Optimizing Distributed Transactions: Speculative Client Execution, Certified Serializability, and High Performance Run-Time

Utkarsh Pandey

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MS
in
Computer Engineering

Binoy Ravindran, Chair
Roberto Palmieri
Chao Wang

August 3, 2016
Blacksburg, Virginia

Keywords: Distributed systems, Transactions, Databases, Verification, Run-time, Concurrency, Paxos, Replication.

Optimizing Distributed Transactions: Speculative Client Execution, Certified Serializability, and High Performance Run-Time

Utkarsh Pandey

(ABSTRACT)

On-line services already form an important part of modern life with an immense potential for growth. Most of these services are supported by transactional systems, which are backed by database management systems (DBMS) in many cases. Many on-line services use replication to ensure high-availability, fault tolerance and scalability. Replicated systems typically consist of different nodes running the service co-ordinated by a distributed algorithm which aims to drive all the nodes along the same sequence of states by providing a total order to their operations. Thus optimization of both local DBMS operations through concurrency control and the distributed algorithm driving replicated services can lead to enhancing the performance of the on-line services.

Deferred Update Replication (DUR) is a well-known approach to design scalable replicated systems. In this method, the database is fully replicated on each distributed node. User threads perform transactions locally and optimistically before a total order is reached. DUR based systems find their best usage when remote transactions rarely conflict. Even in such scenarios, transactions may abort due to local contention on nodes. A generally adopted method to alleviate the local contention is to invoke a local certification phase to check if a transaction conflicts with other local transactions already completed. If so, the given transaction is aborted locally without burdening the ordering layer. However, this approach still results in many local aborts which significantly degrades the performance.

The first main contribution of this thesis is PXDUR, a DUR based transactional system, which enhances the performance of DUR based systems by alleviating local contention and increasing the transaction commit rate. PXDUR alleviates local contention by allowing speculative forwarding of shared objects from locally committed transactions awaiting total order to running transactions. PXDUR allows transactions running in parallel to use speculative forwarding, thereby enabling the system to utilize the highly parallel multi-core platforms. PXDUR also enhances the performance by optimizing the transaction commit process. It allows the committing transactions to skip read-set validation when it is safe to do so. PXDUR achieves performance gains of an order of magnitude over closest competitors under favorable conditions.

Transactions also form an important part of centralized DBMS, which tend to support multi-threaded access to utilize the highly parallel hardware platforms. The applications can be wrapped in transactions which can then access the DBMS as per the rules of concurrency control. This allows users to develop applications that can run on DBMSs without worrying about synchronization. `Serializability` is the de-facto standard form of isolation required by transactions for many applications. The existing methods employed by DBMSs to enforce serializability employ explicit fine-grained locking. The eager-locking based approach is

pessimistic and can be too conservative for many applications.

The locking approach can severely limit the performance of DBMSs especially for scenarios with moderate to high contention. This leads to the second major contribution of this thesis is TSAsR, an adaptive transaction processing framework, which can be applied to DBMSs to improve performance. TSAsR allows the DBMS's internal synchronization to be more relaxed and enforces serializability through the processng of external meta-data in an optimistic manner. It does not require any changes in the application code and achieves orders of magnitude performance improvements for high and moderate contention cases.

The replicated transaction processing systems require a distributed algorithm to keep the system consistent by ensuring that each node executes the same sequence of deterministic commands. These algorithms generally employ `State Machine Replication (SMR)`. Enhancing the performance of such algorithms is a potential way to increase the performance of distributed systems. However, developing new SMR algorithms is limited in production settings because of the huge verification cost involved in proving their correctness.

There are frameworks that allow easy specification of SMR algorithms and subsequent verification. However, algorithms implemented in such framework, give poor performance. This leads to the third major contribution of this thesis Verified JPaxos, a JPaxos based runtime system which can be integrated to an easy to verify I/O automaton based on Multipaxos protocol. Multipaxos is specified in Higher Order Logic (HOL) for ease of verification which is used to generates executable code representing the Multipaxos state changes (I/O Automaton). The runtime drives the HOL generated code and interacts with the service and network to create a fully functional replicated Multipaxos system. The runtime inherits its design from JPaxos along with some optimizations. It achieves significant improvement over a state-of-art SMR verification framework while still being comparable to the performance of non-verified systems.

Dedication

I would like to dedicate this thesis to the following people, without whom my work would not be possible:

Dr. Binoy Ravindran, my advisor, for providing me with the opportunity to perform this research, and for his faith in my work. Dr. Chao Wang for graciously serving on my committee. Dr. Roberto Palmieri, for his great knowledge and guidance in my research, which kept me optimistic and focused towards my goals and also for serving on my committee. Dr Giuliano Losa, for his valuable guidance and support for an important part of my thesis work. Dr Sebastiano Peluso, for his valuable guidance in the research area.

All of the members in the Systems Software Research Group, for their friendship and assistance, including Balaji Arun, Sachin Hirve, Antony Karno, Ian Rossele, Masoomeh Javidi, Cindy Ann, Rob Lyerly, Christopher Jelesnianski, Ho-Ren Chuang, Yuzong Wen, Josh Bockenek, Xiaolong Wu, Dr Sandeep Hans, Dr Antonio Barbalace, Dr Pierre Olivier and others.

All of my family members, especially my parents Mrs Reeta Pandey and Mr J.K. Pandey for their love, support and encouragement to pursue education. My brothers Akshay Pandey and Raveesh Pandey for their presence. My grandparents for their blessings. My teacher Mrs Preeti Tripathi for her valuable presence and unwavering faith in me. Lastly all of my friends for their help, encouragement and support.

Acknowledgments

This thesis is partially supported by Air Force Office of Scientific Research (AFOSR) under grants FA9550-14-1-0187 and FA9550-15-1-0098.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The continuous innovations experienced in the last decade in the area of internet computing have led to the development of high performance and reliable on-line services, which already form an important part of modern life. On-line services make it possible for the end user to perform essential operations like banking or shopping through web-based interfaces. However, web applications only represent the front-end of the services while the user requests are actually processed by a transaction processing systems, which often encompasses a database management system (DBMS). With DBMS, user requests are submitted as transactions that, if successful, will modify the state of one or more shared data repository (or databases) accordingly. A `transaction` by itself, is considered a unit of work consisting of one or more operations, which are either completed as a whole or have no effect at all. DBMS represents the de-facto standard to provide data manipulating applications with transactional support.

`Replication` is a common method used to increase the availability of on-line services. This involves replication of the same service implementation over several nodes, which execute commands enforced by some distributed algorithms in order to reproduce the same final state in each of the involved node. This way, if some of the nodes becomes not available anymore, others can still serve user requests without introducing downtime. Replication makes these services fault tolerant too, as the service can tolerate the failure of nodes. For simplicity, such a replicated system can be referred to as a `distributed system`. The State Machine Approach(SMA) [48] is a well-known technique used for enforcing a common command sequence in distributed systems, which can also provide transaction semantics. We call transactions running on distributed systems are `distributed transactions`.

1.1 Motivation

Many on-line services in production, employ replication and utilize DBMSs to store relevant data. Therefore, improving the performance of DBMS access as well as the replication al-

gorithms, can lead to significant benefits in the overall performance of the service. With the advent of multi-core hardware platforms, the DBMSs support multi-threaded access to utilize the parallelism offered. Devising efficient concurrency control protocols which control the access to the DBMS data is an area of active research. This applies to DBMSs, both when used as standalone server or when deployed in a distributed environment. SMR based algorithms, used to co-ordinate different nodes in a replicated service are generally complex in nature. Subsequently, developing new SMR algorithms incurs a heavy verification cost. Thus, methods to build SMR algorithms which lend themselves easily to verification is also area of active research. However, such ease of verification should not result in compromise of performance. These are the factors which motivate research in `speculative forwarding` in distributed servers, using `serilizability certifier` in DBMSs to relax default concurrency control semantics and high performing `run-time` to deploy easy-to-verify SMR-based replicated system.

1.1.1 Deferred Update Replication (DUR)

Transactional systems widely use replication to ensure fault-tolerance and increase availability. When full-replication is adopted (i.e., all shared objects are replicated across all nodes), the correctness of transactions started on remote nodes is usually guaranteed through a protocol that depends on a reliable total order layer (e.g., Paxos [26]). This approach of designing distributed transactional systems is called State Machine Approach (SMA) [48], where the ordering protocol ensures that each server node executes the same sequence of commands. Such SMA based systems can be classified upon the time when the transactions are globally ordered. `Deferred Update Replication (DUR)` [44] represents a scenario, where the transactions are executed before the total order is reached. Here, a certification phase resolves remote conflicts after the total order is reached. On the other side is `Deferred Execution Replication (DER)` [18, 34], which requires the clients to postpone executing the transactions until a total order is reached. This way, the requests are broadcast to all the nodes and executed once a total order is reached.

DUR and DER based systems find their respective sweet spots under different execution scenarios [23]. DUR based systems execute transactions optimistically, therefore transactions can abort due to remote conflicts. Such system find their best usage scenario with respect to performance, under workloads where transactions running on different nodes hardly conflict. For such an environment, DUR scheme allows for high degree of parallelization due to application threads running locally on each node, thus increasing performance. DER based systems, order the transactions before execution. This makes them immune to remote conflicts. DER schemes allow for better performance and scalability for workloads with medium or high degree of contention among transactions running on different nodes. In this thesis we focus on DUR-based approach because we assume workloads with few remote aborts.

In DUR, each server runs its transactions (client requests) locally and optimistically. The

locally committed transactions on each server are provided a total order by an ordering protocol. The ordered transactions are then sent to each server, where they undergo a certification phase. It determines whether the read operations performed by the each transaction is consistent with respect to other concurrent transactions in the system.

DURs performance advantage over other well-known approaches, such as the Deferred Execution Replication approach [18, 34] where transactions are executed after the imposition of a total order, lies in the fact that, in the latter, the transaction execution is entirely done by all nodes. Instead, in the case of DUR, a transaction is executed only by one node and its updates are propagated to all other nodes. Thus, instead of executing the whole transaction, every node only needs to perform a validation and apply those updates. These tasks are less expensive than executing the transaction as a whole.

