Design and Simulation Of An Adaptive Concurrency Control Protocol For Distributed Real-Time Database Systems

Paul R. Stokes

B.Sc., University of Northern British Columbia, 2003

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

March 2007

Your file Votre référence
ISBN: 978-0-494-28449-0
Our file Notre référence
ISBN: 978-0-494-28449-0

Abstract

In a distributed database system, the Speculative Locking protocol (SL) is a concurrency control protocol that allows for parallelism among conflicting transactions through a method of multilevel lending and versioning. Our protocol, the Adaptive Speculative Locking (ASL) protocol augments the SL protocol with four features: real-time support; simultaneous multi-threading; control of transaction execution through transaction queue management; and finite system memory through the use of virtual memory. The ASL protocol outperformed the SL protocols in all experiments in distributed real-time database systems with high data contention. The most significant improvement was the use of transaction queue management for the management of transaction throughput and virtual memory. In addition, we developed a distributed real-time transaction processing database simulator that provides a full featured, modular simulation system capable of simulating a wide range of protocols within an environment that mimicked the real-world as much of possible.

Contents

List of Tables

List of Figures

Acknowledgements

I would first like to thank Dr. Waqar Haque for his guidance, patience, and support. The opportunity and experience that I have had over the past two and half years has been amazing and is truly priceless. Your attention to detail, high quality of standards, dedication, and knowledge has allowed me to excel academically, this I am grateful for. I would also like to thank Dr. Patrick Mann for his many philosophical conversations about academia, theses, and general methodologies that assisted in steering me in the right direction; always a pleasure. A thanks also goes out to Michael Townrow and Brian Ollenberger for their extreme programming skills in the development of DRTTPS and to my friends for their encouragement. Last and certainly not least, I would like to thank my wife Jennifer and my kids for their support and understanding in the pursuit of my Masters.

The road to my masters' degree has been one with incredible outcomes. Over the past two and a half years, I have tapped potential that I would have not obtained otherwise, including that of critical thinking, research, analysis, and writing skills. This has been a life changing experience and a door to my future.

Chapter 1

Introduction

Database Systems are an integral part of our daily lives and are the core to any global service or business that exists worldwide. A real-time database is a processing system designed to handle workloads whose state is constantly changing [1] and is constrained by time. Many of these systems are distributed in nature for the purposes of data locality and disaster recovery. The application of distributed real-time database systems can be seen throughout many industries including financial services, education, gaming, and aviation. For example, in the stock market, stocks are bought and sold from around the world and the transactions are being recorded in databases that are distributed around the globe. At the same time, those changes are being presented to the world in real-time. As our global economy moves forward, the need to have information at our fingertips that is presented in real-time to make critical decisions will be essential.

A Database System is an organized collection of information that is accessed by a computer program in order to answer queries. Access to the database system is controlled through the concept of transactions. These transactions are logical sequences of read and write operations that execute concurrently in the database system. Transactions executing concurrently must conform to the ACID properties for each transaction. The ACID properties guarantee that: all or none of the tasks of a transaction are performed;

transaction must be in a consistent state; operations must appear isolated from one another; and that a transaction that the result of a transaction is persisted. A Real-time Database system (RTDBS) is a database in which transactions must not only maintain the ACID properties for each transaction, but must also satisfy the timing constraints of each transaction within the system. An extension of RTDBS is the distributed real-time database system (DRTDBS), which is a collection of data sites, containing one or more real-time databases that are interconnected by a communication networks.

The primary goal of a DRTDBS is to maximize transaction throughput while maintaining the serialization of transaction execution. The activity of coordinating concurrent access of these transactions to data is called concurrency control. There are many well-known concurrency control protocols for distributed databases that enforce serialization and permit concurrent access to data. However, very little work has been done with regard to distributed real-time database systems in which transactions are expected to meet deadlines. Our goal is to develop an adaptive real-time heuristics based concurrency control protocol that analyzes performance and resource metrics to improve the transaction throughput in a real-time distributed database system, while minimizing transaction aborts.

1.1. Transactions

A transaction is an atomic unit of data processing in a database system. The purpose of a transaction is to request or modify data in a database. Transactions that require access to local data only, are called local transactions. Transactions that have a number of sub-transactions that require access to data at multiple sites are called global transactions. In

the DRTDBS, transaction execution involves executing sub-transactions at multiple sites to complete a task. In this model, there is a process called the master that is executed at the site at which the transaction originated. The master then has a set of other processes called cohorts that execute on behalf of the transaction at various distributed sites. The cohorts are created by sending a STARTWORK message (which includes the work to be done at each site) from the master to the local transaction manager at each site. When the cohort has completed the work, it sends a WORKDONE message to the master. Once the master has received all of the WORKDONE messages from all cohorts, it executes a commit protocol. A successful execution of a transaction results from the transaction being committed.

From the above description of a transaction, we can clearly see that there are two phases in the lifetime of a transaction: a work phase and a commit phase. The work phase performs the tasks of reading and manipulating data, whereas the commit phase completes the transaction resulting in a commit (making the changes permanent) or in an abort (discarding any changes). Transactions that are executing simultaneously on the same data require a method for guaranteeing the order of execution to be correct. Serialization allows the transactions to be executed concurrently, while maintaining the global order of execution, resulting in each of the transactions being executed serially [2]. In order to protect data and ensure serialization in the work phase, we use a "locking protocol." Within the commit phase, we use a "commit protocol" to protect data and ensure serialization. The locking and commit protocols are two fundamental components required to ensure the serialization of a transaction. These protocols are further discussed in Sections 1.6 and 1.7.

1.2. Partitioned vs. Replicated

An important aspect of distributed databases is the placement of data throughout the distributed system. The locality and replication of data with respect to the execution of transactions can severely affect the performance and integrity of the entire system. Two methods can be used to distribute data between sites in a distributed database system: partitioned and replicated (Figure 1).

1. Partitioned data is the set of data within a database that is distributed to each node, in which there is no intersection of data between the set of nodes. In a database system, this is represented by the distribution of tables or columns of those tables throughout the member nodes for which there is no duplication of data amongst the nodes.
2. Replicated data is the set of data within a database that is distributed to each node, in which there exists an intersection of data between the set of nodes. In a database system, this is represented by the distribution of tables or columns of those tables throughout the member nodes for which there exists duplication of data amongst the nodes.

In a hybrid approach, data can be both partitioned and replicated amongst the set of member nodes.

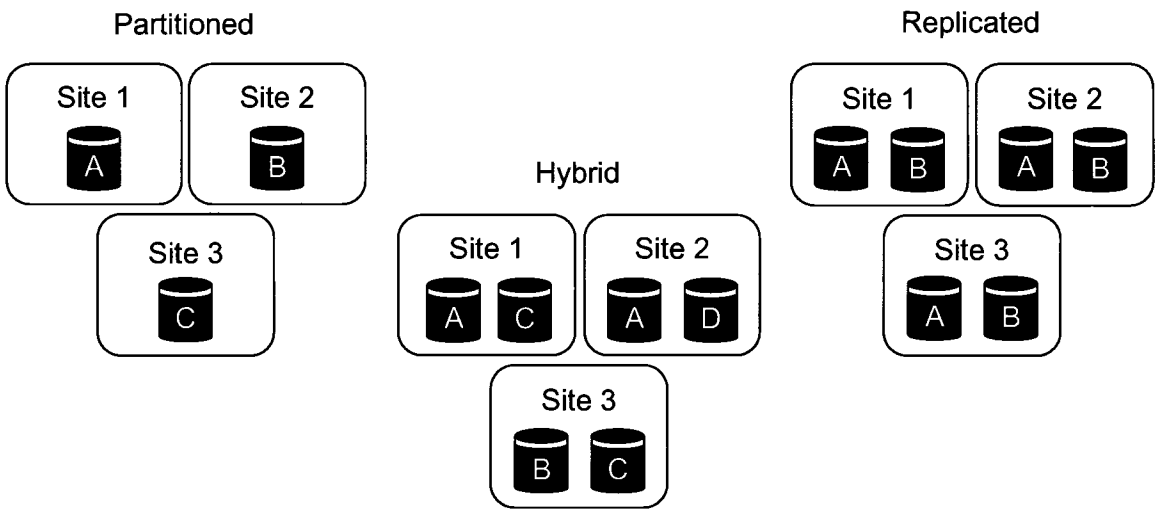

Figure 1: Partitioned vs. Replicated

1.3. Deadlines & Priorities

In a real-time distributed database system, transactions have associated timing constraints in the form of completion deadlines. In order to successfully commit a transaction, the system has to meet these deadlines and comply with the consistency requirements of the transaction. The deadline constraints for global transactions are much greater than that of local transactions, this is due to the time required to process all of the sub-transactions associated with a global transaction. The goal of a DRTDBS is to maximize the number of transactions that commit before their deadlines expire. There are three categories for grouping real-time applications: hard deadline, firm deadline, and soft deadline. For each category, a transaction is assigned a completion deadline and a value that decreases after the deadline has passed which is weighted by the application and is expressed as a function of the completion time. The following are the value functions for the three categories of transactions deadlines (Figure 2):

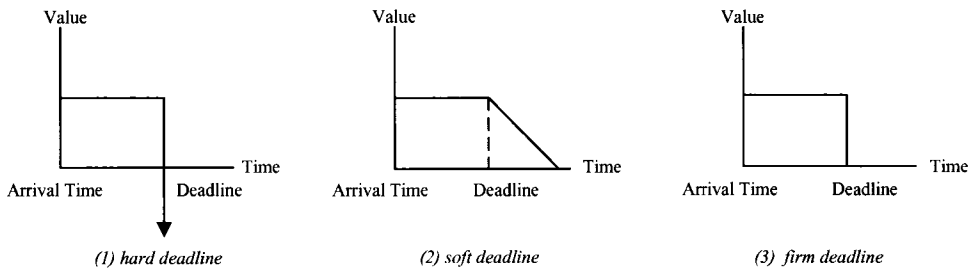

Figure 2: Transaction Deadline Model

1. Hard Deadline: A transaction has been assigned a strict deadline for completion of a task. If the deadline is missed, the transaction is deemed a fatal error or will result in disastrous consequences. The value of the transaction is deemed as negative and has a significant impact on the system.
2. Soft Deadline: A transaction is assigned a deadline and is allowed to miss its deadline and continue processing until the transaction commits. The value of the transaction up to the deadline remains at 100 percent and degrades until the transaction has completed. If the transaction exceeds its tardy time, the amount of time allowed past the deadline, then the value will drop to zero.
3. Firm Deadline: A transaction has been given a strict deadline for completion of a task. If the deadline is missed, the transaction is deemed worthless and is discarded. The value of the transaction at the deadline is zero.

In order to specify the importance of a transaction, a priority is assigned to each transaction. Priority assignment protocols determine which transactions will execute first

and which transaction will be blocked or restarted in case of data conflicts. A priority inversion occurs when a higher priority transaction is being blocked by a lower priority transaction [3]. A priority inversion can cause the high priority transactions to miss their deadlines, which is undesirable. Therefore, transactions need to be scheduled to avoid this situation.

1.3.1. Preemption

Priority inversions can be alleviated by preempting the execution of the blocking transactions using one of following two methods: priority inheritance and priority ceiling protocols. Priority inheritance states that if a lower priority transaction T_L is blocking a higher priority transaction T_H, then T_L will temporarily inherit the priority of T_H. As soon as T_L completes, it will return to its original priority and release all of its locks that was causing the priority inversion [4]. A priority ceiling protocol is an extension of the priority inheritance protocol that sets the priority of a data item to the highest priority of the transactions accessing the item. In order to access the data item, the requesting transaction needs to have a priority greater than that of the highest priority ceiling on that data item. Priority ceiling protocols cannot be used in RTDBS as they do not guarantee serialization of real-time transactions [4].

1.4. Deadlocks

A deadlock occurs when one transaction is suspended and is waiting for a second transaction, and the second transaction is waiting (either directly or indirectly) for the first transaction, which results in a circular wait condition [5]. There are three ways of dealing

with deadlocks: deadlock prevention, deadlock avoidance, and deadlock detection. Deadlock prevention algorithms ensure a deadlock free condition by guaranteeing that at least one of the deadlock conditions (mutual exclusion, hold and wait, no preemption, and circular wait) fail to hold [6]. The deadlock prevention algorithms are relatively easy to implement but they have the cost of high transaction restarts. Deadlock avoidance algorithms (ex. Bankers algorithm) predict deadlocks by dynamically analyzing every request, which requires additional information about the potential use of each resource associated with each process. The information required to make a decision is not always available, which may result in an inefficient transaction execution. These algorithms have lower system overheads and are therefore favoured over prevention algorithms. Deadlock detection allows deadlocks to occur; the deadlock situation is then handled and the system recovers from the deadlock. Whenever an event such as a transaction being queued for access to a locked object occurs, a deadlock detection routine is invoked to detect the deadlock cycle. If a cycle is detected, one of the transactions involved in the cycle is selected as the victim and aborted to resolve the deadlock.

Deadlock detection in distributed databases is complicated in that deadlocks cannot be detected by a single site; inter-site communication is required to detect deadlocks. Two approaches have been adopted for distributed deadlock detection: centralized and distributed. In the centralized approach, a global wait-for-graph is constructed from all local wait-for-graphs within the sites and is stored on a central detector. The centralized approach is highly inefficient due to the amount of bidirectional traffic to the central detector. For the distributed approach, deadlock detection is equally shared between all

sites and is only initiated if the system suspects a deadlock. After a deadlock has occurred, global information is required to determine a resolution. In distributed real-time database systems, transactions are likely to miss their deadlines unless a detection algorithm is invoked immediately, thus allowing the transactions time to restart and complete.

1.5. Locking Protocol

Before a transaction or sub-transaction can access a shared data item, they must request a lock on that data item. When a transaction tries to access the data item, the scheduler determines the state of the associated lock. If the item is not currently locked, then the scheduler obtains the lock on behalf of the transaction. If another transaction already holds an exclusive lock on that data item, then the requesting transaction may have to wait until the current lock has been released. This ensures that a data item can only be accessed by one transaction at a time. Locking is required in order to guarantee serialization of transactions within the system.

In a database system, locks can be divided into two groups, shared or exclusive locks. If a transaction wants to read a data item, it must first request a shared lock on that data item. If a transaction wants to modify a data item, it must first request an exclusive lock on that data item. A data item can be held by multiple shared locks, but can only be held by one exclusive lock. This method allows transactions simultaneous access to the same data item, allowing for concurrent execution of transactions. Several distributed real-time locking protocols exist in literature [7] [8] [9] [10] that try to improve performance of the system by allowing concurrent execution of transactions.

One of the more common locking protocols used today is the two-phase locking or the 2PL protocol. In 2PL, the execution of a transaction can be divided into two phases: growing phase and shrinking phase (Figure 3). During the growing phase, the transaction acquires all of the required locks, but cannot release any locks. During the shrinking phase, the transaction releases its locks but cannot acquire any new locks. In the distributed 2PL algorithm, a transaction that requires read access to a data item sets a read lock on only one copy of the data item. However, in order to update an item, a write lock is required on all copies. As the transaction executes, write locks are obtained and any requesting writes to the data item are blocked until the cohort and its remote updaters have locked all copies of the data item. During the data processing phase, the data that is locked by the cohort is the only data that is updated. Remote copies locked by updaters are updated in the PREPARE phase of the locking protocol. Further, read locks are held until the transaction has entered the PREPARE state, while write locks are held until the transaction has either been committed or aborted.

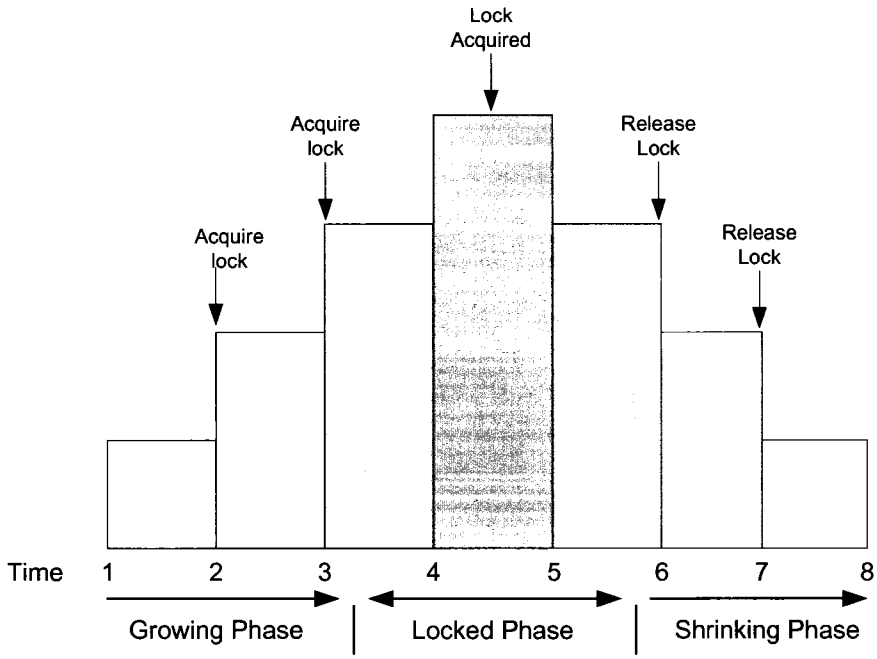

Figure 3: Two-Phase Locking - Growing and Shrinking Phases

To avoid problems such as priority inversions and deadlocks, transactions need to release their locks as soon as possible. A typical approach is to modify the standard two-phase locking (2PL) [11] [2] protocol so that other transactions can access the locked data prior to the locks being released at the end of the commit phase. A second approach is to decrease the steps required by short-circuiting the commit phase as demonstrated in two variants of two-phase commit (2PC) [11]: Presumed commit (PC) and presumed abort (PA) [12] [13]. Depending on whether the master is using PC or PA, the master will either commit or abort the transaction if the master has not heard a response after a specific amount of time has passed. In creating a variation of the 2PC protocol, it is important that the changes still guarantee serialization.

1.6. Commit Protocols

A commit protocol ensures that the work done by local and global transactions is completed and any modifications to the database are successful. It is guaranteed that operations performed by a transaction either all occur or that all operations do not occur, making the transaction atomic. Commit protocols increase the execution time of a transaction due to the message passing and logging that is required to commit a transaction. In addition, commit protocols prevent locks from being released until after the transaction has been committed. This delay could cause a higher priority transaction to wait for a lower priority transaction to complete, resulting in a priority inversion.

In distributed database systems, there are four traditional commit protocols: two-phase commit (2PC) [2], presumed commit (PC) [12], presumed abort (PA) [12], and three-phase commit (3PC) [14]. Each of these traditional protocols have at least two-phases, the prepare phase and the commit phase (Figure 4).

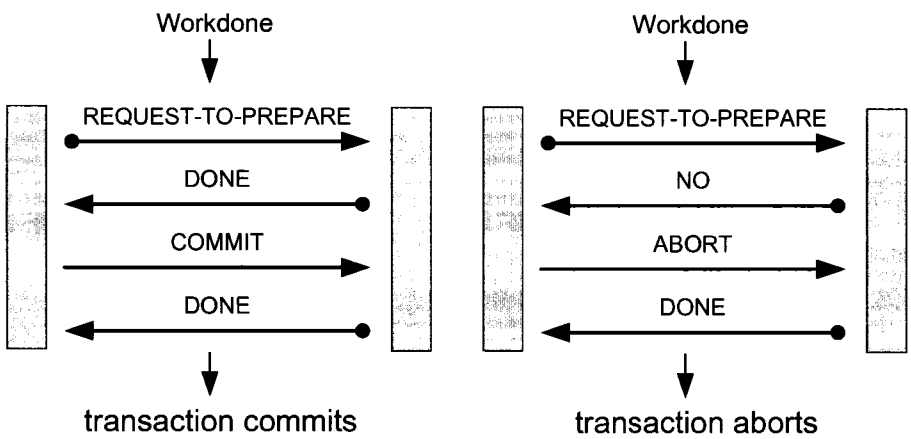

Figure 4: Commit Protocol - Commit and Abort Paths

When a cohort within a site receives information from the master about the operations that it must perform, it creates a new transaction and begins the data processing phase. Upon completion of this phase, the cohort sends a WORKDONE message to the master. Once the master has received all of the WORKDONE messages from all of the cohorts involved in the global transaction, it sends a PREPARE-TO-COMMIT message back to all of the cohorts. For each cohort that is ready to commit, a prepare log is written and a YES message is sent back to the master. The cohort is now in the prepared stage and is waiting for the master to send the next message. Once the master has received a YES (to commit) or NO (to abort) message from all of the cohorts, a global decision is made and the master will unilaterally commit or abort the transaction. If there is a unanimous YES decision, the master writes to the commit log and sends a COMMIT message to all of the cohorts. Each cohort that receives the message writes a commit log, moves to a commit state, and sends an ACK message back to the master. Conversely, if there is an abort message from at least one of the cohorts, the master writes an abort log and sends an ABORT message to all of the cohorts. After the cohorts have received the ABORT message, they write to an abort log, move to an abort state, and send an ACK message back to the master. At this point, the transaction has either committed or aborted and the locks held on the modified data can be released. Several variations of these traditional protocols exist in literature [14] [15] [16] [17] and are discussed further in Chapter 2.

1.7. Concurrency Control Protocols

Concurrency Control is a method used to ensure that transactions can coexist and share data without interfering with one another. A concurrency control protocol (CCP) is a

complex protocol that implements concurrency control among transactions and guarantees that transactions will eventually commit or abort, resulting in atomic transactions. The CCP orchestrates the phases of execution for a transaction and therefore controls the behaviour of the locking and commit protocols. The CCPs used for this research are discussed in Chapter 2 – Literature Review.

1.8. Contribution

One proposed method for addressing the issues around concurrency control in distributed real-time database systems is to use a variation of the speculative protocols [10]. These protocols utilize *before* and *after* images of data objects to allow requesting transactions access to uncommitted (locked) data (through the concept of lending) prior to a cohort committing. Using the before and after images of the cohort, the waiting transaction can now perform a speculative execution using those images, resulting in the creation of its own before and after images. The transaction will then wait for the cohort to terminate and will retain one of its images based on the termination path of the cohort transaction. The number of speculative executions allowed by speculative protocols is restricted due to the additional system resources that are required, the insufficient use of CPU parallelism, and the time constraints imposed by transaction deadlines in distributed real-time database systems. These restrictions reduce the maximum number of possible speculative executions, which in turn causes transactions to wait, resulting in missed deadlines.

In this thesis, we investigate existing speculative based concurrency control protocols and propose an adaptive speculative locking protocol, which will still utilize the fundamental

speculative concept of data lending through before and after images, but will provide the following enhancements:

- Using system performance metrics, the protocol will dynamically determine the level of lending that will occur based on: the system memory utilization, the number of transactions in the queue, and the deadlines of those transactions.
- Remove the SL protocol simulation parameter of unlimited system memory, and impose a constraint of a maximum amount of system memory with the inclusion of virtual swap space.
- Using CPU hyper-threading, provide the ability to simultaneously execute a transactions before and after images through parallel thread execution on a single processor.

