Stochastic Petri Net Analysis of Deadlock Detection Algorithms in Transaction Database Systems with Dynamic Locking

ING-RAY CHEN

Institute of Information Engineering, National Cheng Kung University, No. 1 University, Tainan, Taiwan
Email: irchen@iie.ncku.edu.tw

We develop stochastic Petri net (SPN) models to analyze the best time interval between two consecutive executions of periodic deadlock detection algorithms for two-phase locking database systems with dynamic locking. Our models can accurately estimate 'wait time per lock conflict' automatically and allow the best time interval to be determined as a function of workload intensities and database characteristics. A system designer can apply the SPN tools developed in the paper to choose the best deadlock detection strategy, continuous or periodic, to optimize system performance, when given a set of system characteristics.

Received May 5, 1995; revised September 12, 1995

1. INTRODUCTION

Many existing transaction database systems such as IBM's DB2 and RTI's INGRES ensure serializable executions [1, 2] by running a version of two-phase locking (2PL) [3] in which a transaction consisting of a sequence of access operations should obtain a lock before accessing a data item controlled by the lock and should release all of the locks it owns together when it terminates. A nice property of 2PL is that it allows concurrent transaction executions to produce the same output and have the same effect on the database as some serial executions (where transactions are executed one at a time) of the same set of transactions. However, because of the use of locks in 2PL, lock conflicts may exist among transactions, thus requiring the use of a *lock conflict resolution* method to order the execution of conflicting transactions. Various 2PL lock conflict resolution methods have been proposed in the literature in the past, including those that can cause deadlocks, e.g. general waiting [2], and those that are deadlock-free (due to selective transaction aborts), e.g. running priority [4, 5], cautious waiting [6] and wait-depth limited methods [5, 7]. If the system allows deadlocks to exist, it requires the use of a deadlock detection algorithm. One common strategy is to check the wait-for-graph (WFG) whenever a transaction is blocked. If the blocked transaction is involved in a deadlock cycle, then one transaction in the cycle is aborted to break the deadlock. Otherwise, the transaction waits until it gets its lock. This method is termed 'continuous deadlock detection' in the literature [8].

To model the behaviour of such a transaction database system with continuous deadlock detection, previous models [9–13] require several parameters to be estimated *a priori*. One parameter is the probability of lock conflict per lock request. Another parameter is the probability that a transaction is deadlocked with other transactions when a subsequent lock request is not granted. These two parameters in theory can be estimated fairly accurately by first estimating the number of locks owned by active transactions in the system and then computing the probability that each event can occur [9–12]. Another parameter needed is the wait time for a lock by a blocked transaction. This 'wait time per lock conflict' parameter is difficult to estimate accurately and existing methods for estimating it are often controversial [14]. For example, Irani and Lin suggested that simulation or empirical measurements be used to estimate its value [9]. Pun and Belford estimated it as a function of the throughputs of aborted and terminated transactions, which by themselves are model outputs [10]. As a result, an iterative solution technique is used to repeatedly execute the model until this wait time parameter converges. Hartzman derived an analytical solution for this parameter in an open database system [14]. However, it is obtained based on the dubious assumption that deadlocks never occur and it is not apparent how that approach can be applied to a closed database system for which the population of transactions is fixed.

Recently, Thomasian and Ryu derived an approximate, non-iterative analytical solution for the wait time per lock conflict parameter in closed and open 2PL systems with load control [12, 13]. They ignored the effect of deadlocks in their analysis because deadlock is considered a rare event in typical database systems except in high-capacity systems in which the increased throughput is achieved at the cost of a higher degree of transaction concurrency, or in systems in which hot-spot data items exist which create a high level of data contention. They validated their analytical solution

against simulation results based on continuous deadlock detection. However, deadlock detection overhead was not considered in the simulation study and other alternatives for deadlock detection were not investigated.

Conceptually, for systems in which deadlocks occur frequently, the WFG should be checked *continuously* when a transaction is blocked so that deadlocked transactions are not blocked for too long. On the other hand, for systems in which deadlocks rarely occur (which is more typical), deadlocks can be checked only *periodically* after a few edges have been added to the WFG so that the overhead associated with deadlock detection can be minimized. Kumar [15] in his simulation study concluded that in dynamic locking, deadlocks need not be detected at each blocking and the response time can be improved by periodic rather than continuous deadlock detection. Agrawal *et al.* [8] also compared the performance of continuous and periodic deadlock detection by means of simulation. Under restricted database environment settings and the assumption that a fixed deadlock detection interval is used for periodic detection, it was concluded that continuous detection can perform better than periodic detection, especially when the multiprogramming level is high and there is high contention for data items. There are two potential problems in Agrawal's study. First, the assumption that the deadlock detection interval is fixed for various database environment settings is not justified because an optimal deadlock detection interval should exist for every database environment (i.e. multiprogramming level) and this optimal interval should be used when a comparison is made. Our view is that for every database environment setting, there exists an optimal deadlock detection interval and this optimal interval is a function of the settings of the database environment variables, including user workloads (e.g. the degree of multiprogramming), resource requirements (e.g. disk and CPU processing capabilities) and database characteristics (e.g. the database size, transaction size, number of granules in the database, locking policy and data accessing pattern). Second, the execution time of deadlock detection was ignored in their study, which can create a bias toward continuous or periodic deadlock detection, depending on the interval selected for performing periodic deadlock detection and the frequency with which blocking occurs. In this paper, we eliminate these potential problems. Our goal is to develop an analytical tool rather than a simulation tool that can help choosing the correct deadlock detection approach for 2PL systems with dynamic locking (where locks are requested on demand), when given a set of system characteristics.

To achieve the goal, we develop our analytical model based on stochastic Petri net (SPN) models [16]. There are two salient features in our SPN model. First, although our model still requires the probability of blocking and the probability of deadlock be calculated as model inputs based on system characteristics, unlike previous analytical modeling studies, it does not require the 'wait time per lock conflict' parameter be calculated *a priori* as an input, thus reducing the potential inaccuracy associated with estimating the value of this parameter [14]. In our approach we implicitly model the behavior of a blocked (but not deadlocked) transaction by a Petri net description such that the blocked transaction can simply migrate from one place to another in the net when certain enabling conditions are satisfied. This feature is made possible because a Petri net can keep track of the state evolution of the system. This allows us to dynamically determine the probability of a blocked transaction getting its lock when a particular state transition occurs. To achieve this, we differentiate blocked, non-lock-owner transactions from blocked, lock-owner ones in constructing our Petri net model to provide more precise information on the number of lock owners in the system as the system evolves over time. As a result, unlike the queueing network models proposed in [10, 17], our Petri net model can be solved non-iteratively and efficiently.

Second, two separate Petri net models are constructed separately to evaluate the system performance of continuous and periodic deadlock detection algorithms. These SPN models can be solved easily by using a commercial software package such as Stochastic Petri Net Package (SPNP) [18, 19]. Previous performance models based on queueing network [9, 10, 17] are non-trivial and difficult to apply. For example, in [9, 10, 17], it involves the concept of changing the customer class of a transaction during the transaction's life time as the transaction requests, acquires and releases locks. Consequently, a special solution technique is used to solve their queueing network models. In [10, 17], an iterative solution technique is also used to solve their models. The requirement of using special solution techniques, rather than using a commercial software package (e.g. SHARPE [20]), limits the general applicability of these queueing network models. Although our SPN model involves the concept of changing a transaction's class during a transaction's life time as in [9, 10], the change of customer classes is explicitly modeled by the migration of a transaction from one place to another in the net. Consequently, no special solution method is required to solve the model.

The major contributions of the paper lies in the development of non-iterative Petri net models that can accurately estimate the wait time per lock conflict automatically, taking into consideration the overhead for continuously or periodically executing deadlock detection. The result is that they can be used as an analysis tool in providing guidelines for selecting a deadlock detection strategy. With respect to the simulation model in [8, 15], our Petri net model, being an analytical tool, is much easier to construct and evaluate.

The rest of the paper is organized as follows. Section 2 discusses the background for deadlock detection in 2PL database systems, states our model assumptions and

defines the notation to be used in the paper. Possible database environment variables which must be considered in performance assessment are defined. Section 3 develops an iterative SPN model for 2PL databases with continuous deadlock detection; it also requires the wait time for a lock as a model input as in [10], thus requiring the SPN model to be iteratively solved by SPNP in multiple runs until the wait time converges. A technique for modeling a processor-sharing CPU in the SPN model is described. The validity of the iterative SPN model is demonstrated by comparing the model's outputs in several database environment settings with the results based on a queueing model [10]. In Section 4, modifications to the iterative SPN description are made with the goal of removing the wait time for a lock as a model input. We demonstrate that the new SPN model can greatly improve solution efficiency with a potential of improving solution accuracy. In Section 5, we develop a similar but separate SPN model for analyzing the performance of 2PL with periodic deadlock detection. We illustrate how to identify the correct deadlock detection approach for a system with 2PL dynamic locking using these two separate SPN models, when given a set of system characteristics. Section 6 summarizes the paper and outlines some future research areas.