In DUR systems, transactions run locally without being aware of other transactions on remote nodes. As a result, there is a high possibility of remote conflicts, which can lead to significant aborts after total order is reached. S-DUR [49] introduces the idea of `partitioning` to address remote conflicts. In fully partitioned DUR systems, the data is replicated across all the servers. However transactions running on each server can access only a mutually exclusive subset of the data. Partitioning helps to scale DUR based systems' performance with increase in node-count.

Despite the use of the partitioned access pattern, the performance of DUR based systems is still limited due to two factors : local contention and the rate of post-order certification. Local contention is the contention faced by concurrent transactions running on the same node. This factor can limit the parallelism among local application threads due to conflicts. A generally adopted technique for preventing a local transactions from invoking a certification phase, if it conflicts with another local transaction that has already completed, is to validate the local transaction locally after completion. In case of a conflict, the transaction aborts without burdening the total order layer.

Conflict-aware load balancing introduced in [58], tries to alleviate local contention by assigning transactions to preferred servers. The idea is to serialize conflicting transactions by clubbing them together on the same server. Lock based synchronization techniques are used for local concurrency control. XDUR [1], introduces the idea of addressing local contention by letting active transactions speculative read the shared objects from locally committed transactions awaiting total order. In [44], the authors have described an optimization at commit time to reduce the number of aborts. However their approach works by re-ordering transactions during commit to reduce remote aborts, it does not target the functionality of the committer module itself.

Thus, there is further scope of enhancing the performance of DUR systems beyond partitioning. Local contention management and increasing the rate of post-ordering certification are two promising areas which demand further investigation.

1.1.2 Concurrency Control in Centralized DBMS

Database management systems (DBMSs) are another example of transaction processing systems. State-of-the-art DBMSs support multi-threaded access to exploit the parallelism offered by multi-core platforms. Concurrency control (CC) algorithms interleave read and write requests while giving the illusion that each transaction has exclusive access to the data. Concurrency Control ensures correct system behavior preventing anomalies which can arise in the presence of concurrent access. An anomaly can be defined as any non-desirable interleaving of operations that produces inconsistent results for an application. However, inconsistency depends entirely upon the creator's expectations for their application; for instance, some applications may allow their users to see stale data without any problems, while others require their users to see the most up-to-date information possible.

Serializability [3] can be considered as the gold standard consistency level, because it allows application programmers to focus entirely on developing their programs business logic, without needing to determine any anomalies that may occur from concurrent data usage.

A `schedule` is the actual sequence of concurrent transactions. A precedence graph or transaction dependency graph can be used to determine if a given schedule can be serialized or not. Such a graph has the transactions as its vertices and edges are defined by the dependencies between them (i.e through reading and writing shared objects). The presence of a cycle in such a graph means that the schedule is not serializable. Serializable concurrency control mechanisms aim to provide a cycle-free transaction dependency graph, which is a necessary condition for serializability. A qualitative evaluation of state-of-the-art production level DBMSs (e.g., [41, 53]), which provide strong concurrency control shows that the concurrency control methods can be too conservative given the highly parallel commodity hardware platforms currently available. These CC schemes tend to forbid all dependency cycles, but in doing so they also forbid many serializable schedules.

The ANSI/ISO SQL standards define isolation levels used in DBMS. These isolation levels are explored in [2] along-with the arising ambiguities. A new isolation level `Snapshot Isolation` is introduced, which is aimed at performance improvement at the cost of relaxed consistency as compared to serializability. The work presented in [11] is an attempt to make snapshot isolation serializable by utilizing additional meta-data processing. The technique used, tracks read-write anti-dependencies to prevent cycles in the transaction dependency graph. However, the approach is conservative and disallows many serializable schedules too. A similar idea is presented in [4]. PSSI (precise Serializable Snapshot Isolation) is introduced in [47], which presents a less conservative approach towards serializable snapshot isolation. It further refines the cycle dangerous scenarios identified in [11] . It reduces false positives but has greater overhead.

Many production DBMS systems utilize isolation levels more relaxed than serializability to enhance performance. However, running on these isolation levels can cause anomalies which can be prevented by using serializability. The use of additional meta-data along-

with more relaxed isolation level to guarantee serializability is explored in [51, 52], which presents a serializability certifier called Serial Safety Net (SSN). SSN can be added to relaxed isolation levels (e.g., Read Comitted (RC) , with some modifications) to guarantee overall serializability. SSN tracks anti-dependencies to prevent a cycle in the transaction dependency graph.

As Serializable Transactions (AsR) introduced in [38] present a similar approach to leverage more relaxed isolation levels with external meta-data processing, aimed at enhancing performance. AsR also provides the feature of tightening the DBMS's default isolation level if a transaction aborts repeatedly under a lower isolation level. The serializability certifier used by AsR is based upon visible reads. This version of AsR only gives significant performance gains for high contention cases. Thus, integrating a different, more lightweight serializability certifier in the AsR framework is a viable direction to enhance its performance.

1.1.3 Formalization of SMR algorithms

SMR algorithms are deployed by various on-line services and form the basic infrastructure supporting cloud-computing. Therefore verification and improvement of SMR algorithms can result in improving the performance and reliability for many such services. However, SMR algorithms have traditionally proved difficult to understand and subsequently verify. The complexity involved in formal verification techniques used to ensure the correctness of SMR algorithms, acts as a major hindrance to innovation in this area. Thus devising new SMR algorithms is very costly. The complexity of SMR algorithms makes it easy to overlook catastrophic bugs appearing not only in implementations but also in the high level algorithm. Moreover, concurrency, network behavior, and faults give rise to a number of possible interleavings of actions that is beyond the reach of analysis by testing method. Therefore, to ensure the correctness of an SMR algorithm, one needs to resort to formal methods like model-checking or interactive theorem proving. However, both of these are costly exercises. Model-checking often requires manually building an abstraction of the algorithm to simplify the task of the model checker, and interactive theorem proving is very time consuming. Both require experts trained in the particular tool used for the task.

Distributed algorithms are typically specified in English or pseudo code [14] using different computational models. Synchronous and asynchronous models are commonly used. Synchrony makes it easier to reason about the system, while asynchrony models the network behavior closely. Distributed algorithms are formalized using techniques like Π calculus [35, 36], CSP [20] and I/O automata [32]. These methods provide formal semantics to message passing systems, making it easier to reason about them from a theoretical view point, but not as programming languages. Specification languages like TLA+ [27], +Cal [28] are used to write the formal specification of distributed algorithms. The specification is used to prove the correctness by mechanical proving or through a model checker. However, the specifications are not executable.

Recently, there has been some work in the area of programming languages with verification support. Mace [22] is such a framework which has a built-in model-checker support for verification. It has been used to identify bugs in previously deployed systems. However its model checker cannot prove total functional correctness. In that respect, programming languages like EventML [46] and Verdi [54] are better suited models.

EventML [46] is a functional programming language which can be used to used to write the executable code for distributed systems. The correctness of the implemented programs can be proved mechanically using the Nuprl [7] theorem prover. Verdi [54] starts with a synchronous implementation and transforms it into an asynchronous fault-tolerant system using refinement. Verdi uses Coq framework to prove the correctness of the implementation. By verifying the executable implementations, these frameworks reduce the possibility of error further. However, the algorithms produced by these frameworks suffer from reduced performance.

PSYNC [9] is a domain specific language which can be used to automate verification. PSYNC models the distributed system as a sequence of rounds. It aims to reduce the implementation complexity of distributed algorithms, as well as provides support for automated verification. If a distributed algorithm can be modeled in PSYNC, it can be easily verified. However, it is not easy to model all classes of distributed algorithm in PSYNC. The algorithms modeled in PSYNC give better performance as compared to those modeled in functional language frameworks discussed previously. IronFleet [16] is another framework for creating distributed systems which lend themselves readily to verification. It can be used to prove both safety and liveness properties of distributed system implementations. Ironfleet has a modular structure and attempts to verify all the components of the system. The verification effort involved is also quite high.

An investigation of the verification frameworks reveals that the search for such a model which can be used to specify SMR algorithms, such that the implementation is easy to verify and gives good performance too, is far from over. There is much scope for research in this area.

The contributions made in this thesis explore the previously discussed aspects of transaction processing systems, aimed at improving the performance. `PXDUR` addresses two main factors, which limit the performance of even fully partitioned DUR based systems : local contention and rate of post total-order certification. It inherits the concept of speculative forwarding from XDUR [1], but improves upon the design further by allowing for speculation to occur in parallel. The commit phase is optimized to improve the performance further. PXDUR provides further improvement upon the existing idea of scaling DUR base systems by full partitioning of data [49].

`TSAsR` is built upon the idea of improving the performance of centralized DBMS by combining relatively relaxed default isolation levels with a lightweight serializability certifier. It is fundamentally different from other such protocols (e.g., Serial Safety Net [51, 52]), as it allows a transaction to upgrade its isolation level following repeated aborts. It inherits the idea of hierarchial isolation level upgrade from the existing AsR [38], but utilizes a

different serializability certification layer, one based on timestamps introduced in [51, 52]. It is different from systems described in [11], [4] and [47], since AsR uses isolation levels like Read Committed and Repeatable Read as compared to Snapshot Isolation used by the former.

`Verified JPaxos` is a run-time for formally verified Multipaxos system modeled in HOL [50] language. The run-time is based on the design of JPaxos [25]. It aims at providing a high performance replicated system, which can use formally verified code. It differs from a system like PSYNC [9] as unlike PSYNC we are not forced to model the distributed algorithm in a round based structure. The system defined here assumes correct network behavior by using TCP protocol. In this respect it differs from Verdi [54], which assumes an unreliable network.