In addition to the adaptive speculative protocol contributions, we have developed a flexible and extensible distributed real-time database simulation environment for the purpose of implementation and analysis of the baseline and adaptive protocols.

1.9. Organization

The remainder of this thesis is organized as follows. In Chapter 2, we investigate several approaches to commit and locking protocols proposed in literature that relate to speculative execution. Chapter 3 discusses simulation models and outlines the system model in which the simulation environment was developed for the analysis of the adaptive protocol. Chapter 4 presents the adaptive speculative protocol that improves upon the limitations of the SL protocol. Chapter 5 focuses on the simulation parameters and the

analysis of the simulation results. Finally, in Chapter 6, we summarize the thesis contributions and outline future directions.

Chapter 2

Literature Review

Concurrency control protocols are a core component to database systems and are comprised of an algorithm that guarantees the serialization of transaction processing through the use of locking and commit protocols. This chapter reviews several commit and locking protocols in literature and shows an inheritance of techniques between them. The final protocol presented is our adaptive concurrency control protocol, which is primarily based on the Speculative Locking (SL) protocol [10].

2.1. PROMPT

Permits Reading of Modified Prepared-data for Timeliness (PROMPT) [18] is a commit protocol designed for distributed real-time database systems (DRTDBS) that imposes firm deadlines on transactions. PROMPT was written to enhance the traditional 2PC commit protocol by reducing the amount of blocking that occurs during the commit of a transaction. The PROMPT protocol allows transactions that are requesting access to locked data, the ability to optimistically borrow uncommitted data from a lending transaction. The borrowing transaction can only access the data of the lending transaction, if the lending transaction is in the prepared phase of the commit protocol. Assuming that the lender has allowed the borrower access to the uncommitted data, there are three scenarios that exist around the behaviour of the lender and borrower:

1. Lender receives decision before borrower completes data processing: If the global decision is to commit, then the lender commits normally. However, if the global decision is to abort, the lender and the borrower are aborted.
2. Borrower completes data processing before lender receives decision: The borrower is "put on the shelf" and is not allowed to send a WORKDONE message until the lender has received its decision and has finished processing. If the global decision is to commit, then the lender commits and the borrower is taken off the shelf and is allowed to send its WORKDONE message. However, if the global decision is to abort, the lender and the borrower are both aborted.
3. Borrower aborts during data processing before lender receives decision: The borrower's updates are undone and the lending is nullified.

The performance of PROMPT is further enhanced with several additional real-time features: active abort, silent kill, and healthy lending.

1. In the event of a priority conflict between two transactions, active abort allows cohorts that are not in the commit phase to immediately send an abort message to the master instead of having to wait for the commit phase.
2. Instead of the master invoking an abort protocol when a deadline has been missed, silent Kill allows the cohorts to independently realize the missed deadline and abort without any communication with the master.
3. If a transaction is in the process of committing, healthy lending will disallow the lending of data to that transaction if there is a risk that the borrower will be aborted due to a missed deadline.

With the lending and borrowing of data in PROMPT, if a transaction aborts, the abort does not arbitrarily cascade as transactions are typically expected to commit and more importantly, a borrower cannot simultaneously be a lender. If a lender is in the prepared state, it cannot be aborted unless the transaction deadline at the master has expired prior to a decision. If the lender is in the prepared state, the borrower cannot complete its processing until the lender has committed or aborted. Further, the borrower cannot lend data until it is in the prepared state, which implies that the borrower has committed.

PROMPT shows significant performance improvements over traditional 2PC protocols and reduces priority inversions through the use of active aborts and by optimistically lending uncommitted data to requesting transactions. However, the additional real-time features of silent kill and presumed commit/abort offer little performance improvement and in some cases degrade performance. Further, PROMPT allows for only one level of data lending which still results in priority inversions and transaction aborts. The concepts outlined in PROMPT provide a future direction for extending the protocol to support soft-deadlines and to increase the number of levels of lending, resulting in the reduction of the number of priority inversions and transaction aborts.

2.2. PEP

PROMPT Early Prepare (PEP) [15], is an extension of the PROMPT protocol that reduces the overall master/cohort communication by eliminating a phase in the standard 2PC protocol, resulting in a one-phase commit protocol (1PC). The reduction in communication is achieved by combining the WORKDONE message in data processing with the PREPARE

message of commit processing into one message by using the early prepare (EP) 1PC protocol. Priority inversions are an inherent problem in 1PC protocols and are addressed by incorporating the data borrowing technique from PROMPT into the Early Prepare Framework, resulting in the PEP protocol. PEP also uses Presumed Commit (PC), which in the event of a communications failure, reduces the transactions wait/recovery time by allowing a transaction to commit without contacting the master.

PEP shows performance improvements over the PROMPT protocol by further reducing priority inversions through the use of active aborts. In addition, PEP reduces the message and logging overheads through the use of the 1PC protocol, and incorporates the optimistic lending of uncommitted data as demonstrated in PROMPT. However, as in PROMPT, PEP still suffers from transaction aborts and priority inversions due to missed deadlines.

2.3. Mirror

MIRROR (Managing Isolation in Replicated Real-time Object Repositories) [17] is a concurrency control protocol that extends the optimistic two-phase locking (O2PL) [19] algorithm and is designed for real-time applications with firm-deadlines on replicated real-time databases. The extension of the O2PL protocol is accomplished with the addition of a dynamic state-based conflict resolution method called state-conscious priority blocking.

O2PL handles replicated data optimistically and is identical to 2PL in the absence of replication. When a cohort is updating a replicated data item, the cohort immediately requests a write lock on the local data item. For replicated remote copies of the data item, the write locks are deferred until the beginning of the commit phase. Cohorts initiate replica updaters in the commit phase by passing update information in the PREPARE

message of the commit protocol. Remote updaters must obtain write locks on the remote items before it can act on the PREPARE request. Since write locks are obtained on copies at the end of the transaction, locking occurs rather late in the execution of the transaction.

MIRROR augments the O2PL protocol with two real-time conflict resolution mechanisms, Priority Blocking (PB) and Priority Abort (PA). PB is comparable to conventional locking protocols where a transaction requesting data that is locked is blocked until the current lock has been released. PA attempts to resolve the data conflicts in favour of high-priority transactions. If a requesting transaction has a lower priority than the transaction holding the lock, the requesting transaction is blocked. However, if the requesting transaction has a higher-priority, then lower-priority transaction is aborted and the lock is granted to the higher priority transaction. In order to reduce the cost of aborting a transaction that is near completion, execution is divided in two stages: PA is used in the early stage and PB in the later stage. Determining the point at which to use PB as opposed to PA is defined by the demarcation point. The demarcation point is the point at which the cohort receives a PREPARE message from the master, implying that the cohort has acquired all of its locks and is executing the commit protocol. The demarcation point rules state that if the lock holder has not passed the demarcation point then PA is used, otherwise PB is used.

MIRROR shows performance improvements for the O2PL protocol over the standard 2PL protocol in replicated real-time databases. The interesting characteristic of this protocol is the augmentation of O2PL with a state-based conflict resolution mechanism. The strategy

here is similar to that of PROMPT and PEP for efficiently handling transaction execution and conflict resolution, but does not target distributed real-time database systems.

2.4. AEP

AEP (Adaptive Exclusive Primary) [20] is a distributed file system CCP that dynamically switches between an optimistic and a pessimistic CCP through use of partial system knowledge at each local site. AEP is based on the exclusive write with locking option (EWL) protocol [21] as the optimistic CCP and the primary site locking (PSL) protocol [21] as the pessimistic CCP.

The PSL protocol uses a primary site (PS) to control access to a distributed file system. When a transaction initiates an update on a file, it prepares for a conflict and sends a request for a lock to the PS. If the page is currently not locked, then the PS sets the lock and notifies the requesting site. The transaction then performs an update on the file and sends the results to all sites. However, if the file is locked, the transaction is placed on a queue until a lock can be obtained. The initial conflict check by the transaction for a file lock is the pessimistic nature of this protocol.

The EWL protocol also uses a PS called the exclusive writer and primary site (EW/PS) to control access to a distributed file system. A transaction that initiates an update on a file optimistically assumes that there is no conflict and sends the updated file to the EW/PS. If the file is not locked, then the EW/PS grants the lock and broadcasts the update to all sites. Otherwise, the transaction is placed on a queue and all prior updates that were sent to the EW/PS are discarded.

The AEP protocol dynamically switches between the PSL and EWL protocols, and utilizes the EW/PS to control access to the distributed file system. AEP also keeps track of local active transactions that have initiated locally and have not yet committed. Through a local registry, AEP checks to see if there exists a local transaction accessing the same file. If a transaction is found, then there is a potential conflict. The transaction then sends a lock request to the associated EW/PS and follows the rules of PSL until completion. Conversely, if there is no entry in the registry then the probability of a conflict is lower, so the transaction sends the update to the EW/PS and follows the rules of EWL until completion.

AEP shows performance improvements over the EWL and PSL protocols. The EWL protocol suffers from resource and time penalties caused by transaction restarts. As data contention and transaction computation complexities increase, the probability of transaction restarts also increase due to access conflicts. The PSL protocol suffers from the amount of inter-database site communication required to lock/unlock data and send updates. The advantage of the AEP protocol is the initial check that allows it to adaptively execute the appropriate protocol to avoid conflicts that may result in transaction restarts. AEP does not target distributed real-time database systems, but does provide an adaptive approach to distributed transaction processing.

2.5. Speculative Locking Protocol

The speculative locking (SL) protocol [10] extends the standard 2PL protocol to allow for parallelism among conflicting transactions. In 2PL, a transaction holds an exclusive lock on a data object until after the completion of the commit protocol and then the lock is released. As demonstrated in PROMPT, parallelism can be achieved by allowing transactions that are

requesting access to locked data, the ability to optimistically borrow uncommitted data from a transaction that is in the prepared phase of a commit protocol. PEP used the concept of data lending from PROMPT and further reduced the overall master/cohort communication by eliminating a phase in the standard 2PC protocol. Both PROMPT and PEP showed performance improvements over 2PC, but they are restricted to one level of lending which reduces the effectiveness of parallelism among conflicting transactions. MIRROR used a real-time technique for determining how a transaction in the commit phase would behave in a conflict situation, resulting in the reduction of transaction aborts. The concepts presented in these protocols provide a basis for the features and direction of the SL protocol.

The primary contribution of the SL protocol is the extension of the lending model that is demonstrated in PROMPT, with the addition of allowing for N levels of lending, resulting in N levels of speculative executions. As in PROMPT, the SL protocol allows waiting transactions (borrowers) to access the after-images produced by parent transactions (lenders) during the prepare-phase of the commit protocol. The waiting transaction then performs a speculative execution on the before and after images of the parent, and then waits for the parent transaction to complete. As soon as the parent transaction completes, the waiting transaction decides its outcome based on the parent transaction's outcome. SL's improvement over PROMPT is a trade off between system resources and the improved transaction throughput achieved by performing speculative executions on all outcomes for a transaction at a point in time. To demonstrate the advantages of the SL protocol, Figure

5, Figure 6, and Figure 7 [10] show the completion time of a transaction for normal processing, 2PL processing, and SL processing.

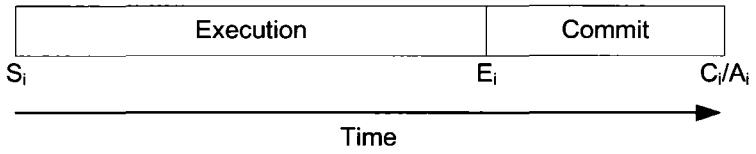

Figure 5: Normal Transaction Processing of Ti

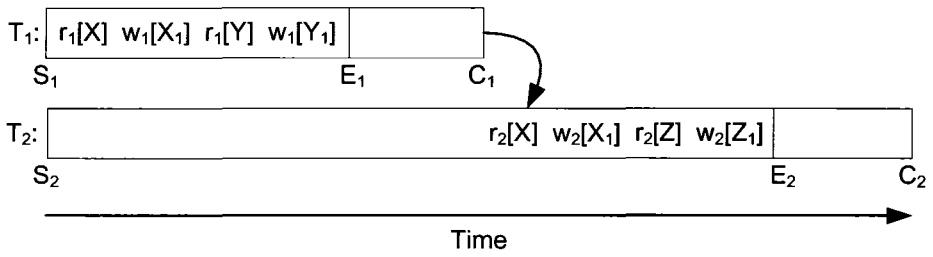

Figure 6: 2PL Processing

In Figure 5, the processing of a transaction starts executing at S_i, ends at E_i, commits or aborts at C_i/A_i, and performs the execution within a linear amount of time. Figure 6 shows the processing of the pages X and Y for transaction T_1 and X and Z for transaction T_2 using the 2PL protocol. If transaction T_1 is reading and writing pages X and Y, Transaction T_2 cannot access the after-image of X until T_1 has completed its commit processing. Figure 7 shows the processing of transactions T_1 and T_2 using the SL protocol.

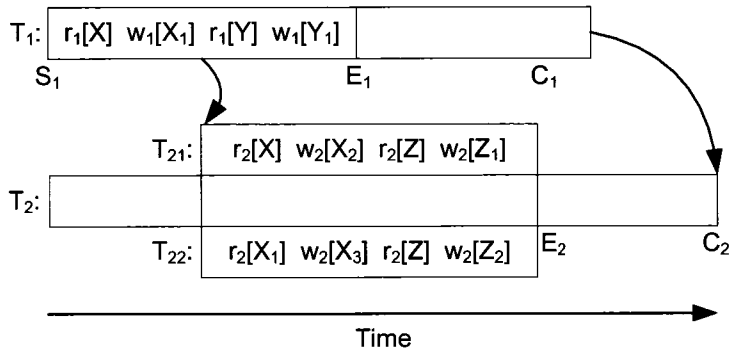

Figure 7: SL Processing

The processing begins with transaction T_1 acquiring locks on pages X and Y. When transaction T_1 completes processing, it releases the locks and produces corresponding after-images X_1 and Y_1. A waiting transaction T_2 can obtain a lock by accessing both the before-image X and after-image X_1. T_2 then carries out speculative executions T_{21} and T_{22} to produce the after-images X_2 and X_3. When transaction T_1 completes processing, T_2 proceeds into the commit phase and retains T_{22} if T_1 commits and T_{21} if T_1 aborts.

As the level of data lending to subsequent transactions increases, so do the speculative executions, which results in an increase of parallel transaction processing. In a naive situation SL(n), if there are n transactions executing, then there are 2^n possible speculative executions required to produce the before and after images, and potentially 2^n termination possibilities if n transactions conflict. Therefore, the SL variants SL(0), SL(1), SL(2) are used to restrict the maximum number of aborted transactions and are defined as follows:

- SL(0) only allows for one preceding transaction to abort and only performs one execution by reading the after-image of the prior transaction. SL(0) has the greatest similarities to that of the PROMPT protocol.

- SL(1) allows for two preceding transaction to abort and the number of speculative executions increases linearly with the number of transactions [10].
- SL(2) allows for two preceding transaction to abort and is sufficient to support ∑n+1 speculative executions for a transaction [10].

To better understand the SL variants, Figure 8 shows the tree growth for speculative executions under SL(n), SL(0), SL(1), and SL(2) [10].

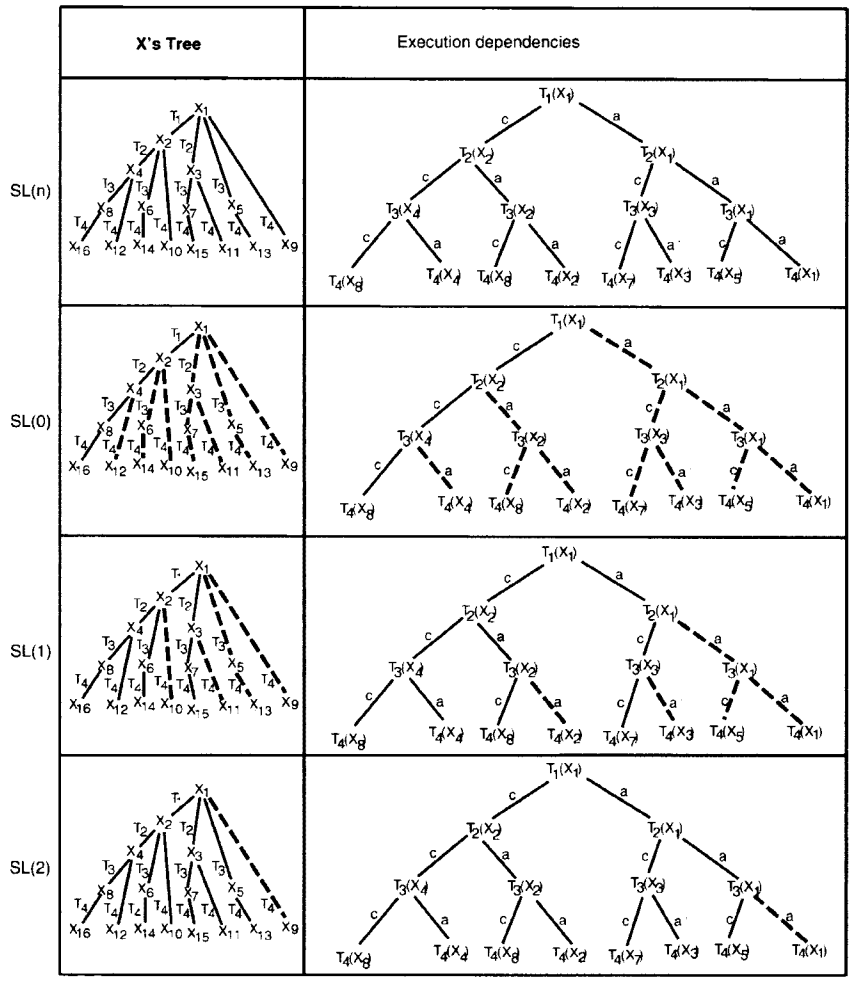

Figure 8: Tree Growth for Speculative Executions

A typical 2PL protocol only marks pages with the state of read or write. For the speculative locking protocol, the read (R) and write (W) states were not sufficient. Therefore, two new locking states, execution-write (EW) and speculative-write (SPW) are created to replace the standard write lock used by 2PL. In the SL protocol, a transaction can only request a read lock (R) and an execution-write lock (EW-lock). When a speculative execution takes place, the EW-lock is used to process the page, and once the after-images are created, the EW-lock is converted to a speculative-write lock (SPW-lock) allowing other transactions to get access to the locked data. Table 1 [10] shows the lock compatibility matrix of 2PL and SL.

Lock Requested By Ti	Lock Held by Tj	
	R	W
R	yes	no
W	no	no

(a)

Lock Requested By Ti	Lock held by Tj		
	R	EW	SPW
R	yes	no	sp_yes
W	sp_yes	no	sp_yes

(b)

Table 1: Lock Compatibility Matrix (a) 2PL and (b) SL

The speculative locking protocol shows performance improvements over the standard 2PL protocol. The use of speculative execution for parallel transaction processing provides a compelling solution to page locking within a distributed database system. However, the SL protocol suffers from its use of infinite, rather than finite, system memory for speculative executions. Further, the amount of processing resources required to perform a speculative execution could lead to a potential degradation in system performance. SL also does not

target distributed real-time database systems, but does provide an excellent approach to locking protocols within a distributed database system.

Chapter 3

System Model and Simulator Design

An important aspect of analyzing a heuristics based protocol is to perform analysis through simulation. This chapter presents a distributed real-time database simulation model and provides a detailed description of the Distributed Real-Time Transaction Processing System (DRTTPS) that was developed for the analysis of the baseline and adaptive protocols. In addition, an in-depth look at the Speculative Locking and Adaptive Speculative Locking protocols is presented.

3.1. Simulation Model

A model is a representation of reality used to understand a situation, predict an outcome, simulate a process, or analyze a problem. A simulator is a system that is based on a model and is used to verify that the model behaves and operates as expected when provided with a set of controlled inputs. Simulators are divided into two categories: discrete and continuous. Discrete simulation is composed of a number of logical expressions that are evaluated at discrete points in time. In continuous simulation, time is controlled by continuous variables expressed as differential equations. Both discrete and continuous simulators utilize mathematical probability distributions to control the level of randomness of components, which in turn provides a more realistic simulation. The process of building simulation models typically involves writing a software application. The model is either the software itself or held within a host system. The software simulator consists of entities,

entity relationships, methods, and logic statements which are used to perform the required actions. The entities are the tangible elements found in the real world, whereas the methods, logic statements and entity relationships define the overall behaviour of the model.

In a distributed real-time database simulator, databases are physically distributed across multiple locations, called sites. These sites can contain one or more server nodes that in turn have one or more databases. The nodes and sites are interconnected by local or wide-area networks, such as a corporate network or the internet. Each node is represented by a hardware system that contains memory, cache, disks, and processors. The databases on each of these nodes are comprised of a set of pages that are located on one or more disks. In a distributed real-time database model, each local database operates in a similar fashion to a centralized database system and consists of four modules (Figure 9) [24]:

1. Transaction Manager: Performs pre-processing on incoming transactions and forwards the transaction operations to the scheduler.
2. Scheduler: Controls the order in which transaction operations will be executed.
3. Recovery Manager: Responsible for aborting or committing a transaction.
4. Cache Manager: Manages the system cache where database read and writes are performed.

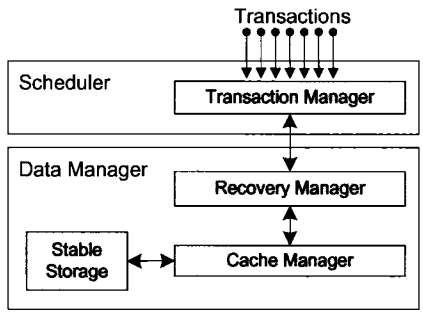

Figure 9: Four internal modules of a database system

3.1.1. Transaction Manager

Transactions interact with the database through the transaction manager as shown in Figure 10. The primary purpose of the transaction manager is to pass transaction operations on to the scheduler. In addition, in a distributed database environment, the transaction manager is responsible for determining which site the transaction is destined for and forwarding that transaction on to the appropriate site's transaction manager.