2. BACKGROUND, ASSUMPTION AND NOTATION

2.1. Assumptions

1. The database is in a closed system in which the multigramming level is MPL. The database contains SZ_{db} data items. Each transaction on the average accesses SZ_{tr} data items. The unit of physical lock is called a granule [21] which contains SZ_{lock} data items. If a transaction locks a granule then it essentially locks all the data items contained in the granule. Of course, when $SZ_{lock} = 1$ each data item is a granule itself and has its own separate lock, thus covering the special case considered in [12, 13]. Not placing a separate lock on a data item may be desirable for performance reasons for certain transaction accessing patterns described below. The database system being modeled is characterized by dynamic locking policy and well-placed transaction accessing pattern [21]. 'Dynamic' locking means that locks are requested one at a time by a transaction as they are needed and there is no pre-determined sequence on the locks. 'Well-placed accessing' means that the data items referenced by a transaction are packed into as few granules as possible. This assumes that a transaction accesses the data items sequentially and that the granule boundaries are optimally located for the transaction. Under well-placed accessing, if SZ_{tr} is the size (number of data items) of a transaction and SZ_{lock} is the size of a granule, then the total number of locks requested by a transaction, denoted by NL, is given by
$$NL = \left\lceil \frac{SZ_{tr}}{SZ_{lock}} \right\rceil \quad (1)$$
We concentrate on well-placed instead of other transaction accessing patterns such as worst-placed and random because performance data for well-placed accessing pattern under continuous deadlock detection are available for comparison [10]. (Worst placement is the opposite of well placement: each transaction requires the maximum number of granules possible. In this case, NL = minimum (SZ_{tr}, SZ_{db}/SZ_{lock}). Random placement means that a transaction will access the database randomly, with each data item being referenced with equal probability. The expression of NL for random placement can be found in [10]). Other transaction assessing patterns can be analyzed in a similar way by our SPN model except a different equation needs to be used for computing NL.
2. The system has one CPU and one data manager. The data manager may keep more frequently accessed data items in cache to minimize I/O overhead. A transaction acquires NL locks to access SZ_{tr} data items in its lifetime. Locks are maintained in the main memory and I/O accesses are not required to get locks. Each lock acquisition is followed by a visit to the data manager to retrieve SZ_{tr}/NL data items controlled by the lock and then a visit to the CPU to perform useful work based on the data items retrieved. When a transaction terminates or aborts, a new transaction immediately enters the system so that the total number of transactions in the system remains MPL.
3. Every lock owner is assumed to own $\lceil NL/2 \rceil$ locks. This assumption is introduced to avoid state space explosion and is also used in previous models. However, we do not assume the number of lock owners, NLO, as a constant as in [10, 17]. In our (non-iterative) Petri net model, NLO is computed dynamically as the system evolves.

Table 1 shows the list of input parameters to our model. Other than these input parameters, there are also

TABLE 1. Input parameters

MPL	total number of transactions (i.e. multiprogramming level)
SZ_{db}	size of (i.e. number of data items in) the database
SZ_{tr}	size of (i.e. number of data items in) a transaction
SZ_{lock}	size of (i.e. number of data items in) a granule or number of data items controlled by a lock
D_{CPU}	total service demand of CPU by a transaction (in sec)
D_{dm}	total service demand of data manager by a transaction (in sec)
$S_{CPU,lreq}$	service demand of CPU for processing a lock request (in sec)
$S_{CPU,lset}$	service demand of CPU for setting a lock (in sec)
$S_{CPU,lrel}$	service demand of CPU for releasing a lock (in sec)
$S_{CPU,node}$	service demand of CPU for checking a node in the WFG

TABLE 2. State-independent computable parameters

NL_{db}	total number of locks (i.e. number of granules) in the database system $NL_{db} = \left\lceil \frac{SZ_{db}}{SZ_{lock}} \right\rceil$
NL	total number of locks demanded by a transaction = total number of visits to CPU and data manager each $NL = \left\lceil \frac{SZ_{tr}}{SZ_{lock}} \right\rceil$
S_{CPU}	service demand of CPU per visit by a transaction (in sec) $S_{CPU} = \frac{D_{CPU}}{NL}$
S_{dm}	service demand of data manager per visit by a transaction (in sec) $S_{dm} = \frac{D_{dm}}{NL}$
$D_{CPU,lrel}$	service demand of CPU for releasing all NL locks (in sec) $D_{CPU,lrel} = NL \times S_{CPU,lrel}$
$D^{continuous}_{CPU,deadlock}$	service demand of CPU for executing a continuous deadlock detection algorithm (in sec) $D^{continuous}_{CPU,deadlock} = N \times S_{CPU,node}$
$D^{periodic}_{CPU,deadlock}$	service demand of CPU for executing a periodic deadlock detection algorithm (in sec) $D^{periodic}_{CPU,deadlock} = N^2 \times S_{CPU,node}$
P_{exit}	probability that after a transaction completes a visit to CPU, the transaction terminates $P_{exit} = 1/NL$

'computable' parameters which can be computed automatically by our model from the input parameters. These computable parameters are divided into two sets, i.e. 'state-independent' and 'state-dependent'. The difference is that the value of a state-dependent parameter can change dynamically as the system goes from one state to another over time, while that of a state-independent parameter remains constant. These two sets of computable parameters are listed in Tables 2 and 3.

TABLE 3. State-dependent computable parameters

NLO	number of lock owners in the system– each lock owner on the average owns one half of the locks it needs, i.e. $\lfloor NL/2 \rfloor$ locks
D_{lock}	wait time for a lock by a blocked transaction (in sec)
P_{g1}	probability that when a (non-lock-owner) transaction requests its first lock, the lock is granted $P_{g1} = \frac{NL_{db} - NLO \times \lfloor NL/2 \rfloor}{NL_{db}}$
$1 - P_{g1}$	probability of blocking for first lock
P_{g2}	probability that when a (lock-owner) transaction requests a subsequent lock, the lock is granted $P_{g2} = \frac{(NL_{db} - \lfloor NL/2 \rfloor) - (NLO - 1) \times \lfloor NL/2 \rfloor}{NL_{db} - \lfloor NL/2 \rfloor}$
$1 - P_{g2}$	probability of blocking for a subsequent lock
P_d	probability of deadlock $P_d = \frac{1 - P_{g2}^{\lfloor NL/2 \rfloor - 1}}{NLO - 1}$

It should be emphasized that the effect of a well-placed transaction accessing pattern is reflected in the value of NL for the state-dependent parameter set and $NL = SZ_{tr}$ covers the special case when each data item has its own separate lock. Other than NLO and D_{lock}, all state-dependent parameters can be estimated *accurately* by using simple probability arguments. Here, P_{g1}, P_{g2} and P_d are estimated as suggested in [10]. Once D_{lock} is known, we can use an infinite service center with no queueing and a service time of D_{lock} to hold all blocked transactions [22].

As stated before, methods for estimating D_{lock} are often controversial. One way to estimate D_{lock} [10] is based on the observation that in the steady state, when a transaction terminates or aborts, a blocked transaction will be unblocked so that the number of lock owners in the system remains the same, i.e.

$$D_{lock} = (N - NLO) \times T_{int}$$

where T_{int} represents the time interval between which a transaction terminates or aborts and NLO represents the average number of lock owners in the system. The methods for estimating NLO will be described later. Since T_{int} is an output of the queueing network model, this method requires T_{int} to be solved repeatedly until it converges. Another way is to estimate D_{lock} as a function of the transaction response time [17]. Since the response time is also a model output, an iterative solution technique is again required.

In Section 3, we develop an *iterative* Petri net model that adopts the same method for estimating D_{lock} as in [10], thus requiring T_{int} to be solved iteratively until it converges. However, unlike the queueing network model in [10], our iterative Petri net model does not require any special solution technique. A software package such as SPNP [18] can solve it. NLO in this case is approximated as a constant as in [10], i.e.

$$NLO = \min\left(N, \frac{NL_{db}}{\lfloor NL/2 \rfloor}\right)$$

In Section 4, we then develop a *non-iterative* Petri net model to eliminate imprecise and error-prone methods for estimating NLO and D_{lock}. By slightly modifying the Petri net description, we keep track of the value of NLO as the system evolves over time. This is achieved by explicitly describing the behavior of a blocked lock-owner or non-lock-owner transaction in the Petri net such that whether or not a blocked transaction can get a lock (when another transaction terminates or aborts) can be determined dynamically as a function of the system state.

Table 4 defines the performance metrics considered in the paper. They can be computed by assigning reward

TABLE 4. Performance measurements

X	system throughput, i.e. number of transactions completed/sec
U_{CPU}	percentage of CPU used for doing useful computation on data items (unit: %), excluding that required for requesting, setting and releasing locks, and for executing deadlock detection code

rates to 'states' in an SPN model. One performance metric, U_{CPU}, is as defined in [10] and is used in the paper only for validating our SPN models. The other performance metric, the system throughput X, is used by the paper as a basis for evaluating system performance.

3. AN ITERATIVE SPN MODEL FOR 2PL WITH CONTINUOUS DEADLOCK DETECTION

Figure 1 shows our iterative SPN modeling a 2PL database system with continuous deadlock detection. A transaction is modeled by a token. There are altogether 13 places and 17 transitions, of which nine are timed (with small rectangular boxes in Figure 1) and eight are immediate (with solid lines). Tables 5 and 6 give the meanings of the nine places with timed transitions from the viewpoint of a transaction.

This SPN enables a transition when the input place contains one or more tokens. If an enabled transition is an *immediate* transition, it will fire immediately. If it is a *timed* transition, it will fire after an amount of time elapsed determined by a random sample from the associated distribution with the transition. Table 7 gives the transition probabilities associated with immediate transitions. To simplify our analysis, the firing times of timed transitions are assumed to be exponentially distributed, thus rendering the Petri net stochastic in nature and susceptible to solution techniques provided by SPNP [18, 19]. The approach described here can be easily extended to Extended Stochastic Petri Net (ESPN) [23] models in which firing times are general distributions.