1.2 Summary of Thesis Research Contributions

The three contributions presented in this thesis all aim to improve the performance of transactional systems by focusing upon their different aspects. The contributions explore distributed transactional systems, centralized DBMS and SMR based distributed algorithm. The major contributions in this thesis are as follows:

1. **PXDUR**, or Parallel XDUR, is an implementation of Deferred Update Replication systems, which represent a well-known approach towards designing fault tolerant replicated transactional systems. PXDUR inherits the idea of using speculation to forward execution snapshots among transactions running locally on a given node to alleviate local contention. It improves upon XDUR [1] by adding parallelism to speculation using a concurrency control algorithm (Parspec) in the XDUR framework, thereby increasing both the performance and flexibility of deployment. This enables the system to utilize the parallelism offered by multi-core platforms when it is possible to do so. PXDUR also optimizes the commit process to improve performance further. PXDUR achieves a performance improvement in the range of 30% – 50% in the best cases over the closest competitors.

2. **TSAsR**, or Timestamp Based AsR, is an extension to the AsR framework [38]. The AsR scheme tries to improve the performance of centralized DBMS transactions by combining more relaxed DBMS isolation levels with an external certifier to guarantee serializability. The existing serializability schemes in DBMSs are lock based and pessimistic. AsR tries to improve upon such conservative eager-locking schemes by utilizing the DBMS's less pessimistic isolation levels and tracking relevant meta-data to achieve serializability. TSAsR adds a timestamp based serializability certifier to the AsR framework. It achieves performance improvement over the existing visible-read based AsR and the default DBMS operation in moderate and high contention scenarios.

3. **Verified JPaxos**, is aimed at developing a run-time for a Multipaxos system, which tries to achieve the ease of verification without compromising performance. The system consists of Multipaxos specification written in HOL logic. The HOL logic is then exported to the Scala programming language through a code-generator. This work takes a pre-existing Scala generated I/O automata describing the Multipaxos algorithm and adds a run-time over it to create a deployable replicated system. The modeling of Multipaxos in HOL and its subsequent verification are out of the scope of this work. The run-time is based on the design of JPaxos [25], which is a high performance Java implementation of Multipaxos. The run-time includes both Java and Scala classes and inherits the design approach and some optimization ideas from JPaxos, to achieve good performance. We achieve a 2.0–2.5x speedup over PSYNC, which is a competitor verification framework, and are about 20% slower than the non-verified implementation of JPaxos.

1.3 Thesis Organisation

The remainder of this thesis is organized as follows. Chapter 2 presents a discussion of the past and related work with respect to the contributions in the thesis. Chapter 3 describes PXDUR, which is designed to enhance DUR-based distributed transactional systems through speculation, parallelism and commit optimization. Chapter 4 presents the experimental evaluation of PXDUR. Chapter 5 presents the TSAsR algorithm, which adds a timestamp based serializability certifier to AsR framework. The external certifier is combined with internal concurrency control measures employed by the DBMSs with more relaxed isolation constraints. Chapter 6 presents the evaluation for TSAsR. Chapter 7 presents the run-time for verified distributed systems. It describes a run-time environment, which can be applied to an easily verified HOL-generated specification of SMR based distributed algorithms. In this work the run-time is based on JPaxos's design and aims at improving the system's performance. Chapter 8 reviews the conclusions formed from this work and some further developments that can extend the given research.

Chapter 2

Past and Related Work

2.1 Deferred Update Replication

Pedone et. al. propose the Deferred Update Replication (DUR) model in [44], and explain its advantage over `immediate update synchronization` method in distributed services. They describe the protocol's properties explaining the idea of executing the transactions locally on individual nodes and sending the results to all the nodes for certification. They underline the need for a global certification pointing out the disadvantage that in DUR transactions executing locally have no way of detecting remote conflicts, as a results many transactions may abort during global certification phase. They try to alleviate the transaction abort rate by using a reordering certification test during the commit phase.

Deferred Execution Replication (DER) [18, 34, 43] is an alternate method of deploying the State Machine Approach based distributed systems. In case of DER, the clients defer the execution of transactions until a global order is established. Unlike DUR, the transactions are not processed optimistically, but they simply broadcast transaction requests to all nodes and wait until the fastest node replies with the order. Then, all the client requests are executed by each of the nodes. DER based systems have both advantages and disadvantages with respect to DUR based systems. DER systems do not suffer from aborts due to contention on shared remote objects because a common serialization order is defined prior to starting transaction (local) execution. Thus, they can give better performance and scalability in medium/high contention scenarios [23]. For DER, the size of network messages exchanged for establishing the common order does not depend on the transactions logic (i.e., the number of objects accessed). The network messages only contain the transaction name and its input parameters, which can prevent the network from saturation and enhance the performance of the ordering protocol. However, a distinct advantage of DUR-based systems over DER is that in DUR systems, the transactions need to be executed only by one node (locally), while all the nodes need to perform a certification after total order. Generally, the cost of

certification is much less as compared to the cost of executing each transaction on every node (as done by DER). Both DUR and DER based systems find their respective sweet spots in different scenarios. DUR systems give best performance when transactions on remote nodes rarely conflict. On the other hand, DER systems are not affected by local contention but are limited due to higher amount of processing required, per transaction. Thus, both the systems target different kind of workloads, therefore their deployment depends upon the requirements of the system and the application.

Sciascia et. al. describe S-DUR in [49], which uses `partitioning` to enhance the performance of the system. They divide the database in partitions, replicate the partitions across the servers and have only local transactions i.e transactions which access only one partition, proceed via the DUR route. Transactions using cross-partition access follow a different commit path. S-DUR can enhance the performance of the DUR system if a higher percentage of transactions are local. However, S-DUR still suffers from local contention.

Speculation is used as a method to anticipate work and order transactions before the establishment of total order by OSARE [43]. OSARE tries to maximize the overlap between the co-ordination of nodes and execution of transactions by speculatively forwarding the post-image of completed transactions which are not delivered finally yet, along the chains of conflicting transactions. The approach is similar to PXDUR, but the deployment is different as it is used for systems where transactions are executed after the establishment of total order.

Peluso et. al. introduce Specula in [45] which is a protocol for Software Transactional Memory (STM) systems. Like OSARE, it uses speculation to overlap the synchronization phase of replicas with execution phase of transactions . Specula removes the execution of the replica synchronization phase from the critical path of execution of transactions allowing threads to pipeline the execution of transactions. Specula is similar to PXDUR as here too, execution threads commit speculatively going on to next operation assuming the success of speculatively committed transactions. However, unlike PXDUR, Specula does not maintain an order among transactions speculatively committed by different threads. Such transactions if conflicting will still abort later.

Conflict-aware load balancing introduced in [58], tries to alleviate local contention by assigning transactions to preferred servers based upon the number of conflicting transactions. The idea is to serialize conflicting transactions by clubbing them together on the same server. It uses lock-based methods for synchronization on the same server.

X-DUR [1] is a DUR based protocol, which alleviates the local contention by allowing the active transactions to speculatively read from the snapshot generated by locally committed transactions awaiting to be globally certified. X-DUR provides two high-level guidelines, which can be applied to existing DUR-based protocol for increasing their performance further:

- `Local Transaction Ordering:` All transactions executing on one node should be pro-

cessed according to a local order. It is worth to note that this order is not necessarily known (or pre-determined) before starting the transaction execution, rather it could be determined while transactions are executing taking into account their actual conflicts.
- `Local Certification Ordering`: Each node should submit completed transactions to the certification layer in the same order as they are speculatively (locally) processed and it should not allow the global ordering layer to change this (partial) order.

This way, the local transaction ordering is always compliant with the final commit order, thus making the speculative execution effective. However X-DUR uses only a single executor thread to schedule parallel transactions issued by clients. This approach tends to serialize the speculative transactions thereby limiting the performance benefit especially on highly parallel platforms.

PXDUR inherits the idea of speculative forwarding and local certification ordering from XDUR [1]. However, it enhances XDUR by adding parallelism to speculation. It also adds the optimized commit feature to increase the performance further. Similar to XDUR, PXDUR can be used to enhance the performance of fully partitioned DUR systems, further. It adds flexibility to the system by providing the option of a configurable number of execution threads. PXDUR's approach is different from the other state-of-art technique to reduce local contention described by Conflict-aware load balancing [58], as [58] does not use speculative forwarding.

2.2 Concurrency control in centralized DBMS

Eswaran et. al. [10] explore the general concept of transactions and their consistency in databases. They define concepts like consistency of a database, the phenomenon of phantoms, transaction scheduling, locking logical datasets instead of explicit items, among other properties.

Berenson et. al. [2] explore and critique the main isolation levels defined by the ANSI/ISO SQL standards. They argue that the standards fail to characterize several popular isolation levels, including the standard locking implementations of the levels. They investigate the arising ambiguities and provide clearer explanation of the isolation levels. They define a new isolation level `Snapshot Isolation` which is less consistent than `Serializability` but gives better performance. Under snapshot isolation, transactions get a snapshot of the system when they start which remains the same for the transaction's lifetime. Thus transactions can be invalidated through concurrent write-write conflicts only. [11], [4] and [47] try to make snapshot isolation serializable by tracking additional meta-data and trying to prevent cycles in the transaction dependency graph.

Li et. al. explore the area of relaxing consistency level for applications in [31]. They allow multiple levels of consistency to co-exist, and divide operations as `Red` and `Blue` operations. The Blue operations are executed locally and are lazily synchronized in the manner of even-

tually consistency, while the red operations are serialized with respect to each other and need to be immediately synchronized across all sites, leading to cross-network communication. They refer to it as RedBlue consistency and have built an infrastructure called Gemini to co-ordinate RedBlue replication. RedBlue replication breaks application operations into generator and shadow operations out of which only shadow operations can become red or blue operations. However, this model places the responsibility of identifying the red and blue operations on the application developer. Some of this analysis can be automated by using a framework like Sieve [30] to automate the process of identifying the consistency level for different operations.

Balanced Concurrency Control (BCC) [57] presents an approach to reduce false abort in Optimistic Concurrency Control (OCC) method by cycle prevention in serial dependency graph. OCC allows transactions to proceed without locking read-set objects and does a validation before the operation write. It aborts the transactions whose read-set objects are modified. BCC tracks an additional data dependency, which can combine with the anti-dependency present in OCC read-set validation to detect the possibility of a cycle in the transactional dependency graph. If no cycle is detected the transaction can commit even though its read-set object(s) is modified before the commit. BCC needs to track anti-dependencies in the system to be able to detect cycles through specific pattern. BCC aims to reduce extra aborts, but only applies to systems using OCC method.