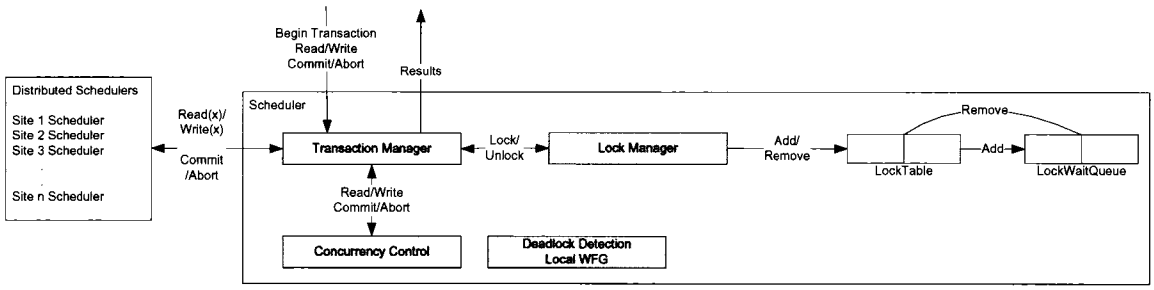

Figure 10: Transaction Manager within the Scheduler

3.1.2. Scheduler

The scheduler (Figure 11) is the component that controls the concurrent execution of transactions through the use of a concurrency control protocol. The scheduler uses the data manager to control the order in which operations of a transaction execute. The scheduler guarantees serialization of transaction operations and avoids cascading aborts by

ordering the execution of transaction operations. In executing a database operation, the transaction manager sends the operation to the scheduler. The scheduler does one of three things: executes, rejects, or delays the operation. If the scheduler passes an execute operation to the data manager, it is executed and the results are returned to the scheduler who in turn passes the results back to the transaction manager. If the scheduler rejects the operation, it informs the transaction manager and an abort is issued. Lastly, if the scheduler issues a delay, the operation is held in a queue within the scheduler and will be processed at a later time with either an execute or reject operation.

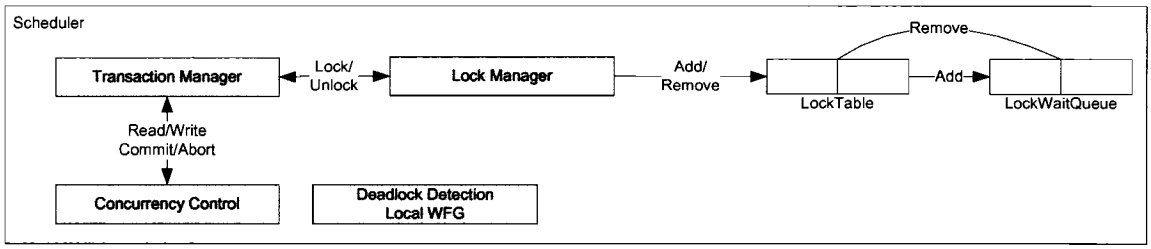

Figure 11: Scheduler

3.1.3. The Recovery Manager

The recovery manager (RM) (Figure 12) is responsible for guaranteeing that the database is consistent. The RM uses several actions to process transaction operations: read, write, commit, and abort. When a transaction is executing normally, the RM uses the cache manager for performing read and write operations against the database. When a transaction commits or aborts, the recovery manager (RM) performs an update or rollback of the transaction operations. The RM is also responsible for recovering incomplete transactions due to system failures. Since the cache manager and recovery manager are

highly dependent on one another, they are grouped together into one component called the data manager.

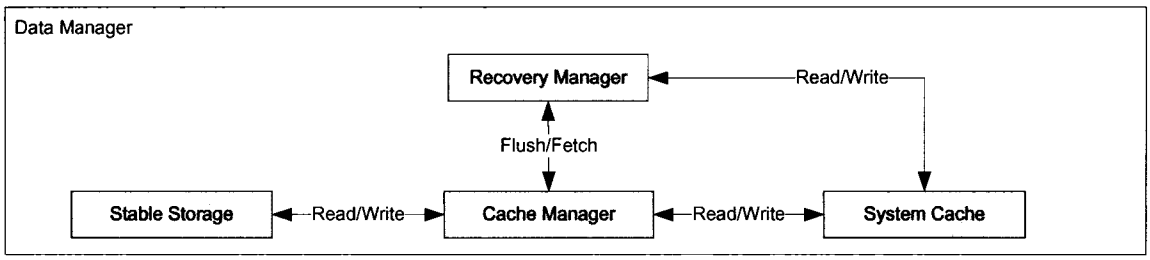

Figure 12: Data Manager

3.1.4. The Cache Manager

A database system stores its pages of data on a stable storage device such as a physical disk, which is typically slow and is prone to failure. In order to provide database performance that is acceptable, a subset of the pages of data can be kept in volatile storage, which is also known as the cache. The cache is considered very efficient, but is also prone to hardware or operating system failures. Managing the cache is the function of the Cache Manager (Figure 12). The cache manager uses two operations, fetch and flush, to retrieve data from stable storage and to remove data from the cache. For example, a flush operation typically occurs when the cache manager fetches data from stable storage and needs to remove or flush the old data from the cache in order to make room for the new data.

3.2. Concurrency Control

Concurrency control ensures that database transactions execute in a serial order and that they conform to the integrity rules and constraints of the database. The responsibility of the transaction manager, cache manager, scheduler, and data manager are to guarantee that these criteria are met. Tight integration between these components is required in

order for the sequence of actions to be processed correctly. The transaction execution model provides details on how a transaction operations move throughout this model.

3.2.1. Transaction Execution Model

When a transaction arrives at a node from across the network or is delivered locally, the transaction manager accepts the transaction into a transaction queue. The transaction manager determines the operations that the transaction will perform and submits a request to the lock manager to obtain any required locks. Once the locks have been acquired, the transaction is passed on to the scheduler. The scheduler then determines the order of execution for the operations and sends them to the recovery manager. The recovery manager records the operations and passes them to the cache manager. The cache manager retrieves the pages from disk for a read/write operation and submits the operation to the processor for processing. Each component within the transaction execution model passes requests to and receives replies back from their child component. The interaction of the components within the system is demonstrated in Figure 13.

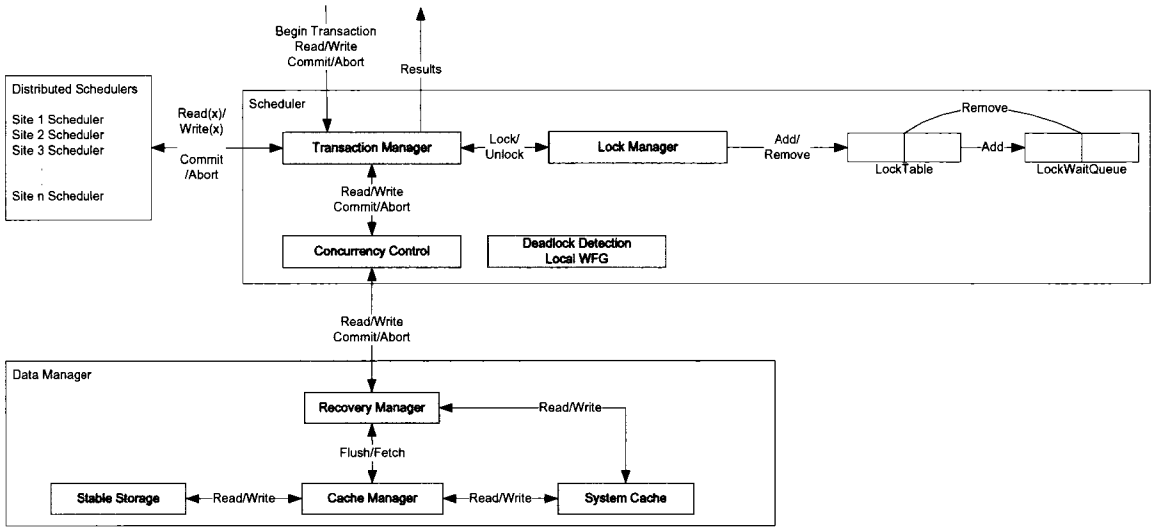

Figure 13: Transaction Execution Model

3.3. Distributed Real-Time Transaction Processing Simulator

DRTTPS (distributed real-time transaction processing simulator) is a discrete event simulator that models a distributed real-time database system. The primary purpose in developing the simulator is to implement new concurrency control protocols and analyze their performance against existing protocols. This section presents a brief discussion about discrete event simulation followed by the architecture of DRTTPS.

3.3.1. Discrete Event Simulation

Discrete event simulation is the study of a complex system that uses a time based approach to model real-world events. The simulations consist of running a multitude of scenarios that are comprised of well-defined sets of parameters and produce statistical results about the complex system. These results are then used to analyze performance, prove a concept, or lead to other outcomes about the system. One method for performing discrete event simulation is to process events in a sequential order with a variable clock. Using an event

queue, events are sequentially ordered in the queue based on the event's execution time. At each sequential time increment, the next event is removed from the queue, processed, and the clock is advanced.

There are two approaches in looking at time within a simulator; time can be driven by a specified interval or by the next event [25]. The interval approach advances the model forward at fixed intervals of time (ex. 5 seconds or x number of ticks). Regardless of whether there are events or not, the simulator will proceed to the next interval. The *next* event approach advances the model forward to the next significant event in time. Despite the amount of time between the previous and next event, the model will move forward to where it can process the next event. The next event model allows the simulator to quickly move forward in time, allowing for simulations to be evaluated quicker. However, there are scenarios where both approaches are valid and could be used.

3.3.2. Simulator Events

In DRTTPS events are processed in a sequential manner and are driven by the next event. The event clock is measured by a unit called a tick. A tick is a discrete amount of time in which one or more events can be executed. These events are actions performed by any component in the simulator such as a message sent between two nodes, a page being processed, a transaction arriving at a node, etc. Events are inserted into the queue based on the order in which they need to execute. For example, if it takes a processor 3 ticks to process a page, an event to "start processing" will be inserted into the queue and subsequently 3 ticks later a "done processing" event will be inserted into the queue. The

processing of events in the queue is achieved by starting at the beginning of the queue and sequentially processing every event until the end of the queue has been reached.

3.3.3. Simulator Topology

In its simplest form, the DRTTPS can be represented by a single database node, which contains one real-time database system. However, the simulator's topology (Figure 14) is typically comprised of one or more sites with one or more nodes within each site. Each node within a site is connected by a local area network and each site is connected by a wide area network. The connections between the nodes and sites are called network connections.

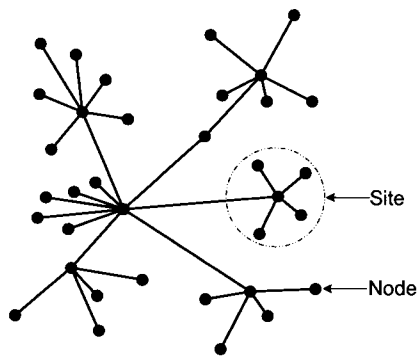

Figure 14: Example Network Topology

Each network connection is a pipe that represents a one way connection between one node or site and another. Each site contains a virtual router and is responsible for all network connections between nodes within a site and for the connections between itself and other sites. In addition, every network node has its own routing table that contains routes to all possible destination nodes. Depending on the network structure, a message traveling from point A to point C may be routed through other nodes or sites to reach its

final destination. If a network connection between two sites fails or recovers, all routing tables are updated to ensure that the routes represent the most efficient path. In the case of a failure, all of the messages that were sent and have not yet arrived at their destination are re-sent, and eventually arrive at their destination. Every network connection contains several important attributes:

- *Source* is the site or node from which a network connection originates.
- *Destination* is the node or site to which the network connection is connecting.
- *Bandwidth* is the maximum number of message units that can be sent down a network connection for each tick.
- *Latency* is the number of ticks a message unit takes to travel through a network connection.
- *External* usage is a percentage of the available bandwidth that is used by other users of the network.

The effective bandwidth of each network connection that is available to our system is calculated by the total bandwidth minus the percentage of external usage for that tick.

3.3.4. Node Architecture and Configuration

The node architecture within the simulator is represented by several hardware and software components that consist of: processor manager, disk manager, buffer, and optionally a workload generator. The processor manager logically organizes and controls all processors in a node. A node can contain one or more processors that can process at most one data page at a time under normal conditions, or multiple pages using simultaneous multi-threading. Instruction and thread level parallelism can be used to achieve

performance gains by issuing instructions from different threads in the same cycle. Hyper-Threading Technology (HT) [22] is the microprocessor simultaneous multi-threading technology (SMT) that DRTTPS uses to support the concurrent execution of multiple separate instruction streams on a single physical processor. Processor managers do not contain any simulator configuration parameters; however processors have two parameters: *Processing Time* which determines the amount of time that it will take to process one page; and *Hyper-Threading* which determines whether Hyper-Threading is enabled and the average percentage of performance improvement achieved for enabling multiple instructions streams.

The disk manager logically organizes all disks in a node. Each disk is a non-volatile storage device which contains a specified set of pages and can perform one read/write operation per tick. Pages can be replicated across multiple disks within the same node without any affect to the concurrency. However, pages that are replicated across multiple nodes require that a replicated concurrency control protocol be configured for each node that belongs to a replicated group. Disk managers do not contain any configuration parameters; however the disks have two parameters: *Access Time* which determines the amount of time that it takes to read or write a page to and from a disk; and *Page Range* which defines the range of pages that are stored on each disk.

The buffer is a volatile storage location that is used to improve the performance of retrieving pages from a disk. When a transaction requests a page for a read/write operation, it is read from disk and placed in the buffer. Pages remain in the buffer until the transaction commits/aborts, at which time the lock on the page is released and the page is

no longer needed. Since the buffer has a size limit, virtual memory or swap is available to allow pages to be written to a swap disk when the buffer becomes full. The swap disk is represented as physical disk and contains only one parameter, *Access Time* which defines the number of ticks that it takes to read or write a page to swap. The buffer contains two parameters: *Number of Pages* which is the maximum number of pages that can be stored in the buffer; and the *Victim Selection Protocol* which is used to decide how pages are to be removed from the buffer when they are no longer needed, and how to determine which pages will be moved to the swap disk.

A workload generator is a component of a node which creates transactions to be submitted at that node. As soon as all of the transactions have been created within the simulator, the workload generator goes dormant. The workload generator has a set of parameters that control the number and nature of the transactions that are to be created and is defined as the following:

- *Size* defines the number of transactions that a workload generator will create during a simulation run.
- *Arrival* is the rate at which transactions enter the system.
- *Slack Time* is the number of ticks that a transaction can be delayed and still meet its deadline.
- *Worksize* defines the number of pages that a transaction will access during its execution.

- *Pages* is a set of unique pages, up to the worksize, that a transaction will access during its execution and is based on a probability distribution over the total number of pages in the simulator.
- *Update* is a parameter that controls the probability of a page being written to or updated during a transactions execution.

A node is characterized by all of the components previously discussed with the addition of some key characteristics that provide the basis for transaction processing:

- *Concurrency Control Protocol* defines the behaviour of concurrently executing transactions within a node and ensures that transactions are executing atomically.
- *Preemption Protocol* controls how the preemption of transactions will occur within a node and is used in the situation of a priority inversion.
- *Deadlock Resolution Protocol* determines the method used to decide which transaction(s) to abort in the event that a deadlock occurs.
- *Priority Protocol* controls the behaviour of all the priority queues within a node including the transaction queue.
- *Replication Protocol* controls the activity of providing access to distributed replicated data across the simulation.
- *Max Active Transactions* defines the maximum number of transactions that may execute simultaneously on a node. If the number of transactions at a node exceeds this value, then transactions are forced to wait in the Transaction Wait Queue. Max active transaction is normally used in the absence of deadlock detection.

- *Transaction Timeout* is a value that defines how long a transaction can wait until it is assumed to be deadlocked and is aborted.

3.3.5. Preemption

If a transaction attempts to obtain a lock on a page, but cannot due to another transaction holding the lock, then an attempt will be made to determine if the transaction holding the lock should be preempted. The preemption protocol for a node determines which, if any, of the waiting transactions should be aborted or priority escalated. In DRTTPS, all transactions that are unsuccessful at obtaining a lock trigger the preemption protocol. The Preemption protocol takes a list of all transactions that are preventing a successful lock and returns a list of transactions that meet a set of criteria for preempting. Three preemption protocol options are available in DRTTPS:

- High Priority: Waiting transactions that have a higher priority will cause the lower priority transaction to be aborted.
- Priority Inheritance: Waiting transactions that have a higher priority will cause the lower priority transactions to inherit the priority of the higher transaction and finish processing rather than be aborted.
- Never Preempt: transactions will never be preempted.

3.3.6. Priority Protocols

A priority protocol is used to determine which object within a priority queue should be processed next. In DRTTPS, priority protocols control the behaviour of all of the priority queues within a node including the transaction queue. The priority protocols in DRTTPS are:

- Earliest Deadline First: The priority is based on the deadline of active transactions. The transaction with the earliest deadline will be given the higher priority in order to meet its deadline.
- First Come First Serve: This priority is a FIFO queue that is based on servicing transactions in the order in which they arrive.
- Shortest Job First: Transactions that require the least of amount of total time to complete will be assigned the highest priority.

3.3.7. Deadlock Resolution

DRTTPS uses deadlock detection to resolve deadlocks among transactions when they occur in the system. The system generates a list of transactions that are involved in the deadlock and determines which transaction will be aborted to resolve the deadlock. There are several options for resolving the deadlock:

1. First Deadlock Resolution: A transaction that appears at the top of the randomly generated deadlock list will be aborted. This is equivalent to choosing a random transaction to abort.
2. Priority Deadlock Resolution: The transaction with the lowest priority will be selected as the victim and aborted.

3.3.8. Simulator Interface

At the core of DRTTPS exists the SetupTool which allows the user of the simulator to graphically create the simulation topology, site structure, node structure, and configuration parameters. In addition, it provides the functionality to run simulations, save simulations,

and export the configuration. The SetupTool is based on the simulation model described in section 3.1, which contains a user defined topology that consists of simulator components and parameters. The topology is defined by one or more sites that are connected by network connections. Each site contains one or more nodes that consist of disks, processors, and other components. The behaviour of these site components are controlled through the creation of parameters.

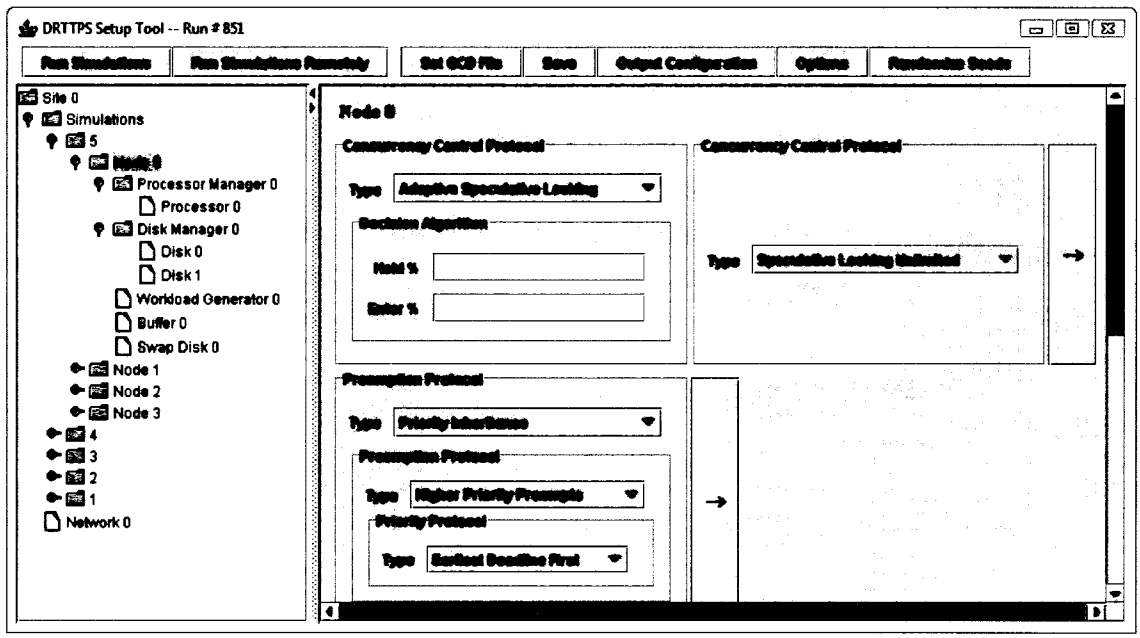

Figure 15: DRTTPS Simulation Setup Tool

The SetupTool interface (Figure 15) consists of two adjacent panels: the left panel that consists of the simulation components (represented as a tree); and the right panel that contains all of the associated component parameters. The SetupTool is completely object oriented and encapsulates all of simulation components and parameters within the SetupTool object. When a simulation is saved, all of the encapsulated objects, including the SetupTool are serialized into one binary image. When the SetupTool loads, it dynamically

generates the entire user interface based on the objects stored within the serialized binary image.

To fully analyze a protocol, simulations must be run using various permutations of parameters. The number of simulations created depends on the number of variation containers specified for each component in the component parameter panel. DRTTPS takes advantage of multiple CPU configurations on the host system to run these simulations in parallel for a quick turnaround time. Once the simulations have completed, the simulation statistics can be saved and opened in the ReportTool which provides the ability to create, view, and export graphs based on the statistics collected.

3.3.9. Design Considerations

At the time of developing DRTTPS, Sun Microsystems Java [26] was chosen as the implementation language based on its object orientated design and for its flexible multi-platform environment in which to develop our simulator. All protocols within the simulator use a common underlying pluggable framework that provides the foundation for developing modular protocols such as the concurrency control protocols (CCP). Every CCP in the simulator is a pluggable component of a node based on a common base class that provides structure, functionality, and interfaces for interacting with other simulator components. Using multiple levels of class inheritance, several categories of CCP protocols were created: Baseline protocols, Level II protocols which inherit from the baseline protocols, and Level III protocols which inherit from Level II protocols (Figure 16).

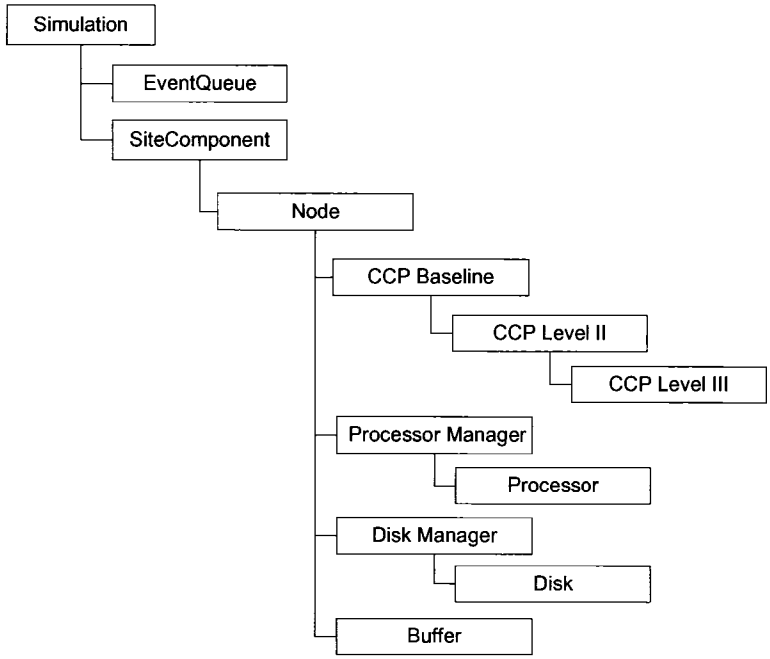

Figure 16: High Level Simulation Class Structure

The simulator architecture was developed in a manner that allowed for the modelling of more complex protocols without having to rewrite the simulator. An example of this would be a concurrency control protocol that assumed infinite memory. While this assumption is valid under certain conditions, other protocols may suggest that memory is finite and that virtual memory is used in the event that system memory is fully allocated. DRTTPS allows for this flexible and dynamic configuration of system components and protocols.