TABLE 5. Meanings of places with respect to a transaction

Place	*Meaning*
p1	the transaction is requesting for the first lock
p2	the transaction is requesting for a subsequent lock
p3	the transaction is setting a lock
p4	the transaction is waiting during the execution of a deadlock detection algorithm
p5	the transaction is releasing all of the locks it owns upon completion
p6	the transaction is releasing all of the locks it owns upon abortion
p7	the transaction is doing useful computation between lock requests
p11	the transaction is retrieving data items controlled by a lock
p12	the transaction is waiting for a lock to be released by other transactions

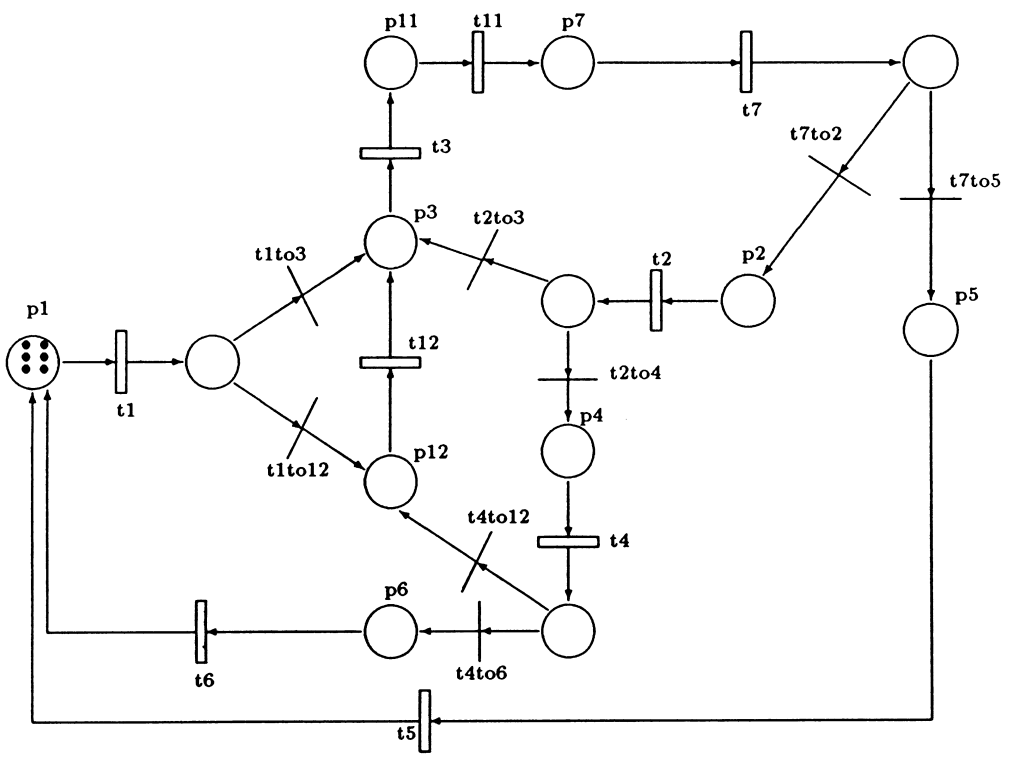

FIGURE 1. An iterative Petri net model for DBS with continuous deadlock detection.

TABLE 6. Transition rate functions

Transistion	*Rate function*
t1	$\frac{1}{S_{\text{CPU,lreq}}} \times \#(\text{p1})/\#(\text{p1}+\text{p2}+\text{p3}+\text{p4}+\text{p5}+\text{p6}+\text{p7})$
t2	$\frac{1}{S_{\text{CPU,lreq}}} \times \#(\text{p2})/\#(\text{p1}+\text{p2}+\text{p3}+\text{p4}+\text{p5}+\text{p6}+\text{p7})$
t3	$\frac{1}{S_{\text{CPU,lset}}} \times \#(\text{p3})/\#(\text{p1}+\text{p2}+\text{p3}+\text{p4}+\text{p5}+\text{p6}+\text{p7})$
t4	$\frac{1}{D^{\text{continuous}}_{\text{CPU,deadlock}}} \times \#(\text{p4})/\#(\text{p1}+\text{p2}+\text{p3}+\text{p4}+\text{p5}+\text{p6}+\text{p7})$
t5	$\frac{1}{D_{\text{CPU,lrel}}} \times \#(\text{p5})/\#(\text{p1}+\text{p2}+\text{p3}+\text{p4}+\text{p5}+\text{p6}+\text{p7})$
t6	$\frac{2}{D_{\text{CPU,lrel}}} \times \#(\text{p6})/\#(\text{p1}+\text{p2}+\text{p3}+\text{p4}+\text{p5}+\text{p6}+\text{p7})$
t7	$\frac{1}{S_{\text{CPU}}} \times \#(\text{p7})\#(\text{p1}+\text{p2}+\text{p3}+\text{p4}+\text{p5}+\text{p6}+\text{p7})$
t11	$\frac{1}{S_{\text{dm}}}$
t12	$\frac{1}{D_{\text{lock}}}$

TABLE 7. Transition probability functions

Transition	*Probability function*
t1to3	P_{g1}
t1to12	$1 - P_{\text{g1}}$
t2to3	P_{g2}
t2to4	$1 - P_{\text{g2}}$
t4to6	P_{d}
t4to12	$1 - P_{\text{d}}$
t7to2	$1 - P_{\text{exit}}$
t7to5	P_{exit}

TABLE 8. Arc multiplicity functions

Arc	*Arc multiplicity*
p12 → t12	#(p12)
t12 → p3	#(p12)

3.1. Description of the iterative SPN model

3.1.1. Places and transitions

There are two modeling concepts in our SPN model. First, we consider the nine places with timed transitions as system service centers in which transactions must get their requests serviced. Places p1, p2, p3, p4, p5, p6 and p7 are seven queuing centers for the CPU, p11 is a queueing center for the data manager, and p12 is an infinite service center with no queueing for holding blocked transactions. Unlike other places, place p12 is not a resource center (i.e. not for CPU or data manager). It can service all of its tokens with a service demand of D_{lock} because a transaction gets its lock in D_{lock} time regardless whether there are also other transactions waiting for their locks. On the other hand, all other places can only service their tokens one at a time since they model physical resources and are queueing centers. We note that in the SPN when a transition is fired, one or more tokens, depending on the *multiplicity* of the associated input arc, will be removed from the input place, and one or more tokens, depending on the *multiplicity* of the associated output arc, will be added to each output place. Therefore, to model the no queueing behaviour of place p12, we can define two arcs having multiplicity greater than 1, i.e. the input and output arcs of transition t12 both have multiplicity equal to the number of tokens in place p12. Table 8 defines the multiplicities of these two arcs in our iterative SPN model. All other arcs only have multiplicity of 1 (which is the default) to model the fact that only one transaction will be serviced at a time in a queueing center.

The second modeling concept concerns the fact that places p1 through p7 all request the service of the CPU. Since the extent of CPU sharing is reflected by the number of tokens in these places, we model this CPU sharing concept [22] by defining the transition rates of t1 through t7 as shown in Table 6 where #(p1) means the number of tokens (transactions) in place 1 and #(p1+ p2 + p3 + p4 + p5 + p6 + p7) means the total number of tokens in places p1 through p7. These transition rates must be defined this way to model the fact that the CPU service rate of each place (p1 through p7) is deteriorated by the ratio of the number of transactions in one place to the total number of transactions in all of the places simultaneously requesting the service of the CPU.

3.1.2. Life profile of a transaction

To better understand the SPN model, we can trace the

life profile of a transaction (a token) in the Petri net. To keep our discussion simple, when referring to the time needed by a transaction at a queueing center, we only mention the transaction's service demand: the waiting time and the inflated service demand are implicitly understood.

A token is initially put in place p1 to get its first lock with a service demand of $S_{CPU,lreq}$. At this point, one of the following two events can occur, i.e. with probability P_{gl} (via transition t1to3), it successfully gets its first lock, or with probability $1 - P_{gl}$ (via transition t1to12), it is blocked because another transaction owns the lock. In the former case (it gets its first lock), the token enters place p3 to set the lock. In the latter case, the token first enters place p12 to wait for another transaction to release the lock before entering place p3.

A transaction sets a lock in place p3 with a service demand of $S_{CPU,lset}$. Then it enters place p11 with a service time of S_{dm} to retrieve the data items controlled by the new lock. It subsequently enters place p7 with a service demand of S_{CPU} to do useful CPU computation based on the data items just retrieved. Then, one of two events can occur: with probability $1 - P_{exit}$ (via transition t7to2), it enters place p2 to request a subsequent lock, or with probability P_{exit} (via transition t7to5), it terminates successfully and therefore enters place p5 to release all of the locks it owns.

When a transaction requests its subsequent lock in place p2 with a service demand of $S_{CPU,lreq}$, one of two events can happen. Either it gets the lock with probability P_{g2} (via transition t2to3), or it is blocked with probability $1 - P_{g2}$ (via transition t2to4). In the former case, it enters place p3 and then follows the same flow pattern as previously discussed. In the latter case, it is blocked and enters place p4, and a deadlock detection algorithm is then executed to check whether the transaction is deadlocked with other transactions. The reason that the transaction may be deadlocked with other transactions at this point is that a transaction in place p2 already owns at least one lock while requesting a subsequent lock. After the deadlock detection algorithm is executed with an average time of $D^{continuous}_{CPU,deadlock}$, one of two events can happen, i.e. with probability P_d (via transition t4to6), the transaction is deadlocked with other transactions, in which case the transaction is aborted and the token enters place p6 to release all of the locks it owns, or with probability $1 - P_d$ (via transition t4to12), the transation is not deadlocked with others. In the latter case, the transaction is blocked and enters place p12 waiting for its lock to be released by another transaction.

When a transaction is in place p5 (the transaction terminates successfully), all of the locks it owns are released with a service demand of $D_{CPU,lrel}$. Then the token transits to place p1 so that a new transaction can immediately take its place. A similar situation occurs for a token in place p6 (the transaction is aborted due to deadlock) except that the average service demand is $D_{CPU,lrel}/2$ instead of $D_{CPU,lrel}$ because an aborted transaction on average owns only one half of the locks.

3.2. Reward assignments to states of the SPN

The state of the SPN is characterized by the distribution of tokens in the places, called a *marking* of the SPN. Initially, a number of tokens corresponding to the degree of multiprogramming is placed in place p1 to start the SPN execution, thus marking the initial state of the system (see Figure 1). Then, as tokens move from one place to another characterized by the distributions of the transition firing times and arc multiplicities of the SPN model, the system migrates from one state to another. Eventually, a steady state is established in which there exists a finite number of states each having a steady state probability.

3.2.1. Calculations of performance measures

In the following, we describe how to compute system performance measures by applying the concept of reward rate assignments [19].