Wang et. al. [51, 52] also discuss the idea of leveraging relaxed concurrency control levels to improve DBMS's performance, while providing serializability through external meta-data processing. They introduce Serial Safety Net (SSN), which is a timestamp based serializability certifier. It can be applied on top of the default concurrency control provided by DBMSs. SSN allows a transaction to commit if its commit will not close a serial dependency graph. SSN determines this by associating a predecessor and successor time stamp to every transaction at the time of commit. It attempts to eliminate cycles from the dependency graph without tracking the full graph, thus it is prone to admitting some false negatives.

The As Serializable (AsR) [38] transaction framework is based on the idea of utilizing relaxed islotion levels, similar to [51, 52]. However, AsR enhances the idea by leveraging the hierarchy of isolation levels discussed in Section 5.1.1. Transactions start optimistically at lowest isolation level using additional meta-data utilized by a serializability certifier to detect inconsistencies. If a transaction aborts repeatedly running at a particular isolation level it can upgrade to tighter isolation levels. The existing implementation of AsR uses a `visible-reads` based method as the serializability certifier. However, this scheme only gives significant benefits only in high contention scenarios.

This work TSAsR, inherits the AsR framework from [38]. However, the serializability certification layer used in this work is completely different from the one used by Niles et. al. in [38]. The serializability certifier used in this work is based on the design of Serial Safety Net described in [51, 52]. The difference lies in the fact that, in this work , SSN is integrated to the AsR framework which tightens the concurrency control mechanism of underlying DBMS

as a transaction aborts repeatedly. SSN presented in [51, 52] was originally designed for multi-version systems, however we use it for a single version system in this work. This work differs from [11], [4] and [47] as it utilizes default isolation levels like Read Committed and Repeatable Read instead of Snapshot Isolation, but can be extended to support snapshot isolation as well. This work though uses read-set and write-set as the level of transactions but unlike OCC methods, it does not use them directly to certify whether it is safe to commit a transaction or not.

2.3 Formalization of distributed algorithms

Distributed algorithms are generally defined in English or via a pseudo code using different computational models. There is no uniform model present to describe the distributed algorithms. The verification of distributed algorithm is a difficult task as well. To address these concerns, recently there have been some attempts to ease the verification effort required for these systems. Such efforts have taken the form of functional languages with built in support for verification (e.g., EventML, Verdi) or domain specific languages like PSYNC which makes it possible for the asynchronous system to be viewed as a synchronous system for the sake of verification.

PSYNC [9] provides a high-level round based control structure aimed at reducing implementation complexity of distributed algorithms. A PSYNC program is defined by a sequence of rounds, and has a lockstep semantics where all the processes execute the same round. A process may send a message in a round and go to another round depending upon messages received. PSYNC's run-time is in Scala and it ensures the lockstep behavior on top of asynchronous networks by using time-out based round updates. The authors prove that the asynchronous system defined by the run-time is indistinguishable from the synchronous lockstep model. They provide a state based verification engine to verify the synchronous model.

Verdi [54] is a Coq framework for implementing and proving correct distributed algorithms. Verdi starts with asynchronous implementation and progressively transforms it using refinement into an asynchronous fault-tolerant one. The correctness of the implemented programs is mechanically proved using the Coq theorem prover. Verdi however gives very low performance.

IronFleet [16] is a framework which supports proving both safety and liveness properties of distributed system implementations. Ironfleet provides a high level spec layer which is used to model the system's specifications. The distributed state machine is defined in Dafny [29] which can also be used for automated theorem proving. The high level specs are integrated to the protocol layer which is extended to support networking. The implementation layer is added to the protocol layer which contains the developer code to run on each node. This layer deals with the practical programming issues. Dafny compiles code to C# to produce

the executable.

EventML [46] is a functional language which can be used to describe distributed systems. It provides many primitives and combinators to implement these systems. EventML programs can be automatically transformed into formulas which can be used to verified the system. EventML has a NuPRL [7] plugin which can be used to verify the algorithm. Like Verdi, EventML too suffers from low performance.

nqsb-TLS or Not Quite So Broken TLS [21] is a re-engineered approach to security protocol specification and implementation. It addresses the various security flaws in Transport Layer Protocols arising due to factors like programming errors, memory management, complex APIs, errors arising due to misinterpretation of complex protocol prose etc. nqsb-TLS consists of functions which serve two roles: 1)They act as the specification of TLS representing state changes as a result of input received. In this form nqsb-TLS can be used as a test oracle to check conformance of traces from arbitrary sources. 2)The same functions can be coupled with I/O code to form the part of main system's TLS implementation. In this respect nqsb-TLS is similar to our system as the same functional code which describes system's specifications can also be used as an executable aimed at making the system more reliable. However TLS finds its utility in a completely different domain of security protocols while we address distributed algorithms.

This work, presents run-time for a formalized Multipaxos system. The replicated system is modeled in HOL (Higher Order Logic). The model describes the system as an I/O automaton [33] which takes specific actions based on the input received, changes the system state appropriately and produces an output. The incentive for modeling the system in HOL is that it simplifies the task of formally verifying the system significantly. However, by itself the system model cannot be deployed. A run-time system is needed to execute the system model, providing it with the input and utilizing its output as per the specifications of the SMR algorithm modeled. Developing such a run-time is the main contribution of this work. The run-time's responsibility is to correctly implement the execution model produced in HOL using the reliable TCP/IP framework for communication, so that the verification is only required for the I/O automaton modeled in HOL.

The approach presented here is similar to that of EventML and Verdi. We implement a system in HOL, verify it and then generate the scala executable. In case of EventML, code is written in EventML first and then exported to NuPerl for verification. Verdi considers network to be asynchronous, while Verified JPaxos uses TCP to model a reliable network free of errors like duplication or data loss. The systems produced by Verdi and EventML also perform poorly. On the other hand, the run-time presented here focuses on performance optimization while staying away from low-level optimizations which can make it unreliable.

Ironfleet is similar to us in some respect as it uses a high level system specification, models distributed protocol in Dafny to make it easy to verify and requires a separate run-time. However, in our case, there is no separation in system specification and the distributed protocol layers. The Multipaxos algorithm is implemented as a distributed protocol in HOL.

Our run-time layer also includes the network unlike Ironfleet where the distributed protocol is extended to support the network. The verification effort required in Ironfleet is quite high which is a limitation.

Lastly, PSYNC is somewhat similar to us as it aims to reduce the verification effort for distributed algorithms. However unlike us, PSYNC provides automated verification if the system can be modeled in its framework, which can be a limitation as well. If an algorithm cannot be described in a round-based manner with lockstep semantics in a straight-forward manner, it can become extremely difficult to model such algorithms in PSYNC. This is especially true for leaderless consensus algorithm which form an important domain in distributed algorithms. Our model, on the other hand, does not enforce a particular program model, as the algorithm is implemented in HOL directly from its semantics.

Chapter 3

PXDUR : Parallel Speculative Client Execution in Deferred Update Replication

In this chapter, we present PXDUR, a DUR based protocol, which addresses both local contention and global certification bottlenecks to improve system performance. PXDUR inherits the speculative forwarding and local certification ordering guidelines from X-DUR and adds parallelism to it. Even though XDUR achieves significant performance gain especially in the scenarios with high local contention, its performance is limited by the use of a single executor thread, which serializes every local transaction irrespective from whether it conflicts with others or not. PXDUR builds upon XDUR by allowing the execution of speculative transactions in parallel. To do so, it borrows the principles at the base of ParSpec, the concurrency control deployed by Archie [17] and used in the context of SMR [48]. With that, it allows to fix the number of execution threads best suited to the particular environment.

Regarding the performance improvement of the global certification phase, PXDUR optimizes that by identifying the scenarios where it is safe to skip the validation of transactions undergoing commit and just apply transaction updates directly. In those cases, this optimization reduces the overhead of the global certification phase, therefore resulting in better performance on the whole replicated transactional system.

PXDUR is implemented in Java and inherits the struc- ture of JPaxos [25]. We evaluate PXDUR using three well- known transactional benchmarks such as Bank, a monetary application, TPC-C [8], the popular on-line transaction processing benchmark, and Vacation [5], a distributed version of the famous application included in the STAMP suite. As competitors, we implemented a DUR system based on the approach described in [58], where conflicting transactions are grouped together and transactions lock the shared objects until they finish, and XDUR. For testbed we use up to 23 nodes available on PRObE [13], a state-of-the-art public cluster. Results reveal that PXDUR benefits due to its parallelism under low con-

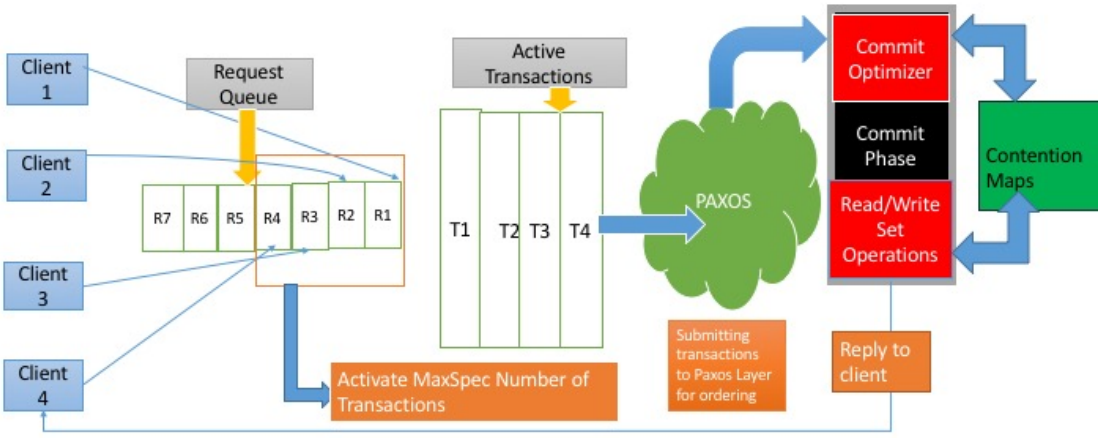

Figure 3.1: PXDUR System Overview

tention scenarios and provides the benefit of speculation under high contention workloads. As an example of our findings, the maximum speed-up observed when running TPC-C is higher than one order of magnitude against the original XDUR and the other competitor.

