Priority-Based Speculative Locking Protocols For Distributed Real-Time Database Systems

Jonas Y. Bambi

B.Sc., Daystar University, 2005

Thesis Submitted In Partial Fulfillment Of

The Requirements For The Degree Of

Master Of Science

In

Mathematical, Computer and Physical Sciences

(Computer Science)

The University Of Northern British Columbia

September 2008

Your file Votre référence
ISBN: 978-0-494-48777-8
Our file Notre référence
ISBN: 978-0-494-48777-8

ABSTRACT

In recent years, a number of concurrency control protocols have been proposed to improve performance within Distributed Real Time Database Systems. Speculative Locking (SL) allows parallelism and execution of multiple instances of conflicting transactions to improve performance in Distributed Database Systems. This research extends SL by proposing two concurrency control protocols: Preemptive Speculative Locking (PSL), and Priority Inheritance Speculative Locking (PiSL). PSL allows preemption and abortion of lower priority transactions when conflict occurs, whereas PiSL allows the lower priority transaction to inherit the priority of the blocked transaction and continue execution. Using a distributed real-time transaction processing simulator that supports nested transaction model, an extensive set of experiments have been conducted which demonstrate that both PSL and PiSL outperform SL. Among the two proposed protocols, PSL performs better when data contention and system load are high. The performance metrics include number of transactions that meet their deadlines, and resource utilization.

preempt and abort any lower priority transaction in case of lock conflict thus giving the higher priority transaction a chance to meet the deadline. PiSL, on the other hand, attempts to prevent any wasted work by avoiding preemption by a higher priority transaction. Instead, the lower priority transaction inherits the priority of the blocked transaction. This gives both transactions an opportunity to meet their deadline whenever possible.

Using a distributed real-time transaction processing simulator that supports nested transaction model, an extensive set of experiments have been conducted to study the proposed protocols under varying system configurations. Both of our proposed protocols have been shown to consistently outperform SL protocol. However, when comparing PSL to PiSL, PSL has been shown to outperform PiSL when data contention and system load are high, whereas PiSL has been shown to have a better performance when data contention and system load are relatively low.

Table of Content

List of Tables

List of Figures

Acknowledgements

First, I would like to thank God for giving me health, strength and patience to finish this work. Secondly, I would like to thank my supervisor, Dr. Waqar Haque, for his timely advice, directions, encouragement and patience through out the course of my research. I also would like to extend my gratefulness to my thesis committee, Dr. Charles Brown and Dr. Ronald Thring, for their time and interest. Finally, I would like to thank my family, here and back in Africa, for their love, support, encouragement and for believing in me.

Chapter 1

Introduction

With globalization, there is a push for an increase in global connectivity, integration and interdependence in the economic, social, technological, cultural, political and ecological spheres. In the technological sphere, multinational networked organizations' need for exchange of information has led to the development of applications that are heavily dependent on globally distributed and constantly changing data. In these applications, transactions must not only execute correctly, but also complete within a specific time frame. For example, in internet stock market applications, computers of a brokerage firm are linked to monitor different stock markets and conduct required trading operations. These computers manage huge amounts of information for stock market as well as client's accounts, which may reside at different sites, and require timely execution of transactions [1]. When a client asks for a stock price or requests a transaction, the system must not only respond in a short amount of time, but also should receive current and accurate market information and the balance of the client's account in order to satisfy both the client and the brokerage firm. Other similar applications include Computer Aided Design and Computer Aided Manufacturing (CAD/CAM), Massively Multiplayer Online Games (MMOG) [10], airline online booking systems, telecommunication systems, e-commerce systems and real time traffic navigation systems. Such applications introduce the need for distributed real-time database systems.

A Distributed Real-Time Database System (DRTDBS) is a collection of multiple, logically interrelated databases distributed over a computer network where transactions have an explicit timing constraint, usually specified in the form of a deadline. In such a system, data shared among transactions is spread over a computer network and transactions are considered successful when they commit within a specified time frame. Moreover, transactions may access the database concurrently and share data. This requires that they maintain the database's logical consistency in addition to their time constraint. This requires concurrency control and priority mechanisms to be in place.

Furthermore, to respond to the need of applications that rely on DRTDBS, a transaction model known as nested transaction has been widely adopted [1, 2, 3, 4, 5]. A nested transaction is considered as a hierarchy of subtransactions in which each subtransaction may contain other subtransactions, or contain atomic database operations (read or write) [4]. A flat transaction, on the other hand, contains only atomic database operations. Referring to the previous example of a stock market application where computers manage huge amounts of information, some subtransactions of the nested transaction will monitor the stocks while others deal with the client account. A failed subtransaction is re-executed without influencing other subtransactions [5]. In the case of a flat transaction, the whole transaction would be rolled back. More details on flat and nested transaction models will be presented in the next section.

A lot of work has been devoted to the study of concurrency control protocols and priority mechanism for DRTDBS. The objectives are to design protocols which can minimize the number of transactions that miss their deadlines while maintaining database consistency [6].

The goal of this thesis is to develop an efficient real time concurrency control mechanism for nested transactions in a DRTDBS.

1.1. Transaction

A transaction is a program/script/query that accesses and manipulates (reads or writes) data items in a database. In a database system, transactions must be processed reliably. To ensure this, a database system must guarantee the properties of Atomicity, Consistency, Isolation and Durability (ACID) for each transaction [7].

Atomicity requires that a transaction runs all its operations in order to have an effect on the database. If the transaction cannot run the totality of its operations, then the database will remain in an unchanged state. Consistency refers to the fact that a transaction is correct; when executed alone or executed with other correct transactions, a transaction should take the database from one consistent state to another. The Isolation property requires that intermediate results of a transaction are not visible to other transactions; transactions must see a consistent database at all times. Durability refers to the fact that transactions' results are permanent in the database once transactions commit, regardless of failures afterwards.

A single transaction might require several queries, each reading and/or writing information in the database. When this happens, it is important to be sure that the database is not left with only some of the queries carried out. For example, when transferring funds from one account to another, even though this process might consist of multiple individual operations (such as

debiting one account and crediting another); it is considered as a single transaction. The transfer of funds can be completed or it can fail for multiple reasons, however atomicity guarantees that one account will not be debited if the other is not credited as well. Consistency, on the other hand, ensures that this transfer of funds does not break the integrity constraint of the database. If an integrity constraint within the database states that all accounts must have a positive balance, then any transaction violating this rule must be aborted. Moreover, isolation guarantees that no operation outside the transaction can ever see the data in an intermediate state; a bank manager can see transferred funds on one account or the other, but never on both accounts– even if he/she runs his/her query while the transfer is still being processed. Finally, durability ensures that once the transfer of funds has been completed, the transaction will persist and cannot be undone, even in case of a system failure.

It is common in database systems to have several transactions run simultaneously and share data. It is important to ensure that these simultaneous transactions do not interfere with each other. One possible way to ensure non-interference of transactions running simultaneously and database integrity is to execute one transaction after another, i.e. serially. Since this is too restrictive, an interleaved or concurrent execution of transactions is more desirable; however, the final result is expected to be the same as that of a serial execution. This is called a serializable execution. Concurrency control algorithms are used to ensure serializable execution of transactions in a database system. Concurrency control mechanisms will be discussed further in the next section.

1.1.1. Flat Transaction vs Nested Transaction model.

Flat transactions are those that have a single start point and a single termination point. In this model, a transaction consists of a set of partially ordered atomic read and write operations. The flat transaction model has been widely used in the area of databases. However, the flat transaction model is unfit to support the requirements of complex and advanced database applications like internet stock trading systems, real time traffic navigation system and Computer Aided Design and Computer Aided Manufacturing(CAD/CAM) [1, 2]. This is due to the fact that these database applications are by nature long, complicated (in term of the complexity of operations involved in performing a transaction) and require access to data items at various network sites [1].

As an example, let us consider an online purchasing application that deals with internet purchasing activity. An internet purchasing activity consists of different steps or operations which include: selecting the product, providing payment information such as a credit card number, providing personal data so that the credit card can be authorized and providing an email address so that the company supplying the product can immediately confirm the customer's order. In the flat transaction model, the whole process of purchasing can be considered as a single transaction in which each operation must complete successfully in order to commit the transaction. If one of the operations fails, the whole transaction will be rolled back. However, if each operation was considered as a subtransaction of the main transaction, a failed operation can be easily restarted without affecting the whole transaction. For example, if the authorization process fails, the system should not abort the whole

transaction, but request the customer to provide the correct personal information. This has led to the concept of nested transactions [11].

A nested transaction extends the flat transaction by allowing a transaction to invoke a primitive operation or initiate a subtransaction [1]. The root transaction, which is not enclosed in any transaction, is called the top level transaction. Parent transactions are transactions that have subtransactions, known as their children. Transactions without any children are called leaf transactions. A nested transaction is modeled as a tree of transactions where subtransactions are either nested or flat transactions. As illustrated in Figure 1, Transaction T_1 represents the top level transaction or root transaction, $T_{1.1}$, $T_{1.2}$ and $T_{1.3}$ are the children of T_1. Transaction $T_{1.3}$ is the parent of $T_{1.3.1}$, $T_{1.3.2}$ and $T_{1.3.3}$. Transactions $T_{1.1}$, $T_{1.3}$ and $T_{1.1.1}$ are nested transactions, whereas $T_{1.2}$, $T_{1.3.1}$, $T_{1.3.2}$, $T_{1.3.3}$, $T_{1.1.1.1}$ and $T_{1.1.1.2}$ are flat transactions and represent leaf transactions. Also, superiors of a given subtransaction include all transactions on the path from the subtransaction to the root, not including the subtransaction itself. For example, transactions $T_{1.1}$ and $T_{1.1.1}$ are superiors of transaction $T_{1.1.1.1}$. Inferiors of a transaction are those transactions which are part of the subtransaction hierarchy spanned by the transaction, not including the transaction itself. As illustrated in Figure 1, transaction $T_{1.1.1.1}$ and $T_{1.1.1}$ are the inferiors of transaction $T_{1.1}$.

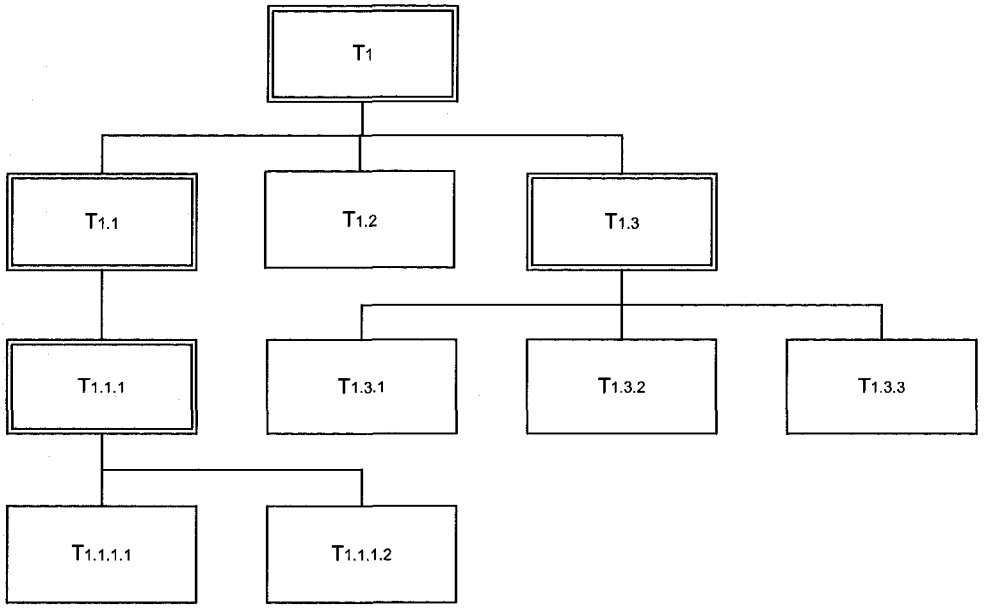

Figure 1: Nested Transaction Model

A top level transaction has all the ACID properties discussed earlier. Hence, top level transactions must be isolated from each other, and, in case of failure, they must be rolled back without side effects to other transactions. Subtransactions, on the other hand, appear atomic to each other and may commit or abort independently. A nested transaction is not allowed to commit until all its subtransactions have committed. However, if a subtransaction is aborted or fails, its parent is not required to abort. Instead, the parent is allowed to perform its own recovery. The possible choices of the parent includes the following: (1) retry the subtransaction, (2) initiate another subtransaction that implements an alternative action, (3) ignore the condition of failure, and (4) abort if the nested transactions do not have enough time to execute completely [1]. The commit of a transaction depends on the outcome (commit or abort) of its superior; even if a transaction commits, the abortion of one of its superiors will undo its effects. All updates of a transaction are considered permanent only when the enclosing top level transaction commits.

Unlike in the flat model where failure of any of the operations within a transaction result in the failure of the whole transaction, a subtransaction failure in a nested transaction model only affects itself and its inferiors. Therefore, the nested transaction model provides a powerful mechanism for both fine tuning the scope of roll backs and ensuring safe intra-transaction concurrency in applications with complex structures. These advantages make nested transaction models especially suitable for real-time distributed environments [1]. The model used in this research is the nested transaction model.

1.1.2. Real time transactions

Real time transactions are transactions which, on top of meeting the correctness requirement, must complete by a specified time, usually in the form of a deadline. Missing this deadline can seriously affect the usefulness of the completing transaction. Hence, with real time transactions, the goal is not only to maintain database consistency, but also to satisfy the transaction deadline. Furthermore, the number of transactions that commit before their deadline must be maximized.

Real time transactions can be divided into three categories: firm, hard and soft [7]. This is illustrated in Figure 2 [12] where firm, hard and soft transactions are compared to non real time transactions. As Figure 2 demonstrates, unlike non real-time transactions, real-time transactions must take transactions deadline into consideration. Depending on the category to

which a real time transaction belongs, missing its deadline may result in a catastrophe or may be ignored:

- Hard deadline transactions are those that must complete their work before their deadlines, otherwise the results can be catastrophic. One can say that a large negative value is imparted to the system if a hard deadline is missed. These are usually safety-critical activities, such as those that respond to life or environment-threatening emergency situations [12].
- Soft deadline transactions are transactions that can execute to completion regardless of their deadlines, but their contributed value to the system decreases as time progresses after the given deadline. Typically, their value drops to zero at a certain point past the deadline. For example, if components of a transaction are assigned a deadline derived from the deadline of the transaction, then even if a component misses its deadline, the overall transaction might still be able to make its deadline [12]. This illustrates the case of a nested transaction where subtransactions are assigned a deadline that is derived from the parent transaction.
- Firm deadline transactions are transactions that are useless once they expire their deadlines, and therefore are aborted and their work is undone (rolled back) if they reach their deadline before completing their work [1]. In a nested transaction environment, a top level transaction might have a firm deadline while its subtransactions have a soft deadline.

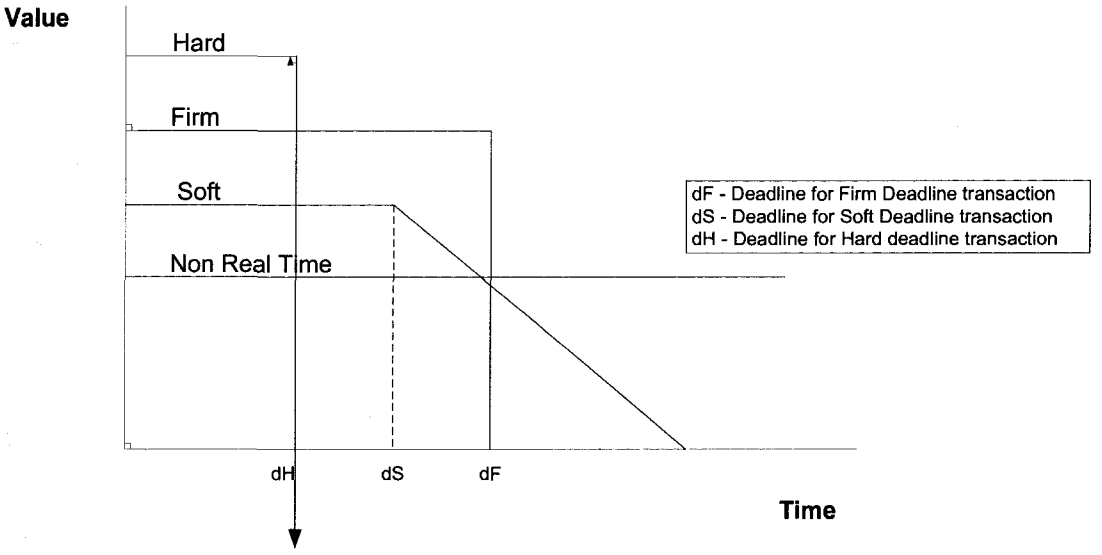

Figure 2: Types of real-time transactions

In order to maintain database consistency and satisfy transaction deadline, real time transactions are assigned priorities so that their access to critical resources (CPUs, disks and data items) can be ordered [7]. Giving priority to transactions determines which transactions should be executed first and which transactions can be safely blocked or restarted in case of a data conflict. Hence, in order to maximize the number of transactions that commit before their deadline, some transactions may be preempted. When two transactions are competing for the same data, the transaction with lower priority can be preempted. This gives the transaction with higher priority an opportunity to complete and release the conflicting data, giving the low priority transaction an opportunity to complete as well.

To illustrate the principle of priority, let us consider a set of transactions with release time ***r***, deadline ***d*** and runtime estimate ***E,*** and the data requirement as shown in Table 1. Transaction T_1 and T_2 both update item X. Therefore these transactions must be serialized. If

the earliest deadline is used to assign priority to transactions, then T_2 has the highest priority, followed by T_3 and finally T_1. Figure 3 and figure 4 [17] are used for this illustration. A time line is shown at the bottom of each figure and a scheduling profile is shown for each transaction. An elevated line means that the transaction is executing on the CPU. A lowered line means that the transaction is not executing. The cross hatching shows when the transaction has a lock on a data object. The cross hatching begins when the lock is granted and ends when the lock is released. Finally, the scheduler assumes that estimates are perfect and ignores the time required to make scheduling decisions or rollback transactions.

Transaction	***R***	***E***	***d***	***Update***
T_1	0	3.5	10	X
T_2	0.5	2	4	X
T_3	1	1.5	5	Y

Table 1: Transactions Data Requirement

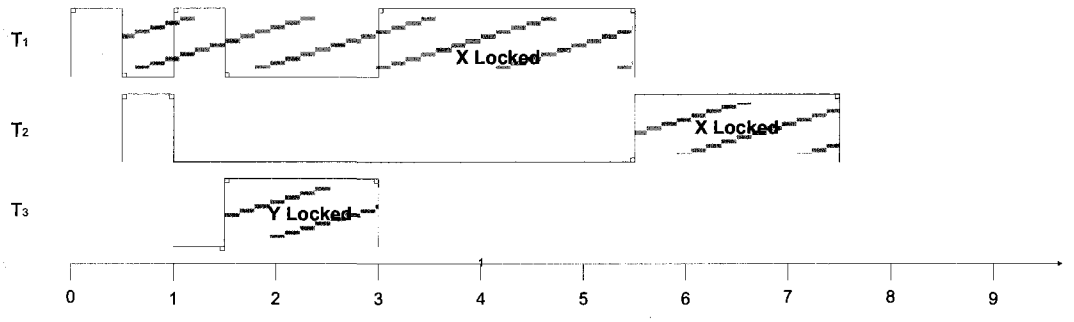

Figure 3: Real Time Transactions without Priority Policy.

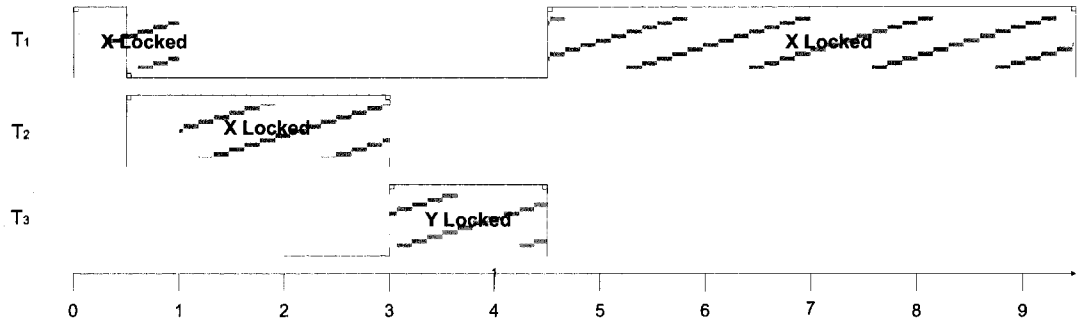

Figure 4: Real Time Transactions with Priority Policy.

In Figure 3, transaction T_1 is the only job in the system at time 0, so it gains the processor and executes until time 0.5, when transaction T_2 arrives. During this time, Transaction T_1 requests and gains an exclusive lock on data object X. At time 1, T_3 arrives in the system and starts executing. Although T_2 has a higher priority than T_1 and T_3, it is not given any special consideration. T_1, T_2 and T_3 timeshare the CPU and T_2 has to wait for T_1 to release the lock on X in order to access X. T_2 must wait for T_1 since the requesting transaction always waits for the lock holding transaction to finish and release its locks. This is true even when the requesting transaction has a very high priority. As a result T_2 misses its deadline. Only T_1 and T_3 meet their deadlines.

In Figure 4, on the other hand, an alternative approach is used. In this approach, one adopted by the High Priority policy, conflict is resolved in favour of the transaction with the higher priority. The favoured transaction, in this case T_2, is allowed to lock the contested object (X) and simultaneously, the lower priority transaction T_1 holding the lock is preempted. By preempting T_1, T_2 is given a chance to meet its deadline, and since T_1 has a long deadline it still meet its deadline as well. Hence, T_1, T_2 and T_3 all meet their deadlines.

1.2. Commit Protocol

In distributed database systems, commit protocols are used to ensure transactions atomicity. This is accomplished through an exchange of messages, in multiple phases, between the participating sites where the operations of a distributed transaction are executed. One of the protocols that are most widely used in practice is the Two Phase Commit (2PC) protocol [15]. In its basic form, 2PC proceeds in two phases: the voting phase and the decision phase [15, 18]. In the first phase, known as the voting phase, the site where the transaction was originally generated (generally called the coordinator) sends a PREPARE message to all the sites involved (participants) in the transaction, asking them to vote on whether to commit or abort the transaction. If, for any reason, any of the participants responds NO, the coordinator decides to roll back the local transaction and sends an ABORT message to all participants. Conversely, if all the received responses are YES, the coordinator then decides to commit the local transaction and informs all the participating sites by sending a COMMIT message. The phase when the coordinator informs the participant about its decision to commit or abort the transaction is the second phase, and it is known as the decision phase. A YES vote indicates that the local operations have been successfully executed, and the update could be made permanent or durable even if a failure occurs. A participant that votes NO can unilaterally abort, whereas a participant that votes YES must wait for the coordinator decision to commit or abort. The participants must also acknowledge the coordinator's decision. This protocol is illustrated in Figure 5 [18].

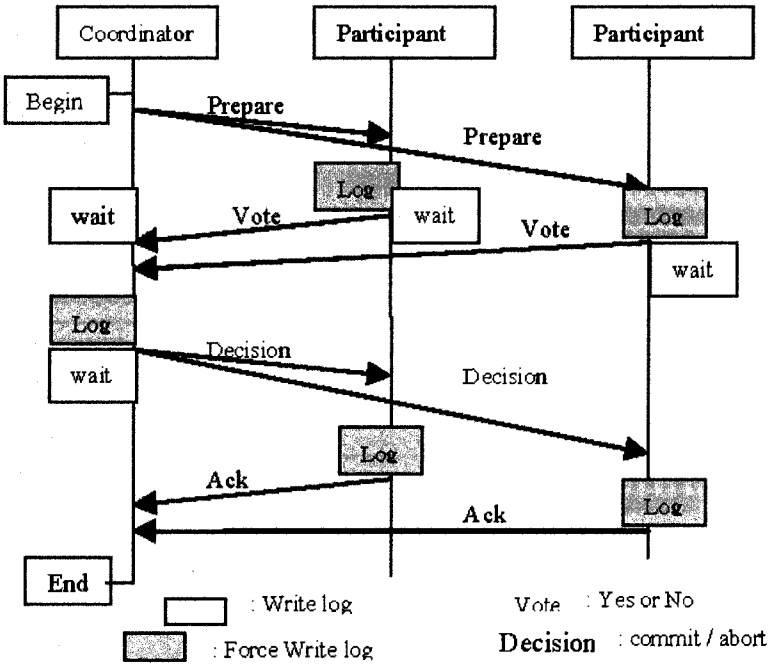

Figure 5: Two Phase Commit

In a nested transaction model, the site at which the top level transaction is committing acts as the coordinator and the rest of the sites at which subtransactions of the committing top level transaction have been executed act as participants. In this model, when a subtransaction commits, it releases its locks and transfers them to its parent. Hence, a subtransaction's commit protocol is different from that of a top level transaction [1]. As a subtransaction commits, it sends a READY TO COMMIT message to its parent's site. In the message, the subtransaction includes the list of all the locks it is holding. Upon receiving the message, for each lock of the subtransaction, the parent's site sends a message to the site holding the lock to inform it of the lock transfer. After the parent's site receives all the messages from its subtransactions, it assesses the messages. Upon completion of the parent transaction, and depending on the type of messages that came from all the subtransactions' sites, a COMMIT

or ABORT message will be sent to all the sites where the subtransactions are located. If it is a COMMIT message, then all the subtransactions will be committed; otherwise, they will all be aborted. Once the message is received by the subtransactions' sites, all the locks that were transferred from the subtransactions to the parent transaction are released. The protocols developed in this research utilize this commit model.