- X and T_{int}: The time interval between which a transaction terminates or aborts, T_{int}, can be computed as in [10] by the reciprocal of the sum of the throughputs of terminating and aborting transactions, i.e.

$$T_{int} = \frac{1}{X + X_{abort}}$$

where X and X_{abort} represent the throughputs of terminating and aborting transactions respectively. These two quantities can be computed by associating *reward rates* with markings of the SPN model. Specifically, for computing the throughput of terminating transactions, X, we assign: (a) a reward rate equal to that of the rate function of t5 (see Table 6) to those markings in which place p5 is non-empty and (b) a reward rate of 0 to all other markings. Then, the average throughput of t5, representing the average throughput of terminating transactions, can be computed as the expected reward rate weighted by marking probabilities. For computing the throughput of aborting transactions due to the deadlock, X_{abort}, a similar reward rate assignment is used, except that t5 is replaced with t6. Once T_{int} is obtained this way in one run, it can be used as an input to the iterative SPN model for the next run until the difference of its values in two successive runs is within 1%. It should be noted that we only need to specify the reward rate assignments as part of the SPN description [17]. The solving of the steady state reward rates is automatically performed by SPNP which first converts the Petri net description into a continuous Markov chain [19] and then solves the Markov chain numerically. Once X is obtained, the average response time of terminating transactions can be computed based on

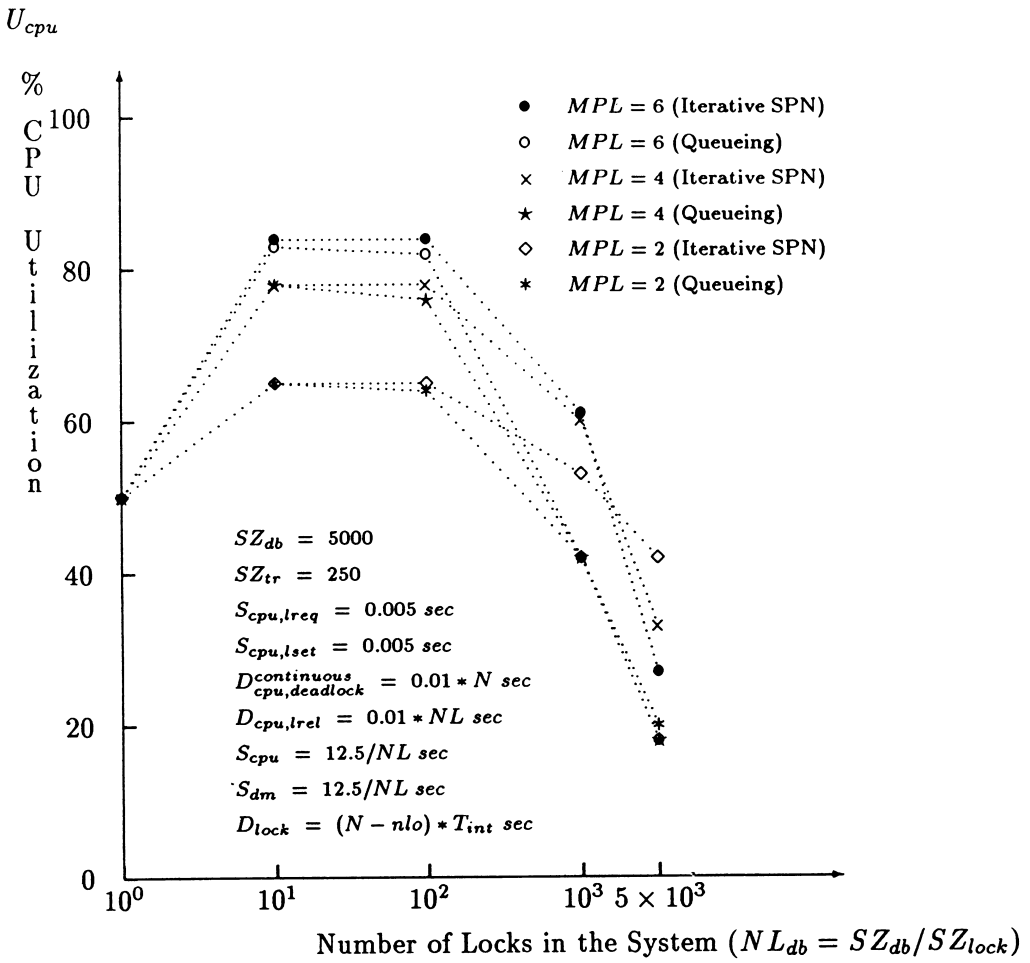

FIGURE 2. Comparing queueing and iterative SPN models for large transaction size.

Little's law [22], i.e. $R = N/X$ where MPL is the multiprogramming level of the system.

- U_{CPU}: The useful CPU utilization of the database system can be computed as a model output by assigning a reward of #(p7)/#(p1 + p2 + p3 + p4 + p5 + p6+ p7) to those markings in which place p7 is non-empty and 0 otherwise. Similar to X, U_{CPU} is computed as the expected reward weighted by marking probabilities.

3.3. Results and comparison

Figures 2 and 3 show the model output for U_{CPU} as a function of $NL_{db} = SZ_{db}/SZ_{lock}$ for $SZ_{tr} = 250$ and $SZ_{tr} = 5$, respectively, with MPL varying from 2 to 6 (MPL from 2 to 6 is chosen to allow comparison to the result reported in [10]). The far left side of these figures represents the case in which the whole database has only one granule and thus there is only one lock in the system, while the far right represents that each data item is a separate granule itself and thus has its own separate lock. Figures 2 and 3 match remarkably well with the data reported in [10] (labeled with 'Queueing' in Figures 2 and 3). There are two results implied in Figures 2 and 3. The first result concerns the effect of locking on system performance. At the one extreme (far right in Figures 2 and 3) where each data item is a granule, the probability of deadlocks among transactions is very high because each transaction must obtain a large number of locks (250 locks for each transaction in Figure 2). The consequence is that most transactions are aborted and must later be restarted, resulting in a low U_{CPU}. At the other extreme (far left in Figures 2 and 3), where the whole database has only one lock or just a few locks, only one or two transactions are allowed to execute at a time and U_{CPU} is low due to a very low level of concurrency. The optimal point therefore exists somewhere between these two extremes.

The second result concerns the effect of transaction size. Figures 2 and 3 show that, when SZ_{tr} is smaller U_{CPU} is less sensitive to the size of a granule. This implies that when the transaction size SZ_{tr} is small, the probability of deadlocks is low even when each data item has its own separate lock. In fact, Figure 3 shows that when SZ_{tr} is small, U_{CPU} is higher when each data item requires a lock (far right) than when there are only a few locks for the entire database (far left), due to a higher level of transaction concurrency.

4. NON-ITERATIVE PETRI NET: REMOVING D_{lock} AS INPUT

In the last section, we developed an iterative Petri net

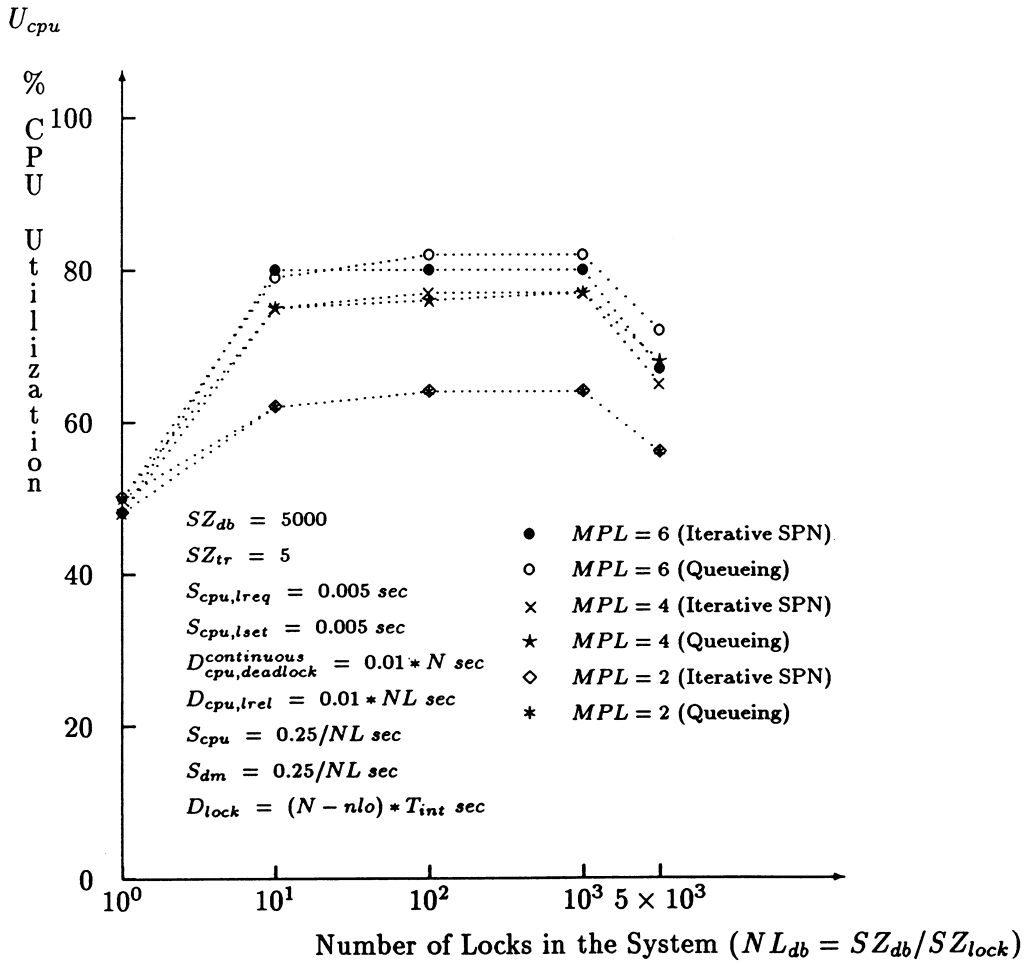

FIGURE 3. Comparing queueing and iterative SPN models for small transaction size.

model with a computational procedure similar to that used in [10] for their queueing network model, i.e. D_{lock} is required as a model input. The difference is that our model does not require any special solution technique — SPNP was the software package used for generating the data in Figures 2 and 3. In this section, we further refine our iterative SPN model so that D_{lock} is no longer required as a model input.