3.1 The Protocol

Figure 3.1 presents a schematic view of PXDURs operations. At each node, client threads, which are the threads in charge of executing the application, do not handle transactions by their own but they enclose them in requests, which are subsequently added into a request queue. PXDUR activates a configurable number of threads (called MaxSpec) to execute the transactions in these requests in parallel.

The concurrency control algorithm used for such parallel execution implements an enriched version of the original XDURs idea of client-side speculation. In practice, it speculatively forwards to the new incoming transactions copies of shared objects updated by those transactions that have committed locally but are awaiting for their global order to be established. The fundamental difference of PXDUR against XDUR is that in PXDUR the speculation happens in parallel, thus speculative non-conflicting transactions do not need to be executed one after the other as in XDUR. Clearly, allowing such pattern mandates an additional mechanism to check whether two transactions conflict while they execute or not. This mechanism is not implemented by XDUR because it allows only a serial speculative execution.

A serialization order is imposed on the speculative trans- actions to ensure that the chain of speculative reads remains consistent even after the total order has been established. In fact, when a set of transaction is entirely executed, their read-sets and write-sets are sent to all nodes through the total order layer. Importantly, the total order layer should not revert the order used by a node to submit its own transactions because, in case of partitioned accesses, it would nullify the benefits of the speculative execution. More specifically, once the ordering layer has imposed a total order among the read-sets and write-sets submitted by locally executed transactions, on each node a committer thread is invoked to verify the consistency of the read objects and apply updates to the database. The committer thread verifies the correctness of a transaction by comparing the version of every object in the read-set object against the most recent committed version of the given object in the database. If the read-set validation succeeds, i.e., no object in the transactions read-set was modified, the transaction is committed by updating shared objects as in the transactions write-set.

In case of partitioned access and given that transactions speculative executed on a node should not be ordered differently by the total order layer, it is guaranteed that local transactions will not be aborted. PXDUR is able to identify such cases, and optimize further the performance of the committer thread by skipping the read-set validation altogether. This feature is described more in detail in Section 2.4.2.

3.1.1 Concurrency Control

In ParSpec, transactions can be classified as `speculatively-committed` (or x-committed hereafter), which are those transactions that have completely executed all their operations and cannot be aborted anymore by other speculative transactions; or `active`, which are those transactions that are still executing operations or they finished them but are not allowed to speculatively commit yet.

Moreover, each transaction T records its speculative order in a field called $T.ID$. This order matches the serialization order that T respected while executing speculatively and it will be used during the transaction certification phase.

As a support for the speculative execution, the following meta-data are used: `abort-array`, which is a bit-array that signals when a transaction must abort; `LastXcommittedTx`, which stores the ID of the last x-committed transaction.

For each shared object, a set of additional information is also maintained for supporting ParSpec's operations:

- The committed version;
- The version written by the last x-committed transaction, called speculatively committed or `spec-version`;
- The `Owner` field, which holds the ID of the currently active writer of the object; NULL if none;

- an array called `readers-array`, which tracks active transactions that already read the object.

The size of the `abort-array` and `readers-array` is bounded by MaxSpec, which is an integer defining the maximum number of speculative transactions that can run concurrently. As clients submit the requests to the request queue, ParSpec extracts the transactions from the request queue and processes them, activating MaxSpec transactions at a time. Once all these speculative transactions finish their execution, the next set of MaxSpec transactions is activated. As it will be clear later, this approach allows a quick identification of those transactions whose history is not compliant anymore with the speculative order, thus they must be aborted and restarted. In the abort-array and readers-array, each transaction has its information stored in a specific location such that, if two transactions Ta and Tb are speculatively ordered, say in the order $T_a < T_b$, then they will be stored in these arrays respecting the invariant $T_a < T_b$. Since the speculative order is a monotonically increasing integer, for a transaction T, the position $i = T.IdmodMaxSpec$ stores T's information. When $abort - array[i] = 1$, T must abort because its execution order is not compliant anymore with the speculative order. Similarly, when an object obj has readers-array[i]=1, it means that the transaction T performed a read operation on obj during its execution. ParSPec's operations can be better understood through Algorithms 1,2 and 3, which represent the read,write and xcommit operations respectively.

Algorithm 1 Read Operation

```
1: Procedure : Read Operation
   Input: Tx, Obj
2: ▷ Check for an active writer
3: if Obj.owner! = 0 ∧ Obj.owner < Tx.Id) then
4:     while Obj.owner < Tx.Id do
5:         Wait
6:     end while
7: end if
8: Mark Tx.Id in Obj.Reader_Array
9: Add Obj.spec_versiontoTx'sReadset
10: Return
```

Transactional Read Operation

Algorithm 1 describes the read operation by Transaction Tx on a shared object Obj. When a transaction Tx performs a read operation on an object Obj, it checks the object's `owner` field to find if it is being modified by a currently active writer (Line 1.2). If the current writer's speculative order (stored in the Obj.owner field) is prior to Tx's order, it is useless for Ti to access the spec-version of Obj because, eventually, the writer will x-commit, and Tx will be aborted and restarted in order to access the writer's version of Obj. In this situation, Tx

waits for the current writer to x-commit and release the object(Lines 1.3 - 1.5). Otherwise, Tx proceeds with the read operation by marking itself in Obj's `reader-array` and adding Obj's `spec-version` to its read-set (Lines 1.7 - 1.9).

Algorithm 2 Write Operation

```
1: Procedure : Write Operation
   Input: Tx, Obj
2: if Obj.owner! = 0 ∧ Obj.owner < Tx.Id then
3:     ▷ Wait for the previous active writer
4:     while Obj.owner < Tx.Id do
5:         Wait
6:     end while
7: end if
8: Obj.owner = Tx.Id
9: Mark Tx.Id in Obj.Reader_Array
10: InvalidateReaders(Obj.Reader_Array, Tx.Id)
11: Add Obj.spec_version to Tx'sReadSet and WriteSet
12: Return

13: Procedure : InvalidateReaders
    Input: Reader_Array, TId
14: Index = TId
15: while Index < MaxSpec do
16:     if Reader_Array[Index] is set then
17:         Set AbortArray[Index]
18:     end if
19:     Increment Index
20: end while
21: Return
```

Transactional Write Operation

In ParSpec, transactional write operations are buffered locally in a transaction's write-set. Therefore, they are not available for concurrent reads before the writing transaction x-commits. The write procedure has the main goal of aborting those speculative active transactions that are: 1. serialized after (in the optimistic order) and wrote the same object, and/or 2. previously read the same object (but clearly a different version). Similar to the read operation, when a transaction Tx performs a write operation on an object Obj and finds the Obj has a currently active writer who precedes Tx in speculative order, Tx waits for the current writer to finish (Lines 2.2 - 2.7). On the contrary, if Obj's owner field is zero (no writer) or $Obj.owner < Tx.Id$ (Obj' current writer is serialized after Tx), Tx will go ahead

and set the Obj's owner field to Tx.Id and marks itself in Obj's reader-array. (Lines 2.8 - 2.9). Since a new version of Obj written by Tx will eventually become available, all speculative active transactions who are speculatively ordered after Tx that read Obj must be aborted and restarted so that they can obtain Obj's new version. The function `InvalidateReaders` is called for this purpose (Line 2.11). Identifying those speculative transactions that must be aborted is a lightweight operation in ParSpec. When a speculative transaction x-commits, its history is fixed and cannot change because all the speculative transactions serialized before it have already x-committed. Thus, only active transactions can be aborted. Object Obj keeps track of readers using the readers-array and ParSpec uses it for triggering an abort on all active transactions that appear in the readers-array after Tx's index (Lines 2.15 - 2.20) . Finally the object's `spec-version` to its read-set and write-set.

Algorithm 3 XCommit Operation

```
1: Procedure : Speculative Commit Operation or XCommit
   Input: Tx
2: ▷ Wait for the previous active writer
3: while LastXcommittedTx! = Tx.Id − 1 do
4:     Wait
5: end while
6: ▷ Check if Tx was aborted by a previous writer
7: if AbortArray[Tx.Id] == true then
8:     Abort Tx and retry
9:     Return
10: end if
11: ▷ Apply the updates from the writeset to shared objects
12: for ∀Obj ∈ Tx.WriteSet do
13:     Update Obj.spec-version with the writeset value
14:     Obj.owner = 0
15: end for
16: LastXcommittedTx = LastXcommittedTx + 1
17: Return
```

XCommit

The Xcommit process is described by Algorithm 3. Speculative active transactions make available new versions of written objects only when they x-commit. This way, other speculative transactions cannot access intermediate snapshots of active transactions. However, PXDUR needs to ensure that the snapshots of shared objects updated by speculative transactions are forwarded to active transactions. When MaxSpec transactions are activated in parallel, multiple concurrent writes on the same object could happen. Thus, ParSpec needs to ensure that if concurrent transactions within a MaxSpec set modify the same object, they

read the object's updated version in the speculative order of the transactions. As an example, consider four transactions T1, T2, T3, T4 that are speculatively active for a MaxSpec value of four, in the given order. T1 and T3 write to the same object Oa, and T2 and T4 read from Oa. When T1 and T3 reach the speculative commit phase, they make two speculative versions of Oa available: OT1a and OT3a . According to the optimistic order, T2's read should return OT1a and T4's read should return OT3a.