An efficient event model that allows for hundreds of thousands of events is an essential requirement for the simulator. A discrete event model based on a global event queue that uses ticks as a measure of time was developed. Java reflection [26] was used to create a generic global event queue for simulation event execution, which avoided the creation of message passing classes and the registration of senders and listeners. In order to create a new event, the following public member function is called:

```
synchronized public void addEvent( Event e)
{
        e.addTimeOffset( t );      //adds the time offset to the event time t
        e.setID( nextid++ )        //sets the ID as the next event ID in the global queue
        e.setEventQueue(this);     // sets the event queue to be the global queue
        events.add(e)              // adds the event object to the global event queue
}
```

The name of the method within the object is then passed to a newly created event object and is placed in the global event queue at a calculated time location. Using Java's reflection, the object is then created and executed by getting the event from the event queue and executing the following methods:

```
Method m = (Method)methodcache.get(name);      // Get the name of the method in the object
result = m.invoke( sc, args );                 // Invoke the method
```

With the assistance of a simple cache of frequently called methods, the overhead in utilizing the Java reflection methods for each event is extremely small.

To analyze the simulator performance, the creation of statistics is obtained through the use of Java's anonymous classes and a mere ten lines of code. This allows us to track almost any value in the system and display it in a graph. The method for creating a new statistic is:

```
Statistic time_usedHyper = new Statistic()
{
        // Function to process an event
        public void process(Event e)
        {
                // If the name of the event is "Start"processing then
                // set a begin interval
                if(e.getName().equals("startProcessing"))
                {
                        beginInterval(e);
                }
```

```
                    // Else if the processing is complete or has aborted, add the end
                    // interval for the event
                    else if (e.getName().equals("processingComplete")
                            || e.getName().equals("abortProcessing"))
                    {

                            add(endInterval(e));
                            if (busy)
                            {
                                    beginInterval(e);
                            }
                    }
            }
    }
    // add the statistic of the time using hyperthreading.
    addStatistic(time_usedHyper);
```

The statistics generated from the simulator are represented by graphs that are displayed in real-time within the running simulation (Figure 17). The ability to zoom into or mark points of interest while the simulation is executing is also available. Once the simulation completes, these graphs and statistics can be viewed using the ReportTool.

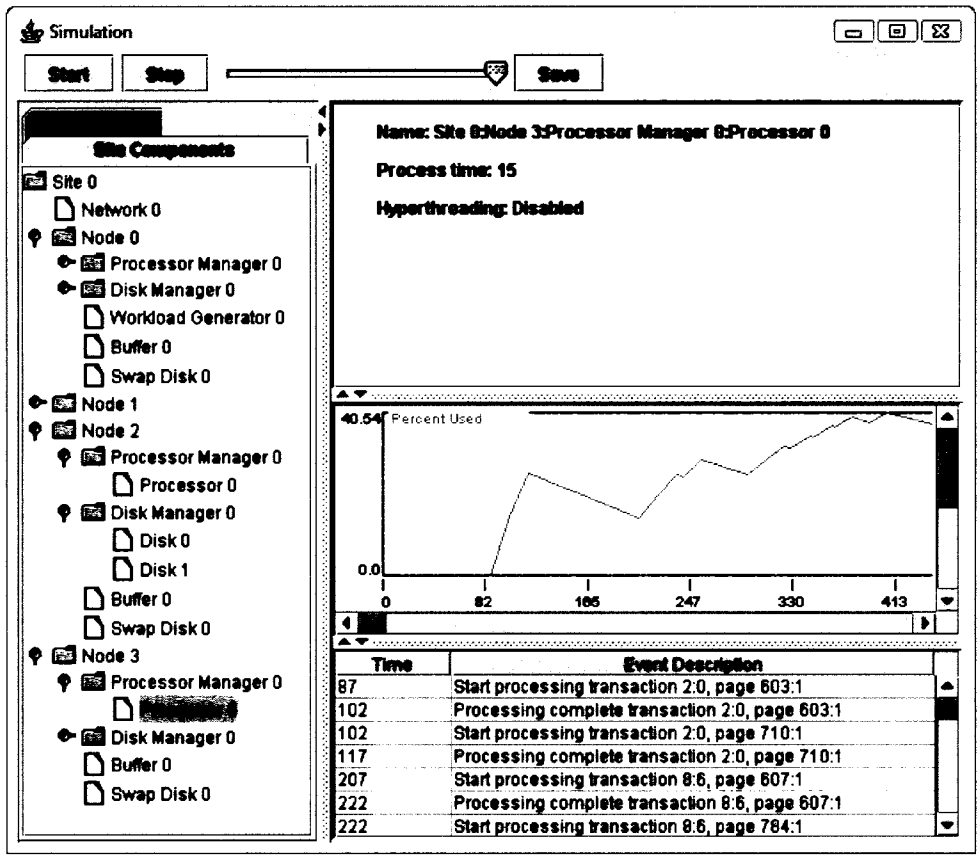

Figure 17: Viewing Graphs in the SetupTool

To save or load the simulator configuration or results, Java serialization [26] was used. Since the simulator is represented by a single object that encapsulates all other simulator components, user interface, statistics, etc., we simply serialize or de-serialize the simulation object to save or load the simulator as illustrated below:

```
public void SaveSimulation()
{
	// Open a file for writting
	FileOutputStream f = new java.io.FileOutputStream(file);
	ObjectOutputStream o;

	// If using the GZip compress the simulation
	If(Simulation.USE_GZIP)
	{
		// Compress the simulation
		GZIPOutputStream g = new GZIPOutputStream(f);
		o = new ObjectOutputStream(g);
	} else
```

```
        {
                // Don't compress the simulation
                o = new ObjectOutputStream(f);
        }

        // Serialize the SetupTool to a file and close
        SetupTool.this.serialize(o);
        o.close();
}
```

Figure 18: Serialization of SetupTool/Simulator

Naturally, this method of saving an application to disk requires a large amount of disk space. We therefore utilized Java's GZip package to compress the objects in memory into a substantially smaller footprint stored on physical disk.

3.4. Speculative Locking Protocol

In DRTTPS, speculative locking is composed of three levels of protocols (Figure 19): Concurrency Control, Abstract Speculative Locking, and variations of Speculative Locking, which includes the adaptive speculative locking protocol.

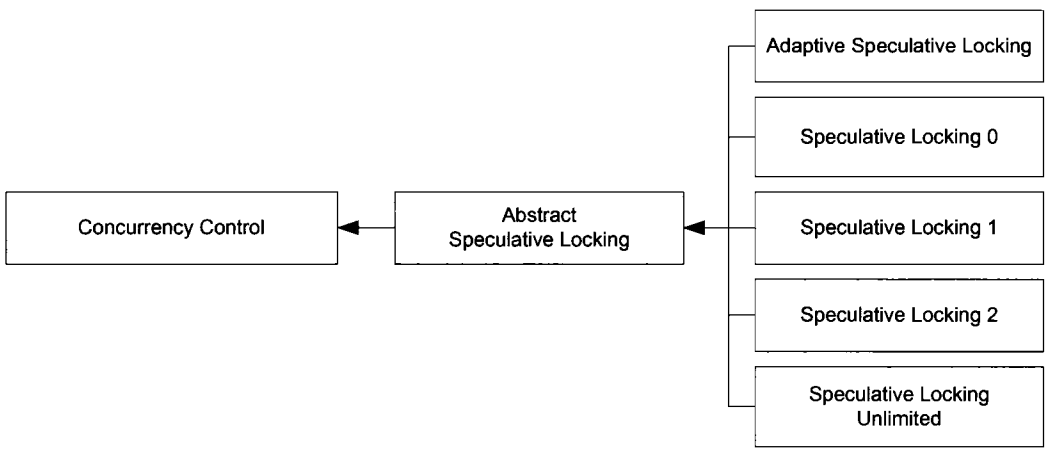

Figure 19: Speculative Locking CCP Dependencies

The concurrency control protocol is the baseline protocol (BP) of all the concurrency control protocols that are created within DRTTPS. The BP provides default methods (Table 2), attributes, and interfaces for all of the CCPs in the system.

Method	Description
startTransaction	Called when a transaction is started to obtain locks on pages
endTransaction	Called when a transaction has ended
doneProcessing	Called when a transaction is finished and all of its pages are ready to commit
tryLock	Attempt to obtain a lock for a given transaction on a given page
Lock	Attempt to obtain a lock
hasLock	Check to see if there are any locks that are help or waiting under this CCP
doUnlock	Release a lock that is held
releaseLocks	Release all locks for a given transaction (used for aborts)
tryToPrempt	Try to preempt a transaction to obtain the required locks
notWaiting	Returns a list of transactions that do not have waiting lock requests
detectDeadlocks	Detect a deadlock and resolve by aborting a transaction
tryActiveWaiting	Try to activate waiting transactions

Table 2: Baseline Concurrency Control Protocol Methods

The Abstract Speculative Locking (Abstract) protocol extends the BP and is an abstract base class for all of the speculative locking (SL) protocols [10]. It overrides many of the BP methods with specific speculative locking functionality and provides the tree structures required for the processing of before and after images. Finally, all of the speculative locking protocols extend the Abstract protocol to provide additional functionality that pertains to each specific SL protocol. The variations of the SL protocol that are used in DRTTPS are SL(0), SL(1), SL(2), SL(Unlimited). The SL protocols are complex objects that inherit all of their functionality from the BP and the Abstract protocol with the addition of version information and abort requirements that are distinct to each protocol. A detailed discussion of the BP is not necessary as the speculative locking abstract protocol extends

the BP and overrides the majority of the baseline methods and functionality with that of SL protocol, such as the speculative tree and locks. Since the variations of SL protocols inherit from the Abstract protocol and only override one method in the Abstract protocol, the discussion that follows focuses on the functionality of the Abstract protocol.

The Abstract protocol has three types of locks: read, execution write (EW), and speculate-write (SPW). The lifetime of a lock is dependent on the type of lock. A read lock can be released right away, whereas an EW lock cannot be released until a transaction has committed or the lock has been changed to an SPW lock. When a transaction starts, it accesses all of the versions in the speculative tree and decides how many versions are required to be locked. As an example, in the situation where a transaction T2 is requesting access to X and X is being written by T1, T2 cannot proceed until the lock is changed from EW to SPW and the page that is being locked is versioned. Further, prior to T2 being allowed to start processing, T2 must have all of its pages locked. In order to understand how the versions are coordinated, we introduce the concept of Lock Groups.

Locks Groups (LG) are used to coordinate the links between the old, new, and aborted pages. The LG contains a list of pages that have been obtained, pages that are waiting to be obtained, and the state of each page. When a transaction requests access to a set of pages (Figure 20), the CCP finds the versions of the pages that are requested within the speculative tree, requests locks on those versions, and stores the locked pages with the type of lock in the CCP. The versions of the pages in the speculative tree do not map one-to-one with the locks, therefore the before and after images of a page are represented by a page set that is locked. Next, the CCP creates a lock group for the transaction and places pointers

to those locks into the newly created lock group. If a transaction wants to determine what locks exist, it can query the CCP or the lock group, but only the lock group knows the relationships amongst the locked pages.

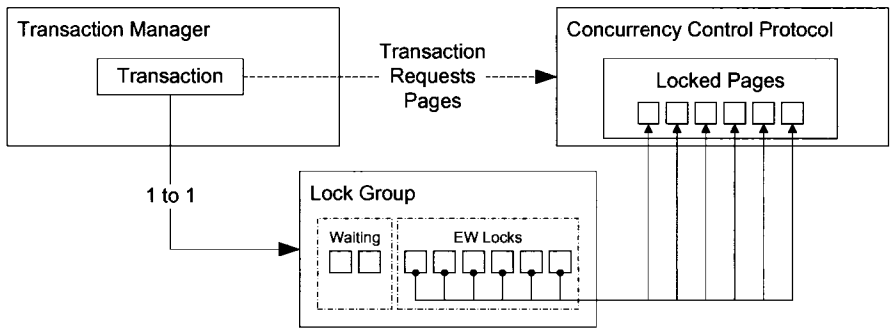

Figure 20: Transaction Requesting Pages

Once the CCP has obtained all of the locks, it informs the transaction manager and goes to sleep. The transaction manager then invokes the data manager which reads the pages from disk and the transaction starts processing. During the processing phase, if a page completes processing, the transaction manager notifies the CCP and the locked page gets downgraded to a speculative write (SPW) lock and is stored in a newly created LG. The transaction manager continues to wait for all of the pages to complete processing and to be converted to SPW locks. Once the SPW locks become available to other transactions, the 2PC protocol begins the commit processing with the prepare-phase, followed by the commit phase. In the commit phase, all of the read pages are released and the pages that have been updated are written. As soon as a page has been written, it releases the lock. After the last page has been written, the transaction completes, and notifies the master which then determines whether it can commit based on the responses from the sub-transactions.

Transaction aborts in the SL protocol are bound to occur and happen for two reasons: transactions are preempted or a global deadlock occurs resulting in a timeout. In the event of an abort, the CCP releases all of the locks for the aborted transaction and goes through the waiting lock queue and discards locks that are no longer required. The CCP then goes into the speculative tree and trims the tree by removing nodes that are no longer required and realigns the tree. The trimming of the tree will remove versions and execute a partial abort on versions that are currently waiting. A broadcast is then sent out to all of the versions of this page so that they do not get processed. The disk queue and swap disk notifications are removed and finally the aborted transaction is placed back in the transaction manager's queue.

Chapter 4

Adaptive Speculative Protocol

The Adaptive Speculative Locking Protocol (ASL) for distributed real-time database systems uses the speculative locking protocol for its underlying architecture and integrates the adaptive nature demonstrated in the AEP protocol. The Adaptive Speculative Locking (ASL) protocol augments the Speculative Locking protocol with several major contributions: hyper-threading, memory management including virtual memory, and transaction queue management.

ASL removes the assumption of unlimited system memory by restricting the local buffer to a maximum size and by providing a page based virtual memory mechanism that uses a memory management algorithm to control memory allocation and de-allocation. When a transaction enters the system and is scheduled, it requests all of the page locks that it requires. The pages are then retrieved from disk and are placed in the local system buffer in a locked state. During transaction processing, transactions perform speculative executions on data, which result in before and after images being allocated in memory. In the event that system memory becomes fully allocated, the pages in the local buffer that are no longer needed or used are overwritten. However, in the situation where there is no free space in the buffer, one of two things could occur:

- The transactions waiting in the transaction queue are held back until the system buffer becomes de-allocated to a minimum level.

- Transactions that are waiting on other transactions for pages are moved to virtual memory, freeing up space in system buffer.

The algorithm used to perform the memory management utilizes partial system knowledge in order to make the appropriate decisions.

In order to improve on the performance of speculative execution, the concept of parallel processing is investigated. Simultaneous multi-threading (SMT) is a cost effective way to utilize instruction level and thread level parallelism to achieve performance gains. Hyper-Threading (HT) [22] is an SMT technology that supports the concurrent execution of multiple separate instruction streams, on a single physical processor. The performance improvement of HT is application and hardware dependent, and has shown up to a 65% performance increase over previous generation processors [22]. In benchmarks, Intel has seen a 21% increase in processor performance for online transaction processing (OLTP) workloads and has shown gains of up to 30% on common server application benchmarks [22]. On a prototype DBMS using TPC-C equivalent benchmarks, it has been shown that HT results in a performance speedup of up to 16% due to the reduction in L2 cache miss rates [23]. For this reason, ASL uses simultaneous multi-threading to improve the overall system processing performance by creating a thread for each speculative execution and allowing the threads to execute simultaneously on the same processor.

Transaction aborts in a DRTDBS occur for many reasons. Deadlocks and transaction timeouts are the most common. In the SL protocol, the maximum number of speculative executions that could occur for transaction T_i was defined by the number of preceding transaction aborts that could occur prior to transaction T_i. In the naive version of

speculative locking SL(n), the number of speculative executions of transactions exploded with data contention. The variants SL(1) and SL(2) were developed to mitigate the number of speculative executions under data contention. In ASL, we use the naive version of SL(n) and have developed a concurrency control algorithm that restricts the number of speculative executions without enforcing a limitation on the number of previously aborted transactions, resulting in the reduction of discarded processing. In ASL, Transaction aborts are also caused by deadlocks

4.1. Hyper-Threading Technology

Hyper-Threading (HT) allows a single processor to execute multiple threads simultaneously, which yields additional processing and improved performance. HT is achieved through the existence of two architectural states per physical processor, which is represented as two logical processors. Each architectural state is allowed to execute an instruction stream, which means that the operating system and user applications can schedule two concurrent processes or threads for each physical processor. However, the processes and threads still share the same processor execution engine, one cache-set, and one system bus interface. This results in a competition for the shared execution resources and the overall performance being less than that of two physical processors. Figure 21 shows a multiprocessor system with two non HT physical processors and Figure 22 shows a system with two architectural states for one hyper-threaded processor.

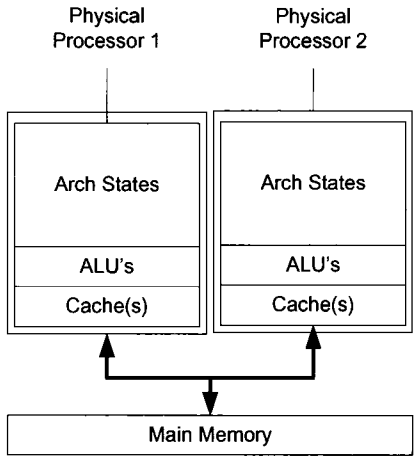

Figure 21: Two non Hyper-Threaded Processors

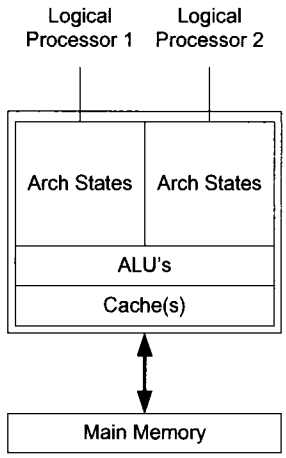

Figure 22: Hyper-Threaded Processor with Two States

The processing of pages takes a certain amount of CPU processing time. In DRTTPS, the amount of time to process a page is measured in *ticks*. As an example, to process one page could take 10 ticks. In a scenario where there is one transaction T_0 with one page to be processed, the amount of CPU processing time is equal to the time required to process a page, as shown in Figure 23.

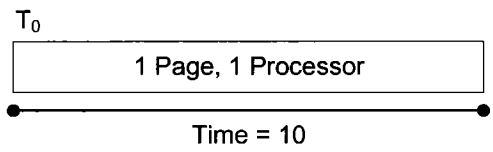

Figure 23: Processing One Page on One Processor with no HT

If there are two transactions T_0 and T_1, and each transaction is processing one page, then the amount of CPU processing required will be equal to that of the page processing time x 2, as shown in Figure 24.

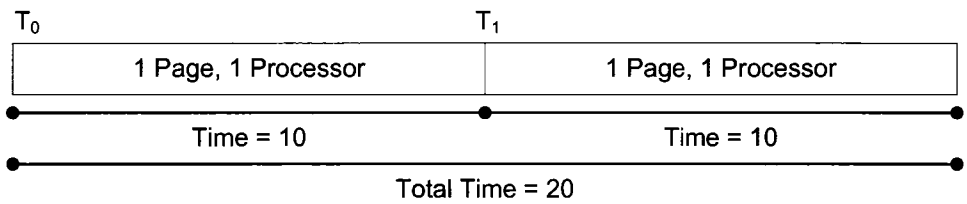

Figure 24: Processing Two Pages on One Processor with no HT

In a system that has hyper-threaded processors, there is an increase in processing that occurs when two threads can simultaneously execute on the same CPU at the same time. The expected performance improvement that is obtained through hyper-threading is measures as a *bonus* (*B*). The rate of expected performance improvement is defined as the amount of work to be done based on the bonus and is called the hyper-threaded rate (*HR*):

$$HR = \frac{1}{2 - 2B}$$

In calculating the hyper-threading rate or the bonus, it is important to note the following:

$$HR \times ticks = work$$

The amount of work is equal to the amount of processing time. Therefore, if the amount of work was 10, then the number of actual ticks required to process a job is:

$$HR \times ticks = 10$$

$$\frac{10}{HR} = ticks$$

If the *HR* that we are trying to achieve is known, then the bonus can be derived from the definition of the *HR* as follows:

$$HR = \frac{1}{2 - 2B}$$

$$HR = \frac{1}{2\,(1 - B)}$$

$$HR(1 - B) = \frac{1}{2(\cancel{1 - B})}\,(\cancel{1 - B})$$

$$(1 - B) = \frac{1}{2HR}$$

$$B = 1 - \frac{1}{2HR}$$

From these formulas, if we are processing two pages on a hyper-threaded CPU and our performance bonus is 30%, then the required time for both jobs to complete would be:

$$HR = \frac{1}{2 - 2B}$$

$$HR = \frac{1}{2 - 2(0.3)} = \frac{1}{1.4} = 0.714$$

$$ticks = \frac{10}{HR} = \frac{10}{0.714} = 14\,ticks$$

The formula to calculate the total time required for both jobs to complete with hyper-threading can be simplified to:

$$(2\ \times work) - (2\ \times work\ \times bonus) = work$$

$$2\ \times (10) - 2\ \times (10) \times (0.3) = 20 - 6 = 14\,ticks$$

Putting this to work, if we have a one processor node with hyper-threading enabled and three transactions T_0, T_1, and T_2 starting at the time 0, 3, and 5 respectively, then Figure 25 shows the number of ticks (*TCKS*) required to complete processing.

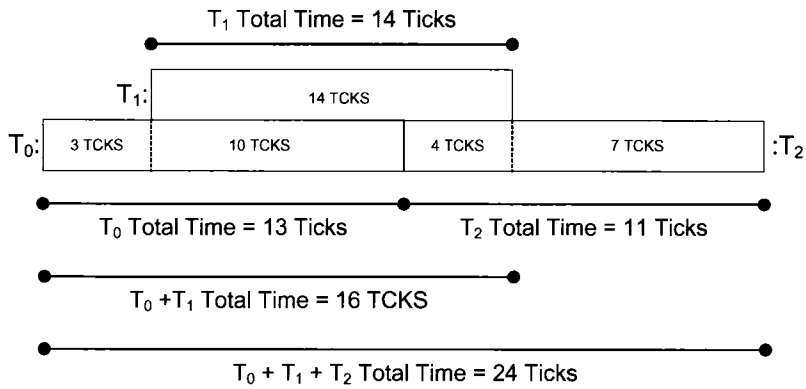

Figure 25: Hyper-Threading of Three Transactions

Transaction T_0 begins processing at time 0, proceeds for 3 ticks, and then transaction T_1 starts processing. At this point, the number of ticks that T_0 has left to process is 7 and the number of ticks that T_1 has left is 10. As transaction T_1 will execute longer than that of T_0, we first calculate the number of ticks remaining for T_0 which will also apply to the partial processing of T_1:

$$2 \times (7) - 2 \times (7) \times (0.3) = 9.8\ ticks$$

Therefore, both T_0 and T_1 continue to process for 9.8 ticks, at which point T_0 finishes processing and T_2 begins processing. Since T_1 and T_2 partially process together, the remaining processing of T_1 will be the total work of T_1 with hyper-threading:

$$2 \times (10) - 2 \times (10) \times (0.3) = 20 - 6 = 14\ ticks$$

Transaction T_2 overlaps with the remainder of T_1, so the beginning of T_2 is calculated as:

$$(T_1\ Total\ Ticks) - (T_1\ and\ T_0\ HT\ time) = T_2\ ticks$$

$$14 - 9.8 = RND(4.2) = 4\ ticks$$

Transaction T_2 processes for the 4 ticks, at which point T_1 finishes, and the remaining processing time of T_2 is then recalculated without hyper-threading by calculating the number of non hyper-threaded ticks:

$$ticks = \frac{work}{HR}$$

$$ticks \times HR = work$$

$$4 \times .714 = RND(2.85) = 3\ ticks$$

$$Total\ Processing\ Ticks - Processed\ ticks = work$$

$$10 - 3 = 7\ ticks$$

From Figure 25, the actual amount of time to process each transaction is greater than the set time to process a transaction. However, the overall time that it takes for all three transactions to execute using hyper-threading is 30% less than the total time it would have taken to process the transactions sequentially.