The assumption used for computing D_{lock} in our iterative SPN model (adopted from [10]) is that in the steady state, when a transaction terminates or aborts, a blocked transaction will be unblocked so that the number of lock owners in the system remains the same. This assumption in general is not justified because a blocked transaction may or may not be a lock owner itself.

With Petri net modeling, we can do a more precise modeling of the behavior of a blocked transaction. We first remove the transition t12 from the iterative SPN so that D_{lock} is no longer needed as an input parameter. We then create a new place p10 to hold only blocked, non-lock-owner transactions, that is, those that are still waiting for their first locks to be released. Blocked, lock-owner transactions, as before, are held in place p12. The purpose of creating place p10 in the non-iterative SPN model is to differentiate blocked, non-lock-owner transactions from blocked, lock-owner ones, thereby providing a more precise information on the number of lock owners NLO in the system as the system evolves over time. With the addition of place p10, NLO can be dynamically determined as

$NLO = MPL$ – (total number of tokens in places p1 and p10).

The behavior of a blocked, lock-owner transaction is modeled as follows. Whenever a transaction terminates or aborts, we allow a token in place p12, if any, to migrate to place p3 with an unblocking probability that is computable as a function of the number of lock owners in the system. This unblocking probability varies on-the-fly as a token in place p12 is considered at a time. Let P'_{12} (P''_{12}) be this unblocking probability for a token in place p12 when a transaction terminates (aborts, respectively). Then

$$P'_{12} = \frac{NL}{NL + (NLO - 2)\dfrac{\lceil NL \rceil}{2}}$$

and

$$P''_{12} = \frac{\dfrac{\lceil NL \rceil}{2}}{\dfrac{\lceil NL \rceil}{2} + (NLO - 2)\dfrac{\lceil NL \rceil}{2}} = \frac{1}{NLO - 1}$$

where, for each probability, the numerator stands for the number of newly released locks by a terminated (aborted) transaction and the denominator stands for the number of locks owned by all transactions in the system except for those owned by the blocked transaction being considered and the transaction that just terminated (aborted, respectively). The unblocking probability is therefore the probability that one of the locks released is the one awaited by a blocked, lock-owner transaction in place p12. (It is possible to do an even more precise calculation of these unblocking probabilities by updating the number of released locks available to a blocked transaction (the numerator term) by conditioning on the probability that some released locks are already allocated to other blocked transactions. However, this would increase the complexity of the model. We plan to study that effect in the near future.) It should be noted that the number of lock owners, NLO, is equal to MPL minus the number of tokens in places p1 and p10, and is dynamically computed by the SPN.

The behavior of a blocked, non-lock-owner transaction waiting in place p10 can also be modeled in a similar way. Let P'_{10} (P''_{10}) be the unblocking probability for a token in place p10 when a transaction terminates (aborts, respectively). Then

$$P'_{10} = \frac{NL}{NL + (NLO - 1)\frac{\lceil NL \rceil}{2}}$$

and

$$P''_{10} = \frac{\frac{\lceil NL \rceil}{2}}{\frac{\lceil NL \rceil}{2} + (NLO - 1)\frac{\lceil NL \rceil}{2}} = \frac{1}{NLO}$$

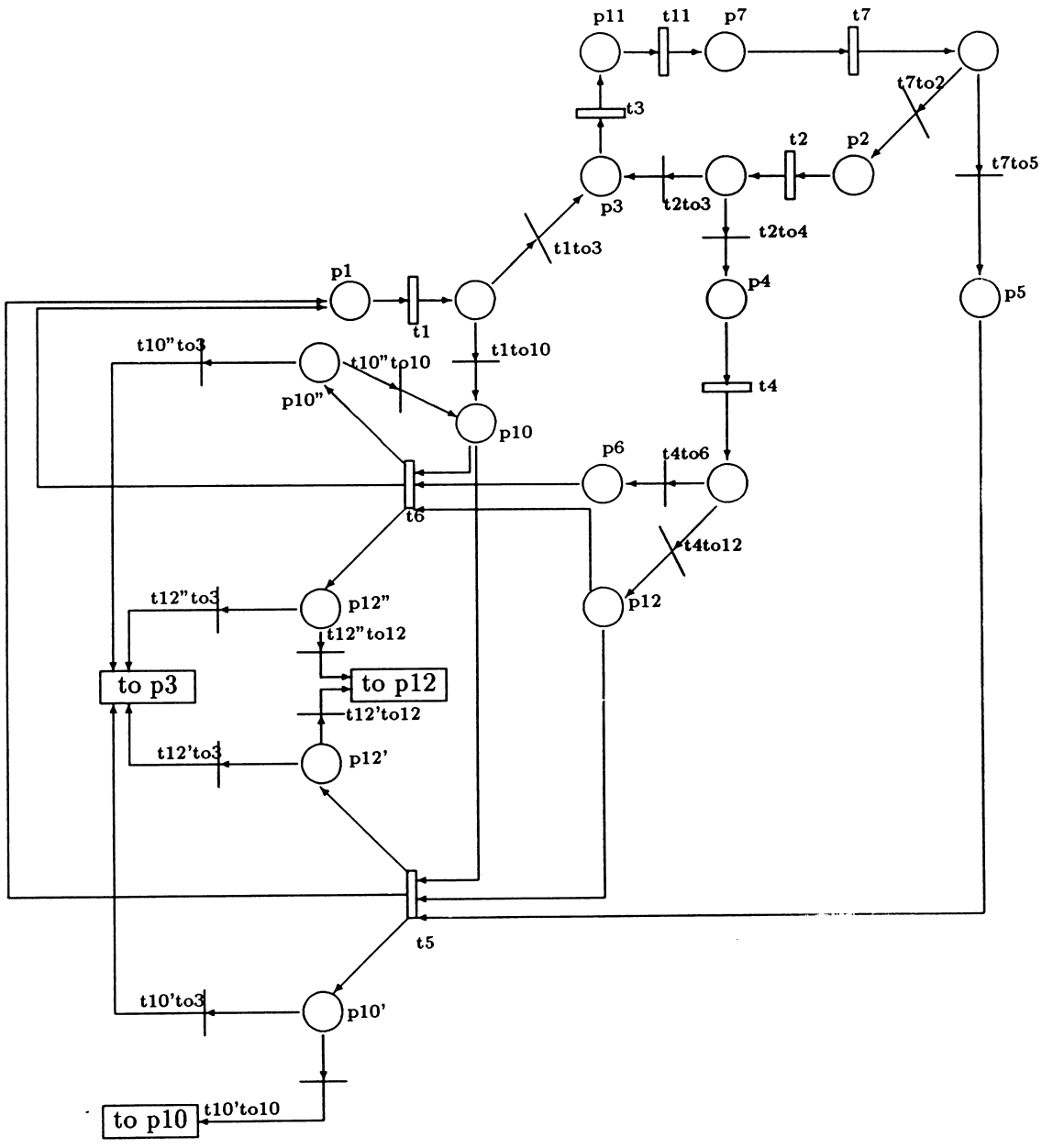

FIGURE 4. A non-iterative Petri net model for continuous deadlock detection.

TABLE 9. Transition probability functions

Transition	*Probability function*
t1to3	P_{g1}
t1to12	$1 - P_{g1}$
t2to3	P_{g2}
t2to4	$1 - P_{g2}$
t4to6	P_d
t4to12	$1 - P_d$
t7to2	$1 - P_{exit}$
t7to5	P_{exit}
t12to3′	P'_{12}
t12to12′	$1 - P'_{12}$
t12to3″	P''_{12}
t12to12″	$1 - P''_{12}$
t10to3′	P'_{10}
t10to10′	$1 - P'_{10}$
t10to3″	P''_{10}
t10to10″	$1 - P''_{10}$

TABLE 10. Arc multiplicity functions

Arc	*Arc multiplicity*
p12 → t5	#(p12)
t5 → p12′	#(p12)
p12 → t6	#(p12)
t6 → p12″	#(p12)
p10 → t5	#(p10)
t5 → p10′	#(p10)
p10 → t6	#(p10)
t6 → p10″	#(p10)

where the denominator term is changed to reflect the fact that a blocked, non-lock-owner transaction does not hold any lock at all, and therefore it has a higher unblocking probability than a blocked, lock-owner transaction which itself holds $\lceil NL \rceil / 2$ locks.