In order to enforce this, ParSpec allows an active transaction to x-commit only when the speculative transaction optimistically ordered just before it is already x-committed. Formally, given two speculative transactions T_x and T_y such that $T_y.Id = T_x.Id + 1$, T_y is allowed to x-commit only when T_x is x-committed. Otherwise, T_y keeps spinning even when it has executed all of its operations (Lines 3.3 - 3.5). Ty easily recognizes Tx's status change by reading the shared field `LastXcommittedTx`. This property ensures that only one speculative version of the object is available when to any transaction for reading. In addition, even though two transactions may write to the same object, they can x-commit and make available their new versions only in-order, one after another. This policy prevents any x-committed transaction to be aborted by speculative transactions.

If a transaction Tx has not been aborted by the time the previous transaction in its MaxSpec set x-commits, the transaction Tx is safe to be x- committed (Line 3.8). In the commit process, all the objects written by transaction Tx are moved from Tx's write-set to the spec-version field of the respective objects and the owner fields of the corresponding objects are cleared (Lines 3.11 - 3.14). This way, the new speculative versions can be accessed from other speculative active transactions. This is followed by incrementing `LastXcommittedTx`, which allows the next transaction in the batch to X-Commit (Lines 3.14 - 3.15).

3.1.2 Handling Conflict Phase

The global certification layer is the distributed component in charge of total ordering certification requests and validating/committing transactions locally. X-DUR inherits the technique for certifying transactions from the DUR model. Specifically, when a batch is delivered, each transaction's read-set and write-set is extracted and certified. The certification consists of validating the read-set against the current (non-speculative) committed versions available and, in case of a successful validation, all speculative written objects are made available to all non-speculative transactions (including the next certification requests). If on the one hand the speculative execution allows to move forward the transactions' progress in case of partitioned accesses, then on the other hand a remote conflict could inevitably force a possible long chain of speculative conflicting transactions to abort. PXDUR addresses this issues by stopping the speculative execution of incoming transactions as soon as a remote abort is detected during the certification phase. In practice, when an abort happens the speculative execution handler sets the spec-versions of all the objects in the aborted transaction's write-set to null. As a result ,all new transactions are forced to read the committed

versions of the objects in question.

As an added optimization, when a transaction is aborted, PXDUR adds all the objects in its write-set to a map called the `Conflict Map` along with the latest committed version number of the given object. Any x-committed transactions which have already read the spec-version of any of these objects will also abort. Inorder to prevent such doomed transactions from burdening the ordering layer, PXDUR scans the queue of x-committed transactions awaiting submission to the ordering layer. Every transaction whose read-set contains any object in the `Conflict Map` is aborted and its write-set objects are also added to the `Conflict Map`. This scan is done whenever a transaction undergoes conventional abort. The commit thread checks the committing transaction's write-set objects against the objects in the `Conflict Map`. If a match is found, and the object's latest committed version is greater than that of the version stored in the map, the corresponding object is removed from the map.

3.1.3 Optimizing the commit

The speculative transaction ordering and the local certification ordering used in XDUR ensures that transactions which are speculatively committed, will not be aborted later unless there is a remote conflict. Thus, if it is possible to identify the scenario where a remote conflict is not possible, the read-set validation step in the commit can be skipped. This can provide a significant speedup in a transaction's critical path and help in improving the overall throughput and latency. The implementation is described in the following paragraphs.

Every node has an array of `contention maps`, which contains one map for each node.The map corresponding to a node contains the objects logically associated to that node according to the partitioned-access pattern. Those objects are suspected to cause aborts if accessed by transactions executed on other nodes. Elements in this map are added and removed by the node commit thread (i.e., the same thread that performs the transaction certification). If the `contention map` for any node is empty, it means that it is safe to skip the read-set validation phase for all local transactions (i.e., transactions which do not access any remote objects). The map operations are explained later.

In a system with limited cross-partition access, the transactions, which trigger remote conflicts are those accessing remote objects. Any transaction T that accesses a remote object is identified by setting the `T.cross_access` flag. In case of cross-partitioned access, another field `T.remoteId` contains the ID of the node whose object(s) the transaction T accesses. When the commit thread encounters a transaction whose `cross_access` flag is set, it performs the following actions regardless of the content of the contention map: 1. it performs the read-set validation for this transaction; and 2. through the `T.remoteId` field, it identifies the remote node number whose object this transaction reads. If the transaction commits, all the remote objects written by this transaction are added to the remote node's contention map. This will cause a read-set validation for all the transactions that are suspected to conflict with T, which includes all the transactions local to the remote node in question

Adding to and removing from the contention maps

Objects are added to the contention map in following two scenarios: 1. whenever a transaction aborts, the objects in its write-set and their latest committed version are added to the given node's contention map; and 2. whenever a transaction accesses objects belonging to a remote node, the remote objects along with their latest committed version are added to the remote node's contention map, if the given transaction commits. Whenever, a transaction is committed, the commit thread checks the corresponding node map where the transaction originated. If this map is not empty, the commit thread checks the committing transaction's write-set objects against the objects in the contention map. If a match is found, and the object's latest committed version is greater than that of the version stored in the map, the corresponding object is removed from the map.

Optimized commit operation and performance

Whenever a transaction's final commit operation s executed, the commit thread checks if the transaction's cross-access flag is not set and the contention map for the transaction's originating node is empty. If both the conditions are true, the read-set validation for the transaction is skipped. This optimization finds its sweet spot under fully partitioned access, since there is no remote access and all the contention maps stay empty. Studies show that even under a small amount of remote accesses (5 percent of accesses), the contention maps do not empty during the course of evaluation test thereby after initial few commits, it is never safe for any transaction to skip the read-set validation. Under this scenario, the additional map operations performed by the commit thread only serve to add to the overhead of the transactions' critical path. Thus it is found beneficial to switch off the feature entirely and force all the transactions to perform the read-set validation. Currently, this is implemented by setting a global flag after a fixed number of transactions are committed without being able to skip the commit phase. In scenarios where the remote access is temporal, the feature can be switched-on again with trivial effort.

Chapter 4

PXDUR : Evaluation

This chapter presents the experimental evaluation of PXDUR followed by the discussion of the experimental observations. We implemented PXDUR in Java, inheriting the software architecture of PaxosSTM [55]. PaxosSTM processes transactions locally, and relies on JPaxos [25] as a total order layer for their global certification across all nodes. We used the PRObE testbed [13], a public cluster that is available for evaluating systems research. Our experiments were conducted using 23 nodes (tolerating up to 11 failures) in a cluster. Each node is equipped with a quad socket, where each socket hosts an AMD Opteron, 16-core, 2.1 GHz CPU. The memory available is 128GB and the network connection is a high performance 40 Gigabit Ethernet. As transactional applications, we leverage Bank, a common benchmark that emulates bank operations, TPC-C [8], a popular on-line transaction processing benchmark, and Vacation, a distributed version of the famous application included in the STAMP suite [5]. We tested two types of workloads: 1. one fully partitioned, where transactions do not conflicts across nodes; 2. one where a given percentage of transactions on remote node accesses remote objects and hence may produce conflicts.

In the fully partitioned configuration, we analyze the behavior of Bank and TPCC under three contention levels: low, medium, high. Each of them differs from others for the total number of shared objects available. Vacation is only studied under high contention. Table 4.1 summarizes all the configurations used.

We enforce the well-partitioned accesses by equally dividing the total number of shared objects per node (e.g., with Bank, at medium contention and 10 nodes, application threads on one node are allowed to access 200 "local" accounts). Within a node, local accesses are uniformly distributed (not skewed).

Read-only profiles are excluded from the evaluation because those transactions can be run locally exploiting a multi-version repository and concurrency control, as in [56, 19, 24, 43]. All clients (i.e., application threads) in the system are balanced among deployed nodes. In order to avoid changing the load of the system while increasing the number of nodes, the

Table 4.1: Details of the contention configurations used for each benchmark.

Application	**Low contention**	**Medium contention**	**High contention**
Bank(accounts)	5000	2000	500
TPCC(warehouses)	700	115	23
Vacation(relations)	NA	NA	100

amount of clients is kept fixed throughout all experiments. In practice, increasing the size of the system does not change the overall load, thus the performance of all tested configurations degrade due to the higher overhead of the total-order layer (e.g., longer broadcast phase, higher number of exchanged messages, network saturation). The best throughput is often reached in the range of 7-15 nodes deployed, where clients are properly balanced so that computing resources of local nodes are not saturated (for example as for the case of 3 nodes). In all experiments the following total numbers of clients are used: 920 for Bank and 700 for TPC-C and 700 for Vacation.

We selected two competitors other than PXDUR. One is the original X-DUR, and the other is a DUR-based system that was implemented on the idea of `Conflict Aware Load Balancing` presented in [58]. The rationale for using Conflict Aware Load Balancing as competitor is that it is a state-of-the-art solution which addresses local contention in DUR based systems. In this approach, conflicting transactions are grouped upon preferred nodes so that they are serialized. This approach uses locks to secure the shared objects until a transaction finishes its certification (commits or aborts). Unlike PXDUR and XDUR, in Conflict Aware Load Balancing, transactions only observe fully committed data, as compared to the speculative data made available by XDUR and PXDUR.

The logical division of shared objects for each node also provides the conflicting client groups on that node. Client threads run transactions in parallel. Each transaction performs a local validation before committing locally. A transaction is aborted locally if it finds an accessed object to be locked by another transaction or if its local validation fails. A transaction is only sent to global ordering layer after it has successfully locked all the accessed objects. The locks are released when the transaction commits or aborts. As a general comment about [58]'s performance, it finds its sweet spot under low contention and low client count.

Given the partitioned accesses, both the systems show high-performance. However, the impact of PXDUR's speculative execution becomes clear when the number of shared objects decreases (i.e., medium/high contention). That is because, although even the simple speculative single-threaded execution of XDUR prevents local transactions from being aborted, it cannot exploit the additional parallelism as PXDUR. Reasonably, if the contention decreases, more transactions will likely not conflict, thus their parallel execution is more effective. Such a trend is clearly visible in most of the reported plots.