1.3. Concurrency Control

Concurrency control is an important mechanism for synchronizing a database access to maintain the database's consistency. Transactions operating on a database system must take the database from one database state to another without losing the database's correctness. As mentioned earlier, serializable execution ensures that transactions that run simultaneously do not interfere with each other. Concurrency control algorithms are used to guarantee serializable execution of transactions in a database system, while maintaining the integrity of the database. Most concurrency control algorithms fall into three categories: lock-based, timestamp-ordered and optimistic. Locking algorithms, especially the Two-Phase Locking (2PL) protocol, are the most widely used algorithms [19].

When two transactions are competing for the same data item, one solution is to make one transaction wait until the other transaction releases the data item common to both transactions. To ensure this, database systems provide locks to data items. If a transaction requests a lock on a data item and the lock has not been given to some other transaction, the requesting transaction can get the lock and keep it as long as the particular data item is being

operated upon. If the requested lock has been granted to some other transaction, then, the requesting transaction must wait. This mechanism ensures serializability within a database system.

To reduce a transaction's wait time, there are two types of locks that can be used, depending on whether a transaction wants to perform a read operation or a write operation on a data item. These are readlocks and writelocks [13]. In a readlock, a transaction locks the data item in a shared mode. Any other transaction intending to read the same data item can also obtain a readlock. In writelock, a transaction locks the data item in an exclusive mode. If one transaction wants to write on a data item, no other transaction may get either a readlock or a writelock on that data item. Regardless of the type of lock, after a transaction has finished operating on a data item, the transaction performs an unlock operation. This operation allows the data item to be made available to other transactions that might be waiting.

In the Two-Phase Locking protocol, all lock operations are required to precede any unlock operation. Hence, the execution of a transaction consists of two phases: the first phase, which is the locking phase, and the second phase, also known as the unlocking phase [8]. The first phase may also be considered as a growing phase in which locks are acquired but may not be released. By releasing the lock, a transaction is considered to have entered the second phase, also known as the shrinking phase. In the shrinking phase, locks are released but new lock may not be acquired. When the transaction terminates, all remaining locks are automatically released. The instance just before the release of the first lock is called the lockpoint [13]. Figure 6 illustrates these two phases as well as the lockpoint.

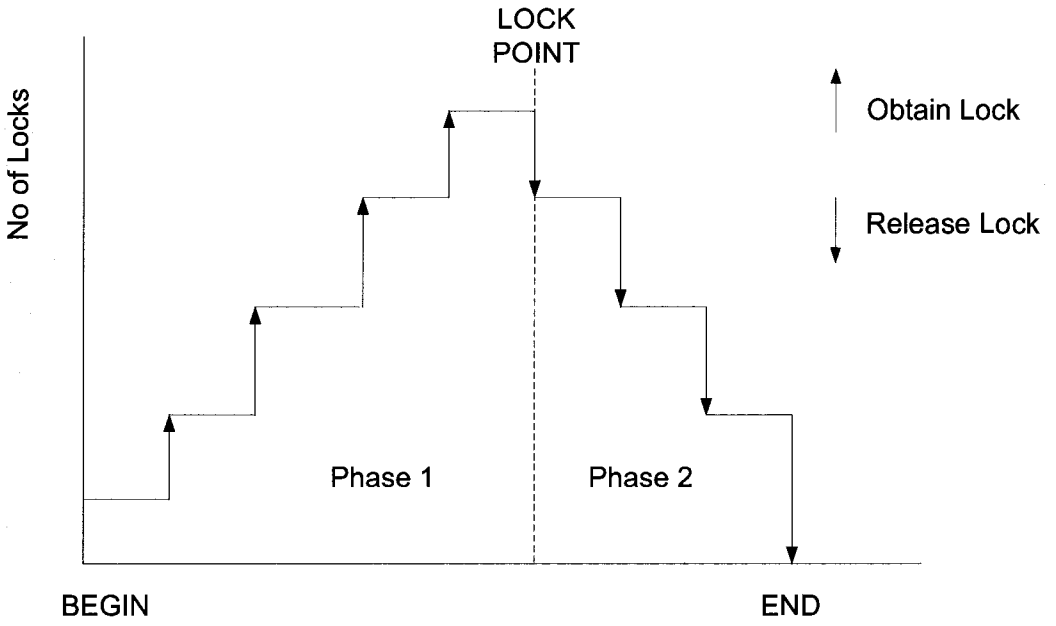

Figure 6: Two Phase Locking and Lock Point

There is a challenge of extending both the serializability argument and the concurrency control algorithms to the distributed execution environment [14]. In this environment, data is spread across multiple sites, and operations of a given transaction may execute at multiple sites. Hence, the serializability argument becomes more difficult to specify and enforce. To address this challenge, the 2PL protocol is combined with the 2PC protocol to create what is commonly known as the distributed two phase lock (Distributed 2PL) protocol [16]. More details about how Distributed 2PL works will be given in the next chapter.

In a real time environment where the execution of concurrent transactions is scheduled to meet timing constraints, transactions are assigned priorities according to their deadline. If 2PL is used as the concurrency control protocol for such transactions, a problem, known as priority inversion, arises [7]. Priority inversion occurs when a high priority transaction is

blocked by a lower priority transaction, which is an undesirable event. For example, let us consider two transactions T_H and T_R competing for the same data item. T_H is holding the lock and T_R is requesting for the lock held by T_H. According to 2PL principle, T_R has to wait until T_H releases the lock. If T_R is of higher priority than T_H, priority inversion occurs. One solution to this problem is to restart the lower priority lock holder, and let the higher priority lock requester get the lock. This protocol is called the High Priority Two-Phase Locking (HP-2PL) protocol [17].

1.4. Contribution

Distributed 2PL, HP-2PL and many other locking protocols, extending the basic 2PL protocol, have been proposed to address the issue of concurrency control in distributed database systems and real-time database systems. For example, real-time database systems require that transactions complete by a specified time. To meet this requirement, the basic 2PL protocol has been extended to take into consideration transactions priority. This resulted in the conception of the HP-2PL protocol.

The Speculative Locking (SL) protocol [9], one of the extensions of 2PL, is an efficient concurrency control protocol, especially in distributed environments. SL improves performance over 2PL by allowing more parallelism among conflicting transactions, without violating serializability criteria. To achieve this parallelism, SL allows a transaction to release its lock on a data item whenever the transaction produces corresponding after-images after its execution. The waiting transaction uses the before and after images of the

uncommitted data item, from its predecessor, to perform speculative executions, and retains one of the executions depending on the termination of the preceding transaction. This parallelism makes SL adequate for distributed environments. However, SL does not take into account transactions' time constraint, making it inadequate for distributed real-time database systems.

This thesis proposes an extension to SL by incorporating the time constraint requirement. It suggests two approaches to enhance SL and make it more adequate for a distributed real-time database environment, where nested soft deadline transaction model has been adopted:

1. The first approach, the Preemptive Speculative Locking (PSL) protocol, involves an integration of the priority driven preemptive scheduling approach with SL to increase the probability of transactions, in DRTDBS, to meet their deadlines. In PSL, a transaction with a higher priority is granted access to data items by preempting and restarting the lock holder transaction/subtransaction with a lower priority. By ensuring that higher priority transactions are not delayed by lower priority transactions, PSL reduces the number of transactions that miss their deadlines. However, in doing so, any work done by a lower priority transaction is lost whenever it is preempted and aborted to advance a higher priority transaction.

2. The second approach, the Priority Inheritance Speculative Locking (PiSL) protocol, attempts to prevent any waste of work that a transaction has already completed. With this approach, any transaction that has been granted access to a data item cannot be preempted regardless of its priority. Instead, PiSL uses the following approach: whenever a higher priority transaction is blocked by a lower priority transaction, the priority of the lower

priority transaction is raised to the priority of the blocked transaction. This ensures that the lower priority transaction completes its execution without being further interrupted and then passes the lock (data item) to the waiting higher priority transaction. With this approach, both the lower and the higher priority transactions get an opportunity to meet their deadlines whenever possible.

A detailed description of PSL and PiSL is provided in chapter 3.

1.5. Thesis Organization

The rest of this thesis is organized as follows: In chapter 2, several approaches to concurrency control in distributed real time databases are investigated. Their strengths and weaknesses, relative to the proposed concurrency control protocols, are discussed. In chapter 3, the proposed concurrency control protocols, PSL and PiSL, are explained in detail. Chapter 4 focuses on the simulation model and analysis of results. Finally, chapter 5 provides the summary of the thesis and outlines future directions.

Chapter 2

Literature Review

Concurrency control ensures that transactions within databases are executed in a correct and safe manner. This is done through mechanisms that coordinate operations within transactions that execute concurrently. Concurrent execution of transactions increases the probability of transactions' processes to interfere with each other, due to access to shared data items. Concurrency control mechanisms allow transactions in a database system to access shared data without interfering with each other. Many studies have been dedicated to concurrency control mechanisms, and various algorithms have been developed. This chapter examines several of these algorithms and their performance, as well as their limitations, in distributed real time database environments.

2.1. Locking vs Optimistic Concurrency Control Mechanisms

Due to its simplicity, Two Phase Locking (2PL) is one of the most commonly used concurrency control mechanisms within conventional Database Management Systems (DBMS) [19]. As previously explained, 2PL requires that each transaction issues lock and unlock requests in two phases, a growing phase and a shrinking phase. In the growing phase, a transaction may obtain locks but not release any lock, and in the shrinking phase, it may release locks but not obtain any new lock. This ensures serializability within DBMS.

Many variations of 2PL have been developed. For example, in Strict Two Phase Locking [20], also known as pessimistic 2PL, a data item is locked when there is an outstanding operation on it. If a lock request is denied, the requesting transaction is blocked until the lock is released. On the other hand, in optimistic based protocols, all locks are allowed to be granted when they are requested, based on the assumption that a conflict is rare [21]. Checking for conflict is preformed when the transaction wishes to commit. For example, Optimistic Two Phase Locking allows all the operations to be executed when they are requested, even if they are not compatible, with the assumption that conflicts between operations will not occur. However, to ensure consistency, Optimistic Two Phase Locking performs a check at the commit stage to confirm that all the operations assumed to be compatible are indeed compatible [20].

Both pessimistic and optimistic based concurrency control protocols provide an acceptable degree of concurrency [13]. However, through simulation studies, it has been shown in [20] that optimistic based concurrency control protocols are especially more effective when network latency is low or when there is a low level of concurrency. When network latency is high or when there is a medium or high level of concurrency, pessimistic based concurrency control protocols perform better. Also, when the rate at which transactions arrive is low, optimistic based concurrency control protocols perform better than pessimistic based concurrency control protocols [13]. Finally, in a system with a significant resource contention, pessimistic based concurrency control protocols perform better than optimistic based concurrency control protocols [26].

Following either the pessimistic or optimistic model, or a combination of both models, many algorithms have been developed to accommodate distributed or real time database environments. These algorithms will be examined in the following sections.

2.1.1. Distributed 2PL

For a distributed database environment, distributed 2PL combines the principles behind 2PL and 2PC. As mentioned earlier, the combination of 2PL and 2PC ensures global serialization [16]. When a transaction is executing, it sets locks directly at the primary execution node, and indirectly, through its subtransactions, at other nodes. The coordinator of these subtransactions initiates the 2PC protocol only after receiving an acknowledgement for all the subtransactions operations. Hence, when the coordinator initiates the 2PC protocol, by sending the PREPARE message, all the subtransactions have surely obtained all the locks that they will need. All locks are held until the transaction is either successfully committed or aborted. Consequently, it is guaranteed that a transaction releases a lock at any given site only after it has finished acquiring locks at all sites. This enforces the 2PL discipline and guarantees global serializability [16].

When there is a lock conflict, the transaction requesting the lock is blocked until the transaction holding the lock releases the lock. This locking may lead to a deadlock situation. A deadlock is a permanent, circular wait condition [22]. A set of transactions is deadlocked if and only if each of the transaction waits for the lock held by other transactions from this set. All transactions from this set are in a waiting state, i.e., are blocked, and none of them will

become unblocked without outside interference. This is illustrated in Figure 6, which represents a wait-for graph containing a cycle. A wait-for graph is a directed graph that indicates which transactions are waiting for which other transactions. Nodes of the graph represent transactions and the edges represent the "waiting for" relationship [23]. In the scenario represented by Figure 6, if T_1, T_2 and T_3 belong to different sites, then a global deadlock situation rises.

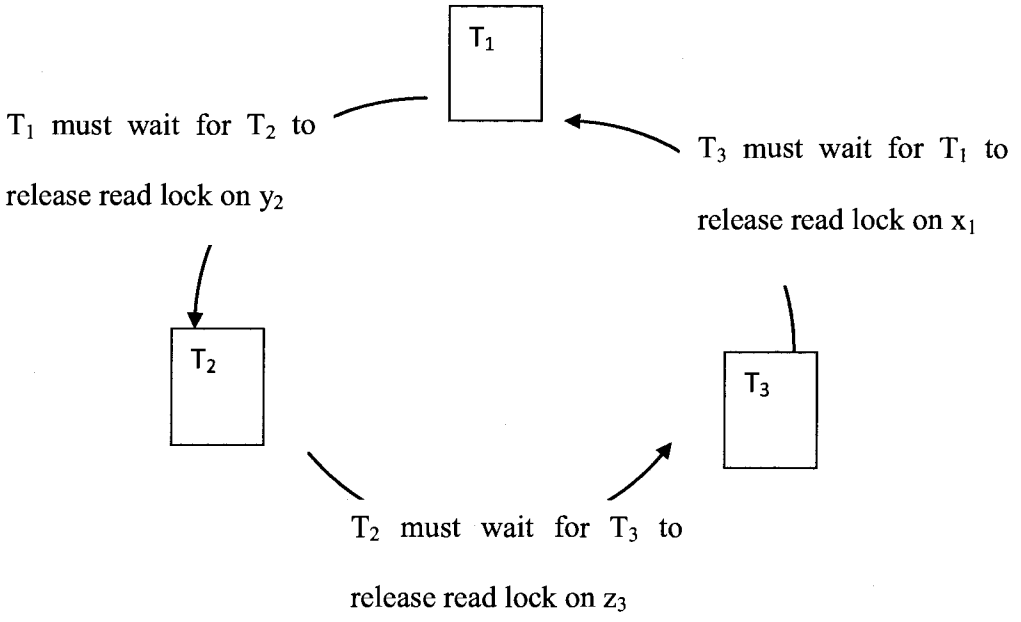

Figure 7: Deadlock Scenario

In a distributed database environment, sites may be connected through a Wide Area Network. As a result, communication between sites can be very slow. This may lengthen the time required for 2PC, which is a part of Distributed 2PL. Consequently, the time needed by a transaction to wait for a lock may increase, as well as the probability of a deadlock situation. In fact, it has been proven that distributed database environments are more susceptible to thrashing (degradation in system performance) than centralized systems because of increased

lock holding times due to inter-node communication and commit protocols [24]. To avoid such a scenario and to improve system performance, a concurrency control mechanism known as distributed wait-depth limited was suggested.

2.1.2. Distributed Wait-Depth Limited

In an attempt to prevent deadlocks, the Distributed Wait-Depth Limited (Distributed WDL) concurrency control mechanism limits the wait depth of blocked transactions to one, by carefully selecting and restarting some of the blocked transactions [24]. The wait depth is the distance of a blocked transaction from the root node(s) of the respective acyclic (deadlock free) waits-for graph [33]. The following example [24] explains distributed WDL. Let us consider a distributed database environment where there is a set of global transactions $\{T_i\}$. Each transaction T_i has an originating or primary node, denoted by $P(T_i)$, with a starting time denoted by $t(T_i)$. The progress made by a transaction involved in a lock conflict or its "length" is denoted by L(T) for transaction T. If T_i has a subtransaction at node k, this subtransaction is denoted by T_{ik}. There are two concurrency control subsystems at each node k, the local concurrency control (LCC) which manages locks and wait relationships for all subtransactions T_{ik} executing at node k, and the global concurrency control (GCC) which manages all wait relations that include any transaction T_i with $P(T_i) = k$ and that makes global restart decisions for any of the transactions in this set of wait relationships. There is a send function that transparently sends messages between subsystems whether they are at the same or different nodes. The general scheme of the distributed WDL is as follows: 1) whenever an LCC schedules a wait between two subtransactions ($T_{ik} \rightarrow T_{jk}$), this information,

consisting of two messages ($T_i \rightarrow T_j$, $P(T_j)$,$t(T_j)$) and ($P(T_i)$, $t(T_i)$,$T_i \rightarrow T_j$), is sent to the GCC of the primary nodes of the corresponding global transactions $P(T_i)$ and $P(T_j)$ and 2) each GCC will asynchronously determine if transactions should be restarted, using their waiting and starting time information [24].

However, due to the fact that in Distributed WDL, LCC and GCC operate asynchronously, a condition may temporarily arise in which the wait-depth of subtransactions is greater than one. Fortunately, such condition will eventually be resolved by a transaction committing or by being restarted by GCC. Moreover, if the subtransaction waiting for the lock belongs to the same node as the subtransaction holding the lock, i.e. $P(T_i) = P(T_j)$, then LCC will only generate information consisting of only one message ($T_i \rightarrow T_j$) sent to their common node [24].

The Distributed WDL algorithm can be demonstrated by a simple example illustrated by Figure 8 [24] which consists of three transactions T_1, T_2 and T_3, with primary nodes 1, 5 and 9. Distributed WDL works as follow:

1. At node 3, T_{13} requests a lock held in an incompatible mode by T_{23}. As a result, the LCC schedules ($T_{13} \rightarrow T_{23}$), and sends messages to the GCC at nodes $P(T_1)$ and $P(T_2)$.
2. Concurrently, at node 7, T_{27} requests a lock in an incompatible mode by T_{37}. The LCC schedules ($T_{27} \rightarrow T_{37}$), and as in the previous case, sends messages to the GCC at nodes $P(T_2)$ and $P(T_3)$.
3. At some later time, these various messages are received and wait graphs are updated by the GCC at nodes 1, 5 and 9. After both messages for the GCC at node 5 are received, there is a wait chain of depth 2.

4. The GCC at node 5 determines, using the local current time and the recorded starting time for each transaction (since $P(T_2) = 5$, its starting time is available locally), that $L(T_2) > L(T_3)$ and $L(T_2) > L(T_1)$. Consequently, following the distributed WDL concurrency control scheme, it decides to restart T_3, and sends a restart message to the transaction coordinator at node $P(T_3) = 9$.
5. The transaction coordinator at node 9 receives the restart message and begins a transaction restart by sending restart messages for all nodes executing a subtransaction T_{3k} and the GCC update messages to the LCC as well as the one at node 5.

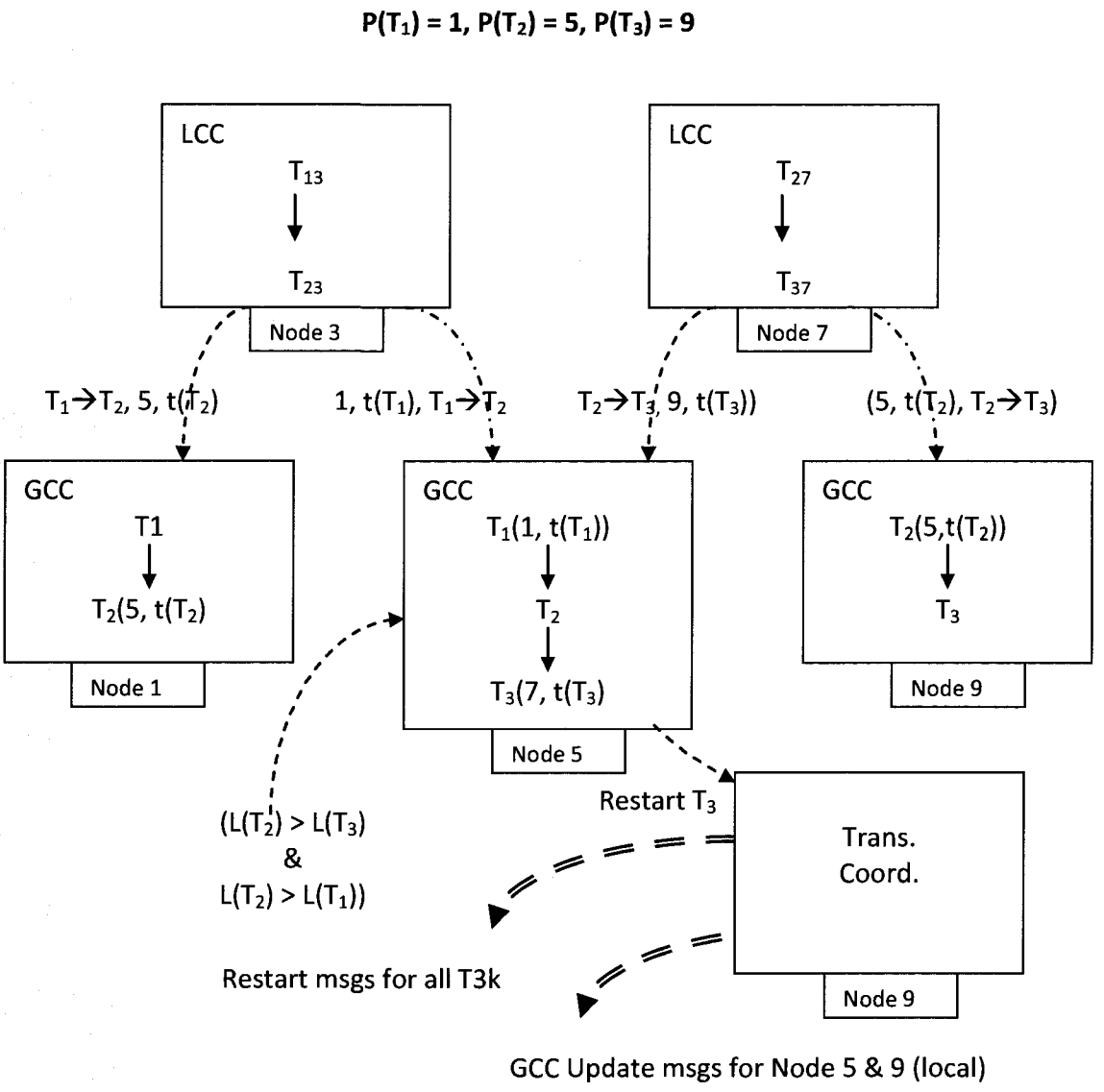

Figure 8: Simple Example of Distributed WDL Method

Similar to distributed 2PL, distributed WDL ensures global serializability. It also has a better performance than distributed 2PL, especially in high processing power systems with a high degree of lock contention. This is due to the fact that transaction blocking, a situation that commonly occurs with distributed 2PL, is reduced by selectively restarting locked transactions. The restarting ensures that deadlocks (local or distributed) are prevented from occurring by limiting the wait-depth of blocked transactions to no more than one.

However, under distributed WDL the number of transaction restarts increases as data contention increases. This situation may degrade the system performance, especially in a system with lower processing power. In fact, it has been shown in [24] that in low processing power systems, distributed 2PL slightly outperforms distributed WDL. Furthermore, not all situations where the wait-depth of blocked transactions/subtransactions is greater than one lead to a deadlock. Hence, by restarting any transaction/subtransaction involved in wait-depth of more than one, the system performance is likely to degrade due to unnecessary restarts. Also, the exchange of messages between nodes, under distributed WDL, is extensive. This can be a serious drawback due to communication overhead. Finally, distributed WDL may still result in a considerable amount of transaction blocking and thrashing in a high data contention environment resulting in a decreased system performance.

2.1.3. Distributed Optimistic Concurrency Control

Optimistic concurrency control protocols delay conflict resolution between transactions until a transaction is near completion. This is based on the assumption that conflict between concurrent transactions is rare. This characteristic allows optimistic concurrency control protocols to be non-blocking and deadlock free.

Optimistic concurrency control protocols operate in three phases: read phase, validation phase and write phase [21]. During the read phase, the transaction reads the values of all data items it needs from the database and stores them in local variables. Updates are applied to the local copy of the data and announced to the database system by a pre-write operation. In the validation phase, the system ensures that all the committed transactions have executed in a serializable fashion. For a read-only transaction, this phase consists of checking that the data values read have not been modified. For a transaction that contains updates, this phase consists of determining whether the current transaction leaves the database in a consistent state, with serializability maintained. Finally, following a successful validation phase, the write phase updates transactions. During the write phase, all changes made by the transaction are permanently stored into the database.

In a distributed database environment, a distributed transaction creates a subtransaction at all the sites where the transaction has operations. To support such an environment, the optimistic concurrency control protocol has been extended into the Distributed Optimistic Concurrency Control (DOCC) protocol. In DOCC, transaction validation is performed at two levels: local

and global [25]. The local validation level involves acceptance of each subtransaction locally. The global validation level involves acceptance of a distributed transaction on the basis of local acceptance of all subtransactions. Similar to distributed 2PL, DOCC involves 2PC principles to ensure global serializability.