Figure 4 shows this non-iterative SPN model incorporating the modeling concepts described above. Compared with Figure 1, transaction t12 and its associated input and output arcs are removed from the net description in Figure 4. Two new input arcs from places p12 and p10 to transition t5, and two new output arcs from transition t5 to places p12′ and p10′, respectively, are added. This is to model that when a transaction terminates from place p5 through t5, a blocked, lock-owner transaction in place p12 (a blocked, non-lock-owner transaction in place p10, respectively) can be unblocked and migrates to place p3. Similarly, two new input arcs from places p12 and p10 to transition t6, and two new output arcs from transition t6 to p12″ and p10″, respectively, are added so that when a

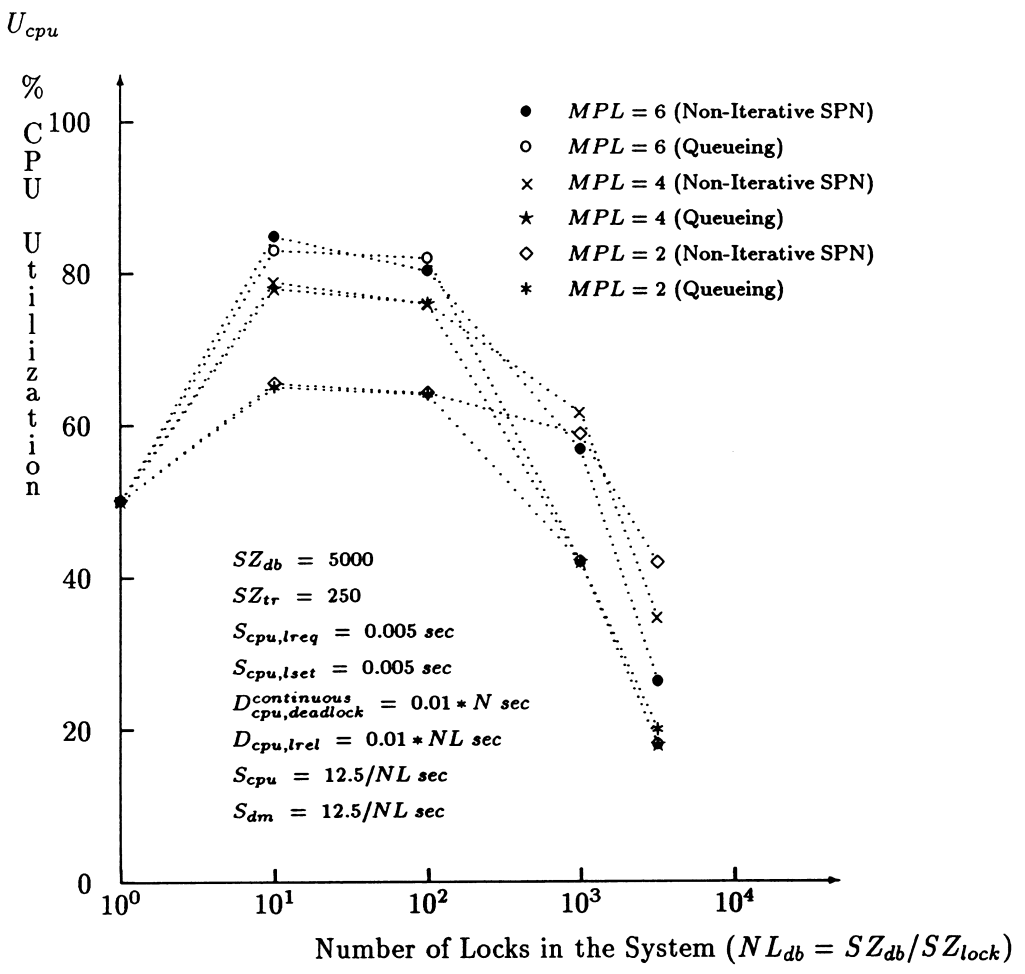

FIGURE 5. Comparing queueing and non-iterative SPN models for large transaction size.

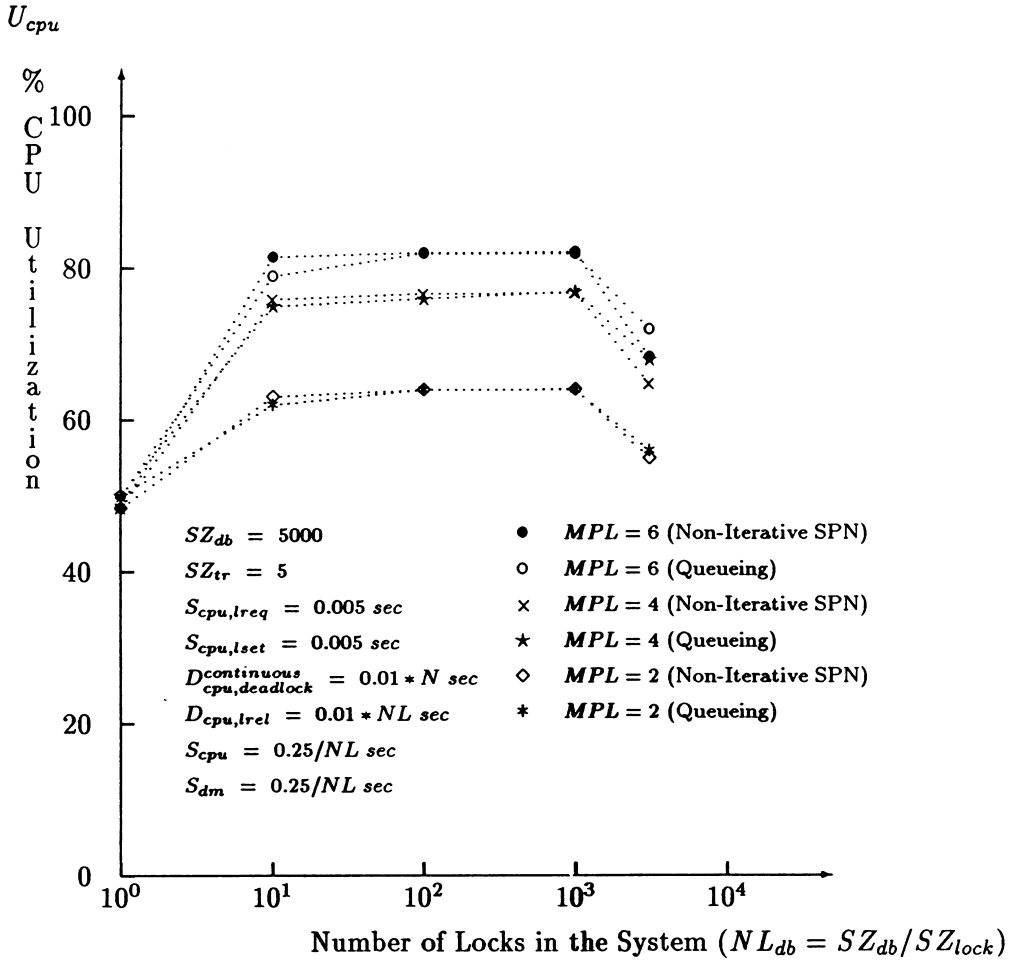

FIGURE 6. Comparing queueing and non-iterative SPN models for small transaction size.

transaction aborts, a blocked, lock-owner transaction in place p12 (a blocked, non-lock-owner transaction in place p10, respectively) can be unblocked and migrates to place p3. The unblocking probabilities of these blocked transactions in places p12′, p12″, p10′ and p10″, are P'_{12}, P''_{12}, P'_{10} and P''_{10}, respectively. Table 9 gives the transition probabilities associated with the non-iterative SPN model. It replaces Table 7.

Another modeling concept of this non-iterative Petri net is noteworthy. Because the state of the system evolves over time, when a transaction terminates or aborts, places p12 and p10 may not necessarily contain a transaction to be unblocked. This situation is modeled by allowing the eight new arcs to have multiplicity equal to the number of tokens in their input places. This allows transition t5 (t6) to fire as long as there is a token in place p5 (p6, respectively) regardless of whether there is a token in place p12 or p10. However, if at the moment when a transaction terminates or aborts, place p12 or p10 is not empty, then the blocked transactions in these two places can be unblocked and migrated to place p3 based on their individual unblocking probabilities computed dynamically. Table 10 defines the multiplicities of these eight new arcs in the non-iterative SPN. It replaces Table 8. Again, all other arcs in the modified SPN have a multiplicity of 1.

Figures 5 and 6 show how the non-iterative SPN fares, for large and small transaction sizes, respectively, when its outputs are compared with those by a queueing network model [10]. The time needed to generate a data point in these figures is reduced by a factor of about 5 when compared with the iterative SPN model developed in Section 3. For all arbitrarily selected database environment settings that we have tested, the outputs generated by the non-iterative SPN model correlate well with those by the iterative SPN model, except that a small discrepancy is observed when the number of locks in the system is very large, i.e. when each data item requires a lock. We therefore conclude that the non-iterative SPN model can greatly improve solution efficiency.

5. NON-ITERATIVE SPN FOR 2PL WITH PERIODIC DEADLOCK DETECTION

In this section, we model 2PL with periodic deadlock detection. With periodic deadlock detection, the system does not check for deadlocks whenever a transaction's subsequent lock request is not granted. Rather, the system checks deadlocks only periodically and, when it does so, it detects and breaks *all* deadlocks.

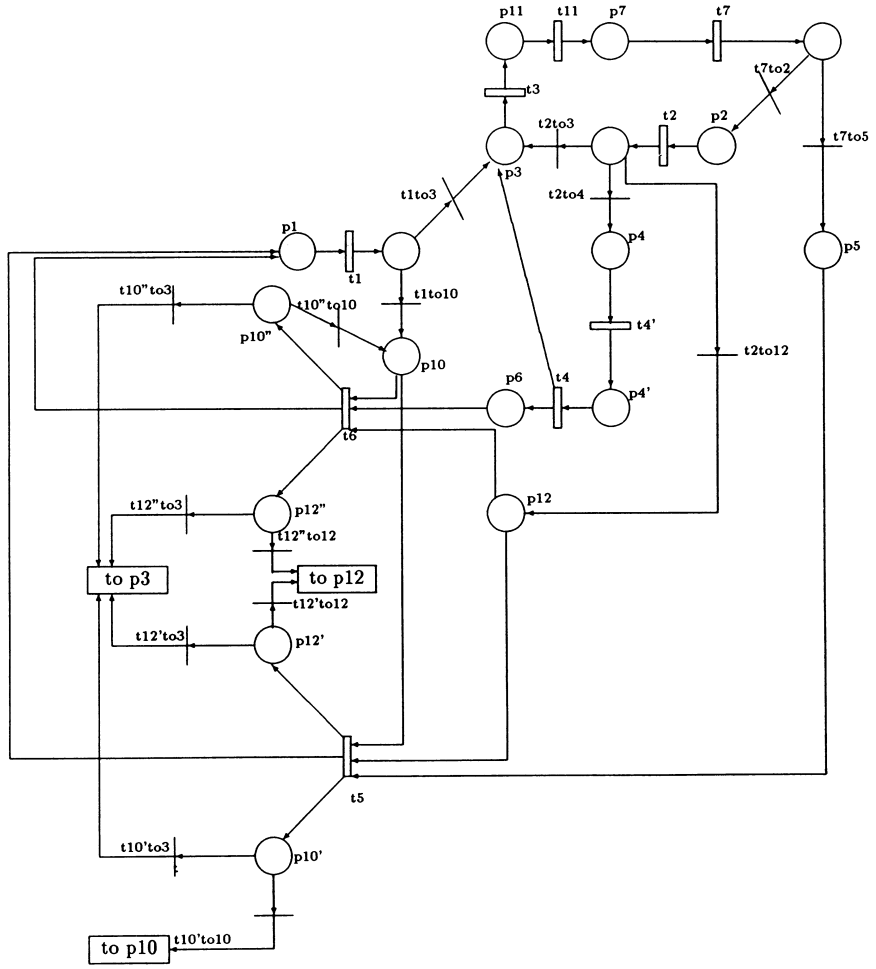

FIGURE 7. An SPN model for 2PL with periodic deadlock detection.