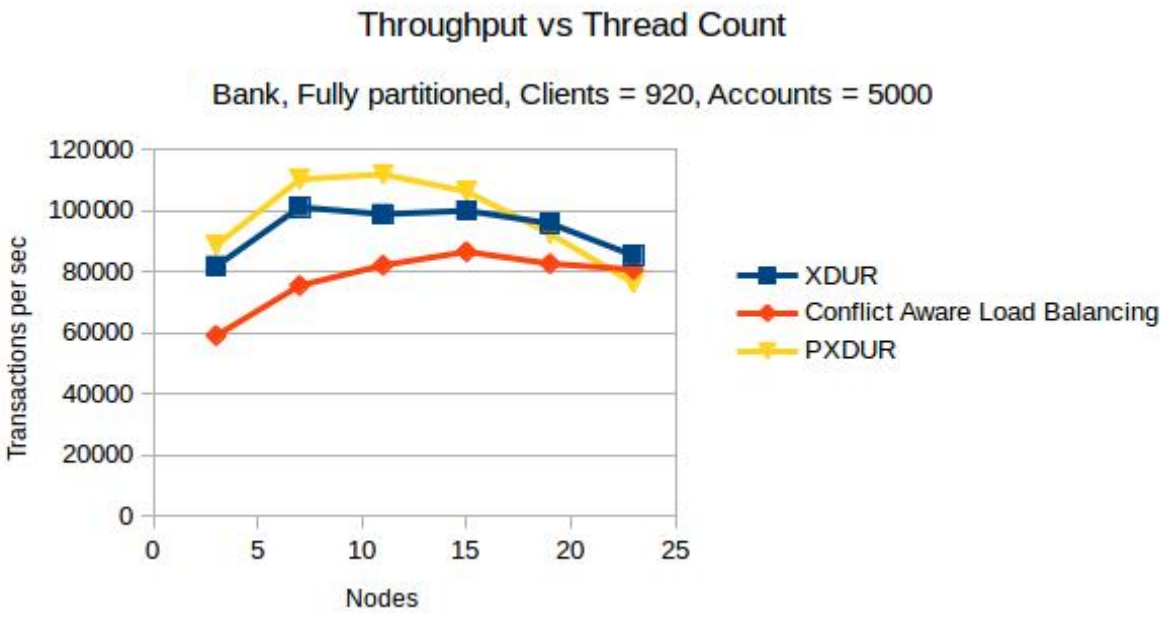

Figure 4.1: Bank low contention

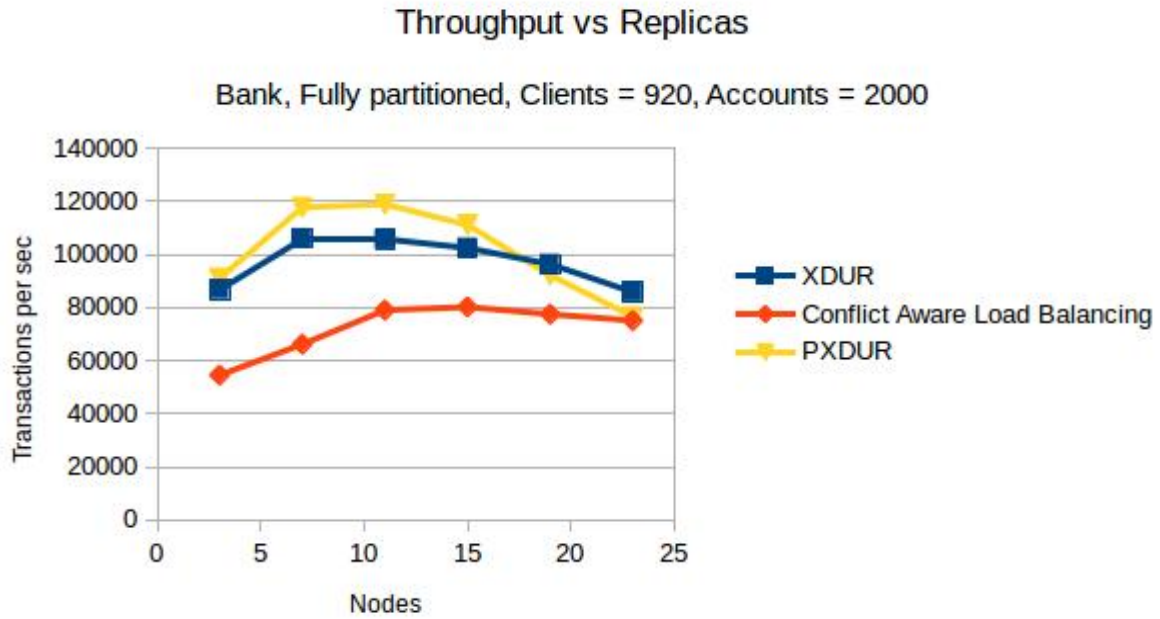

Figure 4.2: Bank medium contention

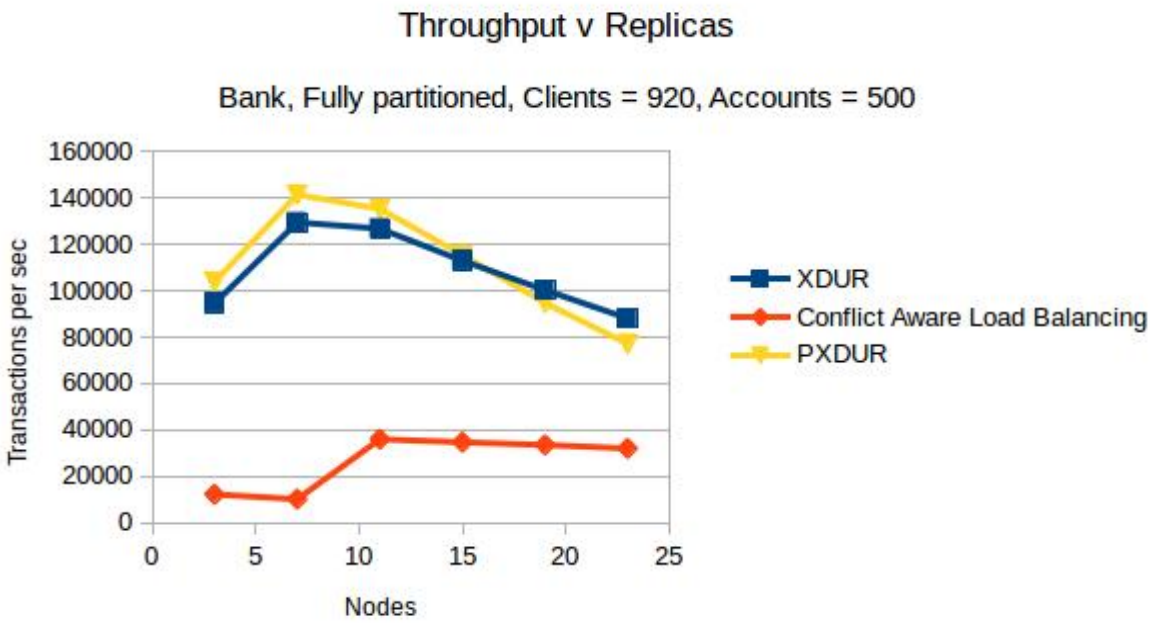

Figure 4.3: Bank high contention

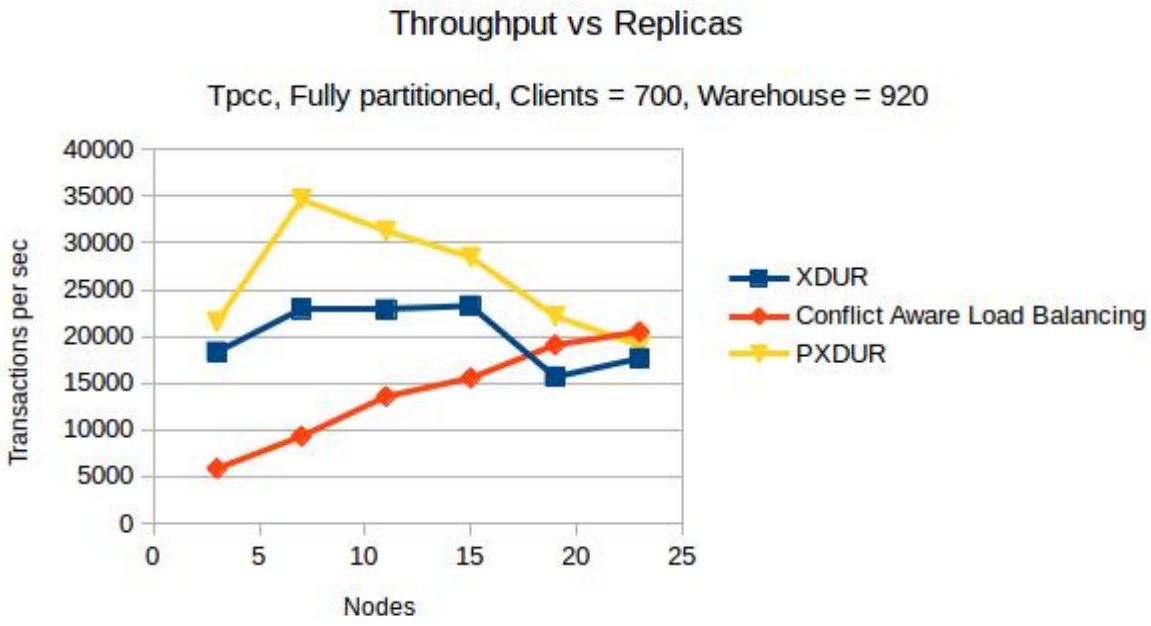

Figure 4.4: TPCC low contention

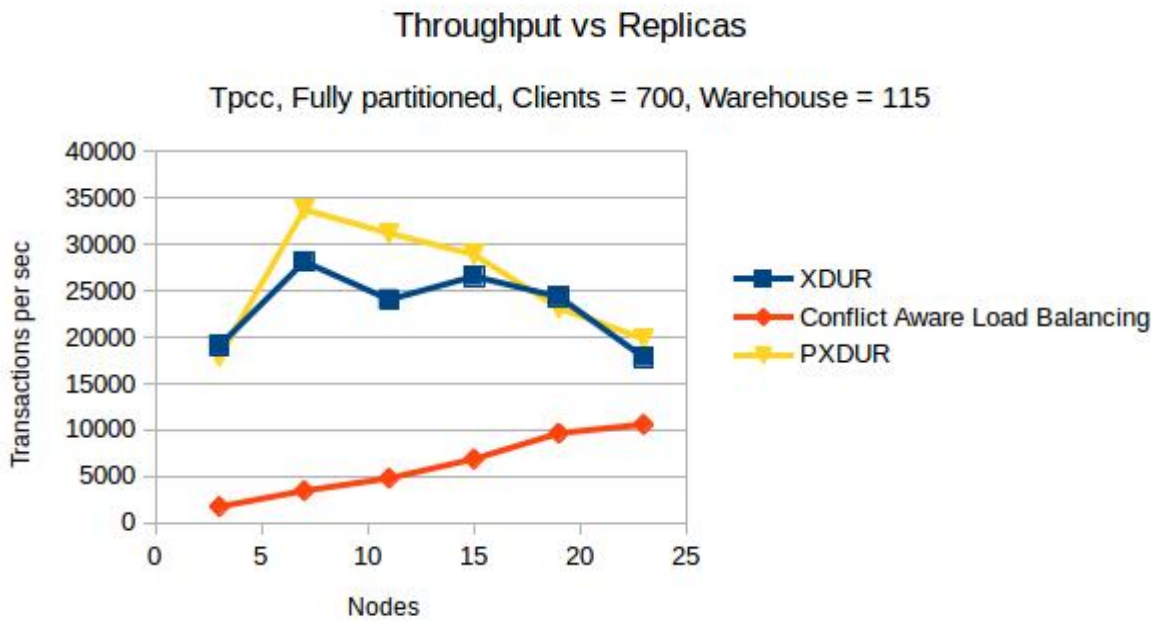

Figure 4.5: TPCC medium contention

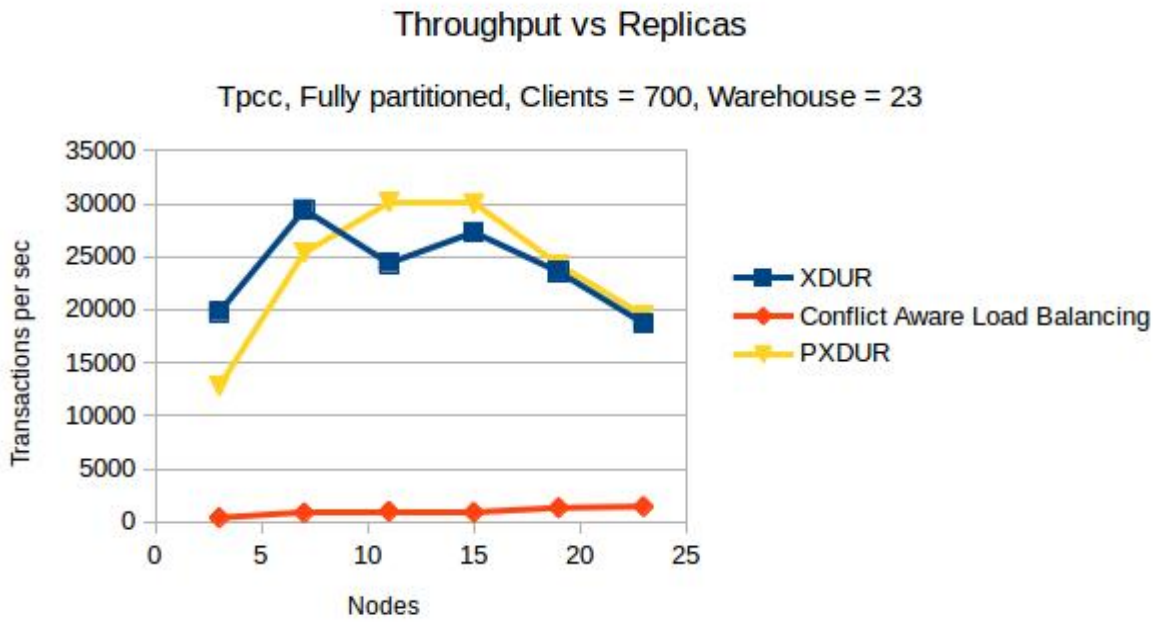

Figure 4.6: TPCC high contention

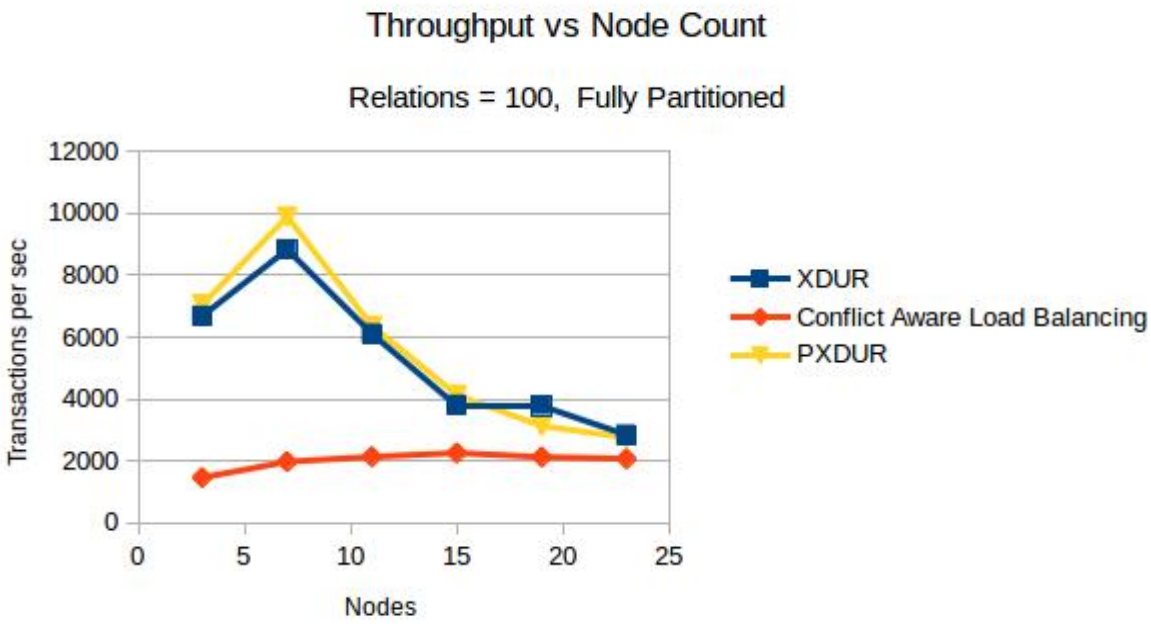

Figure 4.7: Vacation low contention

4.0.1 Bank

Bank is a benchmark characterized by short transactions. Figures 4.1, 4.2 and 4.3 show the results with fully partitioned access under different contention levels. For each of the contention scenario, PXDUR gives better performance than XDUR in the range of 3 – 15 node count. The low and medium contention configurations outline the scenario where PXDUR provides its best performance. For the 11 node case PXDUR gives an improvement of about 13% over XDUR . The performance improvement reduces for the high contention case due to higher number of local aborts occurring for PXDUR under heavy local contention. Single threaded XDUR performs the best for the 19 and 23 node cases. The competitor overtakes PXDUR only for the 23 node case under low contention scenario. As the number of nodes increases, the system performance is limited by the performance of the ordering layer, thus local parallelism provides limited performance benefit. The competitor's performance seems to increase with the increase in node count. This is due to the fact that to keep the overall system load constant we do not increase the total number of clients in the system. Therefore with the increase in node count, the number of clients per node decreases for the competitor thereby bringing down local contention. As expected, the competitor's performance degrades sharply with the increase in contention level in the absence of any measure to alleviate the local contention.

4.0.2 TPCC

TPC-C is characterized by transactions accessing several objects and the workload has a contention level usually higher than other benchmarks (e.g., Bank). PXDUR performs better than XDUR for almost all node counts under low contention (Figure 4.4). PXDUR gives the best throughput and performance improvement over XDUR for the low contention scenario. PXDUR is