4.2. Memory Management

Within the simulator, there are two types of memory: system cache and virtual memory. Cache is a volatile storage area that is managed by the cache manager, in which data is stored for a short duration of time and can be accessed rapidly. Virtual memory is a technique that provides the ability to map non-contiguous memory as contiguous. Virtual memory (Figure 26) in DRTTPS is represented as paged memory that is stored in the system cache and is written or swapped out to a physical disk when the cache becomes full. The process of swapping is an expensive operation as physical disks are much slower than system memory.

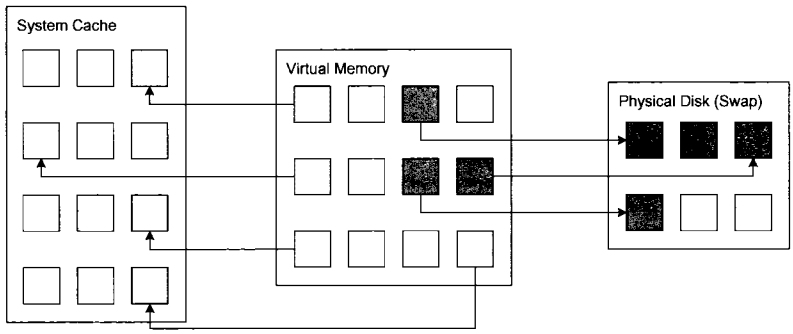

Figure 26: Virtual Memory

When a transaction enters the system and is scheduled to execute, the cache manager needs to reserve or hold space in the cache or swap disk prior to requesting pages from the database. A reserve request will look for space that has never been used, or space that was previously occupied by a transaction that has finished, aborted, or no longer belongs to an active transaction. When adequate space is reserved in cache, the transaction enters the ready state. If there is not enough space, the transaction will wait until there is enough room. From the ready state, the pages required for processing are read from disk into the cache and are locked by the transaction manager. The pages are then processed by the CPU and the locks and holds are then released. For transactions that require a database write, the pages must be in the buffer prior to writing.

In a more complex scenario, if a transaction has three pages reserved and three pages locked in the cache, and an incoming transaction requires space, then the three pages that are held will be swapped to disk and the transaction will wait. Once the waiting transaction proceeds, the pages that are held in swap will be swapped back into the cache and locked. If

everything in the cache is locked, then the incoming transactions simply wait until the active transactions complete or abort.

In the ASL protocol, the before and after images or versions of each page are treated as a single page and require their own slot in the buffer. In order for a transaction to process pages, the versions of those pages must be in cache prior to processing. In the event that versions of a page are swapped out to disk, those versions are stored on the disk as one page to be retrieved as a whole into memory when the transaction requires them. Further, when a new version of a page is created, a copy of the page is created in the cache and has the potential to cause other pages to be swapped out if there is no room in the cache.

4.3. Transaction Queue Management

Transaction Queue Management (TQM) is a feature of the ASL protocol that monitors system performance metrics and makes adaptive decisions on how the ASL protocol behaves in order to improve transaction throughput. In the SL protocol [10], the authors assumed that their system had infinite memory. However, in our system model, we use finite memory and have replaced infinite memory with virtual memory. In order to control the amount of system cache utilization and avoid page swapping, ASL monitors the system cache and controls the flow of transactions that enter the system from the transaction queue.

TQM uses an adaptive decision algorithm that monitors the amount of system cache that is available against the number of transactions waiting in the transaction manager queue, and then makes a decision to either hold or release transactions from the queue. The flow of transactions entering the system is controlled by two parameters, the hold level

(HL) and the enter level (EL). The HL and EL parameters are defined as percentage of the available system cache (ASC). When a transaction is delivered to the transaction manager, it checks the amount of available system cache and estimates the total amount of cache (ETAC) that the actively executing transactions could use. It then estimates the amount of cache that the current transaction (EACCT) will occupy in the system while executing. The ETAC and EACCT are then added together to get a total system cache utilization (TSCU) for both the active transactions and the current transaction. The TSCU is then compared to the TQM hold level. If TSCU > HL then the transaction and any subsequent transactions are held in the queue and prevented from processing. If the TSCU < HL, then the transaction is allowed to proceed.

In the event that TSCU > HL and transactions are held in the transaction queue, the transaction manager will continuously monitor the ASC value against the EL value. When the ASC falls below the EL value, the transactions are released from the queue and begin executing again. When setting the HL and EL values, it is important for the amount of system cache occupied at any point in time to be maximized and for the ASL protocol to minimize the number of pages being swapped. This configuration results in a balance between minimizing data contention and maximizing cache use.

4.4. Summary

Chapter three presented a detailed simulation model that was used to develop the Distributed Real-Time Transaction Processing Simulator (DRTTPS). The simulator is a critical component to not only the implementation of the Speculative Locking (SL) protocol but more importantly our Adaptive Speculative Locking (ASL) Protocol. In Chapter four, we

presented the details of DRTTPS and our ASL protocol using the SL protocol as a base. In the following chapter, we will use the simulator to analyze the performance of our ASL protocol against the SL protocol.

Chapter 5

Simulation Results & Analysis

An extensive set of experiments have been conducted in order to compare the performance of the Adaptive Speculative Locking (ASL) protocol with the Speculative Locking (SL) protocols. The DRTTPS simulator developed for this research was used to conduct these experiments and obtain statistical performance information that is required for this study. This chapter presents a detailed analysis of the results and discusses the performance implications of the ASL protocol design decisions.

5.1. Assumptions

As in any experiment, there are assumptions about the model and system that are being used, and our protocol and simulator are no exception. The following is a list of assumptions that have been made:

- Protocol Assumptions:
  - When a transaction is created the resource required (pages) are known.
  - No rollbacks can occur after a transaction commits.
  - Local deadlocks cannot occur, however global deadlocks can.
  - A transaction has the ability to move any of its held pages into sub-transactions.
- Simulator System

- No hardware failures can occur.
- The network is fully connected.
- Buffer access is instantaneous.
- Disks and Swap Disk are distinct resources
- An exclusive lock is not necessary until a page is written to disk.

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

5.2. Performance Metrics

The primary performance metric of our experiments is the percent of transactions completed on time (PTCT), which is the percent of total transactions that complete before their deadlines. The PTCT value ranges from 0 to 100 percent and is taken to represent all transactions that exist in the system. In our experiments, achieving 100 percent is the goal, but is unrealistic due to the nature of a distributed real-time database system. Stressing the system to high levels of resource utilization provides us with valuable information on how our protocol performs under periods of high load.

In addition to the primary metric, the DRTTPS simulator was designed to produce a multitude of statistics, including that of three other key performance metrics used in our analysis: percent of processor utilization (PPU), percent of disk utilization (PDU), and percent of swap disk utilization (PSDU). These additional metrics are required to analyze the node resource utilization and determine whether the protocols are behaving as predicted. The values of all three metrics range from 0 to 100 percent and are compared to

that of the PTCT results to identify trends in resource utilization for the ASL and SL protocols.

5.3. Experiment 1: Baseline Simulation

The performance evaluation of the ASL and SL protocols begins with the creation of a baseline experiment. The baseline provides a default configuration for all subsequent experiments, for which the default parameters are varied. The baseline workload and system parameters are provided in Table 3. The baseline values were chosen to demonstrate a system that is not under heavy load and to observe how the protocols behave and compare against one another.

Type	Parameter	Meaning	Value
Workload	*ArrivalRate*	Rate at which transactions enter the system	50 ticks
Workload	*WorkSize*	Number of pages accessed by a transaction	4-12
Workload	*Updates*	Percentage Probability that a page will be updated	100
Workload	*SimTransSize*	Number of transactions created in the simulator	100
System	*Nodes*	Number of nodes in the system	4
System	*MaxActTrans*	Maximum number of active transactions per node	30
System	*Processors*	Number of processors per node	1
System	*ProcTime*	Amount of time to process a page	15 ticks
System	*ProcHype*	Processor Hyper-Threading	Disabled
System	*Disks*	Number of disks per node	2
System	*DiskTime*	Amount of time to read a page from the disk	35 ticks
System	*SwapTime*	Amount of time to swap a page (cache <-> disk)	35 ticks
System	*Pages*	Number of pages per disk	100
System	*CacheSize*	Size of the system cache per node	5-80 pages

Table 3: Workload & System Parameters

Of the workload and system parameters, the ArrivalRate is one parameter that requires some discussion. The ArrivalRate is the rate at which transactions enter the system and is defined by the number of ticks between the creation of each transaction in the workload generator. A larger ArrivalRate value will result in a greater amount of time between the

creation of transactions, meaning that transactions will arrive slower. Conversely, a smaller ArrivalRate value will result in transaction arriving faster.

Our first experiment was conducted using the baseline configuration in Table 3 with the ASL protocols Transaction Queue Management (TQM) disabled. By design, the SL protocol assumes there exists infinite system cache, thus all experiments that include the SL protocols follow this assumption. However, the ASL protocol introduces the concept of TQM which assists in dynamically adapting the protocol to an increasing system load. The disabling of TQM causes the ASL protocol to behave as if there was infinite memory as assumed by the SL protocol. This configuration results in comparable performance between the ASL protocol and the SL protocols as shown in Figure 27.

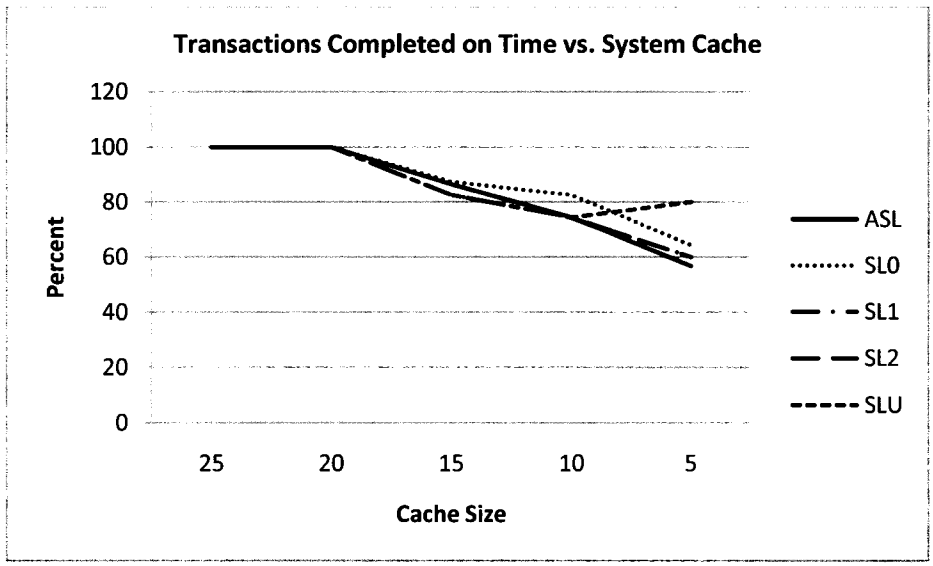

Figure 27: Experiment 1 - PTCT of the baseline with no TQM

From the baseline configuration, we can see that low system loads produce no data contention for cache sizes of 20 or higher and results in a PTCT of 100 percent. For smaller cache sizes, cache contention exists, which results in the degradation of performance due to

the lack of system cache for transaction processing. The degradation in performance causes transactions to miss their deadlines. Further, the lack of system cache proportionally affects other performance metrics as shown in figures Figure 28, Figure 29, and Figure 30.

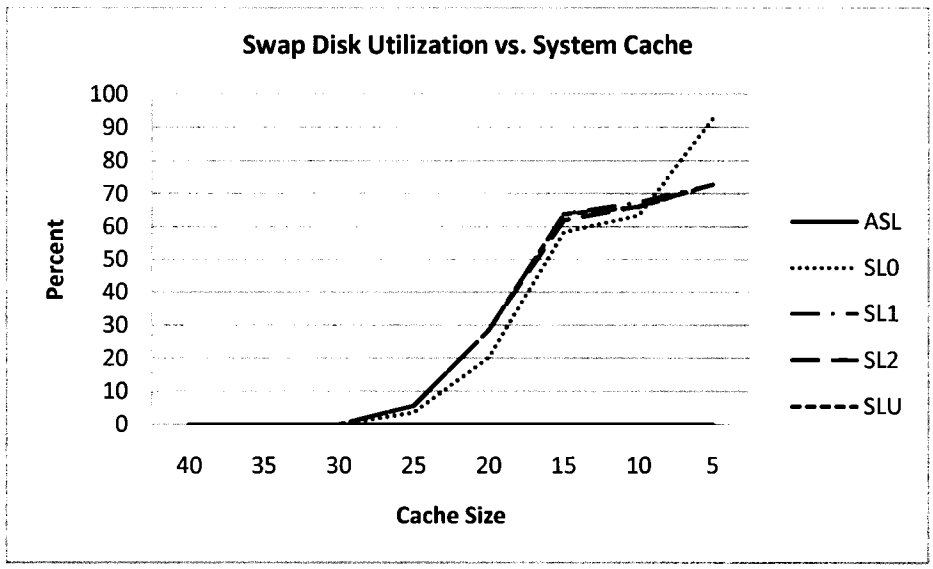

Figure 28: Experiment 1 - PSDU for the baseline with no TQM

In Figure 28 we observe an interesting result, at the same time we see cache contention; we also see the utilization of the swap disk. The increase in cache utilization is caused by the processing of new and existing transactions in the system. As new transactions enter the system, there is an increase in the number of pages being read into cache. The execution of the new transactions may cause other transactions to wait or be preempted. If a transaction has to wait or is preempted and there is data contention, then some or all of the transactions pages will be swapped out to disk. Subsequently, those transactions that have pages on the swap disk will at a future time attempt to swap the

pages back into cache and continue executing. This in turn causes a cycle of page swapping between the cache and the swap disk, resulting in thrashing.

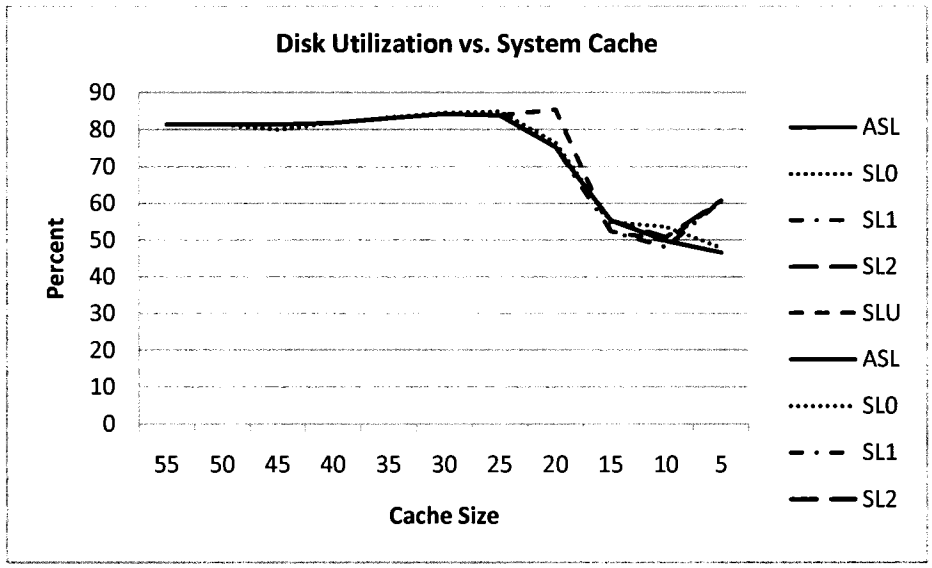

Figure 29: Experiment 1 - PDU for the baseline with no TQM

Once again, in Figure 29 we see the same type of change in performance with disk utilization as we did for the same cache sizes in the PTCT and the PSDU performance metrics. With PTCT and PSDU, we notice that as soon as we see data contention at the cache size of 20, we see a change in the disk utilization. In this case, the utilization of the disk decreases as new pages and updated pages are not being read or written to disk. Instead, we have traded the utilization of the disk read/writes with the increase in swap disk utilization.

The processor utilization in Figure 30 reaches a maximum of 61 percent and is almost identical in performance for all of the protocols. As the cache size begins to decrease, the processor utilization also decreases. This decrease is caused once again by the data contention in the cache. During the paging process from cache to swap disk and vice versa,

the processor is not utilized. Therefore the processor utilization in Figure 30 shows a decline in page processing resulting in the degradation of the PTCT performance.

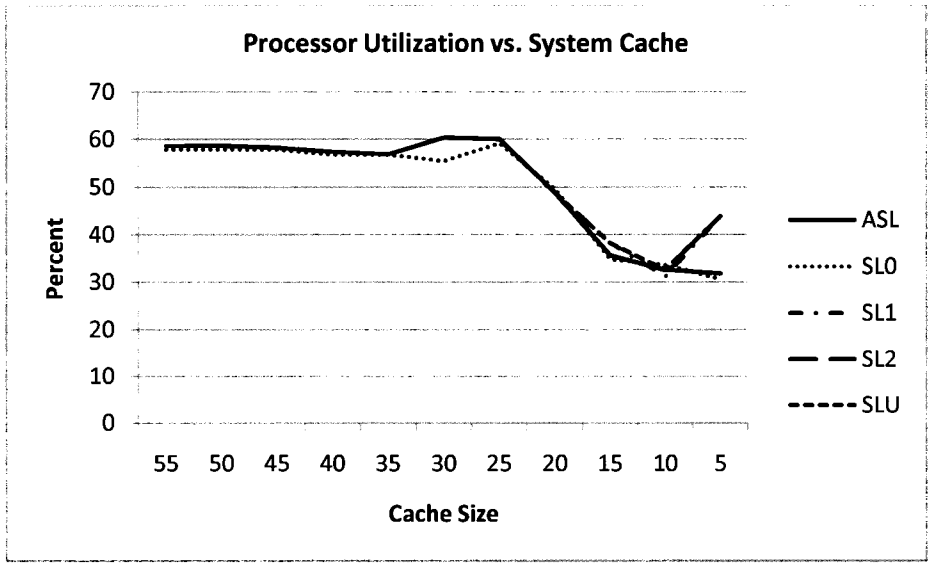

Figure 30: Experiment 1 - PPU for the baseline with no TQM

5.4. Experiment 2: Transaction Queue Management

Transaction Queue Management (TQM) is a feature of the ASL protocol that monitors system performance metrics and makes adaptive decisions on how the ASL protocol behaves in order to improve transaction throughput. In order to control the amount of system cache utilization and avoid page swapping, ASL monitors the system cache and controls the flow of transactions that enter the system from the transaction queue. There are two goals in experiment 2; the first is to observe the overall performance of ASL over the SL protocols, with and without TQM enabled, and second to minimize the impact that ASL has on system resources.

For this experiment, we observe two types of systems, our baseline configuration and a system with a fast arrival rate and a large number of transactions. The parameters for Configuration A (baseline) and Configuration B are shown in Table 4.

Parameter	Configuration A	Configuration B
Disks	2	4
Pages	800	1600
Processors	1	1
ProcHype	Disabled	Disabled
ArrivalRate	50	20
SimTransSize	100	200
Updates	100%	100%

Table 4: Experiment 2 - Configuration Parameters

5.4.1. Configuration A

We first observe configuration A, a system that has a light system load with TQM disabled. All protocols demonstrate 100 percent PTCT for the system cache size higher than 20 as demonstrated in the Figure 27 of the baseline simulation. We can see degradation in performance for cache size values that range from 5 to 20, which is caused by data

contention. We now enable TQM at a hold and enter level of 150 percent. Figure 31 shows the PTCT of the baseline configuration with TQM enabled.

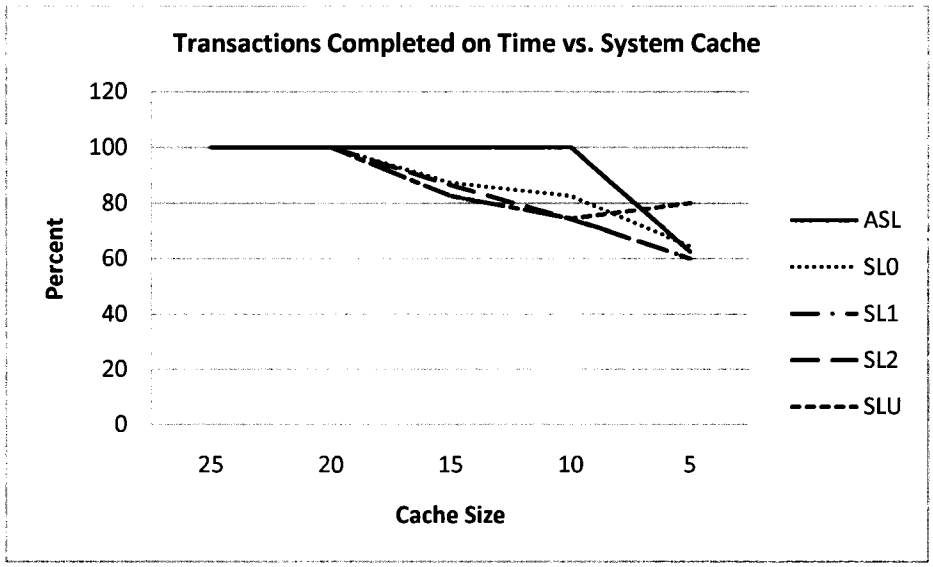

Figure 31: Experiment 2 – Configuration A - PTCT for the baseline with TQM enabled

We observe a performance improvement in the ASL protocol over the SL protocols which results in an increase of transactions completing on time. With smaller cache sizes, all protocols exhibit degradation in performance. Looking at the percentage of swap disk utilization (PSDU) in Figure 32, the ASL protocol swap disk utilization is zero. Consequently, both the disk (Figure 33) and processor (Figure 34) utilizations increase proportionally to the PTCT for the same cache size ranges. This immediately shows that TQM is effective at minimizing data contention, but still suffers from data contention for very small cache sizes.