Unlike distributed 2PL, DOCC provides a non-blocking and deadlock free environment [25] in a distributed database system. However, in DOCC, if a conflict is detected during the validation phase, the transaction must be restarted. This may lead to an unnecessarily high number of transaction restarts, resulting in high overhead and serious system performance degradation, especially when some near-to-complete transactions have to be restarted. Another problem is starvation, a situation where transactions may never succeed due to repeated restarts [25].

2.1.4. Checkpointing Optimistic Concurrency Control

One of the problem with optimistic concurrency control algorithms is that wasted processing incurred due to transactions failing their validation increases rapidly with transaction size (the number of data items accessed by a transaction). Wasted processing can be reduced by a technique known as checkpointing. Checkpointing was suggested in [26], and it is applied at the transaction level. This is accomplished by using volatile savepoints, also referred to as markpoints or checkpoints. The execution state of a transaction preceding access to data items is saved in the memory to make a volatile savepoint. If it is determined at validation

time that the data item has been modified, then the execution of the transaction can be resumed from the latest checkpoint preceding the access to the modified data items.

Checkpointing introduces a trade-off between increased processing and mean transaction response time versus the processing that is saved when a transaction encounters a data conflict [26]. Checkpointing can be beneficial in a system where data contention is high. However, when the level of data contention is low and the checkpointing cost is relatively high, checkpointing is no longer beneficial. In fact, this situation may be unfavourable to system performance due to the needless use of resources in keeping track of all the transaction checkpoints. Another limitation of checkpointing is its potential for becoming complex in a distributed database environment. Also, in a distributed database environment, checkpointing may result in a very high communication overhead due to an extensive inter-node and inter-site communication. To reduce some of the limitations of checkpointing, especially in a distributed database environment, an approach known as Low-Cost Checkpointing was suggested in [27].

In the basic checkpointing approach for distributed databases, there are two kinds of checkpoints: (1) local checkpointing, which refers to the checkpointing process locally at one site and (2) global checkpointing, which refers to a set of local checkpoints, one at each site, with some degree of synchronization among them. The aim of the basic checkpointing is to maintain a transaction-consistent global checkpoint, i.e., the set of local checkpoints, which constitute the global checkpoint. This global transaction-consistent checkpoint can be used for global reconstruction after a transaction restart. Although the global transaction-consistent

checkpoint has the capability of performing a global reconstruction, it has the limitation of being very slow [27].

Low-Cost Checkpointing introduces a technique known as the Loosely-Synchronous Local_Fuzzy Checkpointing (LSLFC) [27] to assist in global reconstruction. The main difference between the loosely synchronized technique and the fully synchronized technique used in the basic checkpoint approach resides in the way synchronization of checkpoints is done. Fully synchronized checkpointing is done only when there is no active transaction in the database system. In this scheme, before writing a local checkpoint, all sites must have reached a state of inactivity. Conversely, in loosely synchronized checkpointing, each site is not compelled to write its local checkpoint in the same global interval of time. Instead, each site can choose the point of time to stop processing and take the checkpoint. A distinguished site locally manages a checkpoint sequence number and broadcasts it for the creation of a checkpoint. Each site takes local checkpoint as soon as possible, and then resumes normal transaction processing. It is then the responsibility of the local transaction managers to guarantee that all global transactions run in the intervals bounded by checkpoints with the same sequence numbers [27].

Hence, LSLFC eliminates the need to maintain a globally transaction-consistent state when checkpointing. Instead, each site takes local fuzzy checkpoints that are loosely synchronized. A fuzzy checkpoint represents any state of the database [27]. The loose synchronization allows the system to be brought back to a globally transaction-consistent state without lengthy log analysis and extensive message exchange. LSLFC does reduce communication overhead and provides better performance than basic checkpointing, but it still does not

eliminate the unnecessary use of resources in keeping track of checkpoint in an environment where data contention is low.

The protocols examined so far provide some advantages for distributed database systems, but they also suffer from various limitations. Pessimistic/locking based algorithms or optimistic based algorithms provide better system performance depending on the system environment. For example, optimistic based algorithms are effective when the network latency is low or when there is a low level of concurrency, whereas pessimistic based algorithms work better when the network latency is high or when there is a medium or high level of concurrency [20]. Consequently, several hybrid algorithms that combine both pessimistic and optimistic approaches have been suggested.

2.1.5. Hybrid Concurrency Control Algorithms

Hybrid concurrency control algorithms are those that combine both locking and optimistic techniques. For example, Optimistic Dummy Locking (ODL) [28] combines a locking method and an optimistic method, by using dummy locks, on top of read and write locks, to test the validity of a transaction. Dummy locks are long term locks, but they do not conflict with any other locks. A dummy lock can be interpreted as a special mark, such that it is possible to check its existence. It works as follows: when a transaction T_i issues a read lock command for an item X, if the data item is not already in its workspace, a read lock is demanded on the data item. When the read lock is granted, a dummy lock is requested on the data item by T_i. A dummy lock request is always granted because it does not conflict with

any other lock. Then, the value of the data item X is read and the read lock is released. A dummy lock can be released by the transaction itself during the validation test or by another transaction T_j when T_j performs an actual pre-write operation (in its own private workspace) on this data item. When the dummy lock of a transaction T_i is released by another transaction T_j, transaction T_i is said to be invalidated and T_i is immediately restarted. However, if transaction T_i terminates successfully, without releasing any dummy locks on any of the data items it holds, then the validation test is applied to T_i. Therefore, the use of dummy locks helps in identifying transactions to be aborted, and such transactions are restarted without performing unnecessary operations. ODL is deadlock free and can improve performance by avoiding unnecessary operations for invalidated transactions. However, ODL can still result in a high number of transaction restarts in cases where data contention is high or when transactions are especially long. Also, keeping track of the dummy locks introduces an extra overhead to the system, especially in a distributed environment.

Another algorithm proposed in [25] suggests an optimistic approach where phase-dependent control is utilized, such that a transaction is allowed to have multiple execution phases with different concurrency control methods in different phases. This algorithm uses optimistic concurrency control in the first phase and locking in the second phase. If the transaction is restarted in the first phase, then the pessimistic concurrency control is used to limit transaction re-executions to one. This approach limits the number of transaction restarts, but suffers from the limitation of both the optimistic and pessimistic approaches.

Finally, a hybrid technique known as optimistic locking architecture, proposed in [29], provides locking for high conflict data items and optimistic access for the rest. This is

achieved through the use of a data structure called lock buffer that maintains an optimal level of locks in the system. This approach enhances the performance of the basic optimistic concurrency control model by automatically providing locking for highly conflict-prone data items. This enhancement is achieved through the design of the lock manager. The lock manager maintains a finite lock buffer. Each slot in the lock buffer holds locks and pending locks requests for a single data item. Hence, the number of data items with active locks cannot exceed the number of slots in the buffer. Whenever a lock request for a data item X is received by the lock manager, the lock manager first attempts to locate X in the lock buffer. If it is found, the lock manager attempts to post the lock in the corresponding slot. The lock request can either be granted or be blocked in the same manner as pure locking, depending on the status of the existing lock on X. If X is not located in the lock buffer, a slot must be located for posting the lock request. If a free slot exists, it will be used; otherwise a victim slot must be selected. If the number of data items for which locks are requested exceed the size of the lock buffer, locks may be evicted from the lock buffer. All transactions affected by such an eviction of locks automatically become optimistic with respect to the evicted data items.

To illustrate optimistic locking architecture, let us consider the following extreme scenarios [29]. In the first scenario, the size of the lock buffer is zero; as a result, all the lock requests get rejected and there are never any active locks. In this scenario all transactions become optimistic with respect to all the data items in their read and write sets, and the system becomes purely optimistic. In the second scenario, the number of slots is greater than or equal to the number of data items in the database; then each lock request can be granted

without evicting any existing data items and locks. In this scenario, the system is no longer purely optimistic. In fact, in this scenario, the system may become purely pessimistic.

Although optimistic locking architecture is self-tuning [29], i.e. it does not require the transaction manager, the transaction or the user to specify which data items or transactions are optimistic, it still suffers from the limitations of transaction blocking and transaction restarts. A transaction usually needs more than just one data item to execute; hence the probability of securing only data items that are not in the lock buffer is very minimal. Also, by reducing the size of the lock buffer to zero, the system becomes purely optimistic. On the other hand, if the number of slots is greater than or equal to the number of data items in the database, then each lock request can be granted and the system must use some locking or validation mechanism to maintain data consistency. Hence, deciding the right size of the lock buffer can be challenging.

2.2. Static Locking vs Dynamic Locking Algorithms

In Distributed Real Time Database Management Systems, every transaction entering the system has an arrival time, deadline, criticality and an estimated execution time associated with it. However, a transaction may or may not need to know all the data items it might access during its execution. In this section, based on whether or not a transaction needs to know all the data items it will access, various aspects of locking techniques and their respective performance in a distributed real time database environment will be examined.

Scheduling algorithms based on locking approach can be classified according to whether they take advantage of the data access pattern of transactions or not, i.e., static or dynamic [8]. When a transaction enters the system, it gets split into subtransactions depending on the location of the required data items. A subtransaction is allowed to lock any given data item only once during its execution time. In Static Locking, a transaction acquires all the locks it needs before it begins its execution. In Dynamic Locking, a transaction may start its execution without acquiring all the needed locks. Hence, subtransactions in Dynamic Locking will request for access to data items as need arises. Subtransactions are then granted access by the scheduler based on their priority.

Moreover, in Dynamic Locking, lock conflicts are resolved by blocking. Hence, in case of conflict, a higher priority transaction arriving in the system will block any conflicting lower priority transaction. The lower priority lock-holding transaction will then be rolled back to the point where it first accessed the contested data item and then wait. In Static Locking, on the other hand, in case of conflict, a higher priority transaction will restart any lower transaction holding any lock needed by the higher priority transaction. Furthermore, in static locking, global deadlock is avoided by ensuring that locks acquired by a transaction during its execution are held until it has committed or aborted. This ensures that global serialization order is the same as the local serialization order.

Dynamic locking based protocols have been shown to perform better than static locking based protocols [30, 31], but they are prone to deadlock [32]. Static locking based protocols, on the other hand, offer the advantage of creating a non-deadlocking environment. This is due to the fact that all the data items needed by a transaction are acquired before the

execution starts. This offers a great advantage in a distributed environment since global deadlock can be avoided. However, using transaction restarting as a mean of resolving conflict between higher priority transactions and lower priority transactions may result in system degradation. Hence, following the static locking model, several non-preemptive real time concurrency control algorithms for distributed real time database systems have been suggested. These algorithms will be examined in the next section.

2.3. Real Time Static Locking Protocols

The sole condition of any static locking based protocol is to acquire all the locks needed by a transaction before it starts its execution. This ensures a deadlock-free environment and can be very beneficial in a distributed database environment where global deadlocks must be avoided. This is due to the fact that a global deadlock can be expensive to be detected and resolved [22]. However, when transaction restart is not used in solving conflicts between higher and lower priority transactions, the sole condition of static based locking protocols may no longer be favourable for distributed real time database environment since blocking time of higher transaction can be arbitrary long. This is due to prolonged blocking as a result of waiting for multiple locks. Also, priority blocking of the lock holding transaction by other intermediate priority transactions can cause system delay. Finally, the commit time is usually lengthy in distributed systems; this may result in further performance degradation. Different approaches have been suggested in [32] to address some of these limitations and a summary of these approaches is presented here-below.

The Real-time Static Two Phase Locking (RT-S2PL) protocol aims to resolve the problem of prolonged blocking [32]. Under RT-S2PL, each lock in the database is labelled with a priority equal to the priority of the highest priority transaction which is waiting for that lock. With this approach, no lower priority transaction can access the lock; only incoming transactions of higher priority than the waiting transaction can access the lock. This reduces blocking of higher priority transactions due to the fact that no lower priority transaction can have access to the lock, unless it came into the system earlier. Hence, no lower priority transaction is allowed to set locks if any of its required locks are awaited by a higher priority transaction even though the locks are free. However, the blocking time of higher priority transactions is unbounded due to the possibility of priority preemption by other intermediate priority transactions preventing the lower priority, lock-holding transaction from using the CPU [32].

The Real-Time Static Two Phase Locking with Resource Priority (RT-S2PL-RP) attempts to reduce the blocking time of higher priority transactions due to priority preemption of CPU in RT-S2PL. To achieve this, RT-S2PL-RP uses the following approach: whenever there is a blocked higher priority transaction, any lower priority transactions is prevented from setting locks, even though the requested locks of these lower priority transactions are not the ones requested by the higher priority transactions [32]. The RT-S2PL-RP approach is unfortunately too restrictive and goes against the principle of concurrency. This has the consequence of degrading system performance.

Finally, Real-time Static Two Phase Locking with Priority Inheritance (RT-S2PL-PI) attempts to use priority inheritance to solve the problem of blocking of lower priority

transactions by implementing the following scheme: whenever a higher priority transaction is blocked by a lower priority transaction, the priority of the lower priority transaction is raised up to that of the higher priority transaction in order to prevent the preemption by other intermediate priority transactions [32]. This scheme does not have the limitations of the RT-S2PL-RP approach, but it is potentially detrimental to the adopted CPU scheduling algorithm. In RT-S2PL-RP, a transaction may be blocked by more than one transaction due to the fact that required locks may be locked by different transactions. Hence, all of these transactions will be raised to the same priority even though their original priorities are different.

To illustrate this limitation let us consider the following example [32] that involves four transactions: T_1, T_2, T_3 and T_4 with priority $T_4>T_3>T_2>T_1$. The locks required by T_4, T_3, T_2 and T_1 are {4,8}, {3,7}, {2,6} and {1,2,3,4,5} respectively. Assume that T_1, T_2 and T_3 successfully obtain their required locks and start their processing. Since T_3 has the highest priority amongst the transactions which have obtained their locks, T_3 is executing and T_1 and T_2 are suspended. When T_4 arrives, it is blocked. If RT-S2PL-PI is used, all T_1, T_2 and T_3 will be set to be the same priority as T_4. Therefore, all the three transactions, T_1, T_2 and T_3, will be executing at the same priority even though their initial priorities are different. This will affect the schedulability of the system. The aim of assigning different priorities to the transactions will become ineffective [32].

However, this limitation can be solved by using two levels of priority [32]. The system should distinguish inherited priorities and original priorities. If the transactions have similar inherited priority, their original priorities have to be compared to decide which transaction

should be selected for execution first. This, unfortunately, adds an extra step whenever an operation needs to be performed within a transaction.

2.4. Speculative Locking Protocol

Speculative Locking (SL) protocol extends the standard 2PL by allowing parallelism among conflicting transactions [9]. In 2PL, a transaction holds an exclusive lock on a data item until completion of commit and then the lock is released. In SL, the waiting transaction is allowed to access the locked data item whenever the lock-holding transaction produces corresponding after-images during execution. The waiting transaction then accesses both the before and after-images and carries out speculative executions and retains one execution based on the termination of the preceding transactions. The before-image and after-image refer to the value of a data item before and after being modified by a transaction. While preserving serializability, SL improves performance over 2PL due to the parallelism among conflicting transactions.

To illustrate SL, let us consider Figure 9 [9]. Figure 9(a) represents the processing a transaction where the notation s_i, e_i and c_i/a_i denote the start of execution, completion of execution and the transaction commit phase where the transaction can either commit or abort. Figure 9(b) represents a 2PL scenario and Figure 9(c) a SL scenario. As illustrated in Figure 9(b), if T_1 is reading and writing pages X and Y, Transaction T_2 will not be allowed to access the after-image of X until T_1 has completed its commit processing. In SL, on the other hand, as illustrated in Figure 9(c), when T_1 completes processing and produces the after-image X'

and Y', T_2 is immediately given access to both the before-image X and the after-image X'. T_2 then carries out speculative executions T_{21} and T_{22} to produce the after-images X'' and X'''. When transaction T_1 completes processing, T_2 proceeds into the commit phase and retains T_{22} if T_1 commits or T_{21} if T_1 aborts. If there is another transaction T_3 waiting for T_2, it will follow the same procedure as illustrated in Figure 10 [9]. Serializability is maintained in SL through the commit dependency that is created between competing transactions.

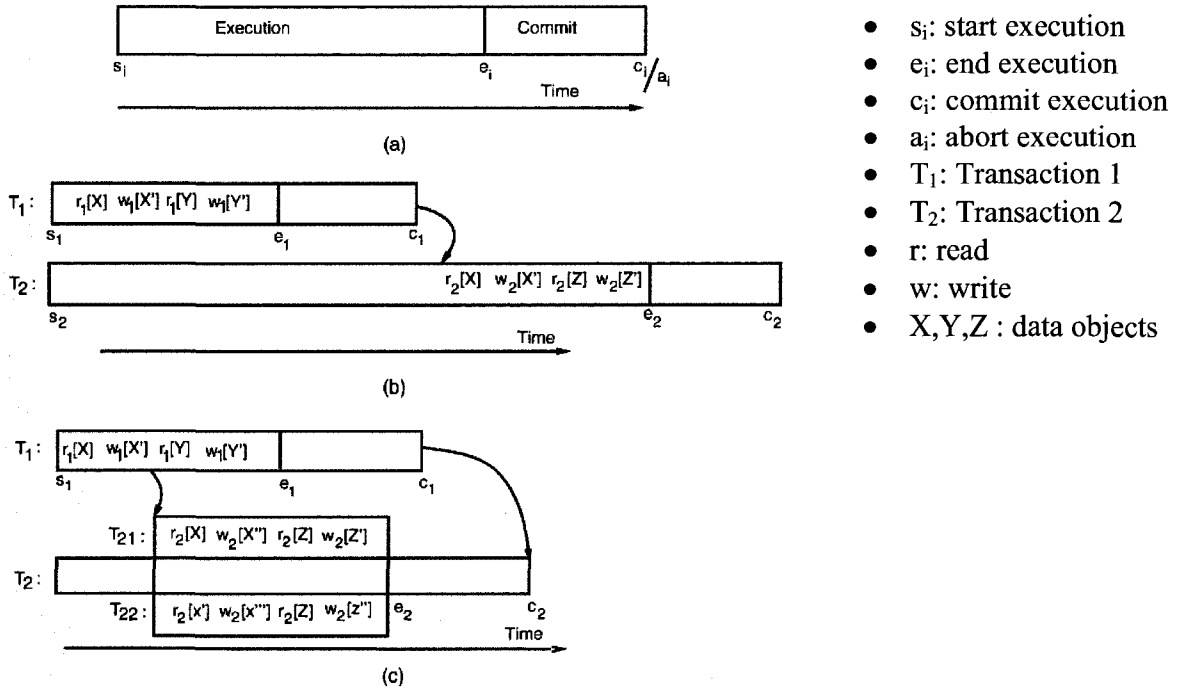

Figure 9: Comparison between 2PL Variants and SL: (a) Processing of T_i, (b) Processing with 2PL and (c) Processing with SL.

SL

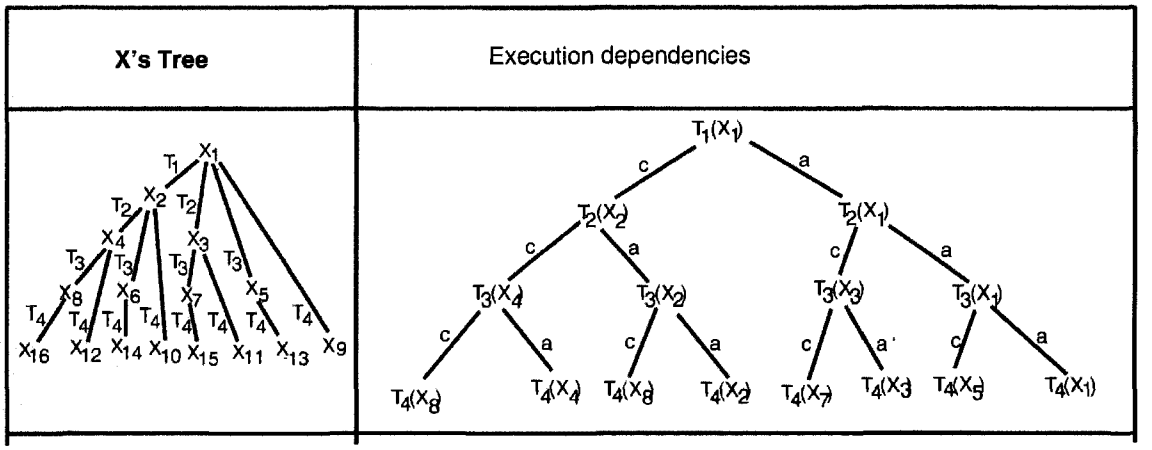

Figure 10: Depiction of Tree Growth and the Speculative Executions

In 2PL, pages can be locked in shared or exclusive mode, i.e. Read (R) or Write (W) mode. A page can be accessed simultaneously by two or more transaction when it is in a read mode. In SL, the write mode is divided into two: execution-write (EW) and speculative write (SPW). A transaction can only request a read lock (r) and an execution–write lock (EW). When a speculative execution takes place, the EW-lock is converted to a speculative-write (SPW) lock allowing other transactions to get access to the locked data. This is illustrated through the lock compatibility matrix of 2PL and SL represented in Table 2 [9]. The lock compatibility matrix of SL shows that only one transaction holds an EW-lock on the data item at any point in time. However, multiple transactions can hold the R and SPW-locks simultaneously. This is different from the 2PL lock compatibility matrix where multiple transactions can hold locks simultaneously only when all the transactions are in the R-lock mode.

Lock requested by T_i	Lock held by T_j	
	R	W
R	yes	no
W	no	no

(a)

2PL

Lock requested by T_i	Lock held by T_j		
	R	EW	SPW
R	yes	no	sp_yes
EW	sp_yes	no	sp_yes

(b)

SL

Table 2: Locks Compatibility Matrix

Sites involved in a Distributed Database Systems (DDBS) are usually connected through a wide area network (WAN). Hence, inter-node or inter-site communication in DDBS can be quite slow. This can affect the execution time of transactions that require remote data access. The most affected part of the transaction execution is the commit phase due to an extensive amount of inter-node/inter-site communication that takes place. The time to commit in such a scenario may account up to 80 percent of the transaction time [9]. This could result in serious performance degradation in 2PL due to the fact that data items become unavailable to the waiting transaction for longer durations. This limitation is eliminated by SL, by allowing speculative executions, resulting in a better system performance in distributed database environment.

Compared to all the protocols (pessimistic based protocols, optimistic based protocols and hybrid protocols) investigated in the above sections, SL offers many more combined benefits for a distributed environment. These include:

1. The ability to allow parallelism in transaction execution to improve concurrency in distributed environment.
2. The ability to not compromise serializability during transaction execution.
3. The ability to allow speculative executions of transactions that alleviate the effect of longer commit time for transactions in a distributed environment.
4. The ability to avoid cascading aborts but still allow early data accessibility.

Hence, from a Distributed Database Management System point of view, SL has the potential to provide better performance than all the other locking and optimistic concurrency control variants analyzed in this chapter. SL improves the throughput performance of DDBSs [9]. However, SL does not take transaction's time constraints into consideration, making it unfit for Distributed Real Time Database System (DRTDBS). This is due to the fact that SL does not take into account the priority of a transaction when scheduling transactions.

Although transaction throughput can be beneficial for DRTDBS, it is crucial to minimize the number of transactions that miss their deadlines. To achieve this, there is a need to modify or extend SL by giving it the capability to take into consideration a transaction priority when scheduling transactions. This requirement necessitates new algorithms: the Priority Based Speculative Locking protocols that will be explained in detail in the next chapter.

Chapter 3

Priority-Based Speculative Locking Protocols.

3.1. System Model

To evaluate the performance of the proposed protocols, the discrete event simulation model described in [35] was used. This model consists of a set of databases which are physically partitioned, in a non replicated manner, over a number of sites connected by a network. These sites can contain one or more server nodes that in turn host one or more databases. The database is modeled as a collection of pages. As illustrated in Figure 11, nodes and sites can be interconnected through a wide area networks or a local area networks.

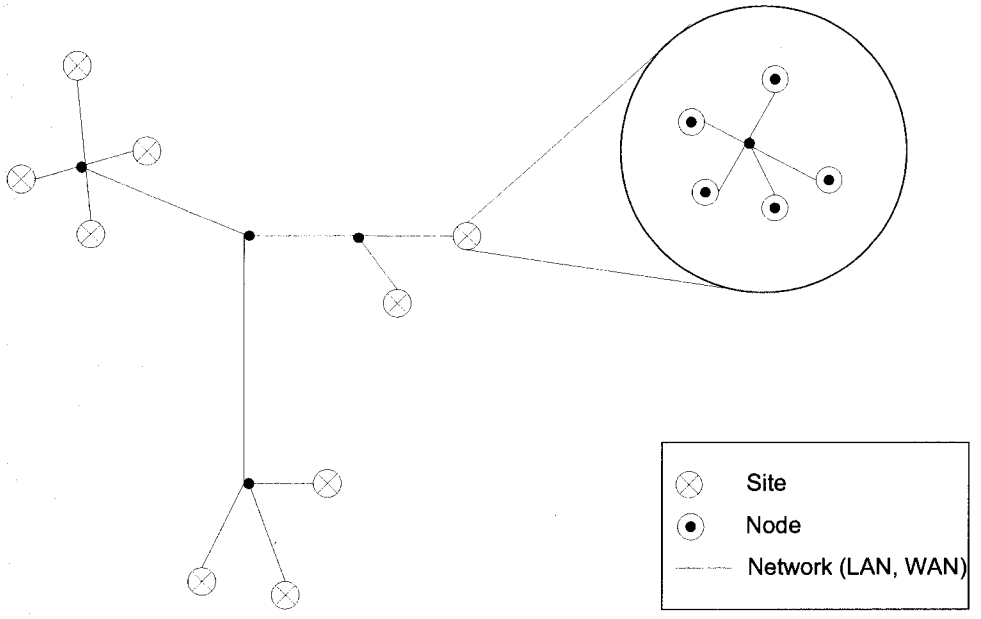

Figure 11 : Sites and Nodes in a Distributed Database System Model

Each node is a representation of a computer system that is comprised of processors, disks, and cache/buffer. Each node may host one or more databases, each of which contains a set of pages that are located on one or more disks. Each site in the model consists of a Transaction Generator which generates transactions, a Transaction Manager which models the execution behaviour of a transaction, a Resource Manager which controls the system resources (processors, disks and buffers) and a Scheduler which implements the concurrency control algorithms (Figure 12).