5.1. Model

Figure 7 shows the periodic SPN model. There are several new modeling concepts. In the following, we illustrate these concepts by explaining the differences between the periodic and continuous SPN models.

- The CPU time required to execute a periodic deadlock detection algorithm (e.g. Warshall algorithm for transitive closure [24]) is of complexity $O(N^2)$, rather than just $O(N)$ as in the continuous case. This fact is reflected by defining $D^{\text{periodic}}_{\text{CPU,deadlock}} = N \times D^{\text{continuous}}_{\text{CPU,deadlock}}$.
- When a transaction in place p2 requests a subsequent lock, there are three possible transitions instead of two, i.e. (a) the lock is granted and thus the transaction goes to place p3; (b) the lock is not granted because the transaction is deadlocked with other transactions, in which case the transaction goes to place p4; and (c) the lock is not granted because another transaction is holding the lock, in which case the transaction goes to place p12. These three state transitions (`t2to3`, `t2to12` and `t2to4` in Figure 7) occur with probabilities of P_{g2}, $(1 - P_{g2})(1 - P_d)$ and $(1 - P_{g2})P_d$, respectively. Table 10 gives the transition probabilities of the periodic SPN model. It is the same as Table 9 for the continuous SPN model except that `t4to6` is eliminated and `t2to4` and `t2to12` have different probability functions.
- Deadlocks are checked only periodically. Therefore, deadlocked transactions in place p4 will stay there for a time period until a deadlock detection algorithm is executed. This is modeled by associating a transition rate of $1/T_{\text{period}}$ with `t4'`, where T_{period} stands for the average time period (in sec) between two successive executions of the deadlock detection algorithm. After a period of T_{period} elapses, all deadlocked transactions in place p4 then migrate to place p4′ at which a periodic deadlock detection algorithm is then executed with a CPU service demand of $D^{\text{periodic}}_{\text{CPU,deadlock}}$ (via `t4`). Then, one half of the transactions are aborted and go to place p6, while one half of the transactions get their locks and go to place p3 (based on the assumption that deadlocks are of cycle 2).
- The execution of a periodic deadlock detection algorithm is a single CPU task, regardless of the number of deadlocked transactions waiting to be resolved in p4′. As a result, the rate functions of `t1` through `t7` are changed as shown in Table 11. This is

TABLE 11. Transition rate functions

Transition	*Rate function*
t1	$\frac{1}{S_{\text{CPU,lreq}}} \times \#(\text{p1})/(\#(\text{p1}+\text{p2}+\text{p3}+\text{p5}+\text{p6}+\text{p7})+n)$
t2	$\frac{1}{S_{\text{CPU,lreq}}} \times \#(\text{p2})/(\#(\text{p1}+\text{p2}+\text{p3}+\text{p5}+\text{p6}+\text{p7})+n)$
t3	$\frac{1}{S_{\text{CPU,lset}}} \times \#(\text{p3})/(\#(\text{p1}+\text{p2}+\text{p3}+\text{p5}+\text{p6}+\text{p7})+n)$
t4	$\frac{1}{D^{\text{periodic}}_{\text{CPU,deadlock}}} \times 1/(\#(\text{p1}+\text{p2}+\text{p3}+\text{p5}+\text{p6}+\text{p7})+1)$
t5	$\frac{1}{D_{\text{CPU,lrel}}} \times \#(\text{p5})/(\#(\text{p1}+\text{p2}+\text{p3}+\text{p5}+\text{p6}+\text{p7})+n)$
t6	$\frac{2}{D_{\text{CPU,lrel}}} \times \#(\text{p6})/(\#(\text{p1}+\text{p2}+\text{p3}+\text{p5}+\text{p6}+\text{p7})+n)$
t7	$\frac{1}{S_{\text{CPU}}} \times \#(\text{p7})/(\#(\text{p1}+\text{p2}+\text{p3}+\text{p5}+\text{p6}+\text{p7})+n)$
t11	$\frac{1}{S_{\text{dm}}}$

$n = 1$ if $\#(\text{p4}') > 0$; else $n = 0$.

due to the fact that (deadlocked) transactions in place p4′ are not being processed one at a time as in other CPU places, i.e. p1–p3 and p5–p7.

- In addition to the eight arcs in Table 10, the periodic SPN model has five more arcs that have multiplicity not equal to 1. Table 13 lists the arc multiplicity functions of the periodic SPN model. One modeling concept that is noteworthy concerns the multiplicity functions of arcs p4 → t4′, t4′ → p4′ and p4′ → t4. While a deadlock detection time interval (via t4′) must elapse even if place p4 contains no token, the execution of a periodic deadlock detection algorithm (via t4), on the other hand, must be executed sequentially following the deadlock detection interval. Consequently, transition t4′ must be inhibited when the deadlock detection algorithm is being executed via transition t4. To model this behavior: (a) the arc multiplicity of p4 → t4′ is the same as the number of tokens in place p4; (b) the arc multiplicity of t4′ → p4′ is equal to the number of tokens in place p4 plus 1; (c) the arc multiplicity of p4′ → t4 is equal to the number of tokens in place p4′ minus 1; and (d) t4′ is disabled as long as there is at least one token in place p4′. This modeling technique allows the elapse of a deadlock detection interval and the execution of the deadlock detection algorithm to occur in a sequential and cyclic manner, even when p4 contains no tokens.

TABLE 12. Transition probability functions

Transition	*Probability function*
t1to3	P_{g1}
t1to10	$1 - P_{\text{g1}}$
t2to3	P_{g2}
t2to4	$(1 - P_{\text{g2}})P_{\text{d}}$
t2to12	$(1 - P_{\text{g2}})(1 - P_{\text{d}})$
t7to2	$1 - P_{\text{exit}}$
t7to5	P_{exit}
t12′to3	P'_{12}
t12′to12	$1 - P'_{12}$
t12″to3	P''_{12}
t12″to12	$1 - P''_{12}$
t10′to3	P'_{10}
t10′to10	$1 - P'_{10}$
t10″to3	P''_{10}
t10″to10	$1 - P''_{10}$

TABLE 13. Arc multiplicity functions

Arc	*Arc multiplicity*
p4 → t4′	#(p4)
t4′ → p4′	#(p4) + 1
p4′ → t4	#(p4′) − 1
t4 → p6	$\lceil(\#(\text{p4}') - 1)/2\rceil$
t4 → p3	$\lfloor(\#(\text{p4}') - 1)/2\rfloor$
p12 → t5	#(p12)
t5 → p12′	#(p12)
p12 → t6	#(p12)
t6 → p12″	#(p12)
p10 → t5	#(p10)
t5 → p10′	#(p10)
p10 → t6	#(p10)
t6 → p10″	#(p10)

5.2. Comparison to continuous deadlock detection

We ran the periodic SPN model under various database environment settings from which we observed the following two results. First, for each database environment setting, we found that there indeed exists an

optimal periodic deadlock detection interval (between two successive executions of the deadlock detection algorithm) for which the system performance is optimized. Second, based on our model outputs, periodic deadlock detection can provide better system performance than continuous deadlock detection only when the deadlock probability is small (i.e. less contention) and the level of multiprogramming is low, in which case since transactions are rarely involved in deadlocks, the system is better off by breaking off rare deadlocks periodically. We also found that even when periodic deadlock detection at optimizing intervals is better than continuous deadlock detection, the improvement in system performance is often insignificant.

Figure 8 displays the model outputs for the system throughput as a function of the selection of the deadlock detection interval (T_{period}) and size of each transaction (SZ_{tr}) for the case when the multiprogramming level is 6 ($MPL = 6$) and the database size is 200. Other cases exhibit similar trends. Figure 8 shows that as SZ_{tr} increases, the contention of transactions increases and consequently the deadlock probability increases, in which case the system is better off by performing the deadlock detection more frequently. As a result, the optimal interval T_{period} shifts from 1000 to 5 sec as SZ_{tr} increases from 2 to 5.

Figure 9 compares two 2PL systems with periodic and continuous deadlock detections in terms of the throughput difference in the two systems. The y coordinate represents the throughput percentage improvement of systems using periodic deadlock detection at optimizing intervals over systems using continuous deadlock detection. We choose the y coordinate

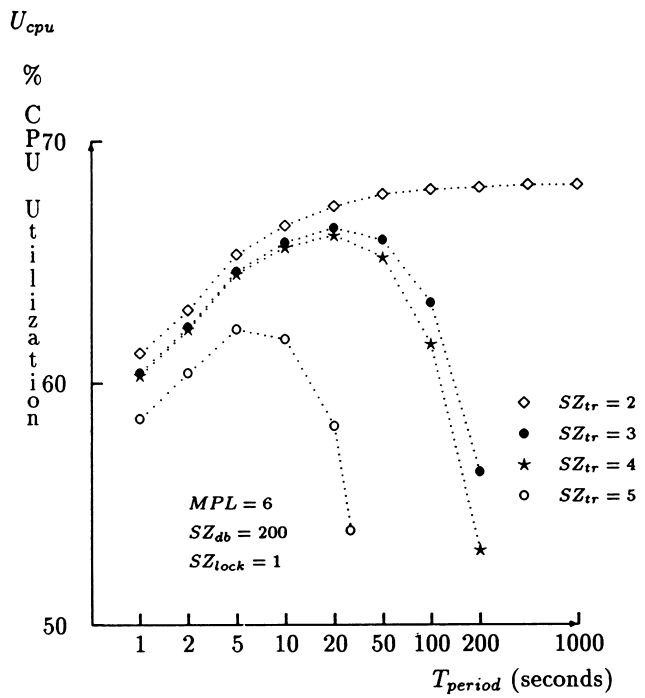

FIGURE 8. Optimizing time intervals under periodic deadlock detection.

