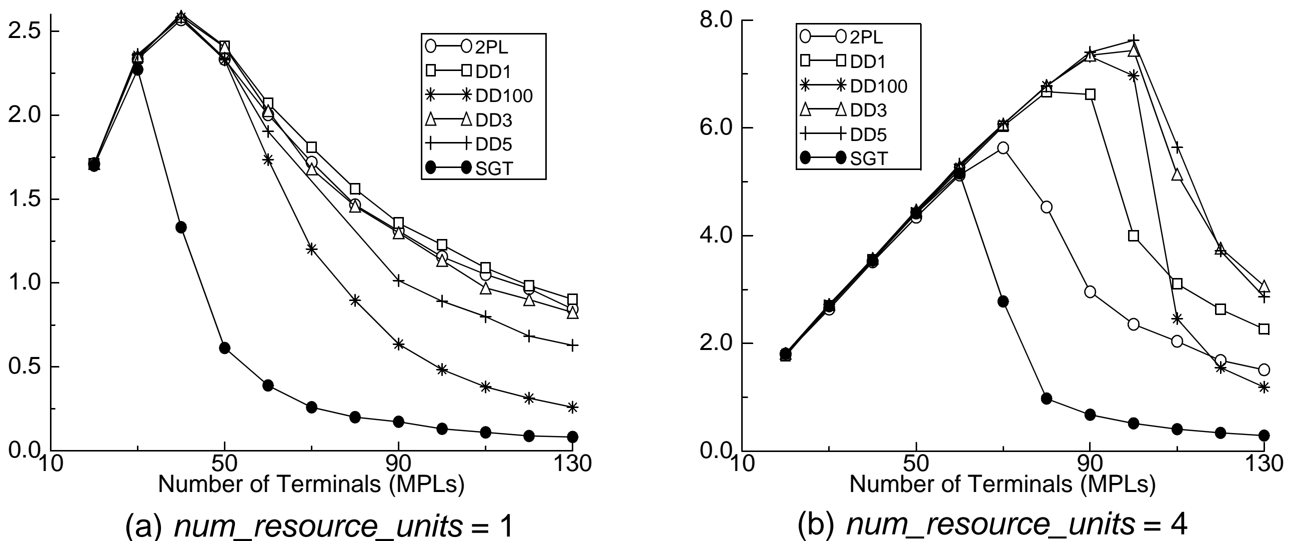
In case of SLTs, DD100 also exploits the most superior throughput results. The reason is same to that of LLTs: *the maximum degree of concurrency with infinitesimal restart ratio*. Note that if an SLT restarts, it can terminate its execution immediately because the SLT is seldom blocked in DD100. The same is true in SGT. However, the throughput of SGT is lower than that of DD100, since the restart ratios of SLTs in SGT are high due to cascad-

ing aborts. Both 2PL and DD1 exploit the worst performance due to their blocking nature.

5.3 Short transaction experiment (SXACT)

In this section, we evaluate the performance of the six CCSs in a traditional workload where there are no LLTs. The intention of this experiment was to find out the protocol overhead of DD*k* in a conventional on-line transaction processing systems, where each transaction is of short duration. In this experiment, *llt_pct* is set to 0 and *short_tran_size* is set to 20.

Figure 11 shows the throughput results of the six CCSs, where *num_resource_units* is set to 1 and 4, respectively. When *num_resource_units* is set to 1, the throughput results of DD1, 2PL, and DD3 are similar. Among them, DD1 outperforms 2PL slightly, and 2PL outperforms DD3. This means that, in DD1, the effect of high degree of concurrency overwhelms that of cascading aborts; while the reverse is true in DD3. As the number of resources increases, DD3 and DD5 outperform 2PL and DD1. This is because 2PL and DD1 begin thrashing primarily due to *data contention*, increasing the number of resources results in marginal performance improvements. On the other hand, since DD3 and DD5 begin thrashing primarily due to *resource contention*, substantial performance improvements can be achieved by increasing the number of resources. Thus, any performance gains that can be achieved by DD*k* are offset due to the limited availability of physical resources. In high MPLs, the performance of DD100 downgrades dramatically due to a number of cascading



(a) *num_resource_units* = 1 (b) *num_resource_units* = 4

Figure 11 SXACT - Average Throughput

aborts. SGT still exploits the poorest performance due to the simple-minded victim selection strategy and thus a number of cascading aborts.