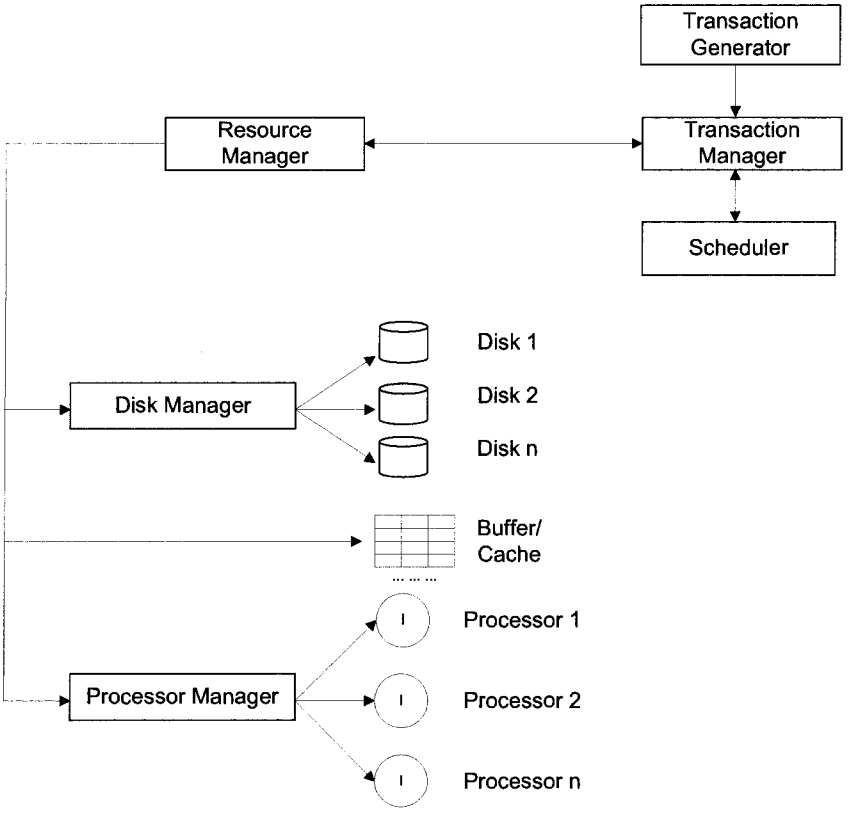

Figure 12 : Simulation System Model

3.1.1. Transaction Generator

The Transaction Generator, also called the Workload Generator, generates transactions and defines their characteristics. The characteristics of a transaction include its inter-arrival time, slack, worksize and update probability [35]. Different statistical distribution models such as constant, normal, Poisson and uniform distributions are used to control the trends of values assigned to these transaction characteristics.

Probability distributions of random inputs such as worksize, slack and inter arrival time must be specified to carry out a simulation. The appropriate choice of a distribution model depends on the trends that best model specific characteristics. For example, the Poisson distribution has been proven to best model inter arrival time. The uniform distribution, on the other hand, is fit to model a quantity that randomly varies between two values but about which little else is known [36]. Hence, in this study, transactions' inter-arrival times are modeled using Poisson distribution, whereas transactions' worksize and slack values use uniform distribution.

Transaction inter-arrival time represents the amount of time between two consecutive transactions generated by the same transaction generator. Thus, a larger inter arrival time translates into fewer transactions arriving, thus representing a low load system. The number of pages a transaction is expected to process when it enters the system is specified by the transaction worksize. It should also be noted that the pages processed by a transaction may or may not have to be written back to the disk. The update probability, set by the Transaction Generator, represents the probability that any given page will be written back to the disk.

The amount of time that a transaction T_i needs to complete depends on the number of pages that are expected to be processed by the transaction (T_i(pages)), the amount of time that a processor needs to process a page (processor ticks) and the amount of time needed to access a page located on a disk (disk ticks) or a swap disk (swap disk ticks). Several different scenarios can be considered:

1. All the pages needed by the transaction are located in the buffer/cache and the cache is located on the same node as the transaction. If the amount of time is represented in ticks, then the amount of ticks needed by a transaction T_i is:

   T_i *ticks* = T_i *(pages in cache) X processor ticks*

2. Some or all the pages needed by the transaction are located on the disk and the cache is not full. Also, the disks and the cache hosting all pages needed by the transaction are located on the same node as the transaction. In this case the amount of time needed will be:

   T_i *ticks = (T_i (pages on cache) + T_i (pages on disk)) X processor ticks +*

   T_i (pages on disk) X disk ticks

3. Some or all the pages needed by the transaction are located on the disk and there is not enough space in the cache. In this case, some of the pages residing in the cache need to be temporarily placed to the swap disk. Considering that the disks, swap disk and cache hosting the pages needed by the transaction reside on the same node as the transaction, the number of ticks needed by T_i is:

   T_i *ticks = (T_i (pages on cache) + T_i (pages on disk)) X processor ticks +*

   number of pages to be moved from cache X swap disk ticks +

   T_i (pages on disk) X disk ticks

4. If one or more pages needed by the transaction reside on a disk or cache located on a different node than the transaction, then the transaction will have to create one or more subtransactions to access the needed page(s). In this case, inter-node communication delay needs to be taken into consideration when estimating the amount of time needed by a transaction to complete.
5. If one or more pages are held by another transaction, then the transaction will have to wait until all the pages it needs are released.

Scenarios 4 and 5 introduce complexities that can make it difficult or impossible to estimate the amount of time needed by a transaction to complete. Hence, when assigning transaction slack, which is an estimate of how long the execution of a transaction can be delayed and still meet its deadline [1, 36], one needs to take into consideration all of the possible scenarios.

The characteristics of the Transaction Generator also include the page range and size. Page range represents the range of pages that transactions generated at this location are expected to access. Size, on the other hand, represents the number of transactions that this Transaction Generator will create.

In the case of a nested transaction model, which is the model used in this study, the Transaction Generator only generates top level transactions, which in turn generate subtransactions. In a distributed database system, data items (represented in this simulation model as pages) may be located on any site or any node. Also, in a database environment which does not support any replication, data items can only be located at one location at a time [34]. Hence, subtransactions are generated depending on the location(s) of the pages that the transaction need to access.

To illustrate this, let us consider Figure 13 which represents a nested transaction in a distributed environment. The Transaction Generator generates T_i on $Node_0$. T_i needs to access data items X, Y and Z located respectively on $Node_1$, $Node_2$ and $Node_3$. As a result, T_i spawns three subtractions $T_{i.1}$, $T_{i.2}$ and $T_{i.3}$. These subtransactions will operate on their respective nodes and locally access and process the data items they need ($T_{i.1}X$, $T_{i.2}Y$ and $T_{i.3}Z$). After processing is done, all of the subtransactions will report the result back to transaction T_i.

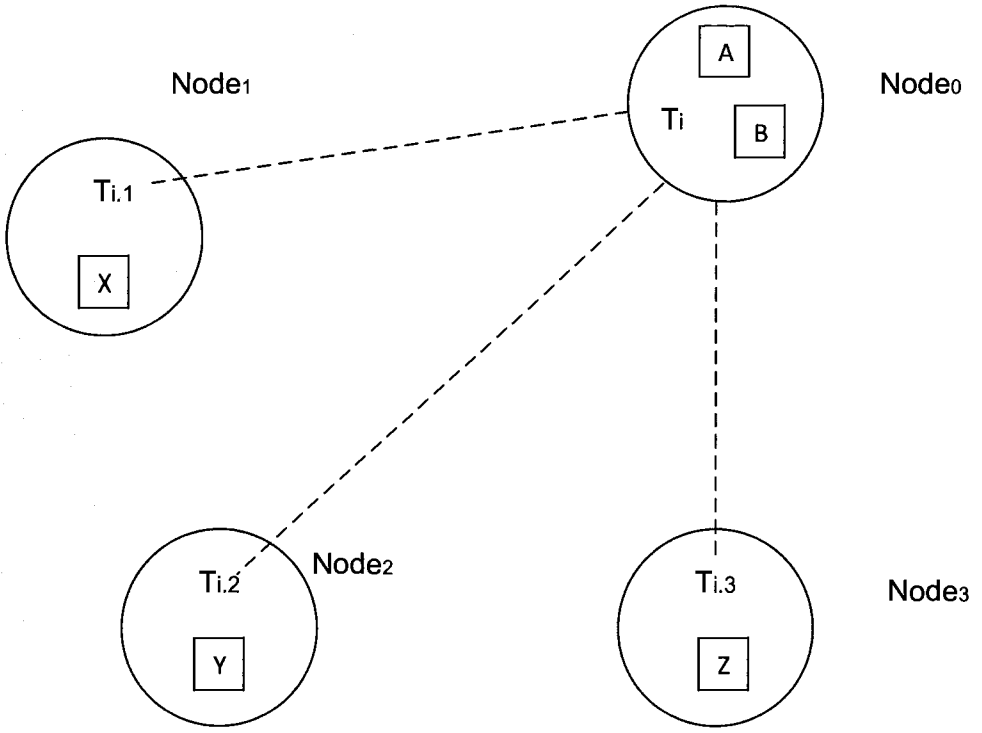

Figure 13 : Nested Transaction Model in Simulation

Another characteristic of a transaction executing in a real-time environment is its priority. Priority assignment policies manage how transaction time constraints are used to assign a priority to a transaction. Several priority assignment policies are used for scheduling

transactions. These include: (1) First Come First Serve, a policy that assigns the highest priority to the transaction with the earliest arrival time; (2) Earliest Deadline First, where the transaction with the earliest deadline is assigned the highest priority; (3) Shortest Job First, where the transaction that requires less work is given a higher priority [1]. Of these, the policy that assigns priority to a transaction based on its deadline has been broadly used [1, 7] and has been found to be the best policy in terms of success ratio in most cases [2]. For a nested transaction, we have adopted the model that assigns subtransactions the same priority as their parent transaction.

3.1.2. Transaction Manager

The Transaction Manager manages the execution of transactions from the time they enter the system until they leave the system. The primary purpose of the Transaction Manager is to pass operations within a transaction to the concurrency control manager and establish which site the transaction is destined for [35]. For example, in the case of a nested transaction, if the operation is subtransaction activation, the Transaction Manager will forward the subtransaction to the appropriate site's transaction manager. Other purposes include controlling the maximum number of active transactions and managing transactions waiting queues.

3.1.3. Scheduler

The Scheduler, also called the Concurrency Control Manager, is responsible for the coordination of data access in order to keep the database in a correct and consistent state at all times. This is achieved through the use of concurrency control protocols. Furthermore, the Concurrency Control Manager ensures serialization of transactions operations and avoids cascading aborts by ordering operations within a transaction [36]. Finally, as the coordinator of data access, the Concurrency Control Manager is in charge of handling deadlocks.

When a transaction requests a lock, it sends an access request to the Scheduler. The Scheduler validates serializability with other concurrent transactions and then forwards the request to the cache or disk. In case of a data conflict with an existing transaction, this validation may lead to blocking of the request or preemption of the transaction holding the requested data item. This blocking/preemption of transactions prevents both the loss of serializability and future cascading aborts. Also, depending on the concurrency control protocol used, if there is no data conflict, the Scheduler may ensure that the priority of the transaction is taken into consideration in prioritizing data access for the requesting transaction. Finally, the Scheduler is in charge of checking for any possibility of deadlock. If a deadlock is detected, one of the transactions involved in the deadlock will be aborted. The choice of the transaction to be restarted depends on the deadlock resolution model adopted for the simulation.

3.1.4. Resource Manager

The Resource Manager manages shared access to system resources. These resources include processors, disks, network and cache. Processors are managed through a processor manager and disks are managed through a disk manager. A processor represents a computer processor and can only process one page at a time. However, each node can host more than one processor. A disk is a representation of a non-volatile storage on a computer. A disk may contain many pages, but only allows one operation (either a read or a write) to be performed on one of its pages at a time. The network represents the links that connect different nodes and sites. Finally, a cache, also known as buffer, represents a volatile computer memory.

Before a transaction can access a page, the page has to be moved from the disk into cache. A page held in cache will not be released until the transaction which requested the page is either completed or aborted. If the cache is full, then some of the pages that are not locked will be swapped to a physical disk known as the swap disk to create space on the cache. If all of the pages in the cache are locked, then depending on the priority of the transaction requesting access to pages, the incoming transaction may either be blocked until there is enough room in the cache, or a group of pages locked by a lower priority transaction may be swapped out to the disk. Once there is room in the cache, any pages that were swapped to the disk are returned to the cache.

Another simulation parameter is the maximum number of active transactions [35]. This parameter sets a limit on the number of transactions that can be simultaneously active at a node. If this limit is met, then all other transactions are forced to wait outside the node until

there is space available in the node. Hence, the higher the value of the maximum active transactions parameter, the higher the number of active transactions in the system may be. This gives the maximum active transaction parameter a certain level of control over the system load and data contention within the system.

3.2. Description of Priority-Based Speculative Locking Protocols

The Priority-Based Speculative Locking Protocols use the Speculative Locking (SL) protocol's underlying architecture [9], but incorporate the notion of transaction priority when scheduling transactions, in order to improve performance within Distributed Real-Time Database Systems (DRTDBS). The Priority-Based Speculative Locking Protocols have been demonstrated to improve performance in DRTDBS due to the fact that they address both distributivity and time constraint issues.

In a distributed database environment data items can be distributed over multiple sites. These sites communicate through a network structure composed of local area networks (LAN) and wide area networks (WAN). WAN communication, required for remote data access, takes up a considerable portion of transaction execution time. SL addresses this issue of data distributivity by allowing more parallelism between conflicting transactions without violating the principle of serializability [9]. This results in an increase in transaction throughput within distributed database systems. In distributed real-time database systems, however, the key goal is to minimize the number of transactions that miss their deadlines. Hence, increased

transaction throughput is not enough if one aims to improve performance in distributed real time database systems. This makes SL inadequate for distributed real time database systems. To illustrate this limitation of SL, let us consider the diagram (Figure 14) [9] representing a SL scenario:

- $\mathbf{T_1}$, $\mathbf{T_2}$, $\mathbf{T_3}$ and $\mathbf{T_4}$ represent different incoming transactions in a specific order, i.e. $\mathbf{T_1}$ came in the system earlier than $\mathbf{T_4}$
- $\mathbf{x_1}$, $\mathbf{x_3}$....$\mathbf{x_n}$ represent data items
- **a** represents an abort situation
- **c** represents a commit situation
- $\mathbf{T_i}$ $\mathbf{x_i}$ represents transaction T_i locking data items x_i

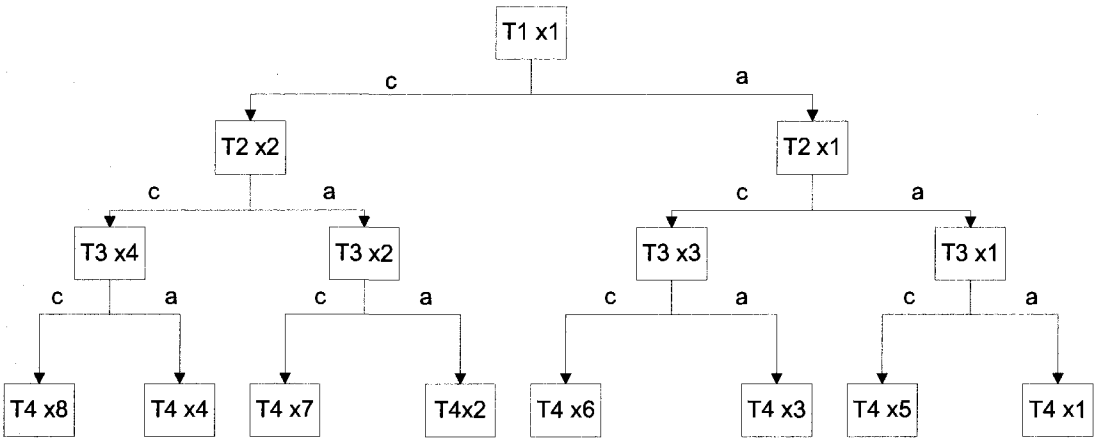

Figure 14 : Speculative Locking Scenario

- T_1 is the first transaction to come into the system and naturally it accesses the data item it needs (x_1).

- Before T_1 is done with x_1, i.e. before it has committed x_1, T_2 comes into the system and requests access to x_1 as well. This creates an access conflict on x_1 between T_1 and T_2
- To avoid unnecessary waiting of T_2, the before and after images of the data item upon which there is conflict, in this case x_1 (the before image) and x_2 (the after image), are given to the waiting transaction T_2.
- As a result T_2 proceeds with two speculative executions and will wait until T_1 has committed to decide which version of its execution should be kept.
- If T_1x_1 commits, T_2x_2 will be kept, otherwise in case T_1 aborts, T_2x_1 will be kept.
- The rest of the speculative executions from other incoming transactions follow the same principle.

Using the SL approach, no incoming transaction has to go through unnecessary waiting. However, a certain commit dependency is created between transactions, i.e. T_1 has to commit before T_2 and T_2 has to commit before T_3 and so on. This commit dependency is necessary to ensure serializability, which is required in order to maintain information consistency. However, this may cause some transactions to miss their deadlines. For example, if T_4 is of higher priority than T_1, T_2 and T_3, but came into the system last, T_4 may miss its deadline due to the fact that it has to wait for T_1, T_2 and T_3 to commit before it can commit. To address this limitation, each transaction's priority needs to be taken into consideration when scheduling transactions. Two approaches are suggested in this study: priority preemption and priority inheritance. These approaches lead to the two proposed protocols, Preemptive Speculative Locking (PSL) and Priority Inheritance Speculative Locking (PiSL)

3.2.1. Preemptive Speculative Locking

Preemptive Speculative Locking (PSL) takes transaction's priority into consideration when scheduling transactions. This allows access to data items in a prioritized manner. PSL extends SL by allowing any incoming higher priority transaction to preempt and abort any lower priority transaction, in case of a lock conflict. To illustrate this, let us consider Figures 15, 16 and 17, which represent different PSL scenarios.

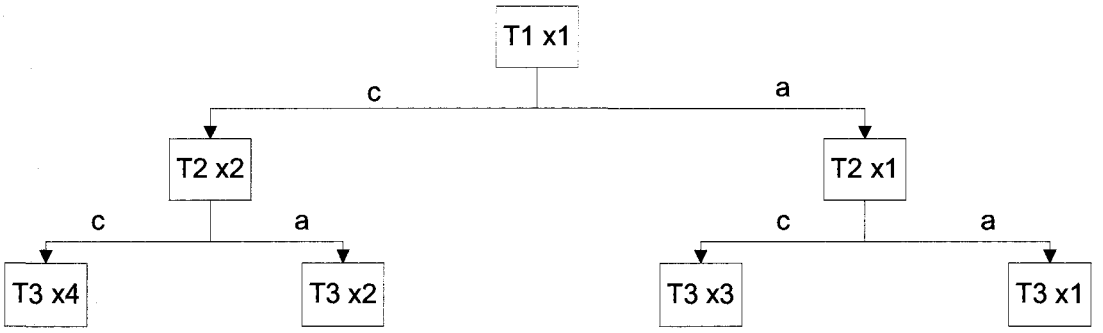

Figure 15 : PSL structure before preemption

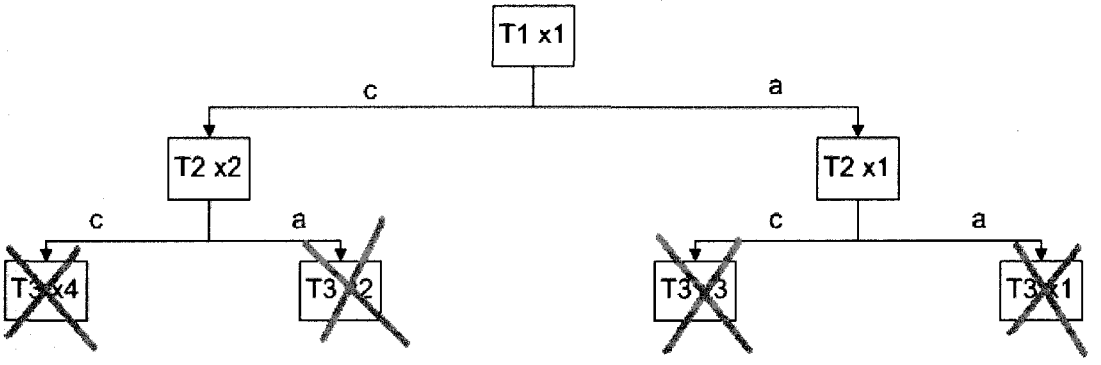

Figure 16 : PSL structure during preemption

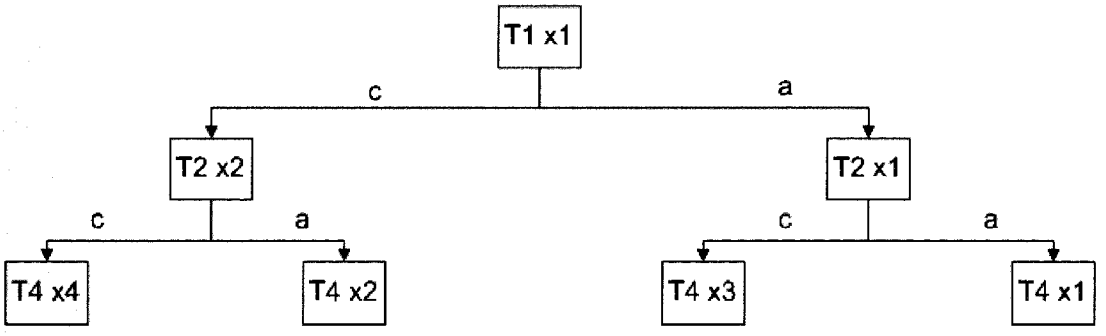

Figure 17: PSL after Preemption

Suppose we have a transaction T_4 coming into the system while T_3, T_2 and T_1 are executing (Figure 15). T_4 is of higher priority than T_2 and T_3. Under PSL, T_4 will abort T_3 (Figure 15) to allow T_4 to access resources held by T_3 (Figure 16). By aborting T_3, T_4 is given a chance to finish within its time limit. If T_4 had to be attached to the tree as a 4^{th} level transaction (Figure 13), as in SL, T_4 would have to wait for T_1, T_2 and T_3 to commit before it could commit. Hence, by preempting the lower priority transactions (T_3), T_4 is given a greater chance of meeting its deadline.

However, it should be noted that T_2 is not aborted, in spite of the fact that it is of lower priority than T_4. In order to allow access to data items requested by T_3, T_2 must first complete its operations on the corresponding data items. Under PSL, only transactions that have not completed operations on the data items requested by higher priority transactions are aborted in case of conflict.

PSL favours transactions with higher priority in order to give them a chance to meet their deadline. However, in doing so, PSL causes execution work done by lower priority transactions to be lost because these are preempted and aborted in order to advance a higher priority transaction. For example, by preempting and aborting T_3 to advance T_4, any work already done by T_3 is lost (Figure 16, 17). As a result T_3 will have to restart its execution and may miss its deadline. Hence, with PSL, T_3 risks its chance of meeting its deadline in order to give T_4 a chance to meet its deadline. This situation may result in three scenarios:

1. T_3 has enough time and it still meet its deadline in spite of being restarted. This presents the best case scenario, where all transactions (T_3 and T_4) meet their deadlines.
2. T_3 does not have enough time to meet its deadline. In this case T_4 meet its deadline, but T_3 misses its deadline due to being restarted.
3. T_4 still misses its deadline despite being favoured, or T_4 is preempted by another incoming higher priority transaction. In this case, all the work achieved by T_3 is needlessly lost.

To avoid situations similar to the second scenario, and especially the third scenario, a different approach needs to be adopted to address the issue of transaction priority. In this approach, a transaction's priority is still taken into consideration in scheduling transactions, but transaction restart is conditional. When a higher transaction needs to access a data item held by a lower priority transaction, the higher priority transaction checks if the lower priority transaction is close to completion, and if its request for the data item can be delayed. This involves checking the status of each data item held by the lower priority transaction. If the lower priority transaction is a nested transaction, checking the status of each of its data

items will involve sending messages across the network, to each site where a subtransaction of the nested transaction is located. This may result in a very high and expensive inter node message exchange, which can result in the higher priority missing its deadline or in both transactions missing their deadlines.