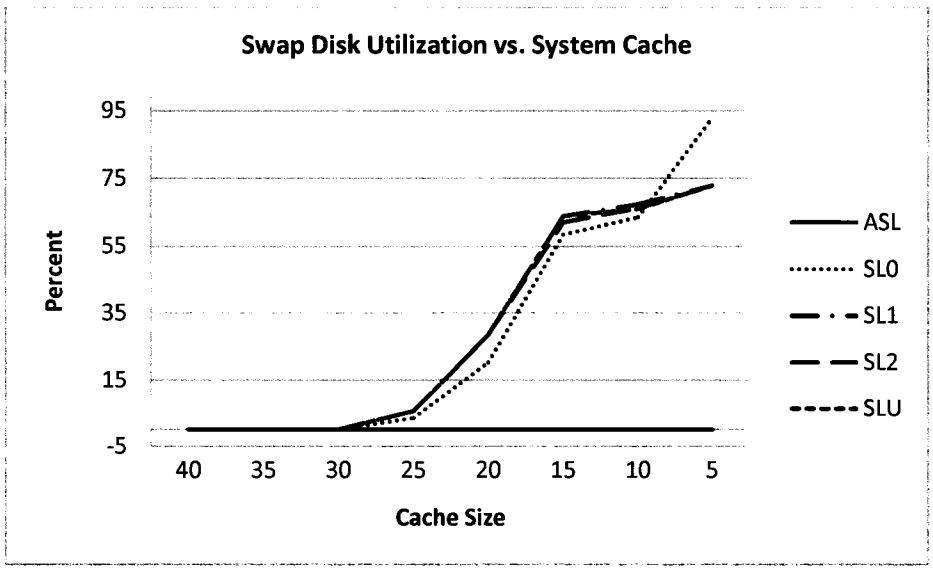

Figure 32: Experiment 2 – Configuration A - PSDU for the baseline with TQM

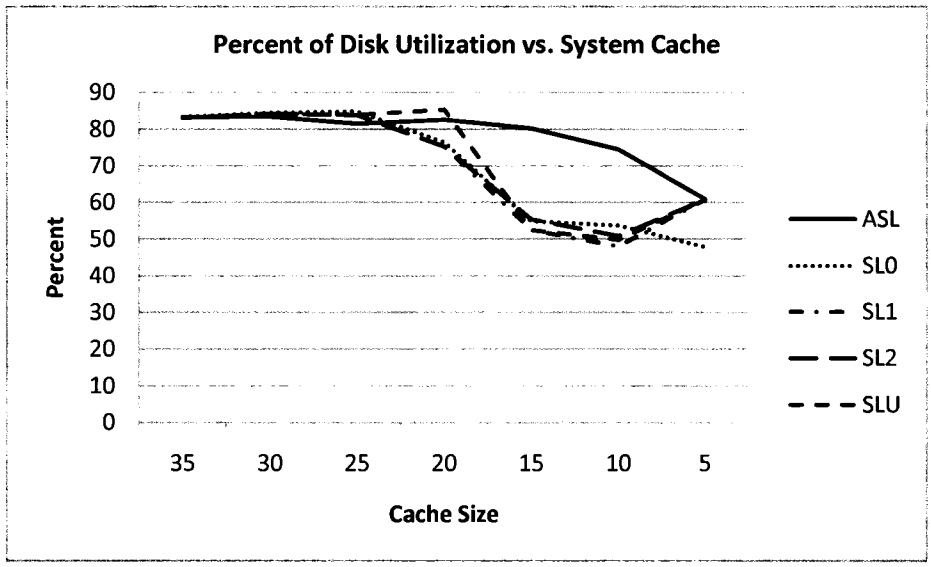

Figure 33: Experiment 2 - Configuration A - PDU for the baseline with TQM

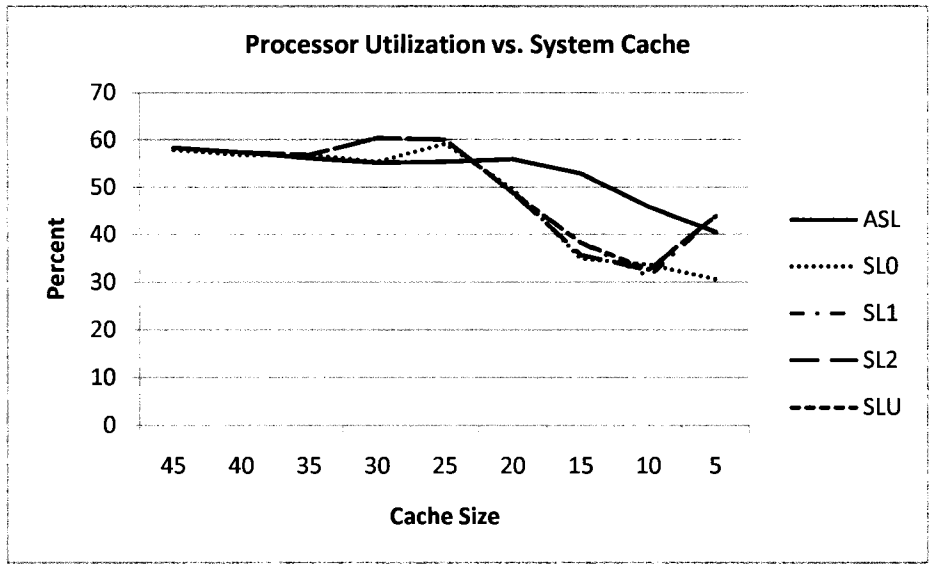

Figure 34: Experiment 2 - Configuration A - PPU for baseline with TQM

5.4.2. Configuration B

In Configuration B, the PTCT of a system that is heavily loaded with TQM disabled is presented in Figure 35. Under these load conditions and the assumption that there is infinite cache, the ASL protocol does not perform any better than that of the SL variants. Once again, this results in a high level of data contention, which causes a high utilization of swap disk as shown in Figure 36. Since pages are being swapped and not processed, the amount of page processing declines resulting in a decline in processor and disk utilization and therefore resulting in poor performance. Configuration B with TQM disabled responds no differently than that of Configuration A with TQM disabled. The main difference is the heavy system load, which further increases data contention, causing more thrashing, and less processor and disk utilization. The discussion of the performance implications of light and heavy system loads will be discussed in Experiment 3 and 4.

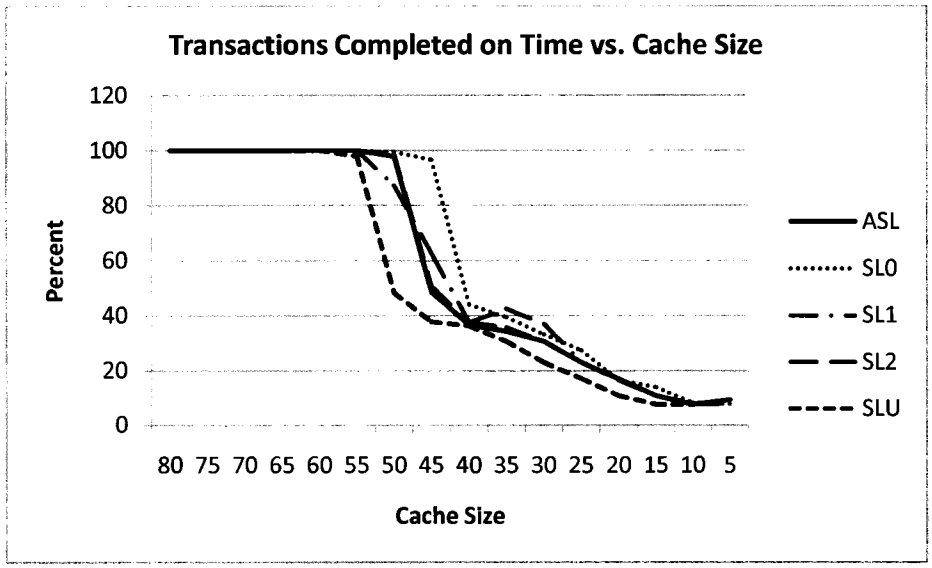

Figure 35: Experiment 2 - Configuration B - PTCT for heavy load with TQM disabled

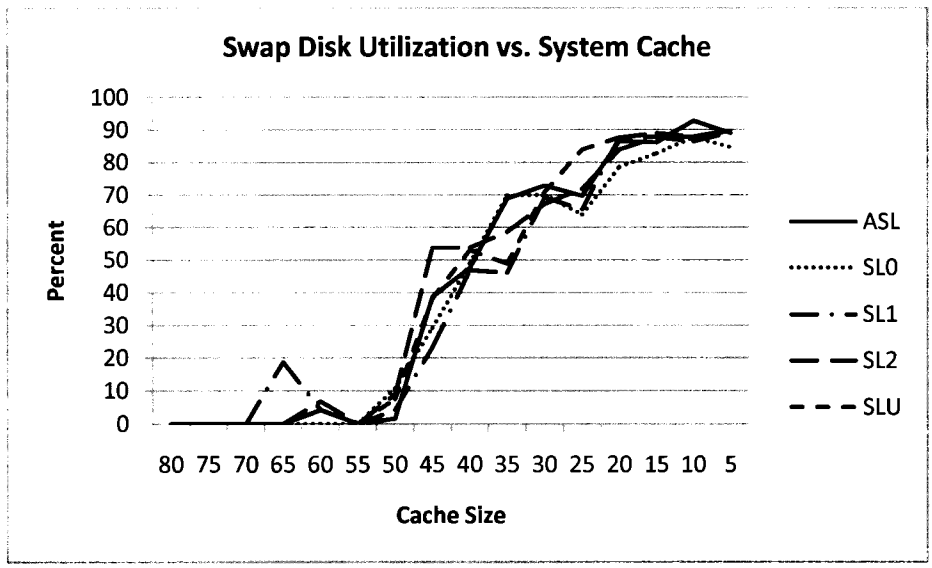

Figure 36: Experiment 2 - Configuration B - PSDU for heavy load with TQM disabled

We now enable a hold and enter level of 150 percent as we did in the Configuration B. Figure 37 shows the PTCT of a heavy loaded system with TQM enabled at 150 percent. Immediately, we can see a performance improvement in the ASL protocol over the SL protocols. The increase in system load causes the SL protocols to experience significant

data contention, whereas the ASL protocol has data contention, but through the use of the TQM, it is mitigated.

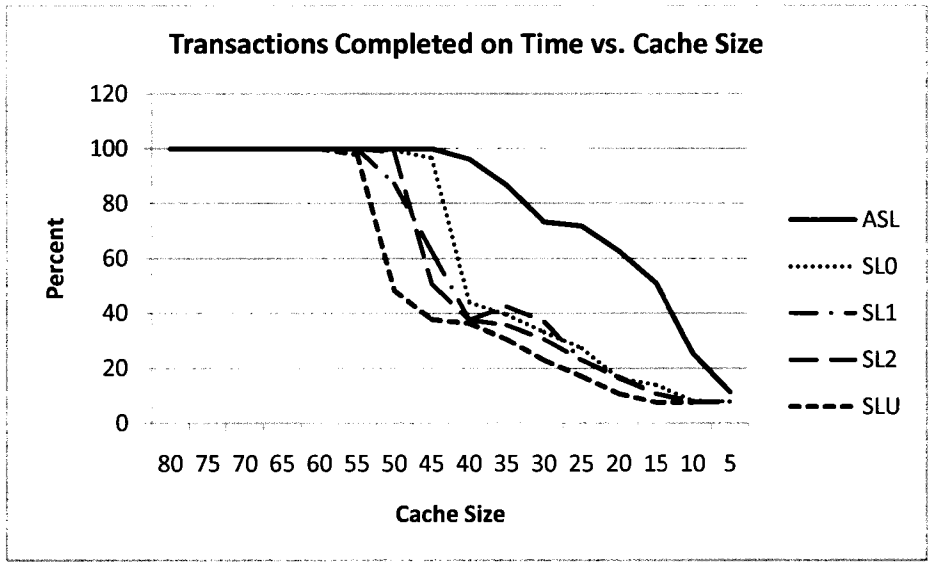

Figure 37: Experiment 2 - Configuration B - PTCT for heavy load with TQM enabled

Looking at the PTCT and comparing it with that of the PSDU (Figure 38), we can see that the amount of ASL swap disk utilization is substantially less than that of the SL protocols and at the point in which the ASL and SL protocols begin to degrade in performance, the swap disk utilization increases rapidly. This confirms the previous observations of the link between data contention and the PTCT. The same observation applies to the processor and the disk, as the data contention increases, the processor and disk utilization decreases.

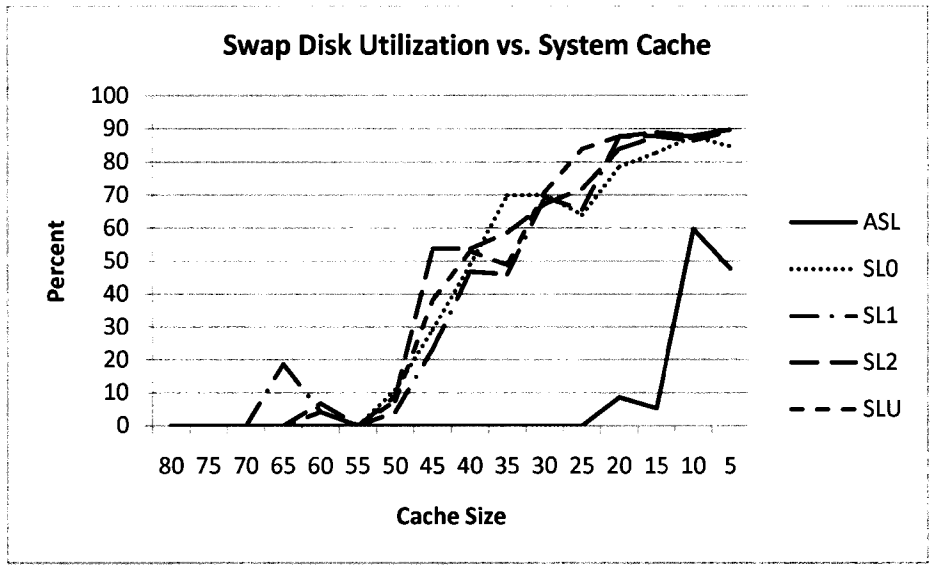

Figure 38: Experiment 2 - Configuration B - PSDU for heavy load with TQM enabled

5.4.3. Hold and Enter

Throughout all of the experiments in Chapter 4, we have set the TQM hold and enter values to both be 150 percent. This was done in order to provide a certain level of consistency throughout the experiments and to show the impact of the changes that were made to individual workload or system parameters. In this experiment, we take the same system configuration that was used in Experiment 2 (Configuration B) and change the hold and enter values to be between 75 percent and 175 percent to determine the effect on the PTCT. In Figure 39 we can see that setting the hold and enter values to 125H-125E and 175H-175E, we get comparable performance results to that of 150H-150E. Further, setting the values to 150H-100E, we get identical values to that of 150H-150E. From these runs, we can see that setting the hold and enter values above 120 percent results in comparable PTCT performance.

We can also observe in Figure 39, that if the hold and enter values are 100H and 75E, we get a degradation in system performance. The reason for this is that the system cache plus the incoming transactions cannot exceed the system cache size, and when the hold back is enabled, the transactions cannot be released until the system cache is 75 percent full. This causes greater data contention and results in a decrease in PTCT performance. Therefore, throughout our experiments, we consistently set the hold and enter values at 150 percent, which results in a more stable performance.

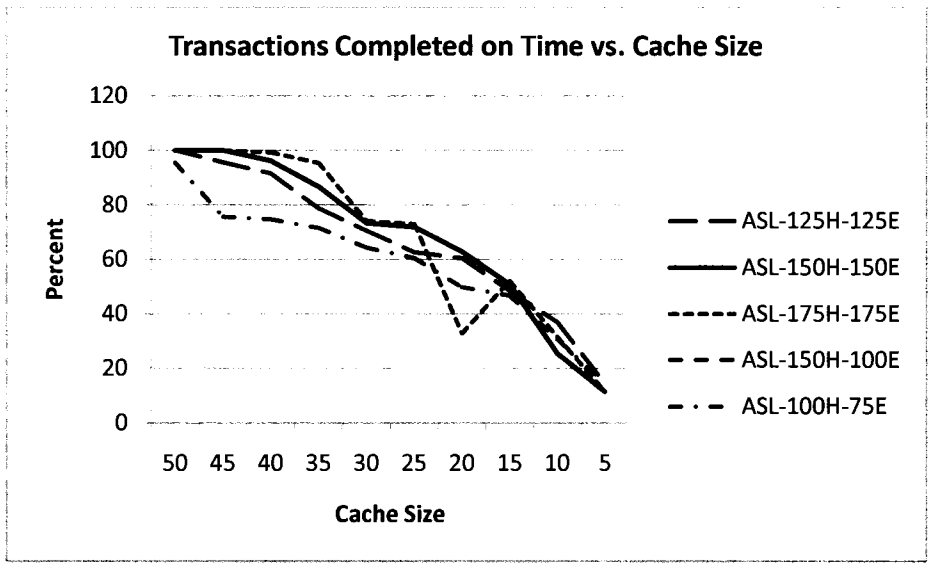

Figure 39: Experiment 2 - Hold and Enter Values: 125H-125E; 150H-150E; 175H-175E; 150H-100E; 100H-75E

5.5. Experiment 3: Arrival Rates

In this experiment, we look at the impact that arrival rates have on the performance and behaviour of the ASL and SL protocols. In the baseline experiment, we observed that the PTCT for a system with 100 transactions and an arrival rate of 50 yielded 100 percent PTCT for system cache sizes greater than 20. In experiment 2, configuration B, we saw a decline in performance of a system with 100 transactions and an arrival rate of 20. The goal of this experiment is to change the arrival rates and observe the impact on the PTCT performance and resource utilization.

We will use a four node system with four disks containing 100 pages each for a total of 1600 pages, one processor with hyper-threading disabled, and the ASL TQM enabled at 150 percent hold and enter. For this experiment the *SimTransSize* is set at 200 and the arrival rate will be varied between 15 and 50. Figure 40 shows the PTCT of a simulation with the ArrivalRate set at 50. We will use this PTCT as the starting point for the comparisons in this experiment.

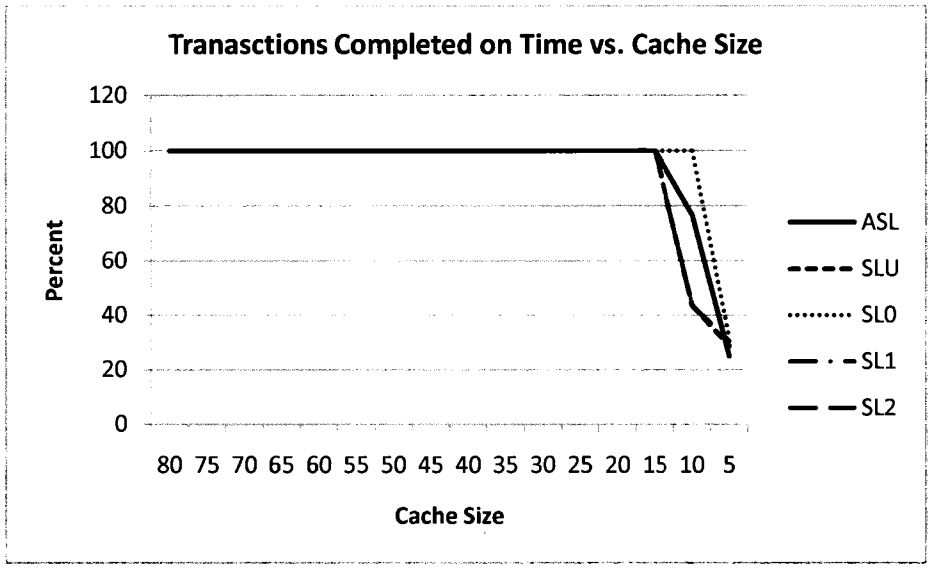

Figure 40: Experiment 3 - PTCT of SimTransSize 200 and ArrivalRate 50

As in the baseline experiment, we can see from Figure 40 that the performance of the ASL protocol is comparable to that of the SL protocols due to the low system, resulting in minimal data contention. Next, we decrease the ArrivalRate value to 30, which in turn increases the frequency in which transactions are arriving. Figure 41 shows the PTCT performance as a result of increasing the arrival rate of transactions. In increasing the arrival rate of transactions by 20, the performance of ASL is only slightly affected. However, the performance of the SL protocols does suffer from the increase in the arrival rate and is caused by data contention as it was in previous experiments.

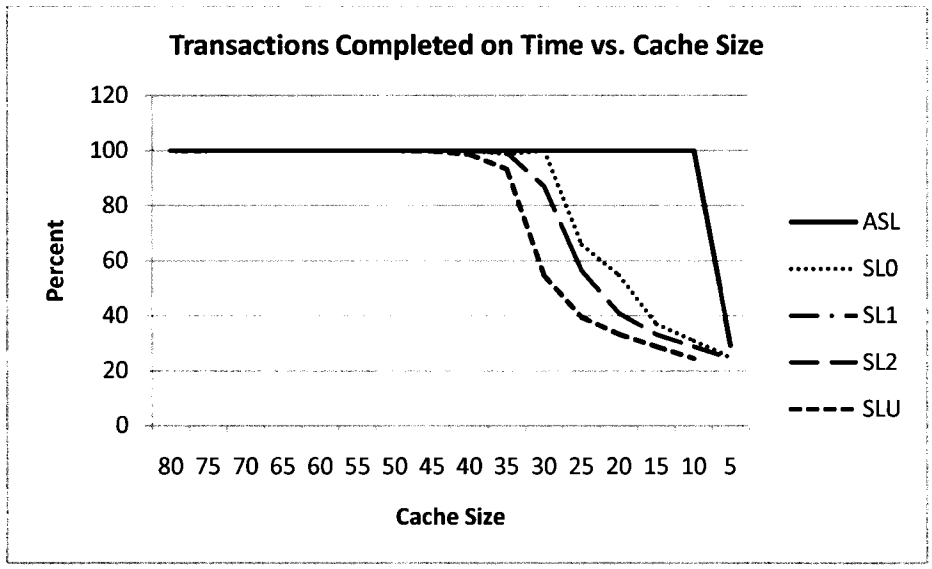

Figure 41: Experiment 3 - PTCT of SimTransSize 200 and ArrivalRate 30

Next we further decrease the ArrivalRate value to 20, which will once again increase the frequency of transactions arriving, and we can see from Figure 42 that the performance of the ASL and SL protocols has once again degraded. This time, the ASL protocol was more adversely affected by the increase in the arrival rate. Looking at the PSDU in Figure 43, we can see that the swap utilization for both protocols has increased significantly from the baseline, confirming that there is data contention which is causing the decrease in performance.