5.4 Summary of the results of other experiments

The results from three other experiments are worth noting here (see [13] for details). All of the experiments changed the value of some parameters in LXACT. The first experiment increased the database size to 2000 to make a rather low contention workload. In this experiment, DD1 outperformed 2PL even in case of SLTs with high MPLs due to fewer data conflicts. This result could be anticipated from Figure 7(b). The other two experiments were conducted to evaluate the performance of the six CCSs under *biased* workload, where the percentage of write operations in SLT and that in LLT are different. One experiment with read intensive SLTs and write intensive LLTs was intended to approximate an environment where there are short-term queries and large-scale update transactions. In this experiment, LLTs request more write locks than uniform workload, and thus there are more chances to create dependency cycles. Since the frequent aborts of LLTs have more critical effects on DD*k* than 2PL due to cascading aborts, the performance gain of DD*k* was reduced. The other experiment with write intensive SLTs and read intensive LLTs was intended to approximate an environment like banking system, where there are frequent short-term updates and infrequent large-scale read transactions to collect statistical information. Since most write operations were performed by SLTs and DD*k* does not allow LLTs to read uncommitted data of SLTs, the possibilities of introducing cascading aborts can be reduced. Fewer cascading aborts in turn result in decreasing the restart ratios of SLTs. As a result, in case of SLTs, both DD3 and DD5 exploited superior throughput than 2PL until MPL with 110 and than DD1 for every MPL.

6 Concluding Remarks

In this paper, we proposed an extension to 2PL, named conditional blocking (CBL), for managing long-lived transactions (LLTs). The major contribution of CBL is to properly balance the aborting effect and the waiting effect due to LLTs, and thus it can attain more concurrent executions than 2PL with fewer aborts than SGT. This is achieved by two notions: (1) *restricting the dependency depth* to nonnegative integer d, so that transactions with larger dependency depths than d should be blocked and those with smaller dependency depths than d can access already locked data with conflicting mode, and (2) *judicious victim selection strategies* so that a victim is chosen to minimize the effect of transaction abort.

We have explored the performance of CBL under a variety of database workloads and system configurations. Our experimental results showed that CBL exhibits substantial performance improvements over 2PL and SGT when LLTs are present. Among CBLs with different values of d, CBLs with small d outperformed CBLs with large d when LLTs access large number of data. This is due to the increased restart ratios of LLTs in CBLs with large d, resulting from cascading aborts and the large number of locks held. On the other hand, the order inverted when LLTs execute long period due to user interaction while they access small number of data. This is because the restart ratios of LLTs are reduced due to fewer transaction conflicts. Hence the degree of concurrency becomes a key factor of the performance in this workload. Lastly, we found that the performance characteristics of CBL are highly dependent on the availability of the underlying physical resources, that is, substantial performance improvements can be achieved by increasing the number of physical resources. This property of CBL is well matched with the trend of ever-increasing CPU and I/O speeds.

References

[1] N. Barghouti and G. Kaiser, Concurrency Control in Advanced Database Applications, ACM Computing Surveys 22(3) (1991) 269-317.

[2] A. Elmagarmid, *Database Transaction Models for Advanced Applications* (Morgan Kaufmann Publishers, 1992).

[3] H. Salem, H. Garcia-Molina, and J. Shands, Altruistic Locking, ACM TODS 19(1) (1994) 117-165.

[4] P. Dasgupta and Z. Kedem, The Five Color Concurrency Control Protocol: Non-Two-Phase Locking in General Databases, ACM TODS 15(2) (1990) 281-307.

[5] H. Garcia-Molina and K. Salem, SAGAS, Proceedings of the 13th ACM SIGMOD Conference (1987) 249-259.

[6] F. Bancilhon, W. Kim, and H. Korth, A Model of CAD Transactions, Proceedings of the 11th VLDB Conference (1985) 25-33.

[7] S. Yeh, C. Ellis, A. Ege, and H. Korth, Performance Analysis of Two Concurrency Control Schemes for Design Environments, Information Sciences 49 (1989) 3-33.

[8] P. Klahold, G. Schlageter, R. Unland, and W. Wilkes, A transaction model supporting complex applications in integrated information systems, Proceedings of the 11th ACM SIGMOD Conference (1985) 388-401.

[9] P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, (Addison-Wesley, 1987).

[10] B. Jiang, Deadlock Detection is Really Cheap, ACM SIGMOD RECORD, 17(2) (1988) 2-13.

[11] R. Agrawal, M. Carey, and M. Livny, Concurrency Control Performance Modeling: Alternatives and Implications, ACM TODS 12(4) (1987) 609-654.

[12] H. Schwetman, *CSIM User's Guide for use with CSIM Revision 16*, (MCC, 1992).

[13] H. Cho, *Concurrency Control for Long-Lived Transactions in Collaborative Design Applications* (Ph.D. Thesis, KAIST, 1994).