To illustrate this, let us consider two transactions T_1 and T_2 that are competing for the resource X. T_2 is of higher priority than T_1, but T_1 was granted access to X before T_2. T_1 has also been granted access to other data items A, B, C, D and E. Suppose that T_1 and T_2 are located on $node_0$ and A, B, C, D and E are located on $Node_1$, $Node_2$, $Node_3$, $Node_4$ and $Node_5$, respectively. In order to access all the data items located outside the node where T_1 is located, T_1 must spawn subtransactions $T_{1.1}$, $T_{1.2}$, $T_{1.3}$, $T_{1.4}$ and $T_{1.5}$ to access A, B, C, D and E, respectively. If the conditional restart approach is used to solve the data conflict between T_1 and T_2 over X, the Transaction Manager of $node_0$ will need to exchange messages with the Transaction Managers of $Node_1$, $Node_2$, $Node_3$, $Node_4$ and $Node_5$ to check the status of all the subtransactions ($T_{1.1}$, $T_{1.2}$, $T_{1.3}$, $T_{1.4}$, $T_{1.5}$) in regards to the data items (A, B, C, D, E) they are accessing. Based on the information gathered from this exchange of messages and the current status of T_2, the Scheduler will decide either to proceed with the restart of T_1 or not. This extensive exchange of messages, especially in a WAN environment, may result in a delay that may cause both transactions to miss their deadlines. Missing of both transactions' deadlines is possible if the Scheduler decides that T_1 need to be restarted when T_2 is close to missing its deadline. Hence, while conditional restart can be beneficial in a non-distributed database system, it is risky to adopt it in a distributed environment where inter-node communication can be expensive and time consuming.

3.2.2. Priority Inheritance Speculative Locking

Priority Inheritance Speculative Locking (PiSL) attempts to prevent wasting any work that a transaction has already completed. In this approach, if a transaction is already executing, or it has been granted access to a data item, it cannot be preempted and aborted regardless of its priority. PiSL uses the following approach in order to solve data conflict between transactions: whenever a higher priority transaction is blocked by a lower priority transaction, the priority of the lower priority transaction is raised to the priority of the blocked transaction. This approach ensures that the lower priority transaction completes its execution without interruption and then passes the lock (data item) to the waiting higher priority transaction as soon as the lock is available.

To illustrate this, let us consider the following diagrams (Figures 18-21)

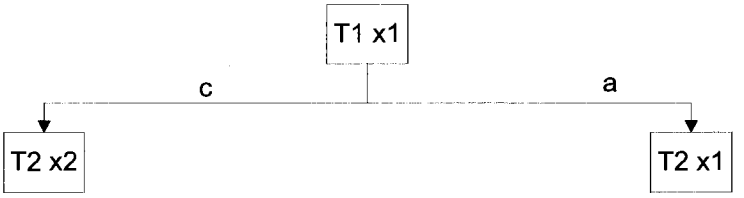

Figure 18: PiSL before any lock conflict

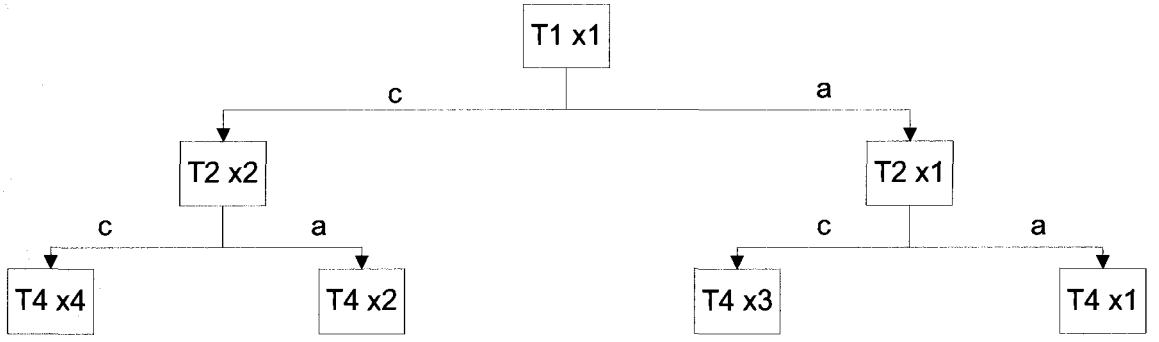

Figure 19: PISL during transfer of transaction priority

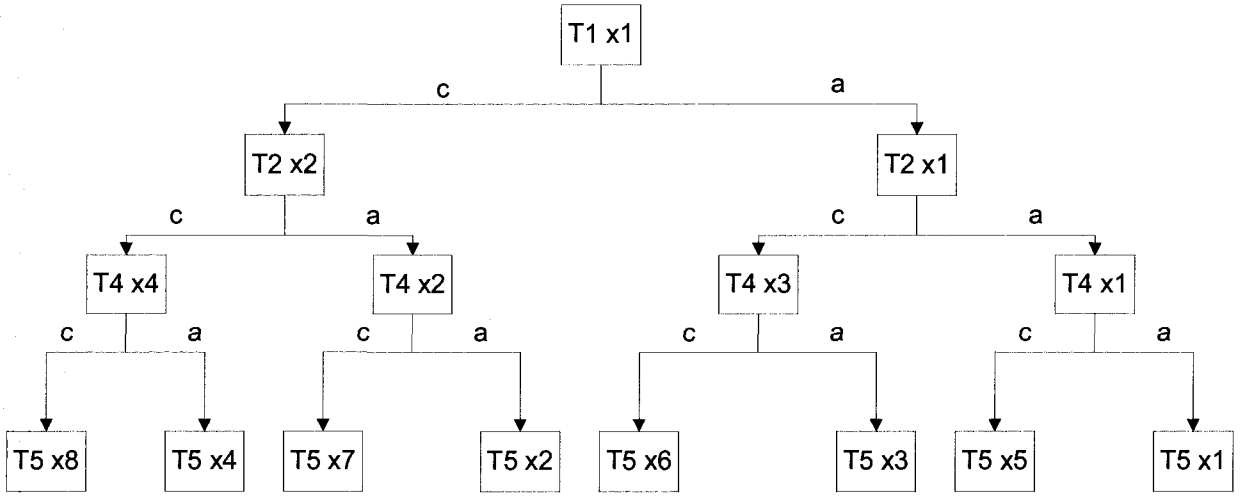

Figure 20 : Figure 19: PISL during transfer of locks

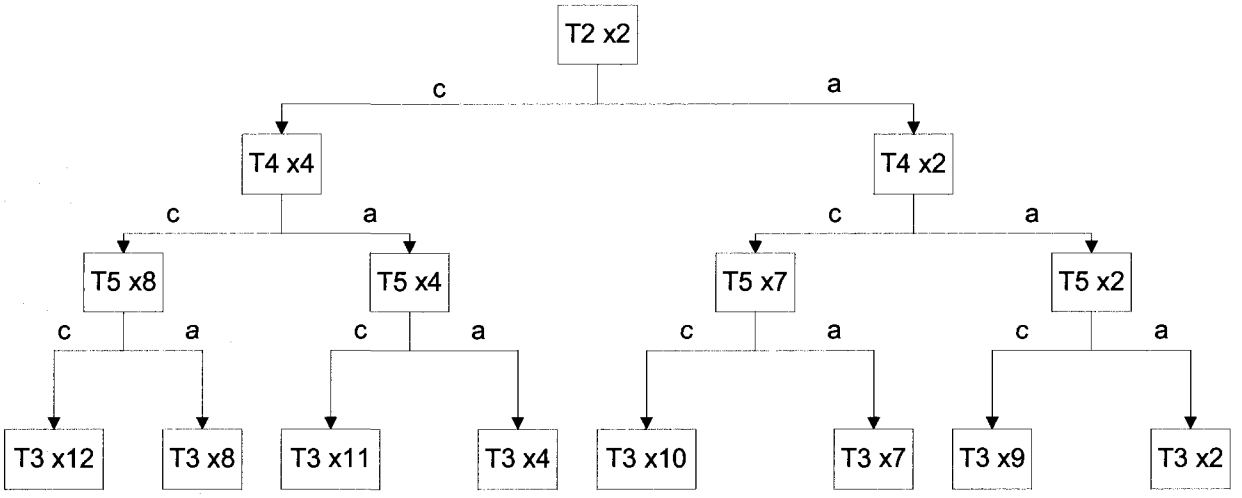

Figure 21: PISL during commit and transfer of locks

Let us consider 5 transactions (T_1, T_2, T_3, T_4 and T_5) that are competing for the same data item x_1. The priority of these transactions is as follow $T_2<T_1<T_3<T_5<T_4$. Transaction T_1 is the first transaction to enter the system. At this point, there is no hold on the data item x_1, and thus, T_1 is allowed to lock x_1. Later on, T_2 enter the system; however, since T_2 is of lower priority than T_1, T_2 waits until T_1 produces an after-image x_2 to start execution. After T_1 produces the after-image, x_2, T_2 is allowed to access both x_1 and x_2 (Figure 18). As soon as T_2 accesses x_1 and x_2, T_3 enters the system. T_3 is of higher priority than T_1 and T_2, but instead of preempting and aborting T_2, as it is done with PSL, T_3 passes its priority to T_2. This gives T_2 a chance to quickly complete its execution work, as it will be favoured in the priority queue of the processor. Then, once it finishes, T_2 grants T_3 access to the locks it is holding. However, suppose that as T_3 waits to be granted access to the lock, T_4 enters the system. T_4 is of higher priority than T_3 and passes its priority to T_2. In this situation, as soon as T_2 produces after-images of the lock it has been holding, access to the lock will be granted to T_4 rather than T_3 (Figure 19). As transaction T_4 is being granted access to the lock, T_5 comes into the system. The arrival of T_5 does not make any change to the existing transactions priorities, since T_4 is of a higher priority than T_5; however, as soon as T_4 is done with its execution, access to the lock is directly granted to T_5 which is the next highest priority transaction (Figure 20). Finally, T_1 commits and T_5 completes its execution; as a result T_2 retains x_2 and discards x_1 and T_3 is finally granted access to the lock (Figure 21).

PiSL takes into consideration a transaction's priority when scheduling transactions, but at the same time, it ensures that no work completed by other transactions is wasted. This approach solves the problem of useless restarts of lower priority transactions raised by PSL. By taking

each transaction's priority into consideration during the scheduling process, PiSL gives higher priority transactions a better chance to meet their deadlines. PiSL achieves this by ordering transaction access to data items in a prioritized way, whenever possible. By preventing any preemption and abortion in the system due to transaction priority, lower priority transactions are given a better chance to meet their deadlines as well. Although the approach adopted by PiSL does not guarantee that all the transactions will meet their deadlines, it may be useful in cases where a lower priority transaction has already accomplished a lot of work, or in case where a high priority transaction runs the risk of being preempted by another higher priority transaction as illustrated between T_3 and T_4 in Figures 18-20.

However, under some conditions, PiSL may not be beneficial. These conditions include:

1. If the high priority transaction is close to missing its deadline.

2. When there are not enough resources to quickly process the lower priority transactions that inherited their priority from the higher priority transaction.

3. When the lower priority transactions that inherited their priority from the higher priority transaction still need a lot of time to accomplish their work.

Chapter 4

Simulation Results and Analysis

An extensive set of experiments has been carried out to study and compare the performance of the Priority-based Speculative Locking protocols (Preemptive Speculative Locking (PSL) and Priority Inheritance Speculative Locking (PiSL)) with the Speculative Locking protocol. The Distributed Real Time Transaction Processing System (DRTTPS) simulator [35] was used to conduct these experiments and collect statistical data for performance analysis that is presented in this chapter.

4.1. Assumptions

Some assumptions were made in order to conduct these simulation experiments. These assumptions are as follows:

- When a transaction is created, all the pages required by the transaction are known.
- When a transaction needs to access a data item that is located on a different node than the node where the transaction originated, a subtransaction will be created.
- Once a transaction commits, it cannot be rolled back.

- The hardware (disks, swap disks and processors) are free of failure.
- Access to cache or buffer is instantaneous.
- The network is fully connected.

The above assumptions simplify the design of the simulator, but they do not impact the modelling of a realistic scenario.

4. 2. Performance Metrics

The main performance metric used for performance analysis is the percent of transactions completed on time (PTCT). The PTCT value ranges from 0 to 100 percent and represents all the transactions existing in the system. In the upcoming experiments, achieving a PTCT of 100 percent is the goal. However, due to resource limitations and the nature of distributed real-time database systems, expecting a PTCT of 100 percent for any of the protocols is unrealistic.

The other performance metric used for this experiment is related to the utilization of resources within the distributed real-time database system. These resources are located on each node that constitutes the distributed system and include the processors, disks and swap disks. These additional metrics comprise: (1) percent of processor utilization (PPU), (2) percent of disk utilization (PDU) and (3) percent of swap disk utilization (PSDU). PPU, PDU

and PSDU values vary between 0 and 100 percent. These additional metrics provide an insight into the trends of the utilization of resources for PSL, PiSL and SL protocols.

4.3. Baseline Configuration

The performance evaluation of PSL, PiSL and SL protocols start with the creation of a baseline configuration. The parameter values chosen for the baseline configuration provide a blueprint for the upcoming experiments. The explanation and values chosen for the baseline parameters are provided in Table 3. These values represent a moderately loaded system where transactions are not allowed to share data items. For example, as mentioned earlier, the update probability, set in the Transaction Generator, represents the probability that any given page will be written back to the disk. By setting the *Update* parameter to 100, every data item that is accessed by a transaction will be updated. This prevents transactions from sharing the same data items. In the upcoming experiments, different system parameters are varied to analyze their impact on the performance of PSL, PiSL and SL.

Parameter	Meaning	Value
InterArrivalTime	Time separating the arrival of two consecutive transactions into the system	75 ticks
WorkSize	Number of pages accessed by a transaction	4-12
Update	Percent Probability that a page will be updated	100
SimTransSize	Number of transactions created in the simulator	200

Nodes	Number of nodes in the system	4
MaxActiveTrans	Maximum number of active transactions per node	30
Processors	Number of processors per node	1
ProcTime	Amount of time to process a page	15 ticks
ProcHype	Processor Hyper-Threading	Disabled
Disks	Number of disks per node	2
DiskTime	Amount of time to read a page from the disk	35 ticks
SwapTime	Amount of time to swap a page: from cache/buffer to disk and vice versa	35 ticks
Pages	Number of pages per disk	100
CacheSize	Size of the system cache/buffer per node	75 pages

Table 3: Baseline Configuration parameters

4.4. Running Simulations

The DRTTPS, described in [35], provides a set of tools that can be used to carry out simulations. These tools provide graphical user interfaces to create, manage and analyse results from simulations. These include the setup tool, the simulator tool and the report tool that are used to carry out the simulations whose results are presented in this chapter.

The setup tool (Figure 22) allows the user of the DRTTPS to create the simulation topology, the site structure, the node structure and set the values for the configuration parameters. Also,

the setup tool provides the functionality to initiate the running of simulations and save configurations. As mentioned earlier, the simulation model consists of a number of sites connected by a network. This constitutes the simulation topology. Moreover, each site contains one or more nodes that consist of components such as processors, disks and buffers. The behaviour of these components is controlled through the manipulation of parameters.

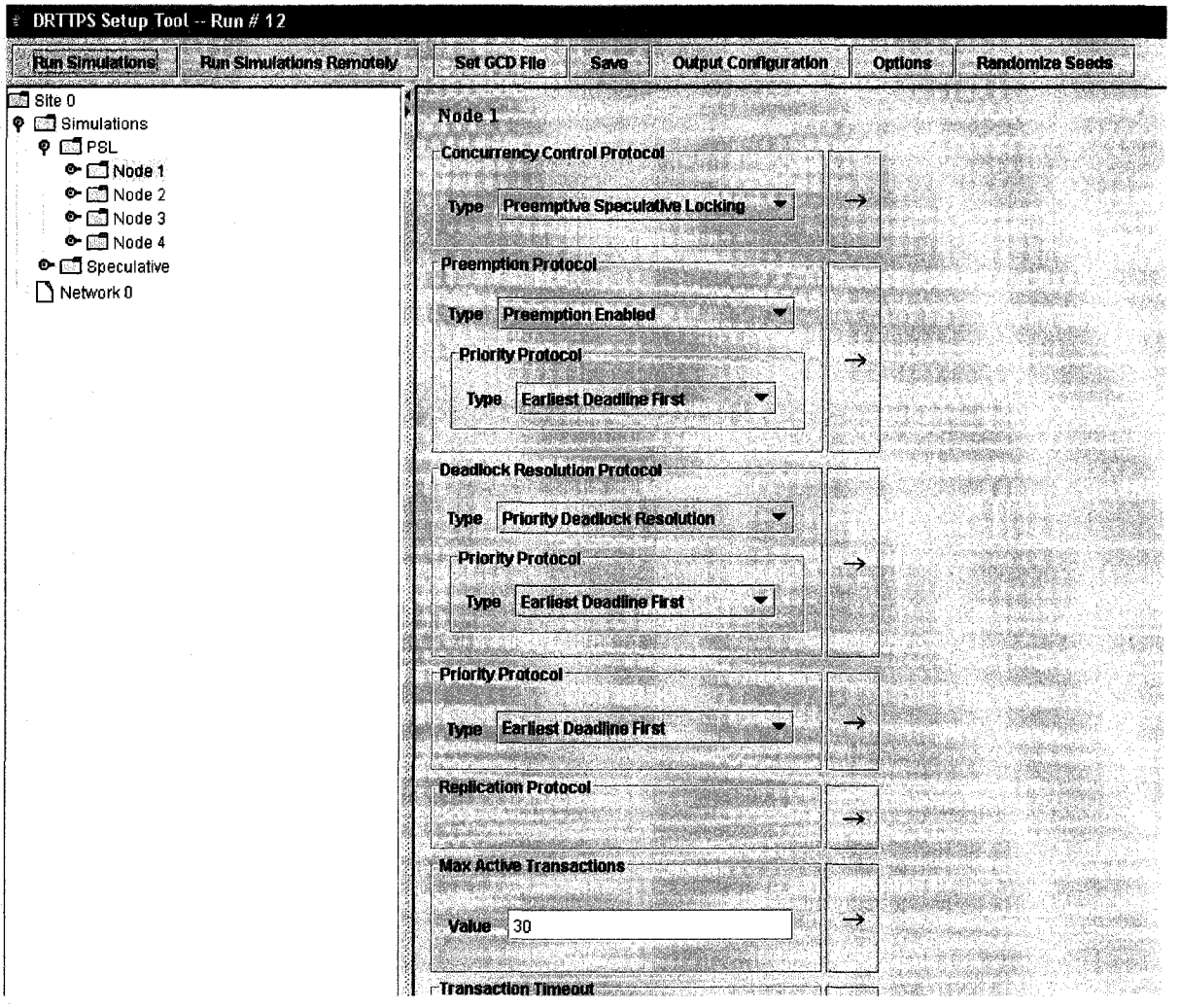

Figure 22: DRTTPS Setup Tool

The simulator tool (Figure 23) takes the configuration from the setup tool and runs the simulations, generating simulation statistics. These statistics are represented by graphs that are displayed in real time. The simulator tool also provides a real time description of event portraying the progress within each component of a node. This provides an insight on how the components interact during the simulation.

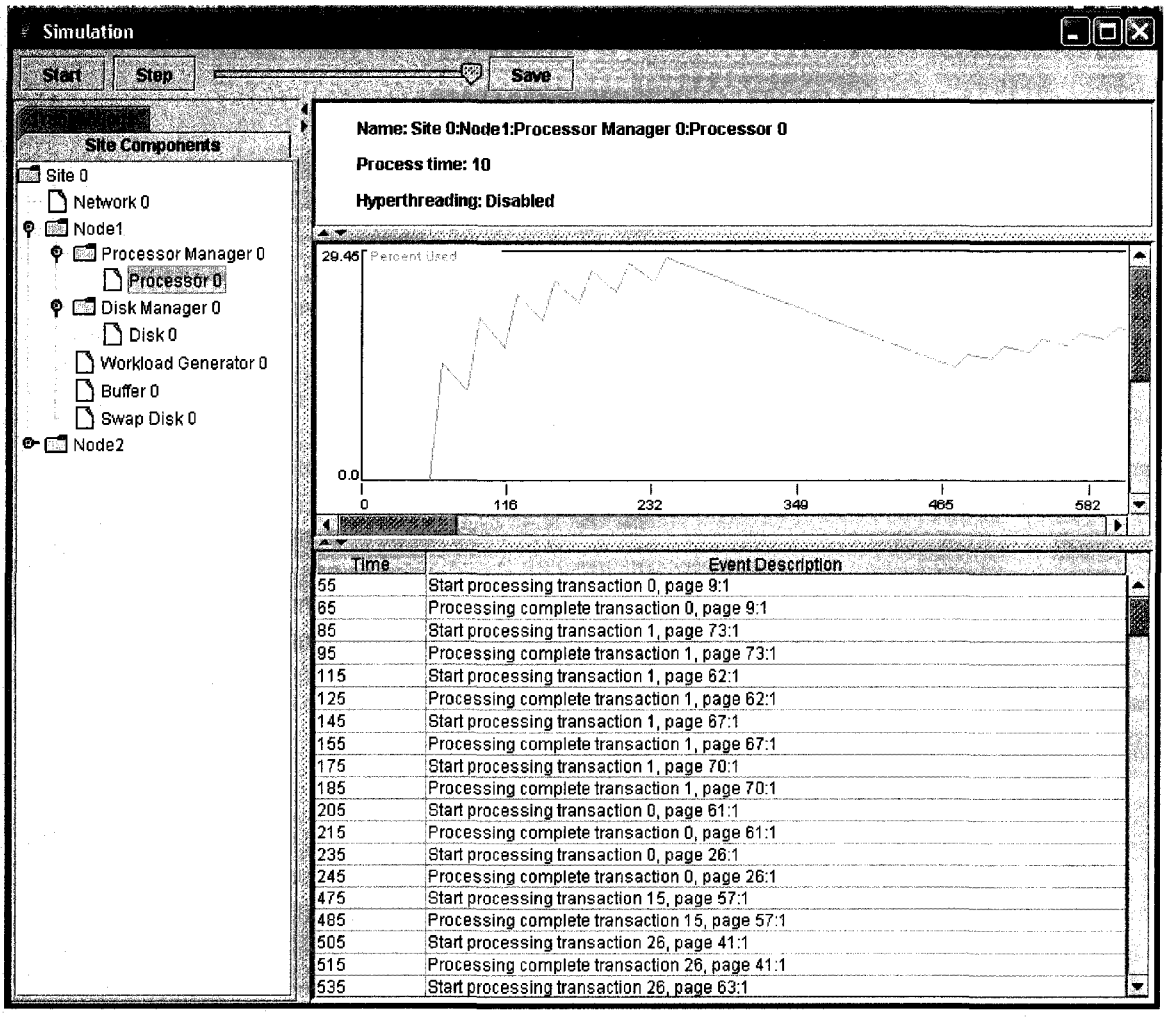

Figure 23: DRTTPS Simulator Tool

The report tool (Figure 24) is used to view the graphs and statistics generated by the simulator tool once the simulations complete. The statistics contained in the report tool are used to generate the graphs that are presented in the upcoming experiments.

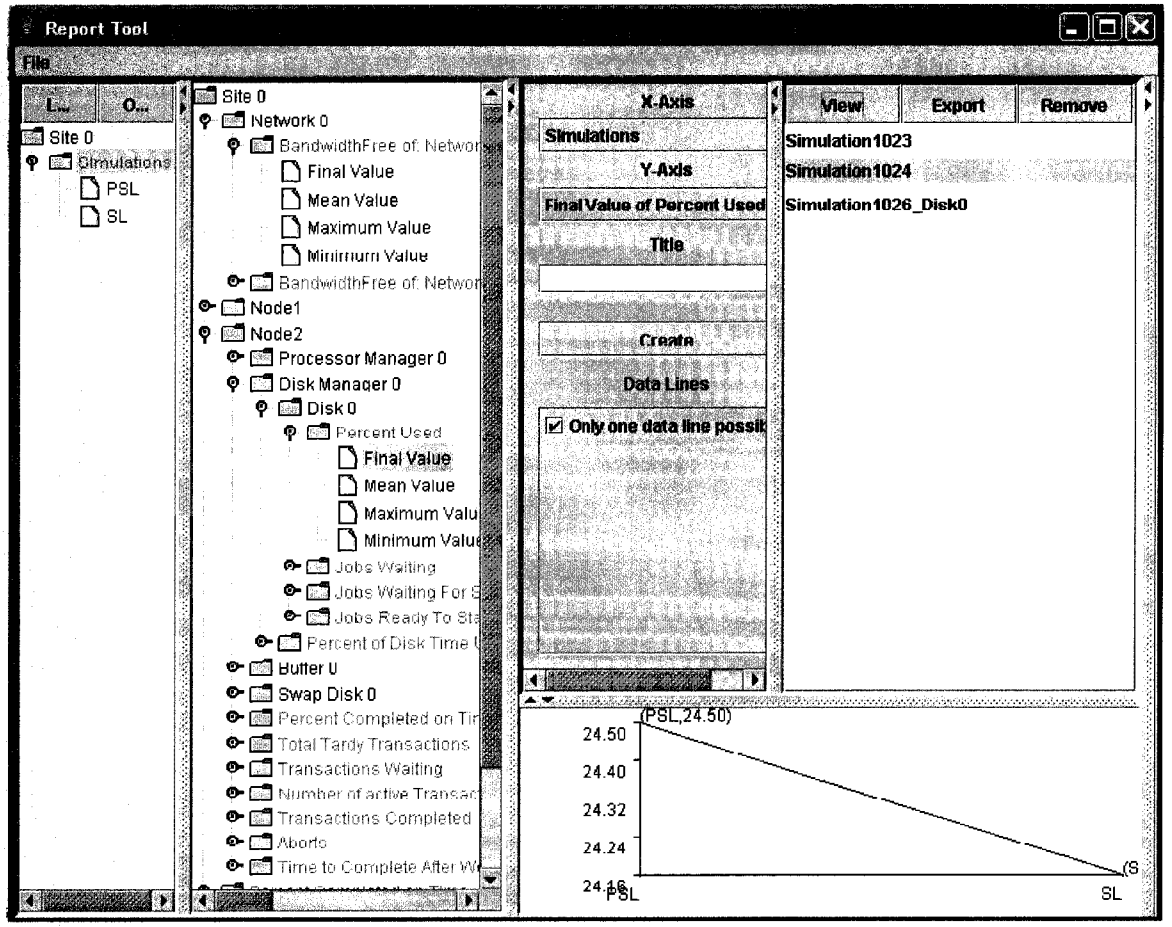

Figure 24: DRTTPS Report Tool

4.5. Experiment 1: Baseline Simulations.

Using the baseline configuration, this first set of experiments analyses the impact of the *InterArrivalTime, WorkSize, Processors, MaxActiveTrans* and *CacheSize* on the performance of PSL, PiSL and SL protocols. These parameters are varied to observe how they affect the behaviour of these protocols. This also allows the comparison of performance of protocols against one another.