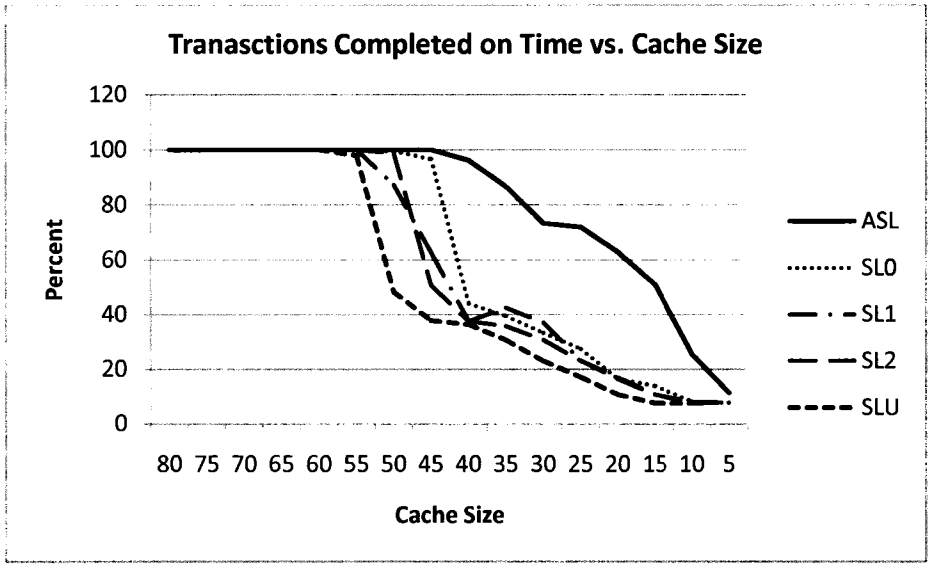

Figure 42: Experiment 3 – PTCT of SimTransSize 200 and ArrivalRate 20

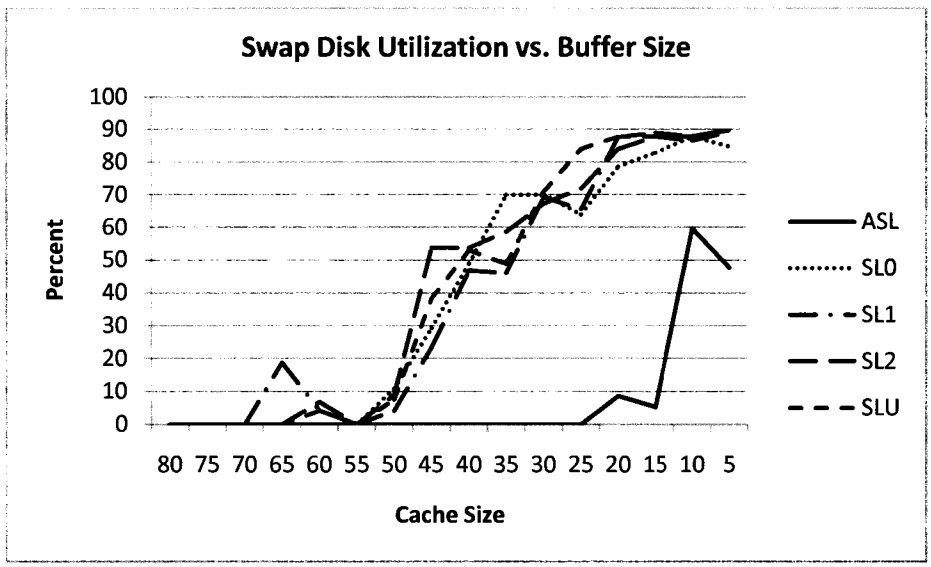

Figure 43: Experiment 3 - PSDU of SimTrans 200 Arrival Rate 20

In comparing the PTCT of the previous figures, we can confirm that if we keep the system and workload parameters constant and increase the arrival rate of transactions, the PTCT performance of the system decreases as expected due to data contention and notably the ASL protocol still performs better than the SL protocol.

5.6. Experiment 4: Transactions

In this experiment, we look at the impact that transaction loads have on the performance and behaviour of the ASL and SL protocols. In the baseline experiment, we observed that the PTCT for a system with 100 transactions and an arrival rate of 50 was 100 percent for system cache sizes greater than 20. In experiment 2, configuration B, we saw a decline in performance of a system with 200 transactions and an arrival rate of 20. The goal of this experiment is to maintain consistent workload and system parameters, but change the transaction loads to observe the impact on the PTCT and resource utilization.

We will use a four node system with four disks containing 100 pages each for a total of 1600 pages, one processor with hyper-threading disabled, and the ASL TQM enabled at 150 percent hold and enter. For this experiment the arrival rate is set at 20 and the SimTransSize will be varied between 100 and 400. Figure 44 shows the PTCT of a simulation with the SimTransSize set at 100.

We will use this PTCT as the starting point for the comparisons in this experiment. The PTCT performance is very similar to that of previous experiments with a light system load. Next we increase the SimTransSize to 200, and see from Figure 45, that both the ASL and SL protocol suffer from performance degradation. As the number of transactions increase, the PTCT performance decreased on average of about 20 percent for the ASL protocol and on average of about 40 percent for the SLU protocol. Next, we increase the simTransSize to 300 and see the results in Figure 46.

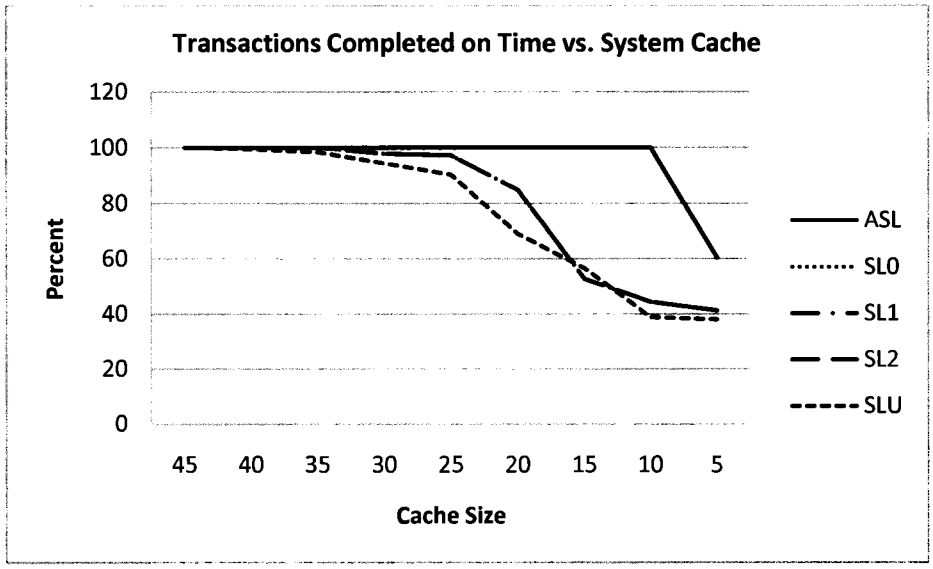

Figure 44: Experiment 4 - PTCT of SimTransSize at 100

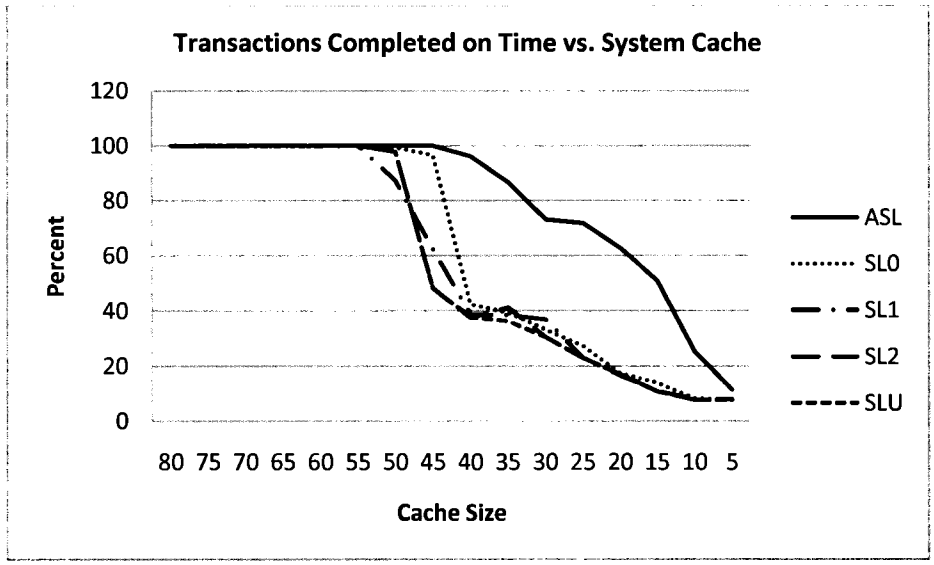

Figure 45: Experiment 4 - PTCT of SimTransSize at 200

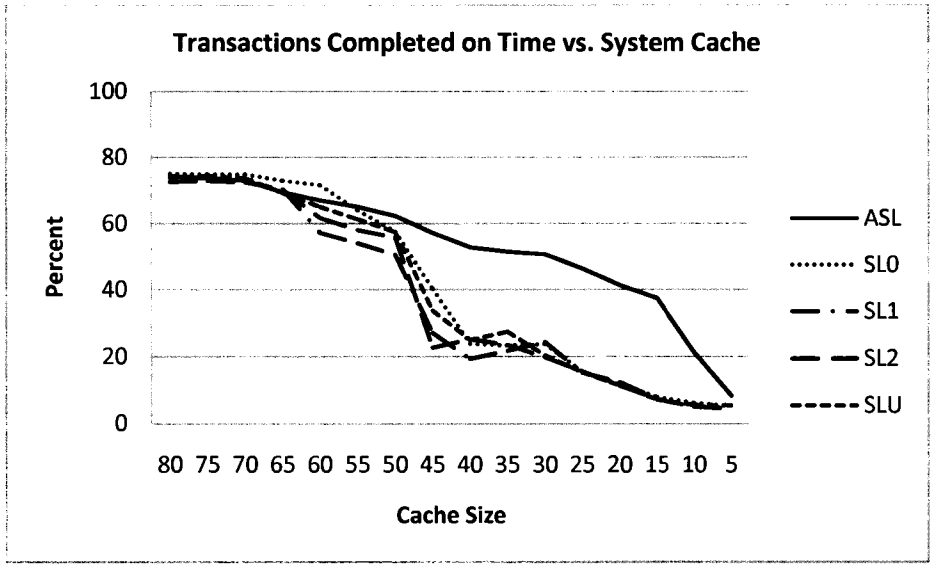

Figure 46: Experiment 4 - PTCT of SimTransSize 300

Comparing the PTCT of SimTransSize 300 to that of SimTransSize 200, we notice a substantial decline in the PTCT performance for both protocols. Based on the arrival rate of the transactions and the number of transactions in the system, we can see that both protocols degrade in performance. Once again this is due to data contention, which causes transactions to miss their deadlines. It is interesting to note that by increasing the SimTransRate by 100 that our performance for all protocols has decreased by roughly 35 percent more. It can also been seen that the gap between the ASL and SL protocols is decreasing. We now increase the SimTransRate to 400 and observe the results in Figure 47. The overall performance of all of the protocols has decreased and as in the previous figure, there are no transactions that are completing on time in the cache size range of 5 to 80.

We have observed from this experiment that as the number of transactions in the system increases, the PTCT performance decreases. At cache size 80, the percent complete has decreased from 100 percent with 100 transactions down to 62 percent with 400

transactions at an arrival rate of 20. In addition, PTCT performance with respect to the number of transaction is also heavily dependent on the arrival rate. An increase in the number of transactions and an increase in the arrival rate will result in a decrease in performance.

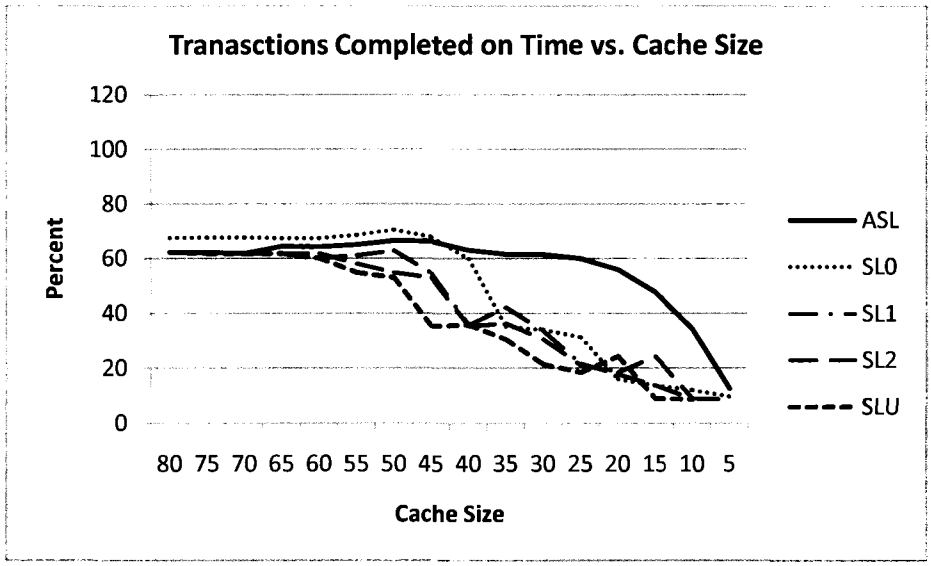

Figure 47: Experiment 4 - PTCT of SimTransSize 400

5.7. Experiment 5: System Resources

In this experiment, we look at the impact that system resources have on the performance and behaviour of the ASL and SL protocols. We look at two types of system resources: disk and processors. The goal of this experiment is to vary the number of disks and processors and observe the impact on the PTCT.

For this experiment, we use a four node system with ASL TQM enabled at 150 percent for both the hold and enter. The transaction *ArrivalRate* is 25 with a *SimTransSize* of 100. We begin the experiment with four disks containing 100 pages each for a total of 1600 pages and one processor without hyper-threading.

In Figure 48, we see the default configuration for this experiment with four disks and one processor. The PTCT is no different to what we have seen in previous experiments. Next we increase the number of disks to eight, for a total of 3200 pages, the result can been seen in Figure 49.

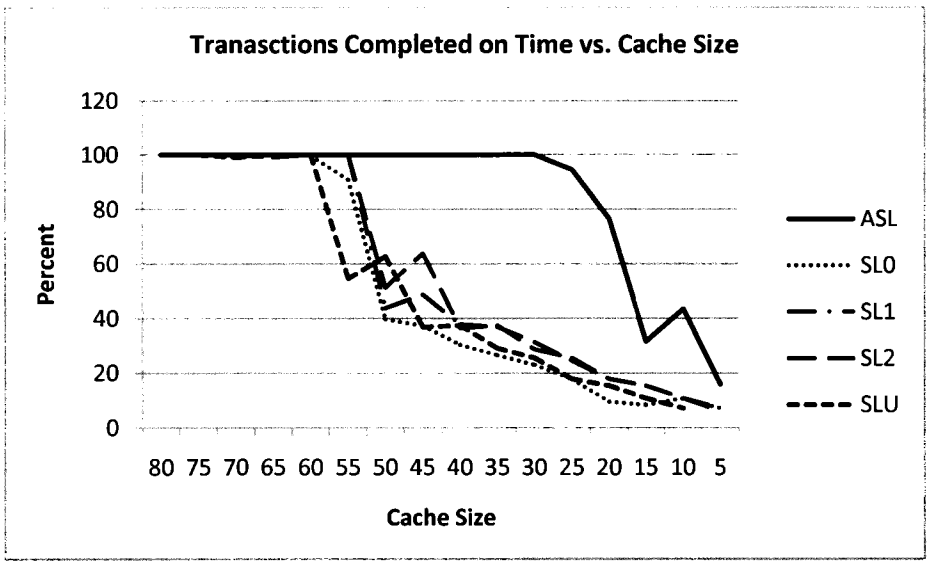

Figure 48: Experiment 5: PTCT of 4 disks and 1 processor

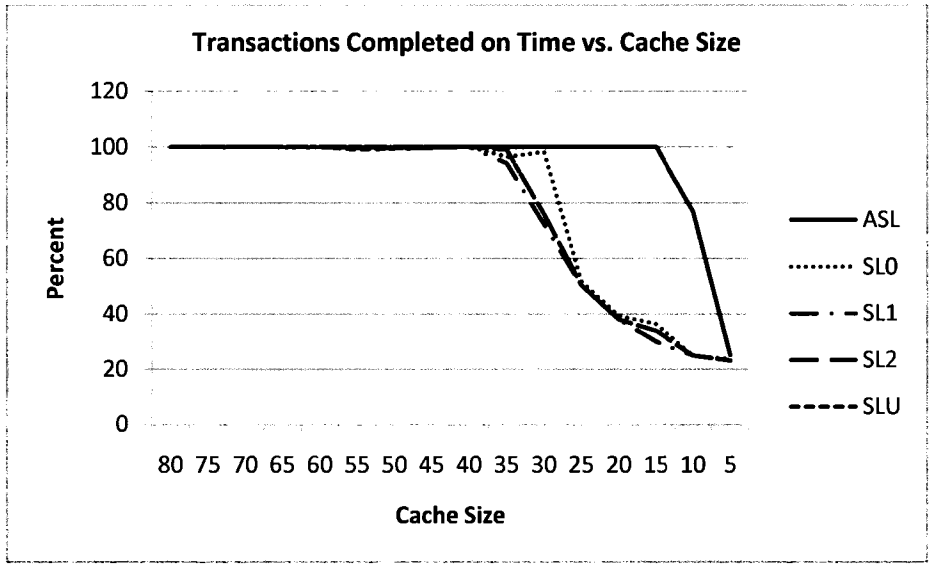

Figure 49: Experiment 5: PTCT of 8 disks and 1 Processor

The increase in the number of disks has improved the performance of the PTCT due to the distribution of pages. Instead of only having 1600 pages in the system over four disks, we now have 3200 pages over eight disks. With the small number of disks, there will be more contention for the disk resources as the 100 transactions will be distributed across the four disks of the four nodes. In case where there are 3200 pages, the 100 transactions are now more widely distributed. The increase in the number of disks has improved the PTCT performance as there is less contention for disk. The overall PTCT performance will still be affected by the total number of transactions and the arrival rate.

Next, we take the original configuration, add a processor per node, and observe the PTCT in Figure 50. The increase in the number of processors does not provide us a significant increase in performance; in some cases it causes a decrease. The reason for this is that the system processor is not being fully utilized and therefore adding a second processor provides no real benefit to the simulation.

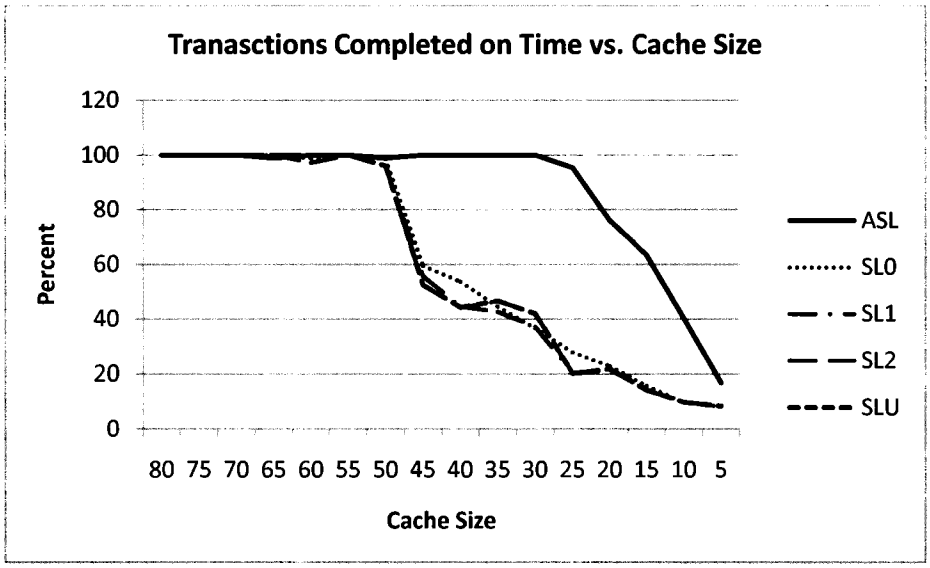

Figure 50: Experiment 5 - PTCT of 4 disks 2 processors

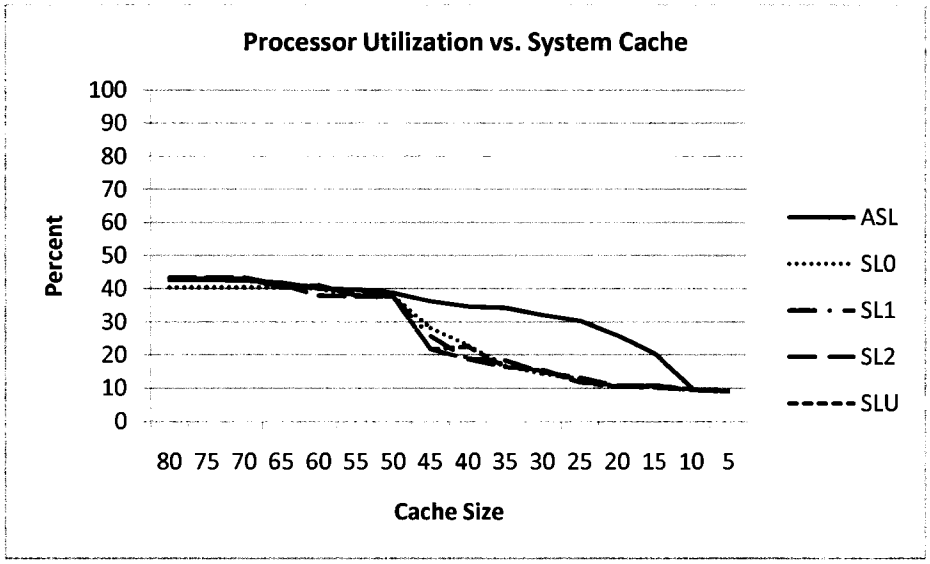

Figure 51: Experiment 5 - PDU of 4 disks 2 processors

5.8. Experiment 6: Page Updates

In the previous experiments, we have assumed a worst case scenario in that all database pages that are being accessed by all transactions are also updated. As transactions are created, the workload generator assigns an update percentage to the pages of a transaction. The update percentage ranges from 0 to 100 percent, zero being no writes and one hundred being 100 percent writes. The goal of this experiment is to analyze the performance of the protocols when the percentage of page updates is reduced from 100 percent.

For this experiment, we use a four node system with four disks containing 100 pages each for a total of 1600 pages, one processor with hyper-threading disabled, and the ASL TQM enabled at 150 percent for both the hold and enter. The transaction *ArrivalRate* is 25 with a *SimTransSize* of 400. We begin the experiment with an update probability of 100 percent and decrease it to 50 percent to see the effect on the PTCT.

In order for a page to be written to disk, it must first exist in cache. In the simplest case, if the pages of a transaction are loaded into cache, processed by the processor, and then are written to disk directly from cache, then the only impact on performance is the read and write of the page to disk. However, in the situation where a transaction has pages in cache and pages in swap, or even all of its pages in swap, performance will be negatively impacted if the page needs to be written to disk. A page that has been processed and swapped out to disk must first be read back into cache prior to the page being written to disk. This process impacts the time that it takes for transactions to complete, especially if transactions are waiting for other transactions to complete first.