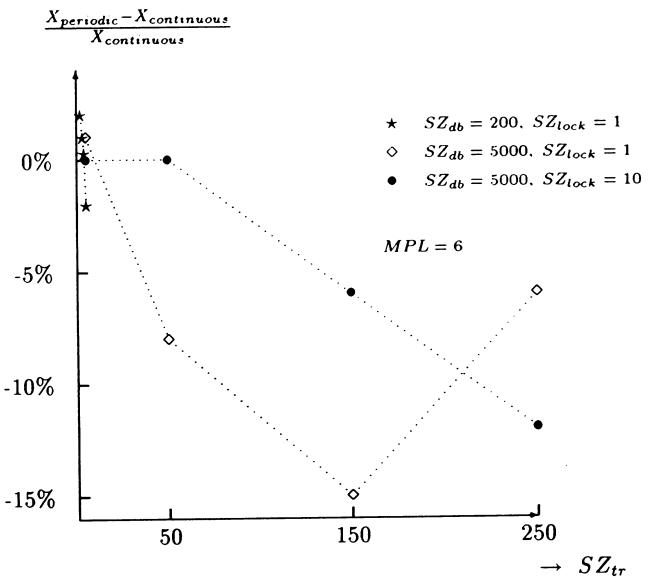

FIGURE 9. Difference in throughput percentage for periodic and continuous deadlock detection policies.

this way to ease the presentation of the results because different database environments may yield significantly different system throughputs possibly by an order of magnitude. The x coordinate represents the size of a transaction (the number of transaction is $MPL = 6$) to analyze the effect of transaction size. As can be seen in Figure 9, even for conditions under which periodic deadlock detection is better than continuous deadlock detection (e.g. when $SZ_{\text{db}} = 5000$ and $SZ_{\text{tr}} = 5$), the improvement in system throughput over continuous deadlock detection is relatively small. Conversely, for the conditions under which continuous deadlock detection is better than periodic deadlock detection, the difference in system throughput is much more noticeable. This result suggests that periodic deadlock detection may not improve system performance by too much in a centralized database system even at optimizing deadlock detection intervals possibly because the cost of continuous deadlock detection in centralized systems is relatively small (as compared to distributed database systems) and therefore for database environment settings for which there is a reasonable level of data contention (e.g. when $SZ_{\text{db}} = 5000$ and $SZ_{\text{tr}} = 50$ for $MPL = 6$ transactions), the system throughput can only be improved by resolving deadlocks as soon as possible by using continuous deadlock detection. Nevertheless, Figure 9 demonstrates that when the deadlock probability is low, systems with periodic deadlock detection can still perform better than systems with continuous deadlock detection, although the improvement in performance is less significant. In general, the SPN models developed in the paper can help a system designer determine the conditions under which periodic deadlock detection can perform better than continuous deadlock detection.

6. SUMMARY

In this paper, we have developed Petri net models to analyze the behavior of 2PL database systems with continuous and periodic deadlock detection. Our objective is to simplify the computational procedure required for obtaining system performance measures as model outputs. As opposed to existing performance models, our Petri net model can be defined and solved easily by using a commercial software product such as SPNP which has been used for generating the data in this paper. An important feature of our Petri net model is that the wait time for a lock can be implicitly described in the Petri net definition. This eliminates the need to use an iterative procedure to ensure that the estimate of the wait time must eventually converge. We have demonstrated that this saving in computation time in our non-iterative SPN model (for 2PL with continuous deadlock detection) does not compromise solution accuracy by comparing its outputs with those reported in [10] based on a queueing model. Furthermore, because our SPN model computes the unblocking probability of a blocked transaction dynamically as a function of the system state without making any ad hoc assumption, it is likely that the behavior of a blocked transaction can be modeled more precisely.

For 2PL database systems with periodic deadlock detection, our analysis indicated that there indeed exists a best deadlock detection interval under which the system performance is optimized, and that periodic deadlock detection can perform better than continuous deadlock detection in 2PL database systems when the deadlock probability is low. We suggest that our SPN models be considered as a prediction tool to help determine the exact condition under which periodic deadlock detection is better than continuous deadlock detection or vice versa. A possible application of the tool is to use it to design a database system that can dynamically switch between continuous and periodic deadlock detection policies based on a priori knowledge on the workload distribution of the system in a time cycle (e.g. 24 h in a day with a distribution of peak and slow hours) so as to optimize the performance of the system.

The comparison result between periodic deadlock detection and continuous deadlock detection based on the model outputs suggests that continuous deadlock detection be used for database systems for which some contention of data items is expected. The reason is that the cost of continuous deadlock detection in centralized systems is small. One possible future research area is therefore to apply the modeling concepts developed in the paper to compare the performances of continuous and periodic deadlock detection algorithms in distributed 2PL database systems where the cost of continuous deadlock detection is high.

REFERENCES

[1] Bernstein, P. A., Shipman, D. W. and Wong, D. W. (1979) 'Formal aspects of serializability in database concurrency control.' *IEEE Trans. Software Eng.* **5**, 203–216.

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

[3] Eswaran, K. P., Gray, J. N., Lorie, R. A. and Traiger, I. L. (1976) 'The notions of consistency and predicate locks in a database system.' *Commun. ACM*, **19.**

[4] Franaszek, P. A. and Robinson, J. T. (1985) 'Limitations of concurrency in transaction processing.' *ACM Trans. Database Syst.*, **10**, 1–28.

[5] Franaszek, P. A., Robinson, J. T. and Thomasian, A. (1992) 'Concurrency control for high contention environment.' *ACM Trans. Database Syst.*, **17**, 304–345.

[6] Hsu, M. and Zhang, B. (1992) 'Performance evaluation of cautious waiting' *ACM Trans. Database Syst.* **17**, 477–512.

[7] Franaszek, P. A., Haritsa, J. R., Robinson, J. T. and Thomasian, A. (1993) 'Distributed concurrency control based on limited wait depth.' *IEEE Trans. Parallel Distributed Syst.* **4**, 246–264.

[8] Agrawal, R., Carey, M. J. and McVoy, L. W. (1987) 'The performance of alternative strategies for dealing with deadlocks in database management systems.' *IEEE Trans. Software Eng.*, **13**, 1348–1363.

[9] Irani, K. B. and Lin, H.-L. (1979) 'Queueing network models for concurrent transaction processing in a database system.' In *Proc. ACM SIGMOD Int. Conf. Management of Data*, pp. 134–142.

[10] Pun, K. H. and Belford, G. G. (1987) 'Performance study of two phase locking in single-site database systems.' *IEEE Trans. Software Eng.*, **13**, 1311–1328.

[11] Tay, Y. C., Goodman, N. and Suri, R. (1985) 'Locking performance in centralized databases.' *ACM Trans. Database Syst.*, **10**, 415–462.

[12] Thomasian, A. and Ryu, I. K. (1991) 'Performance analysis of two-phase locking' *IEEE Trans. Software Eng.*, **17**, 386–402.

[13] Thomasian, A. (1993) 'Two-phase locking performance and its thrashing behavior.' *ACM Trans. Database Syst.*, **18**, 579–625.

[14] Hartzman, C. S. (1989) 'The delay due to two-phase locking.' *IEEE Trans. Soft. Eng.*, **15**, 72–82.

[15] Kumar, V. (1990) 'Performance comparison of database concurrency control mechanisms based on two-phase locking, timestamping and mixed approachs.' *Information Sci.*, **51**, 221–261.

[16] Peterson, J. L. (1981) 'Petri Net Theory and the Modeling of Systems. Prentice-Hall, Englewood Cliffs, NJ.

[17] Thomasian, A. (1982) 'An iterative solution to the queueing network model of a DBMS with dynamic locking.' In *Proc. 13th Computer Measurement Group Conf.*, pp. 252–261.

[18] Ciardo, G., Muppala, J. K. and Trivedi, K. S. (1989) 'SPNP: stochastic Petri net package.' In *Proc. 3rd Int. Workshop on Petri Nets and Performance Models*, pp. 142–151, Kyoto, Japan.

[19] Muppala, J. K., Woolet, S. P. and Trivedi, K. S. (1991) 'Real-time systems performance in the presence of failures.' *IEEE Comp.*, May, 37–47.

[20] Sahner, R. A. and Trivedi, K. S. (1991) *SHARPE Language Description.* Duke University.

[21] Ries, D. R. and Stonebraker, M. (1979) 'Locking granularity revisited.' *ACM Trans. Database Syst.*, **4**, 210–227.
[22] Lazowska, E. D., Zahorjan, J., Graham, G. S. and Sevcik, K. C. (1984) *Quantitative System Performance: Computer System Analysis Using Queueing Network Models.* Prentice Hall, Englewood Cliffs, NJ.
[23] Dugan, J. B. *et al.* (1984) 'Extended stochastic Petri nets: applications and analysis.' In Gelende, E. (ed), *Performance 84*. Elsevier, Amsterdam.
[24] Sedgewick, R. (1988) *Algorithms*, 2nd edn. Addison Wesley, Reading, MA.
[25] Beeri, C. and Obermarck, R. (1981) 'A resource independent deadlock detection algorithm.'. In *Proc. 7th Int. Conf. on Very Large Data Bases*, pp. 166–178.
[26] Mitra, D. and Weinberger, P. J. (1984) 'Probabilistic models of database locking: solutions, computational algorithms and asymptotics.' *J. ACM*, **31**, 855–878.