4.5.1. Arrival Rate

In this experiment the value of the *InterArrivalTime* parameter is varied. The *InterArrivalTime* determines the number of ticks that separate the creation of two subsequent transactions, by the workload generator. The metric used in this experiment is the PTCT, and the results are presented in Figure 25.

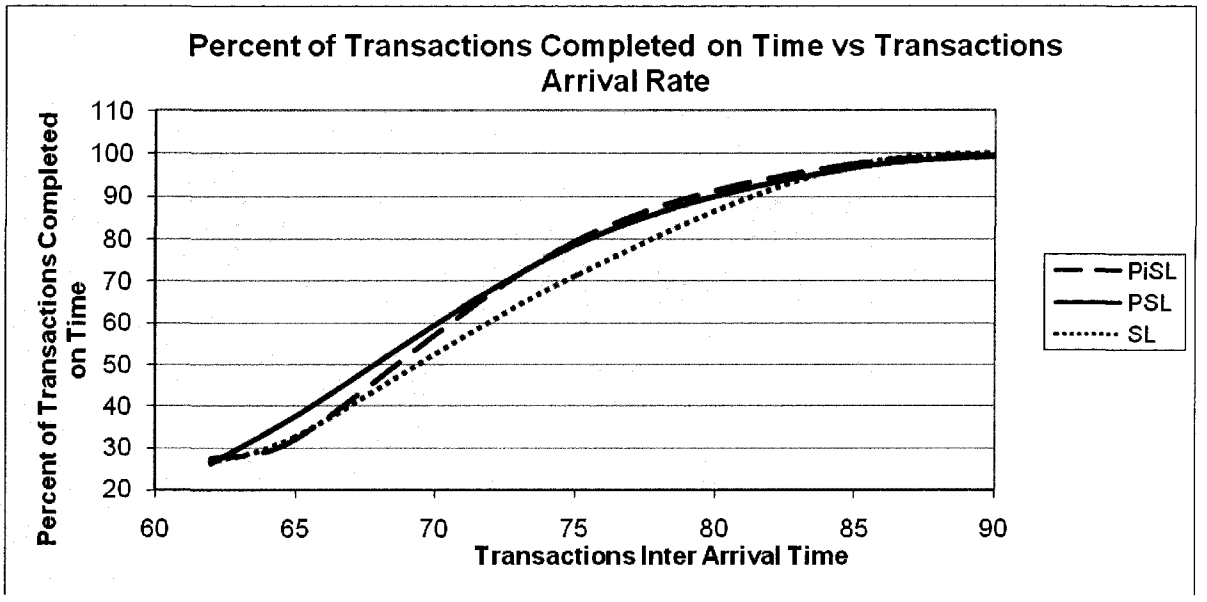

Figure 25: Experiment1-InterArrivalTime: PTCT for baseline configuration

Figure 25 shows that the *InterArrivalTime* has a direct impact on the performance of all the protocols. At lower values, the performance of each of the protocols is low. As the *InterArrivalTime* increases, the performance of each of the protocols also increases. This is due to the fact that a slower arrival rate of transactions results in a lower system load, whereas a faster arrival rate of transaction in the system results in a higher system load. Also, the higher the system load, the higher the competition for data items and the lower the performance of the protocols and vice versa. However, it should be noted that regardless of the value of the InterArrivalTime, PSL and PiSL outperform SL.

4.5.2. Work Size

In this experiment, the value of the *WorkSize* parameter is varied. The *WorkSize* parameter represents the number of pages that each transaction in the system needs to process in order to complete. The value of the *WorkSize* parameter is specified in the form of a range of numbers (2-12, 3-12… 6-12). This range defines a set of possible values for the number of pages to be processed by each transaction within the system. For example, a *WorkSize* of 4-12 specifies that each transaction within the system must process between 4 pages and 12 pages. The actual number of pages accessed is determined by a chosen distribution that uses the specified range. This experiment allows the analysis of the impact of *WorkSize* on the performance of PSL, PiSL and SL. PTCT is the metric used to measure performance in this experiment. The results from this experiment are presented in Figure 26.

By determining the number of pages that each transaction in the system need to process, the *WorkSize* parameter indirectly determines how long a transaction needs to stay in the system. The more pages that a transaction needs to process, the longer it will likely take to complete. Also, the period that a transaction stays in the system has a direct impact on the system load. The longer transactions remain in the system, the higher the system load will be. Further, the more pages each transaction in the system needs to process, the higher the probability of data conflict will be, due to a higher level of competition for data items. Due to these two factors (system load and probability of data conflict) the *WorkSize* has the potential to affect the performance of PSL, PiSL and SL.

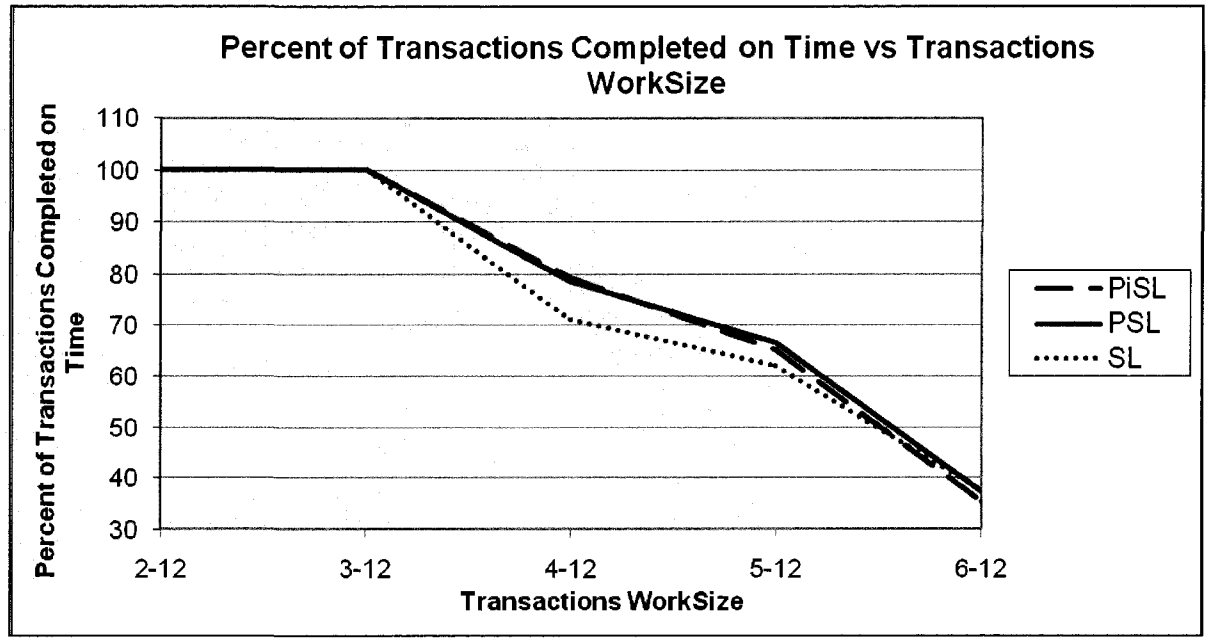

Figure 26: Experiment 1-WorkSize: PTCT for the baseline configuration

Figure 26 shows that when the *WorkSize* has lower values, the performance of each of the protocols is relatively high. This is due to the fact that when the *WorkSize* is low, the system is less loaded, and data conflict is relatively rare. However, as the *WorkSize* increases, the system load increases, as well as the probability of data conflict. This results in decreased performance for each of the protocols. However, Figure 26 clearly shows that when one compares the protocols to each other, PSL and PiSL perform better than SL regardless of the workload.

4.5.3. Maximum Active Transactions

In this experiment, the value of *MaxActiveTrans* parameter is varied. The *MaxActiveTrans* parameter determines the maximum number of transactions that are allowed to be active at the same time within a node. As soon as this maximum value is reached within a node, the system will no longer allow any other transaction to enter the node. Consequently, all incoming transactions are queued outside the node until there is space available in the node again. Hence, if the value of the *MaxActiveTrans* is low, few transactions can be simultaneously active in a node. On the other hand, if the value of the *MaxActiveTrans* is high, then many active transactions are allowed to reside simultaneously in a node. It should also be noted that when a transaction is queued outside a node, it is in an inactive mode, meaning it is not allowed to access any resources or data items located within the node. Therefore, if a transaction is queued outside a node for too long, it may miss its deadline. Consequently, if the value of the MaxActiveTrans is set to a very low value, the resources of a node may be underutilized.

This experiment allows us to evaluate the impact of the *MaxActiveTrans* parameter on the performance of the protocols. The metric used for this experiment is the PTCT. The results from this experiment are presented in Figure 27.

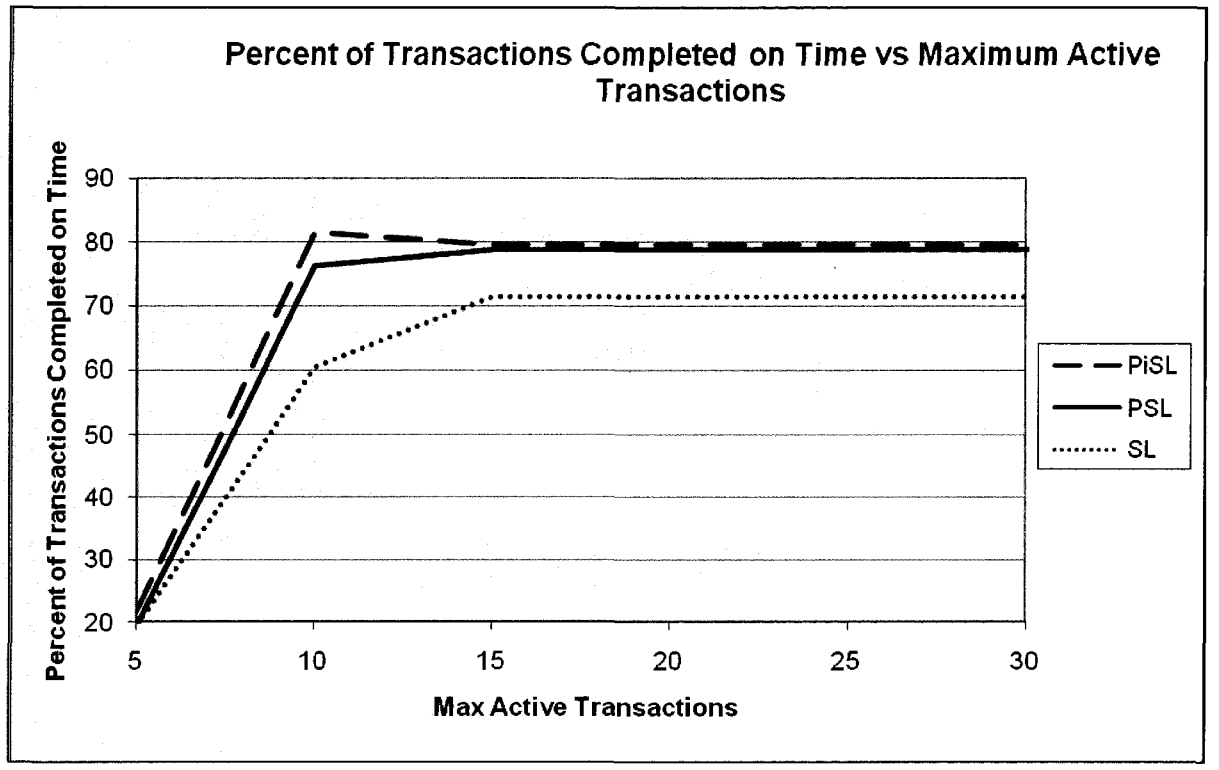

Figure 27: Experiment 1-MaxActiveTrans: PTCT for the baseline simulation

Figure 27 shows that at low values of *MaxActiveTrans*, the performance of each of the protocols drops. This is due to the underutilization of resources within each node. As the *MaxActiveTrans* value increases, the performance of the protocols quickly increases. However, something interesting happens: instead of the performance of the protocols continuously increasing as the *MaxActiveTrans* increases, performance stabilizes at a certain point. As the value of *MaxActiveTrans* increases, the number of active transactions within each node increases. This eventually results in maximum resource utilization within each node. However, due to the fact that resources in a node are limited, and data items are accessed in a controlled manner, transactions that are waiting for access to resources or data items are queued within the node. Hence, after the *MaxActiveTrans* value hits a certain point,

any further increase to this value no longer impacts the performance of the protocols. However, it should be noted that regardless of the value of *MaxActiveTrans*, PiSL and PSL once again outperform SL protocol.

4.5.4. Number of Processors

In this experiment, the *Processor* parameter is varied. The *Processor* parameter represents the number of processors per node. As mentioned earlier, each node can have one or more processors. Each processor can only process one page at a time. This experiment allows us to evaluate the impact of the number of processors per node on the performance of the protocols. The metric used for this experiment is the PTCT. Figure 28 presents the result of this experiment.

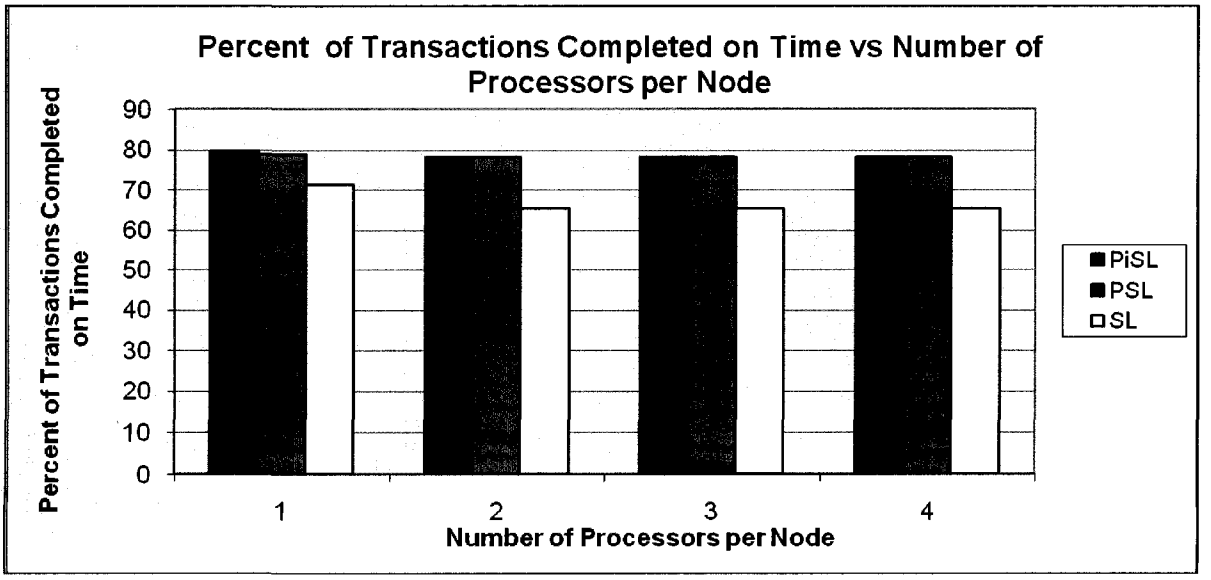

Figure 28: Experiment 1-Processors: PTCT for the baseline simulation

Figure 28 shows that increasing the number of processors per node does not result in an increase in performance for any of the protocols. In fact, when there is only one processor per node, the performance of the protocols is slightly higher. As the number of processors increases to 2 processors per node and more, the performance of the protocols slightly decreases then stabilizes for any *Processor* value greater than 2. The slight decrease in performance as the number of processors increases from 1 to 2 is due to processor overhead. However, regardless of the number of processors, PiSL and PSL outperform SL. The reason why the number of processors per node does not affect the performance of the protocols is due to the fact that processors are being underutilized. Thus increasing their number does not improve performance. This will be further discussed in the upcoming experiments on system resources utilization.

4.5.5. System Cache

In this experiment, the *CacheSize* parameter is varied. The *CacheSize* parameter represents the size of system cache, also known as the buffer, at each node. The value of *CacheSize* determines the number of pages that the system cache can hold. As mentioned earlier, when a transaction requires access to a page, it first checks if the page is in the system cache. If the page is not available in the system cache, then the page must be fetched from the disk and loaded into the system cache before it can be accessed by the transaction. All the operations that the transaction needs to perform on the page occur while the page resides in the system cache. A page is moved back to the disk only when a transaction is done with it. Hence, when the number of pages held in the system cache is equal to the value of the *CacheSize*

parameter, no more pages can be loaded in the system cache. In this case, transactions that need access to pages that are not in the system cache will either have to wait, or some of the pages residing in the system cache will have to be swapped out to the swap disk to create space in the system cache.

In this experiment, the impact of the system cache on the performance of the protocols is analyzed. The metric used for this experiment is the PTCT. The result of this experiment is presented in Figure 29.

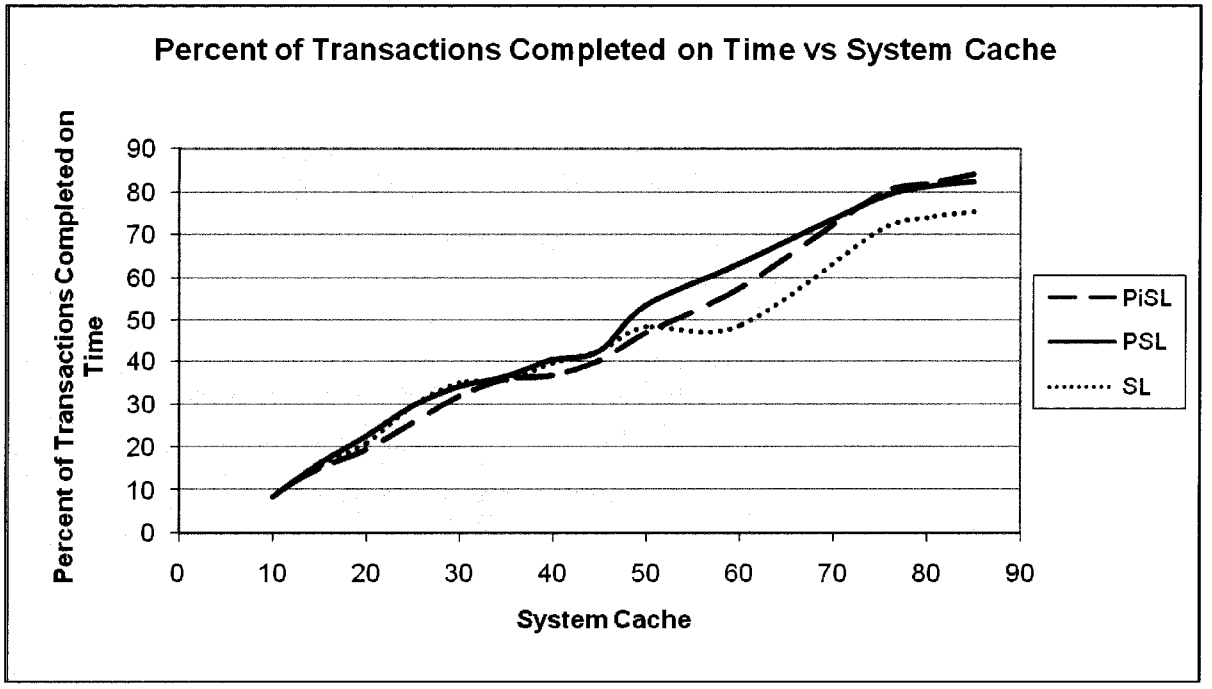

Figure 29: Experiment 1-System Cache: PTCT for baseline simulation

Considering that transactions perform operations only on pages residing in the system cache, in speculative based protocols, all of the speculative executions being performed place a

considerable demand on the system cache. Hence, the size of the cache has a direct impact on the performance of the PiSL, PSL and SL protocols. In this experiment, it can be observed that the performance of the protocols increases as the size of the system cache increases. At low *CacheSize* values, the performance of PiSL is below the performance of PSL and even SL at times. However as the value of *CacheSize* increases, PiSL performance picks up and surpasses the performance of SL and ends up slightly outperforming PSL. In subsequent experiments, more investigation will be conducted to further analyze the impact of the system cache size on the performance of PiSL, PSL and SL.

4.6. Experiment 2: System Resources Utilization

In this set of experiments, the utilization of system resources (disks, processors and swap disks) is analyzed under various system cache values. A comparison of the different protocols regarding their utilization of resources is also explored. Furthermore, this experiment provides some clarification on some of the protocols behaviours that were not fully explained in Experiment 1.

4.6.1. Swap Disk Utilization.

When a transaction needs to access a page, and there is no space available in the cache, then some of the pages currently in the cache may temporarily be swapped to a swap disk to create space in the system cache. Hence, the swap disk is used only when there is not enough space in the system cache. This experiment allows us to analyze the utilization of the swap disks by

each of the different protocols as the system cache size varies. The metric used in this experiment is the PSDU.

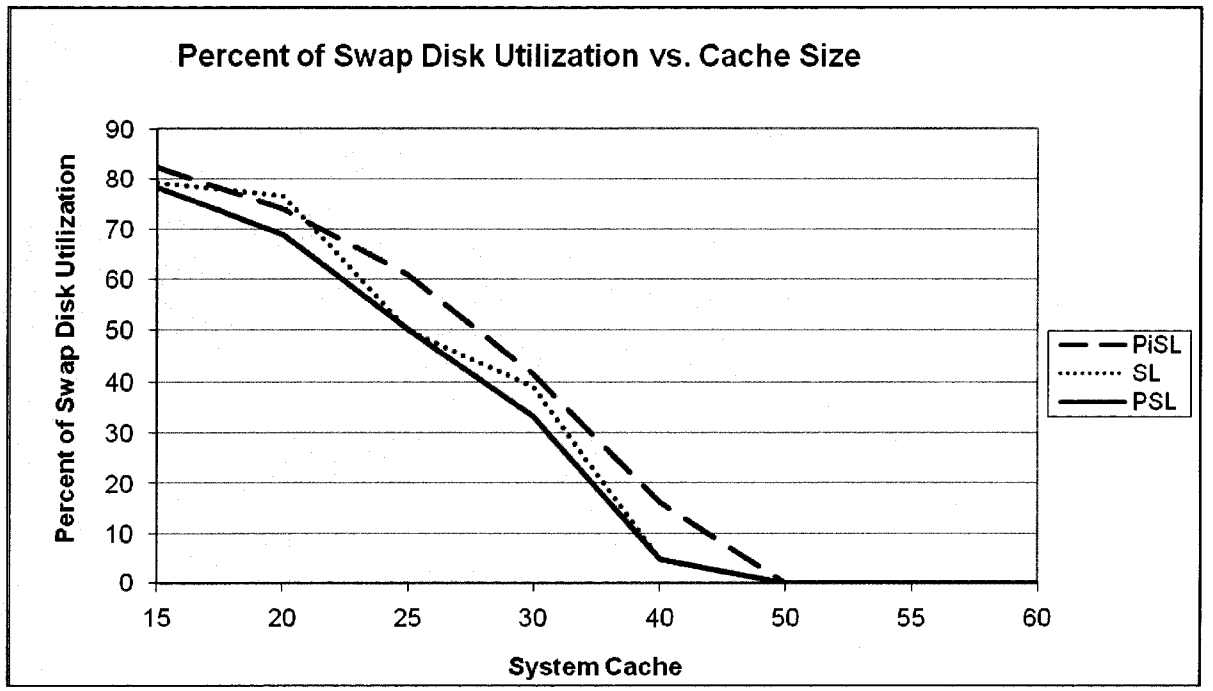

Figure 30: Experiment 2-Swap Disk: PSDU – System Resource Utilization

Figure 30 shows that the utilization of the swap disk is significantly dependent on the size of the system cache. For smaller system cache, the utilization of the swap disk increases, going as high as 80% for a cache size of 15. This is due to the fact that demand on the system cache increases as transactions enter the system, since the cache must hold all the pages required by active transactions. As the system cache runs out of space, some of the pages currently residing in it must be swapped to the swap disk in order to make more cache space available. This results in increased utilization of the swap disk. Another interesting result that needs to be noted in the comparison of the protocols in term of the utilization of the swap disk, is that PSL seems to utilize the swap disk least. This observation is further explored in upcoming experiments.