Figure 52 shows the PTCT of a medium loaded system with 100 percent updates. We can see from the PTCT that the percentage of transactions that complete on time slowly declines as the cache size decreases. As demonstrated in previous experiments, the reduction in performance is once again caused by data contention and thrashing, which results in transactions waiting.

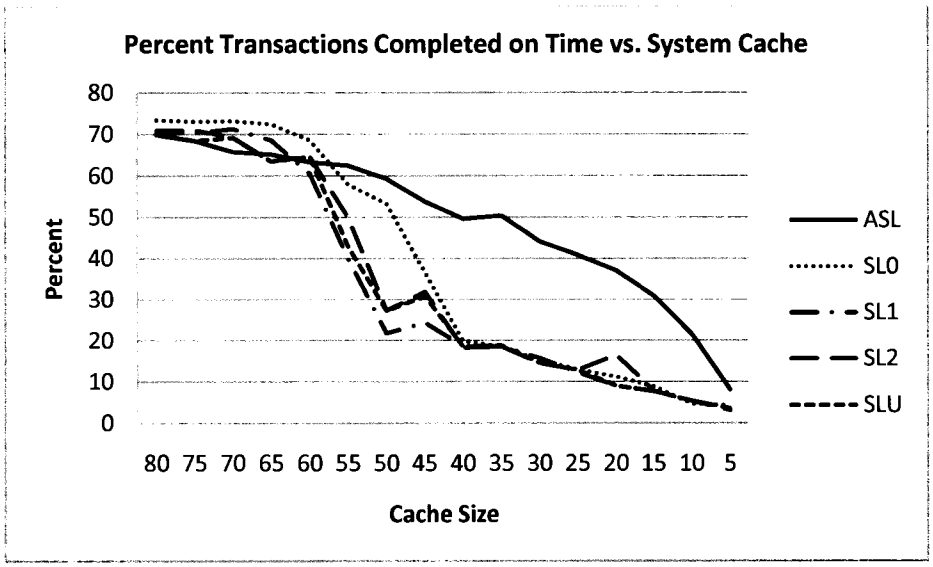

Figure 52: Experiment 6 - PTCT of 100% page updates

Now we will reduce the percentage of page updates to 50 percent and observe the results in Figure 53. Comparing Figure 52 to that of Figure 53, we can see that there is an improvement in the percent of completed transactions for the entire range of the system cache. The PTCT graph has shifted to the right, resulting in the cache size range of 50 or higher to perform at 100 percent and the range of 5 to 50 percent to significantly increase from that of Figure 52.

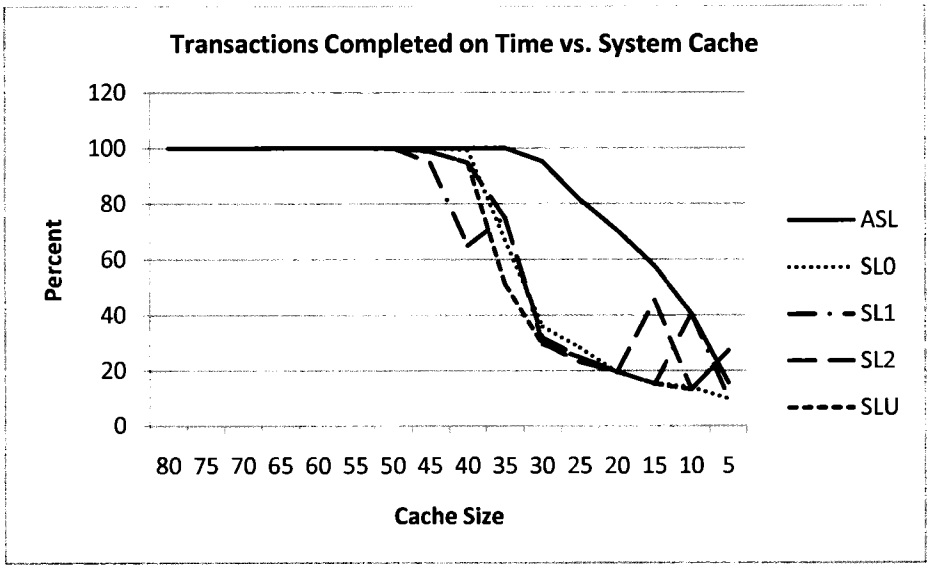

Figure 53: Experiment 6 - PTCT of 50% page updates

Reducing the number of updates reduced the data contention in the cache. Figure 54 and Figure 55 show the comparison of swap disk utilization for both the 100 percent and 50 percent page updates.

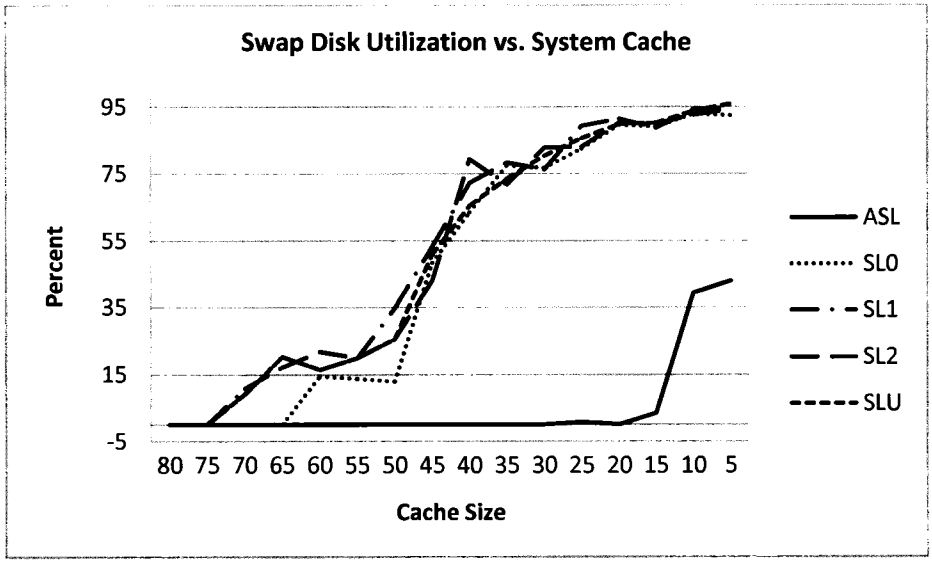

Figure 54: Experiment 5 - PSDU of 100% page updates

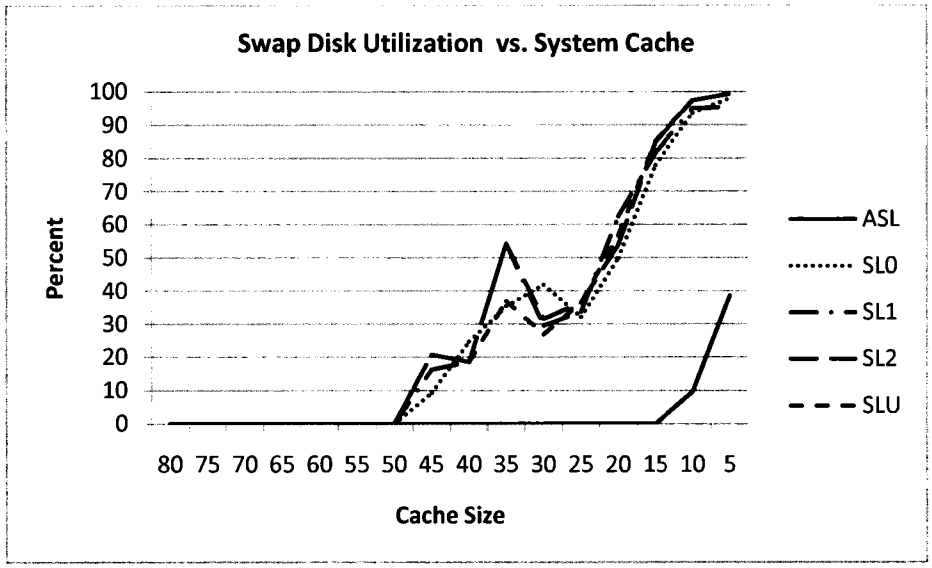

Figure 55: Experiment 6 - PSDU of 50% page updates

We can see that the swap disk utilization for both protocols declines significantly when the number of page updates decreases. The impact in performance and the utilization of swap confirm the fact that if pages are updated, they must be in memory prior to being written to disk. If there are 100 percent updates, the amount of cache required to hold all of the updated pages grow significantly, which results in the use of swap disk. Lastly, we note that the significant difference in performance of the ASL protocol to that of the SL protocols is due to ASL's TQM being enabled.

5.9. Experiment 7: Hyper-Threading

The goal of the next experiment is to demonstrate the effective use of hyper-threading with the ASL protocol. The SL protocol does not provide any support for the use of hyper-threading; therefore it is not analyzed in this experiment. This scenario is important for looking at system resource utilization and trying to leverage existing hardware technologies available to us to gain an improvement in performance. In order for hyper-threading to be effective, the system processor needs to be either under significant load or transaction pages have to be scheduled to process at the same time. In many of the previous experiments, the percent of processor utilization ranged from 40 percent to 90 percent and the number of pages scheduled to process at the same time was limited due to the light system load. Our approach to proving an increase in performance is to first observe a heavily loaded system in which hyper-threading is disabled and then demonstrate an improvement in performance with the same system when hyper-threading is enabled.

For this experiment, the configuration consists of a four node system with four disks containing 100 pages each for a total of 1600 pages, TQM enabled at 150 percent for both hold and enter, and one processor with hyper-threading disabled. The transaction *ArrivalRate* is 15 with a *SimTransSize* of 400 and an update probability of 100 percent.

In Figure 56, we can see that under heavy load, the ASL protocol does not perform well, and shows the PTCT result being less than 60 percent. As in previous experiments, the overall PTCT performance degrades under the heavy load due to data contention, resulting in transaction timeouts.

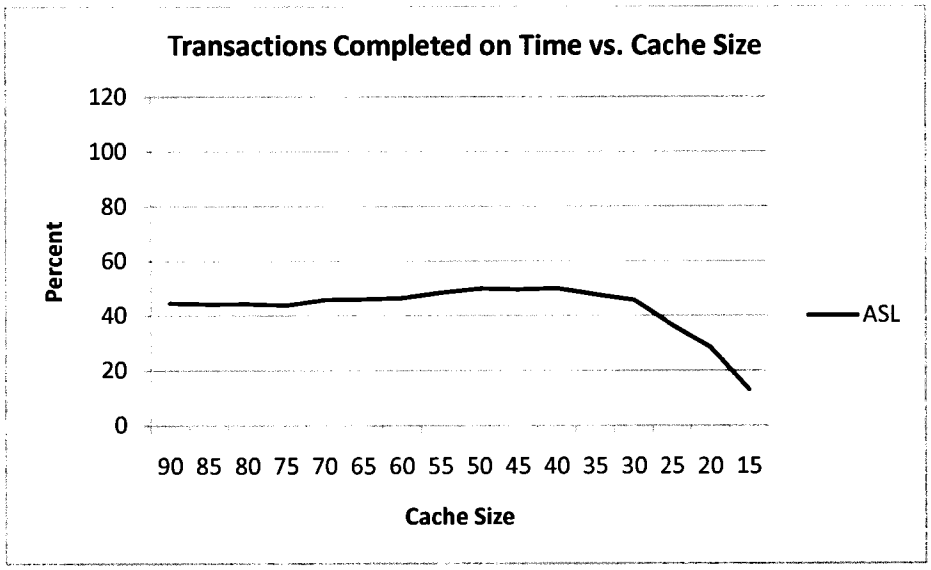

Figure 56: Experiment 7 - Heavy loaded system without hyper-threading

When we enable hyper-threading, as in Figure 57, we see that there is a slight improvement in PTCT for the cache size range of 40-90. This shows that hyper-threading can improve the performance of transaction execution.

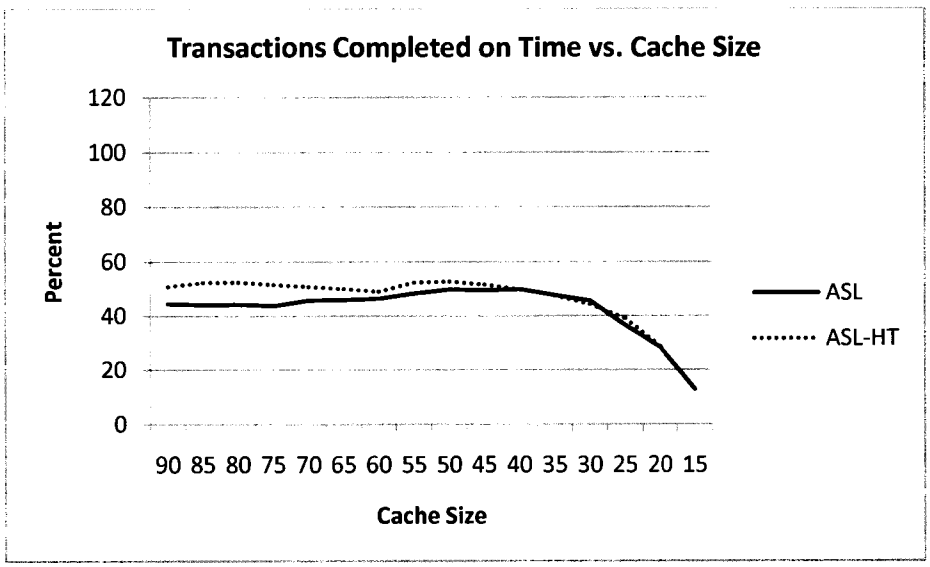

Figure 57: Experiment 7: Heavy loaded system with hyper-threading

However, under heavy system loads data contention increases and results in fewer pages being processed by the processor. As data contention increases, the number of transaction timeouts increases, which results in transactions missing their deadlines. Transaction timeouts significantly impact the transaction throughput, resulting in an under utilization of system resources.

As discussed earlier for hyper-threading to be effective, transaction pages need to be simultaneously processed on the processor. In analyzing this problem further through the running many simulations with varying workloads and system parameters, we discovered that in very few cases was hyper-threading providing us an increase in performance, due to the nature of the protocol. In the cases where there was an improvement, it was negligible.

Chapter 6

Conclusion & Future Direction

Distributed real-time database systems are not all that common in today's world. The majority of the database systems that exist in industry are real-time centralized or replicated database systems. There has been extensive research into RTDBS, but not in distributed RTDBS. The Speculative Locking protocol [10] provides a good approach to solving concurrency control issues in distributed RTDB systems. However they have made some assumptions about their model that would make it difficult to determine how their protocol would behave in real-world scenarios. We have addressed these limitations in this thesis.

Using our DRTTPS system, we have performed the analysis of an extensive set of experiments and simulations using a varying range of workload and system parameters. As our ASL protocol enhances the SL protocol, our performance comparisons are based on the analysis of these protocols. Our experiments demonstrated the following:

1. The ASL protocol outperforms the SL protocols in all experiments in systems with high data contention.
2. Restricting the system cache and implementing virtual memory had a significant performance impact on all of the protocols. Higher system loads caused a larger amount of data contention, which resulted in thrashing.

3. The ASL Transaction Queue Management had the greatest impact on the performance of ASL. TQM prevented the use of swap by mitigating cache contention, resulting in a large increase in performance.
4. Transaction load levels and arrival rates dramatically affected the performance of both the ASL and SL protocols. In most cases, both protocols reacted equally to the increase in load with ASL consistently outperforming the SL protocols. In addition, in several situations, the increased load caused the SL protocols to degrade at a faster rate than the ASL protocol.
5. An increase in system resources such as processors and disk had unexpected results on system performance. An increase in processors provided no additional performance improvement as the processors were not being heavily utilized. Secondly, the increase in disks distributed the transaction load across more disks, which reduce data contention on the disk. The amount of reduction in data contention was dependant on the other workload and system parameters.
6. Updates to database pages require that pages be in cache prior to being written to disk. If there were 100 percent page updates in the system, all protocols were susceptible to data contention. In the case where page updates was less than 100 percent, the overall performance improved for all protocols. However, in both cases, due to the ASL TQM, the ASL protocol performed better than the SL protocol as it mitigated data contention.
7. The effect of hyper-threading had a negligible effect on the performance of the ASL protocol as there was only a small amount of simultaneous processing of pages on

the processor. This lack of processing was due to the amount of data contention in the system under heavy loads.

6.1. Future Work

There are two directions that are of interest for future work in distributed RTDBS. The first would be to take the ASL protocol and implement it in a prototype system and see how it responds to real-world workloads. In using our simulation model, we discovered that using a fully encapsulated object oriented model requires large host system resources when running simulations. If the protocol was implemented in a prototype system, those limitations would be avoided and the true performance would be seen.

The second direction would be taking DRTTPS and evolving it from a simulation system into an actual real-time distributed database system. Currently, few commercial distributed RTDB systems exist and most of the research to date has evolved around simulation systems that were built by researchers. Taking this concept to the next level could open up many avenues for not only researchers, but for industry.

Bibliography

[1] **Buchmann, A.** Real Time Database Systems. *Encyclopedia of Database Technologies and Applications.* 2005.

[2] **Eswaran, Kapali P., et al.** *The Notions of Consistency and Predicate Locks in a Database System.* 11, 1976, Communications of the ACM, Vol. 19, pp. 624-633.

[3] **Sha, Lui, Rajkumar, Ragunathan and Lehoczky, John P.** *Priority Inheritance Protocols: An approach to Real-Time Synchronization.* Carnegie Mellon University, 1987. pp. CMU-CS-87-181.

[4] **Lam, Kwok-wa, Son, Sang H. and Hung, Sheung-lun.** *A Priority Ceiling Protocol with Dynamic Adjustment of Serialization Order.* 1997. 13th International Conference on Data Engineering. pp. 552-563.

[5] **Obermacrk, R.** *Distributed Deadlock Detection Algorithm.* 1982, ACM Transactions on Database Systems, Vol. 7, pp. 187-208.

[6] **Yeung, Chim-fu and Hung, Sheung-lun.** *A new deadlock detection algorithms for distributed real-time database systems.* 1995. 14TH Symposium on Reliable Distributed Systems. pp. 146-153.

[7] **Sha, L., et al.** *A Real-Time Locking Protocol.* 1991, IEEE Transactions on Computers, Vol. 10.

[8] **Kuo, T.-W., Kao, Y.-T. and Shu, L.C.** *A Two-Version Approach for Real-Time Concurrency Control and Recovery.* 1998. Third IEEE International High-Assurance Systems Engineering Symposium. pp. 279-287.

[9] **Bernstein, Philip A. and Goodman, Nathan.** *An algorithm for concurrency control and recovery in replicated distributed databases.* 1984, ACM Transactions on Database Systems, Vol. 8, pp. 596-615.

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

[11] **Gray, J.** Notes On Database Operating Systems. *Operating Systems: An Advanced Course.* London : Springer-Verlag, 1979.

[12] **Mohan, C., Lindsay, B and Obermarck, R.** *Transaction Management in the R* Distributed Database Management System.* 1986, ACM Transactions on Database Systems, Vol. 11.

[13] **Lampson, B. and D.Lomet.** *A New Presumed Commit Optimization for Two-Phase Commit.* 1993. Proc. of the 19th Conference of Very Large Databases.

[14] **Skeen, D.** *Nonblocking Commit Protocols.* 1981. Proc. of ACM SIGMOD Intl. Conf. On Management of Data.

[15] **Haritsa, Jayant R. and Ramamritham, Krithi.** *Adding PEP to Real-Time Distributed Commit Processing.* 2000. 21st IEEE Real-Time Systems Symposium. pp. 37-46.

[16] **Haritsa, Jayant R. and Ramamritham, Krithi.** *Real-Time Commit Processing, Real-Time Database Systems: Architecture and Techniques.* s.l. : Kluwer Academic Publishers, 2001.

[17] **Xiong, Ming, et al.** *MIRROR: A State-Conscious Concurrency Control Protocol for Replicated Real-Time Databases.* 1999. International Workshop on Advance Issues of E-Commerce and Web-Based Information Systems. pp. 10-29.

[18] **Haritsa, J. R., Ramamritham, K. and Gupta, R.** *The PROMPT Real-Time Commit Protocol.* 2000, IEEE Transactions on Parallel and Distributed Systems, Vol. 11, pp. 160-183.

[19] **Carey, M.J. and Livny, M.** *Conflict detection tradeoffs for replicated data.* 4, 1991, ACM Transactions on Database Systems, Vol. 16, pp. 703-746.

[20] **Tai, Ann T. and Meyer, John F.** *Performability Management in Distributed Database Systems: An Adaptive Concurrency Control Protocol.* 1996. 4th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems. pp. 212-216.

[21] **Chum, W.C. and J.Hellerstein.** *The Exclusive-writer approach to updated replicated files in distributed systems.* 1985, IEEE Trans. Computers, pp. 489-500.

[22] **Marr, T., et al.** *Hyper-threading technology architecture and microarchitecture.* 2002, Intel Technology Journal, Vol. 6, pp. 4-15.

[23] **Hassanein, Wessam M, Hammad, Moustafa A. and Rashid, Layali.** *Characterizing the Performance of Data Management Systems on Hyper-Threaded Architectures.* s.l. : IEEE Computer Society , 2006. Proceedings of the 18th International Symposium

on Computer Architecture and High Performance Computing (SBAC-PAD'06). Vol. 00, pp. 99 - 106 . ISSN:1550-6533 .

[24] **Bernstein, Philip A., Goodman, Nathan and Hadzilacos, Vassos.** *Concurrency Control and Recovery in Database Systems.* s.l. : Addison Wesley, 1987.

[25] **Ball, Peter.** *Introduction to Discrete Event Simulation.* Algrave : s.n., 1996. 2nd DYCOMANS workshop on "Management and Control : Tools in Action. pp. 367-376.

[26] **Microsystems, Sun.** *Java Language Specification v1.4.2".* Palo Alto California : Sun Microsystems, 2003.

[27] **Beeri, C., et al.** *A theory for nested transactions.* 1983. Proc. of the second annual ACM symposium on Principles of distributed computing. pp. 45-62.

[28] **Vidyasankar, K.** *A Non-Two-Phase Locking Protocol for Global.* 1991, IEEE Transactions on Knowledge and Engineering, Vol. 3, pp. 256-261.

[29] **Lee, J. and Son, S.** *Concurrency Control Algorithms for Real-Time Database Systems.* 1994, PhD Dissertation.

[30] **Lam, Kam-Yiu, et al.** *Impact of priority assignment on optimistic concurrency control in distributed real-time databases.* 1996. Third International Workshop on Real-Time Computing Systems Application. pp. 128-135.

[31] **Saha, Prabodh.**One-Phase Real-Time. *Masters Dissertation.* 1999.

[32] **Traiger, Irving L., et al.** *Transactions and consistency in distributed database systems.* 1982, ACM Transactions on Database Systems, Vol. 7, pp. 323-342.