4.6.2. Disk Utilization

This experiment analyzes the utilization of the regular disk by each of the different protocols as the system cache size is varied. The metric used in this experiment is the PDU.

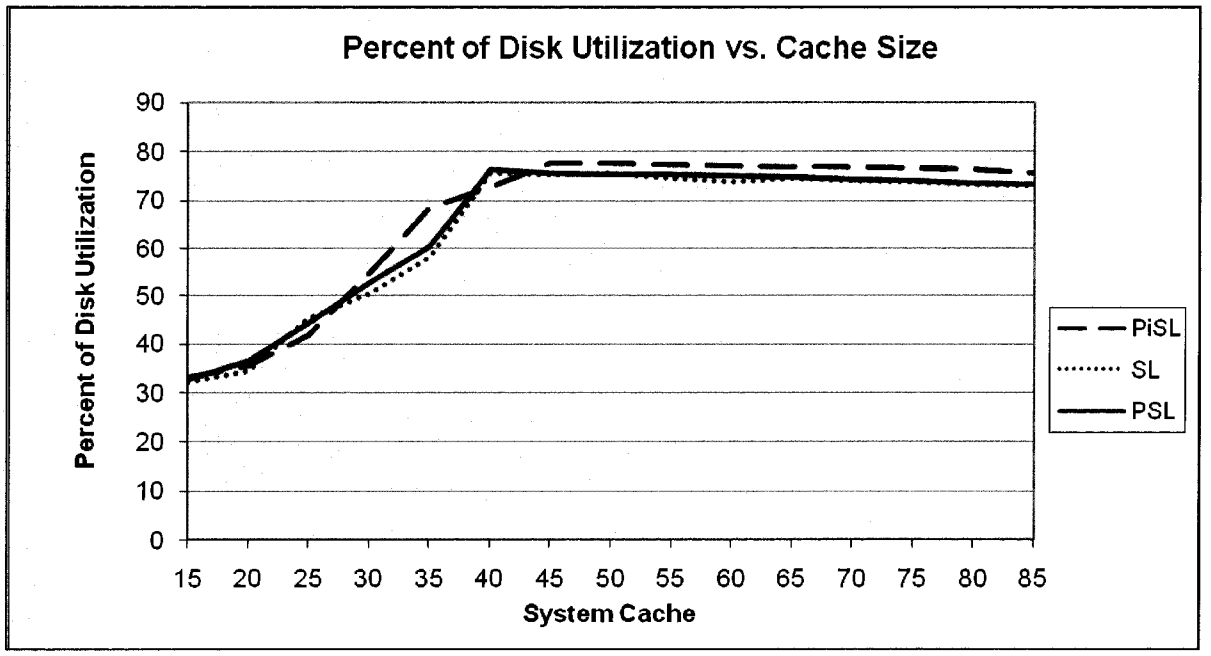

Figure 31: Experiment 2-Disk: PDU – System Resource Utilization

Figure 31 reveals some interesting output: the size of the system cache where PDU starts to drop corresponds to the size of the system cache where the PSDU starts to increase (Figure 30). This is due to the fact that pages are not being read or updated in this situation, due to an intensive movement of pages between the system cache and the swap disk. In this case, the utilization of the disk is essentially traded for utilization of the swap disk. This results in an overall decrease in the performance of each of the protocols. Also, when comparing the disk

utilization of PiSL, PSL and SL, it can be observed from Figure 31 that the overall disk utilization is the same for each of the protocols.

4.6.3. Processor Utilization

In this experiment, the utilization of the processor is analyzed for each of the different protocols, as the system cache size varies. It should be noted that, only one processor is used per node for this experiment. The metric used for this experiment is the PPU, and the result is presented in Figure 32.

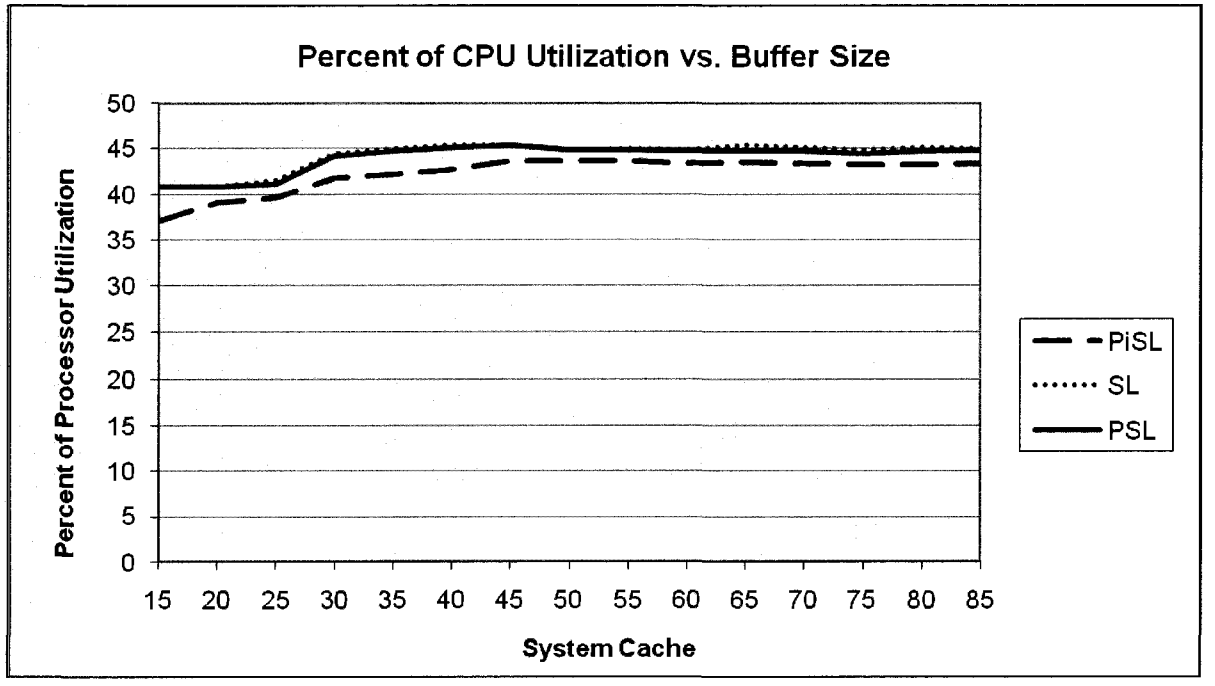

Figure 32: experiment 2-Processor: PCU – System Resource Utilization

Examining Figure 32, one may make two major observations. First, similar to what was observed in Figure 31, one may note that there is a slight decrease in processor utilization when the size of the system cache is too low. Once again, this is due to the fact that pages are moving back and forth between the system cache and the swap disks instead of actually being processed. Similar to the disk, the utilization of the processor is traded for the utilization of the swap disk.

The second observation is related to experiment 1. Figure 32 reveals that the PPU stabilizes at a certain system cache size, and thereafter does not change despite of an increase in the value of the system cache. This reveals that the processor is never fully utilized. This output clarifies the results of the analysis of the impact of the number of processors on the PTCT in experiment 1 (Figure 28). In that experiment, it was found that increasing the number of processors per node did not affect the increase in the PTCT. It can now be concluded that the lack of impact of the number of processors on the PTCT, as observed in Figure 28, is due to the fact that the processor is never fully utilized.

4.7. Experiment 3: Small System Cache

In this set of experiments, the size of the system cache is reduced to analyze the overall performance of the protocols under different system loads and resources availability. The impact of the *InterArrivalTime*, *WorkSize* and *MaxActiveTrans* parameters on the performance of the protocols is analyzed for low system cache sizes. For this experiment the value of the *CacheSize* is reduced to 60. The performance metric used in this experiment is PTCT.

4.7.1. Arrival Rate

In this experiment, the impact of the *InterArrivalTime* on the performance of the protocols, when the system cache size is small, is analyzed. The result of this experiment is presented in Figure 33.

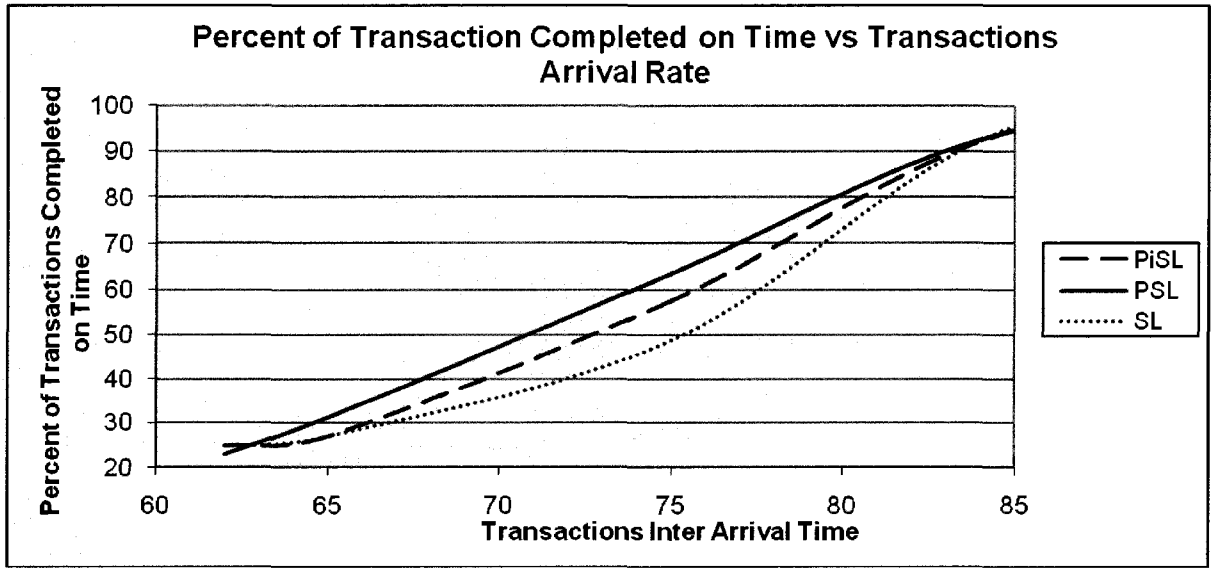

Figure 33: Experiment 3-InterArrivalTime: PTCT for Small System Cache

As in experiment 1, the performance of the protocols increases as the value of the *InterArrivalTime* parameter increases. However, in contrast to experiment 1 (Figure 25), PSL performs best here, followed by PiSL and finally SL. It should also be noted that, as shown by Figure 33, the difference in performance between PSL and SL is bigger in this experiment than in experiment 1. This is due to the fact that with PSL, when a lower priority transaction is aborted, its respective pages are removed from the system cache, freeing up space in the system cache. Moreover, as shown in experiment 1- Figure 29, the size of system cache has a direct impact on the performance of all the protocols. A small system cache has a negative impact on the performance of all the protocols. Hence, by freeing some space within the system cache, especially for system with small system cache where more space is really needed, PSL further increase the overall system performance.

4.7.2. Work Size

In this experiment, the impact of the *WorkSize* on the performance of the protocols is analyzed, in conditions where the system cache size is small. The result of this experiment is presented in Figure 34.

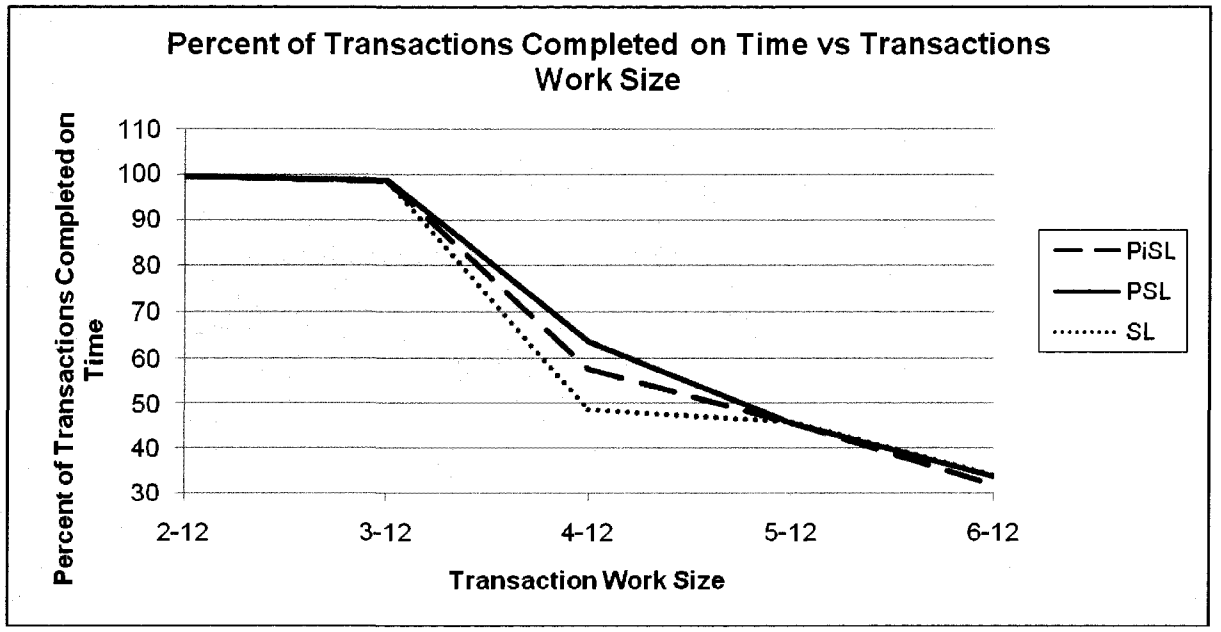

Figure 34: Experiment 3-WorkSize: PTCT for Small System Cache

As in experiment 1, the performance of the protocols drops as the value of the *WorkSize* increases. Unlike in experiment 1 (Figure 26), PSL has the best performance, followed by PiSL and finally SL. However, as shown in Figure 34, when the *WorkSize* reaches a certain value, the gap between the protocols starts to decrease, as so does the overall performance of each of the protocols. This is due to the fact that the *WorkSize* influences the system load as well as data contention. As the *WorkSize* increases, the system load and data contention increase. In a case of a system with limited system cache, once the *WorkSize* reaches a certain value, the demand on the system cache added to the increased data contention adversely affect all the protocols and result in poor system performance.

4.7.3. Maximum Active Transactions

In this set of experiments, the impact the *MaxActiveTrans* parameter on the performance of the protocols is analyzed in conditions where the system cache size is small. The result of this experiment is presented in Figure 35.

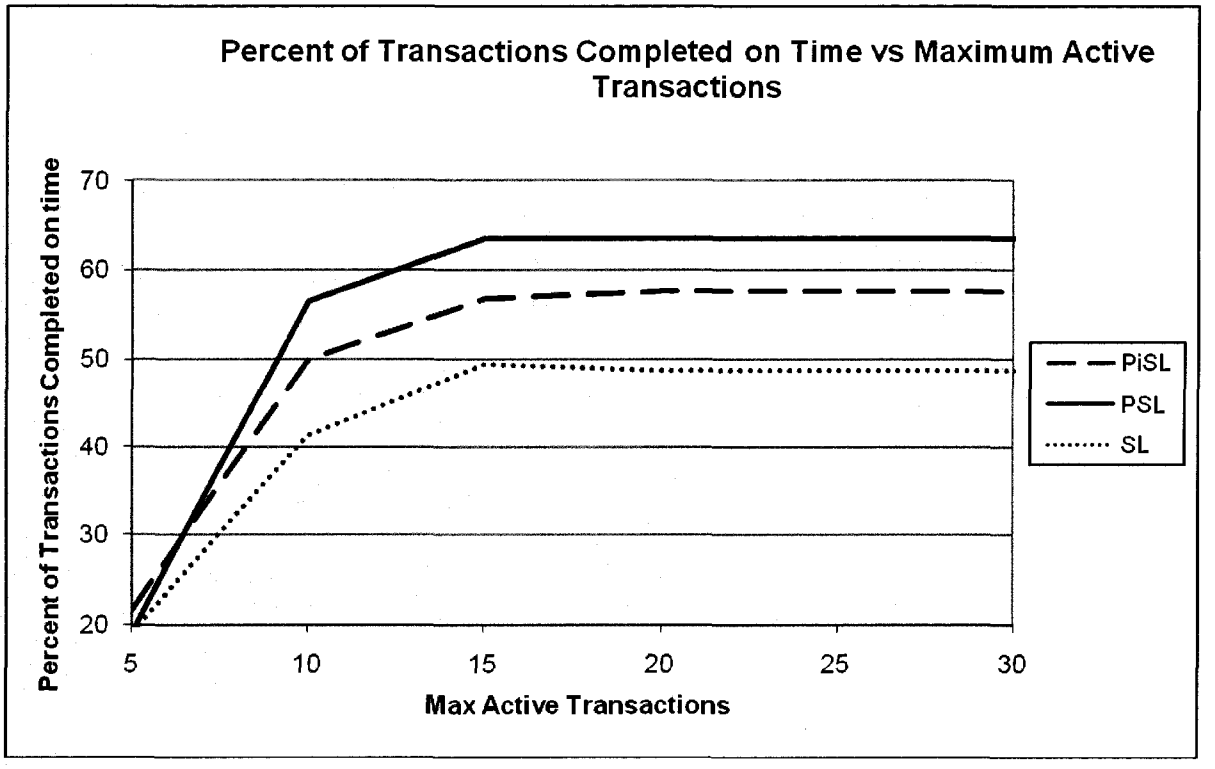

Figure 35: Experiment 3-MaxActiveTrans: PTCT for Small System Cache

The output of this experiment shows the same trend as the output of experiment 1(Figure 27). However, PSL shows a higher level of performance here compared to the other protocols, followed by PiSL, and finally SL. It should also be noted that the gap, in terms of performance, between PSL and SL has significantly increased. This is due, as previously explained, to the ability of PSL to free up space in the system cache when the space is really needed.

4.8. Experiment 4: Large System Cache

This experiment is similar in nature to experiment 3. However, in this experiment, the value of the *CacheSize* parameter has been raised to 85. Hence, this experiment allows us to analyze the impact of the *InterArrivalTime*, *WorkSize* and *MaxActiveTrans* parameters on the performance of the PiSL, PSL and SL protocols when the system cache size is relatively large. The metric used for this experiment is the PTCT.

4.8.1. Arrival Rate

In this experiment, the impact of the *InterArrivalTime* on the performance of the protocols, under large system cache, is analyzed. The result of this experiment is presented in Figure 36.

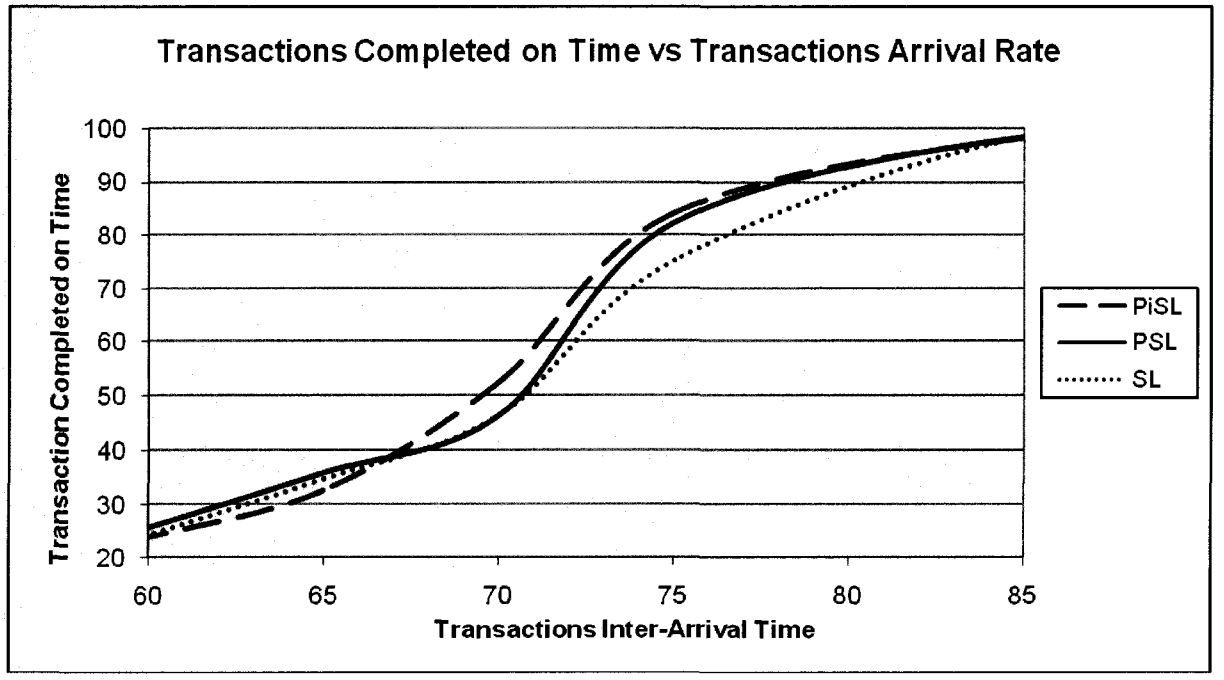

Figure 36: Experiment 4-InterArrivalTime: PTCT for Large System Cache

As in experiment 1 (Figure 25) and experiment 3 (Figure 33), the performance of each of the protocols increases as the value of the *InterArrivalTime* increases. However, in contrast to experiment 3 (Figure 33), the performance of PiSL is better than the performance of PSL, except when the value of the *InterArrivalTime* is too low. As mentioned earlier, when the value of *InterArrivalTime* decreases, the system load increases. Hence, as shown in Figure 36, PiSL does not perform well when the system load is too high, even when the system cache is relatively large.

As the system cache increase, the ability of freeing up space in the system cache provided by PSL looses its impact on the performance of the protocols. In this case the performance of PiSL is better than the performance of PSL. This is due to the fact that with PiSL, there is no abortion of transaction that results in wasting of work. Moreover, in the case of a large system cache, there are enough resources to quickly process the lower priority transactions that inherited their priority from the higher priority transaction. This gives a chance to both higher and lower priority transactions to meet their deadline. However, as the *InterArrivalTime* decrease, there is more and more demand on the system cache due to the increased system load. This results in the decrease of performance for PiSL.

4.8.2. Work Size

In this experiment, the impact of the *WorkSize* on the performance of the protocols, under large system cache situations, is analyzed. The result of this experiment is presented in Figure 37.

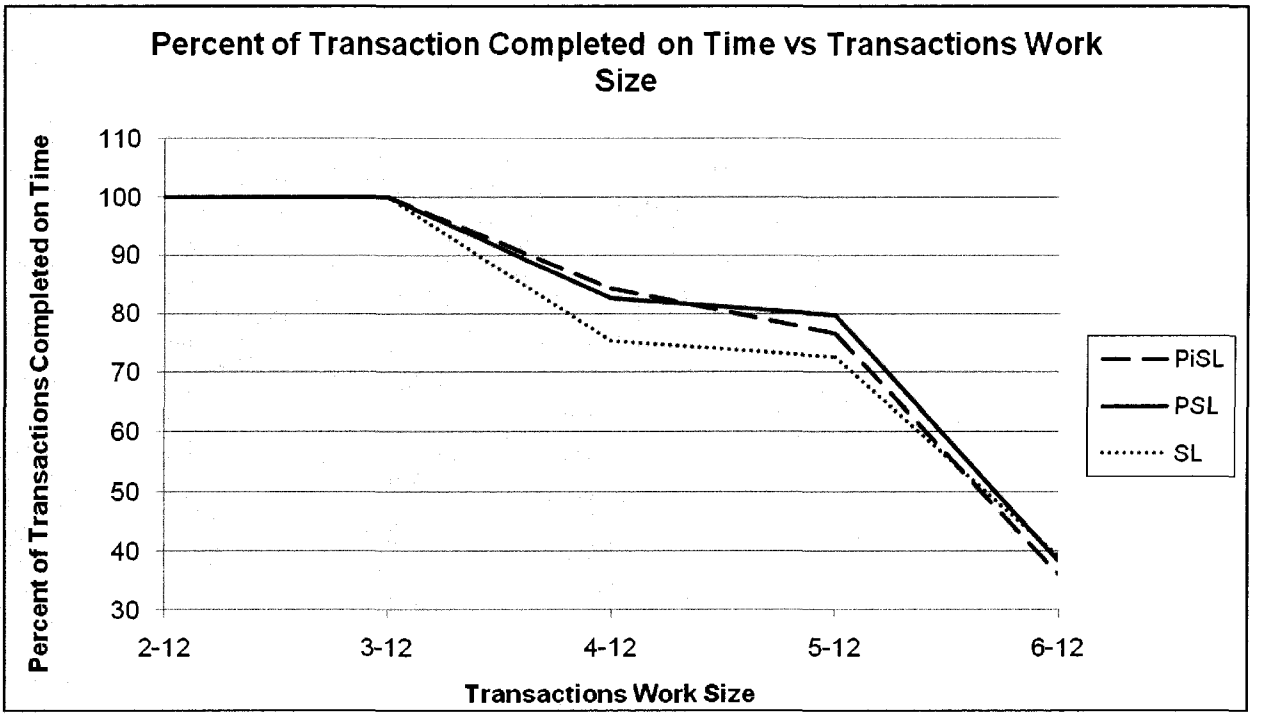

Figure 37: Experiment 4-Work Size: PTCT for Large System Cache

The output of this experiment is similar to the output in experiment 1 (Figure 26) and experiment 3 (Figure 34); the performance of the protocols decreases as the value of *WorkSize* increases. However, in contrast to experiment 3 (Figure 34), the performance of PiSL does not lug behind PSL. In fact, for low *WorkSize* values, PiSL slightly outperforms

PSL. However, as the value of the *WorkSize* increases the performance of PiSL drops below the performance of PSL.

This is due, as previously explained, to the ability of PiSL to avoid waste of work, by preventing any transaction abortion, combined with the fact that, in a system with large system cache, there are enough resources to quickly process lower priority transactions that inherited their priority from higher priority transactions. However, as mentioned in experiment 1, the *WorkSize* indirectly determines the system load. Hence, as the *WorkSize* increases, the system load increases, as well as the demand on system cache. This negatively affects the performance of PiSL, but, at the same time, the ability of PSL to free up space within the system cache gives PSL a better performance than PiSL.

4.8.3. Maximum Active Transactions

In this experiment, the impact of the *MaxActiveTrans* on the performance of the protocols, under high system cache situations, is analyzed. The result of this experiment is presented in Figure 38.

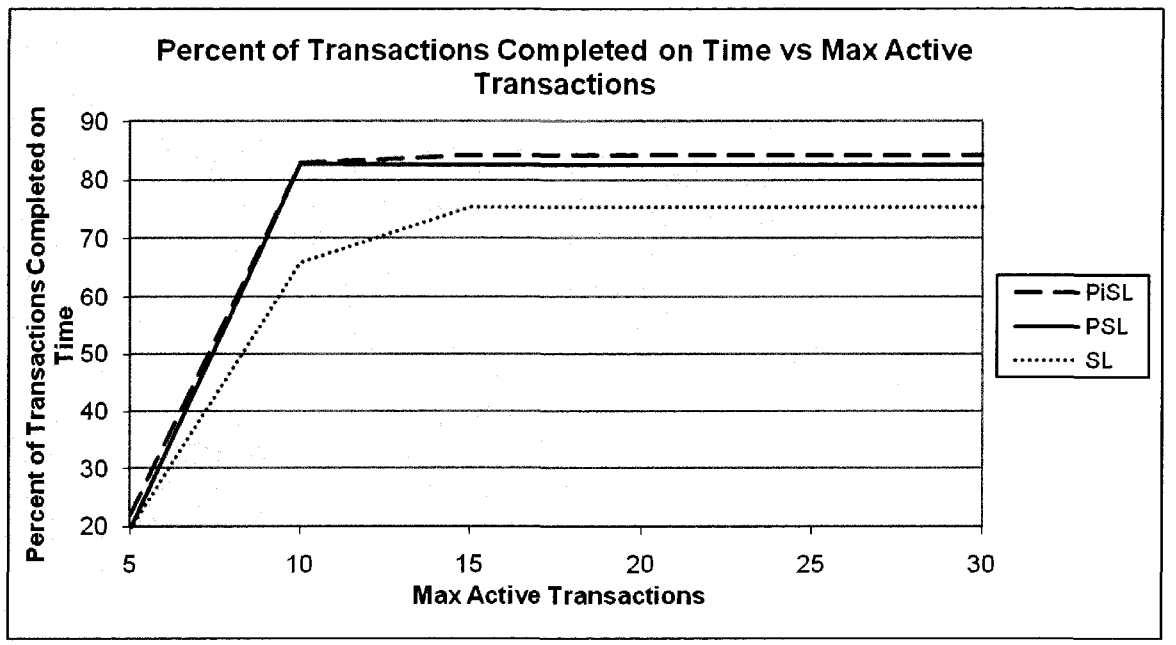

Figure 38: Experiment 4-MaxActiveTrans: PTCT for Large System Cache

The trend in performance for the protocols in this experiment is similar to what was shown in experiment 1 (Figure 27) and experiment 3 (Figure 35). However, in this experiment, unlike in experiment 3 (Figure 35), the performance of PiSL is better than the performance of PSL. This is due, as previously explained, to the ability of PiSL to avoid waste of work combined with the advantage that a large system cache provides to PiSL.

4.9. Experiment 5: Swap Disk

The impact of the *CacheSize* on the utilization of the swap disk was analyzed in Experiment 2 (Figure 30). The output of that experiment showed that the swap disk is only used when the value of *CacheSize* is less than 50. In the previous experiments, the swap disks were not utilized since the value of *CacheSize* was greater than 50. In this experiment, the behaviour of the PiSL, PSL and SL protocols is analyzed when the swap disk is used. Hence, in this experiment, the value of *CacheSize* is reduced to 20.

This set of experiments serves two purposes:

1. The analysis of the impact of *WorkSize* and *InterArrivalTime* on the performance of the protocols, when swap disks are being utilized. These two parameters have been chosen due to the fact that they control the system load and data contention within the system.

2. As the impact of *WorkSize* and *InterArrivalTime* is investigated, the utilization of the swap disk is studied, under various system loads and data contention levels, for the different protocols.

The metric used for this experiment is the PTCT and the PSDU. The results of these experiments are presented in Figure 39-42.

4.9.1. Work Size

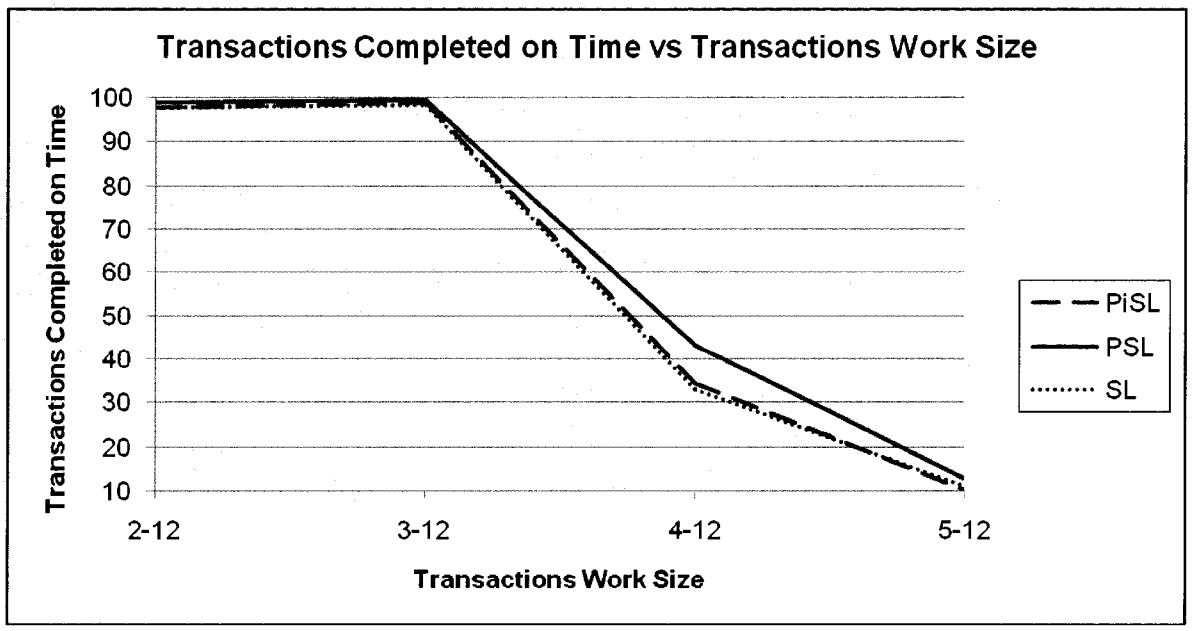

Figure 39: Experiment 5-WorkSize: PTCT – Swap Disk

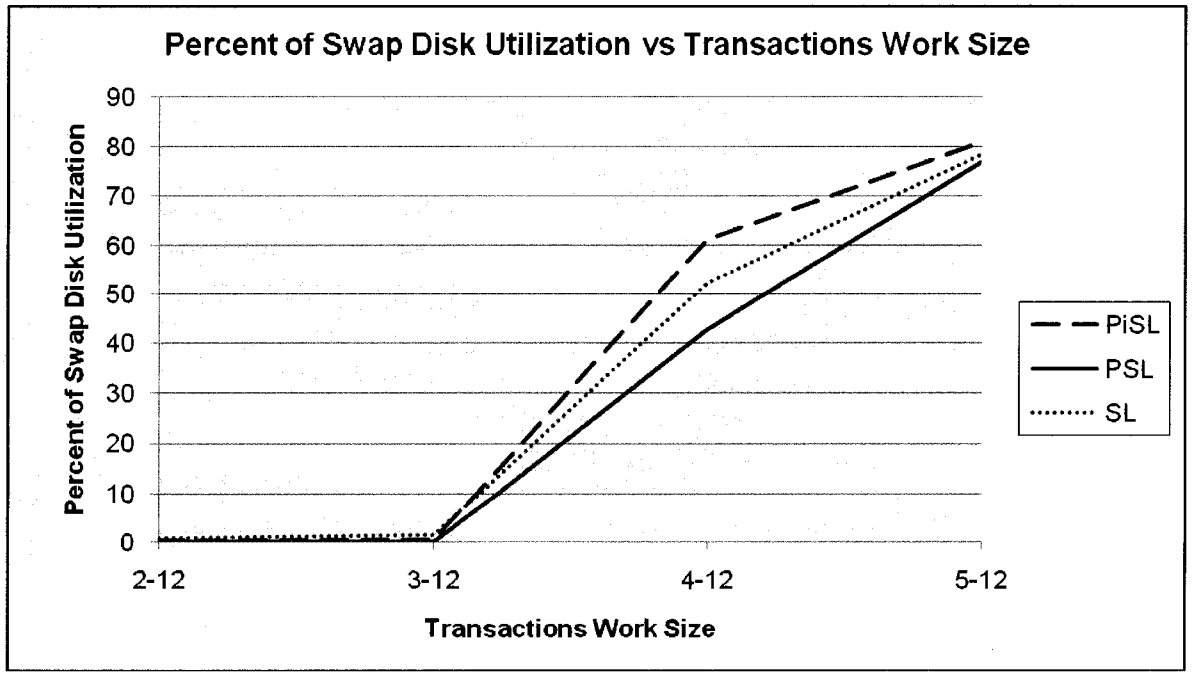

Figure 40: Experiment 5-WorkSize: PSDU – Swap Disk

Figure 39 and Figure 40 show that the utilization of the swap disk is not totally dependent on the size of the system cache. Even though the size of the system cache used in this experiment is very small, the swap disk is not utilized when transactions work sizes are smaller. Hence, it can be concluded that the utilization of swap disk is dependent on both the size of the cache and the amount of data contention.

When the swap disks are not being utilized, all of the protocols come close to a perfect performance. However, as data contention increases, the swap disks start being utilized, and at this point, the performance of all the protocols starts to drop. This is due to the increase in movement of pages between the system cache and the swap disks. In this case, as shown in experiment 2 (Figure 31, Figure 32), the utilization of the disk and the processor is traded for the utilization of the swap disks. This results in decreased performance for each of the protocols. However, when the protocols are compared to each other, PSL outperforms the other protocols.

Finally, it should be noted that when comparing the performance of PSL and PiSL, there seems to be a direct connection between their performance and their utilization of the swap disks. Figure 40 shows that PSL uses less swap disks than PiSL. This is due to the fact that with PSL, when a lower priority transaction is preempted and aborted, its respective pages are removed from the system cache, freeing up space within the system cache which results in lower utilization of the swap disk, and increases the overall system performance.

4.9.2. Arrival Rate

Figure 41 shows the impact of the *InterArrivalTime* on the performance of the protocols when the swap disks are being utilized. The behaviour of the protocols is similar to what has been observed in previous experiments; the performance of each of the protocols increases as the value of *InterArrivalTime* increases. However, when considering both Figure 41 and Figure 42, it may be noted that at low *InterArrivalTime* values, swap disks utilization is relatively high and the performance of all of the protocols is poor. Conversely, a high InterArrivalTime results in the swap disks being utilized less which results in better performance for each of the protocols.

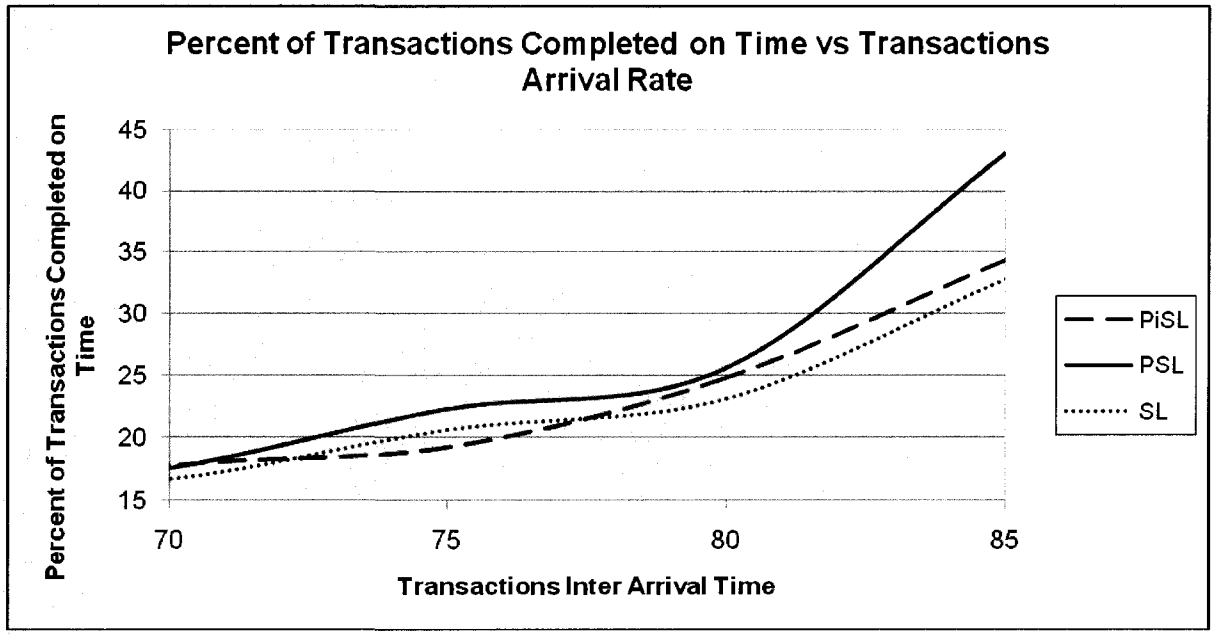

Figure 41: Experiment 5-InterArrivalTime: PTCT – Swap Disk

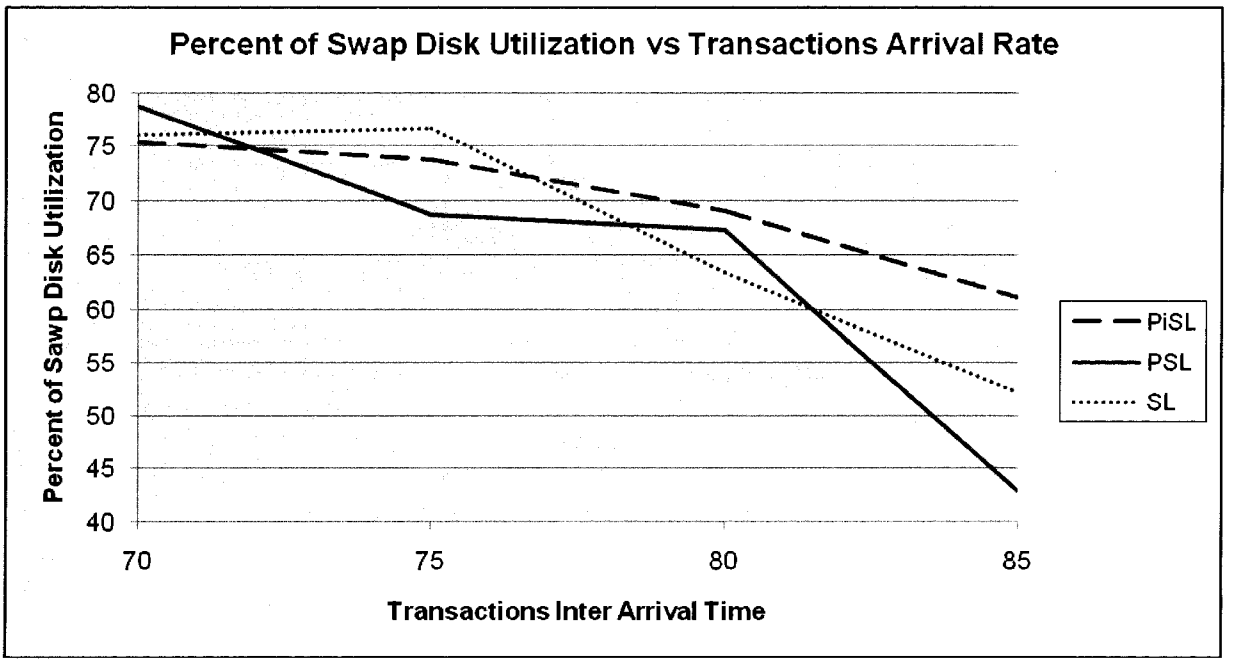

Figure 42: Experiment 5-InterArrivalTime: PSDU – Swap Disk

Chapter 5

Conclusion and Future Direction

With globalization, the need for exchange of information has led to the development of applications that are heavily dependent on globally distributed and constantly changing data. To support these applications, there is a need for distributed real time databases. The Speculative Locking protocol [9] provides an efficient approach in solving concurrency control issues within distributed database systems. SL improves the throughput performance of distributed database systems, but does not take transaction's time constraint into consideration, making it unfit for distributed real time database systems. In this thesis, this limitation was addressed by extending SL into two new concurrency control mechanisms (PSL and PiSL) that take into consideration the distributivity and time constraint issues of distributed real time database systems.

An extensive study has been carried out using the DRTTPS simulator to compare the performance of the proposed protocols. These experiments have demonstrated the following:

1. The Priority-based Speculative locking protocols consistently outperform the SL protocol.

2. System load and data contention levels have a direct impact on the performance of all the protocols.

3. When data contention and system load are too high, PSL outperforms PiSL. For low data contention and low system loads, PiSL outperforms PSL.

4. When the system is forced to use the swap disks, due to a small system cache and high data contention, PSL uses the swap disks less than PiSL and SL. In this case, PSL provides a better performance than PiSL.

5.1. Future Work

PSL provides better performance when there is a high amount of data contention and a high system load. On the other hand, PiSL performs better in systems with lighter loads and lower data contention. Our future study will involve the development of a protocol that will dynamically switch between PSL and PiSL depending on the status of the system. This protocol will take advantage of the strengths of both the PSL and PiSL protocols.

Another interesting study that could be conducted in the future would be implementing PSL and PiSL in a real world distributed real time database system or in a prototype system where real world transactions would be used to study the behaviour of these respective protocols.

References

1. Chen, Y., and L. Gruenwald. "Effects of Deadline Propagation on Nested Transactions in Real-Time Database Systems," *Information Systems,* Special Issue on Real-Time Database Systems, 21 (1), 1996, 103-124.

2. M. Abdouli, B. Sadeg and L. Amanton, "Scheduling Distributed Real-Time Nested Transactions," Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2005, 208-215.

3. Chen, H-R. and Y.H. Chin, "An Efficient Real-time Scheduler for Nested Transaction Models," Ninth International Conference on Parallel and Distributed Systems, 2002, 335-340.

4. Abdouli, M., L. Amanton, B. Sadeg and A. Alimi, "Using Imprecise Concurrency Control and Speculative Lending of Prepared Data-item in Distributed Real-Time Nested Transactions," Ninth IEEE International Symposium on Distributed Simulation and Real-Time Applications, 2005, 298-306.

5. Chen, H.R., Y.H. Chin, "Scheduling value-based nested transactions in distributed real-time database systems," *Real-Time Systems*, 27(3), 2004, 237-269.

6. Lam, K.Y., V.C.S. Lee, S.L. Hung, B.C.M. Kao, "Impact of priority assignment on optimistic concurrency control in distributed real-time databases," Third International Workshop on Real-Time Computing Systems Application, 1996, 128-135.

7. Dogdu E. *Real-Time Databases: Extended Transactions and the utilization of Execution Histories*. PhD thesis, Western Reserve University, 1998.

8. Mittal A., and S. P. Dandamudi, "Dynamic versus Static Locking in Real-Time Parallel Database Systems," 18th International Parallel and Distributed Processing Symposium, 2004, 32-41.

9. Reddy P. K., and M. Kitsuregawa, "Speculative Locking Protocols to Improve Performance for Distributed Database Systems," *IEEE Transactions on Knowledge and Data Engineering*, 16(2), 2004, 154-169.

10. Steinkuehler C. A., "Learning in massively multiplayer online games," Proceedings of the 6th international conference on Learning sciences, 2004, 521-528.

11. Moss, J.E.B., "An Introduction to Nested Transactions". *COINS Technical Report 86-41*. 1986.

12. Ramamritham, K., "Real-time databases," Distributed and Parallel Databases 1, 1993, 199-226.

13. Bharat B., "Concurrency Control in Database Systems," *IEEE Transactions on Knowledge and Data Engineering*, 11(1), 1999, 3-16.

14. Ozsu, T. and P. Valduriez, *Principles of Distributed Database Systems* (second ed.). Prentice Hall, Englewood Cliffs, NJ, 1999.

15. Liu, L., D. Agrawal, and A. El Abbadi," The performance of Two-Phase Commit Protocols in the presence of site failures". *Technical Report TRCS94-09*. Department of Computer Science, University of California, Santa Barbara, 1994.

16. Levy E., H. Korth and A. Silberschatz, "An optimistic commit protocol for distributed transaction management," *Proc. of ACM SZGMOD Conj.*, 1991.

17. Abbott, R., H. Garcia-Molina," Scheduling real-time transactions: A performance evaluation," *ACM Transactions on Database Systems,* 17(3), 1992, 513-560.

18. Nouali, N. et al. "A Two-Phase Commit Protocol for Mobile Wireless Environment". *In Proc. 16th Australasian Database Conference*, 2005, 135-143.

19. Chiu A., B. Kao and K. Lam, "Comparing two-phase locking and optimistic concurrency control protocols in multiprocessor real-time databases," Joint Workshop on Parallel and Distributed Real-Time Systems, 1997, 141-148.

20. Climent A., M. Bertran, F. Babot and J. M. Muixi, "Performance Analysis of Speculative Concurrency Control Algorithms based on Wait Depth Limited for Distributed Database Systems," Second International Symposium on Parallel and Distributed Computing, 2003, 64-71.

21. Kung H. T., J. T. Robinson, "On optimistic methods for concurrency control", *ACM Transactions on Database Systems,* 6(2), 1981, 213-226.

22. Krivokapić N., A. Kemper and E. Gudes, "Deadlock detection in distributed database systems: a new algorithm and a comparative performance analysis," *The International Journal on Very Large Data Bases*, 8(2), 1999, 79-100.

23. Bernstein, P. A. and Nathan Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys (CSUR)*, 13(2), 1981, 185-221.

24. Franaszek P.A., J.R. Haritsa, J.T. Robinson and A. Thomasian, "Distributed Concurrency Control Based on Limited Wait-Depth," *IEEE Transactions on Parallel and Distributed Systems*, 4(11), 1993, 1246-1264.

25. Thomasian, A., "Distributed Optimistic Concurrency Control Methods for High-Performance Transaction Processing," *IEEE Transactions on Knowledge and Data Engineering*, 10(1), 1998, 173-189.

26. Thomasian, A., "Checkpointing for Optimistic Concurrency Control Methods," *IEEE Transactions on Knowledge and Data Engineering*, 7(2), 1995, 332-339.

27. Lin, Jun-Lin and M.H. Dunham, "A low-cost checkpointing technique for distributed databases," *Distrib Parallel Databases*, 10(3), 2001, 241-268.

28. Halici U., A. Dogac, "An Optimistic Locking Technique for Concurrency Control in Distributed Databases," *IEEE Transactions on Software Engineering*, 17(7), 1991, 712-724.

29. Akintola, A A; Aderounmu, G A; Osakwe, A U; Adigun, M O, "Performance Modeling of an Enhanced Optimistic Locking Architecture for Concurrency Control in a Distributed Database System," *Journal of Research and Practice in Information Technology*, 37(4), 2005, 365-380.

30. Hung, S. L. and K. Y. Lam, "Performance comparison of static vs. dynamic two phase locking protocols, "Journal *of Database Administration* 3(2), 1992, 12-23.

31. Thomasian, A., and I. K. Ryu, "Performance analysis of two-phase locking," *IEEE Transactions on Software Engineering,* 17(5), 1991, 386-402.

32. Lam, Kam-Yiu, Sheung-Lun Hung and Sang H. Son, "On Using Real-Time Static Locking Protocols for Distributed Real-Time Databases," *Real-Time Systems*, 13(2), 1997, 141-166.

33. Thomasian, A., "A Performance Comparison of Locking Methods with Limited Wait Depth," *IEEE Transactions on Knowledge and Data Engineering*, 9(3), 1997, 421-434.

34. El Abbadi, A. and S. Toueg, "Availability in partitioned replicated databases," *Proceedings of the 5th ACM Symposium on Principles of Database Systems*, 1986, 240-251.

35. Haque, W., and P. Stokes, "Simulation of a Complex Distributed Real-Time Database System," *Proc of High Performance Computing Symposium (HPC 2007),* Norfolk, VA, 2007.

36. Stokes, P. R. *Design and simulation of an adaptive concurrency control protocol for distributed real-time database systems.* M.Sc. thesis, University of Northern British Columbia, 2007.

37. Law, Averill M. *Simulation Model Analysis* (4th edition). McGraw-Hill, NY, 2007.