able to improve upon XDUR by as much as 50% for the 7 node case under low contention. For the medium contention scenario depicted in Figure 4.5, PXDUR outperforms XDUR for 7, 11 and 15 nodes. PXDUR's maximum gain over XDUR amounts to approximately 30% for the 11 node case. PXDUR's performance and performance gain over XDUR decrease with increase in contention level. Under high contention scenario (Figure 4.6) the perform improvement is minimal and PXDUR outperforms XDUR only for the 11 and 15 nodes cases. This is expected since with the increases in contention, the synchronization overhead for the multi-threaded PXDUR also increases as more transactions abort locally and retry. The competitor's performance shows a trend similar to that of Bank benchmark. The competitor outperforms both PXDUR and XDUR for the 23 node case under low contention. However the competitor's performance shows a significant degradation with the increase in local contention.

4.0.3 Vacation

Vacation is more similar to TPC-C than Bank in terms of composition of transactions, but the overall contention is lower (as in Bank). Vacation was tested only in single high contention configuration shown in Figure 4.7. The performance trend is similar to that shown by TPC-C and Bank under high contention. PXDUR slightly outperforms XDUR for 3 – 15 node counts. The biggest performance improvement is obtained for the 7 node scenario where PXDUR shows a 12% improvement over XDUR. XDUR gives the best performance for the 19 and 23 node cases.

The reason for this trend is the fact that under high contention, PXDUR has to deal with a higher degree of local contention, which results in many transactions aborting and restarting locally. XDUR benefits from the serialization of every transaction by the single execution thread as more transactions have the possibility of accessing conflicting objects under high contention. As expected, the competitor's performance is the worst for all the node counts. This result is consistent with the other two benchmarks where the competitor's performance degrades under high contention scenarios without any measure to alleviate contention.

4.0.4 Remote Conflict Scenarios

In this section we present the results for the scenarios where the nodes can access remote objects thus making remote contention possible. Figure 4.8 shows the throughput for each

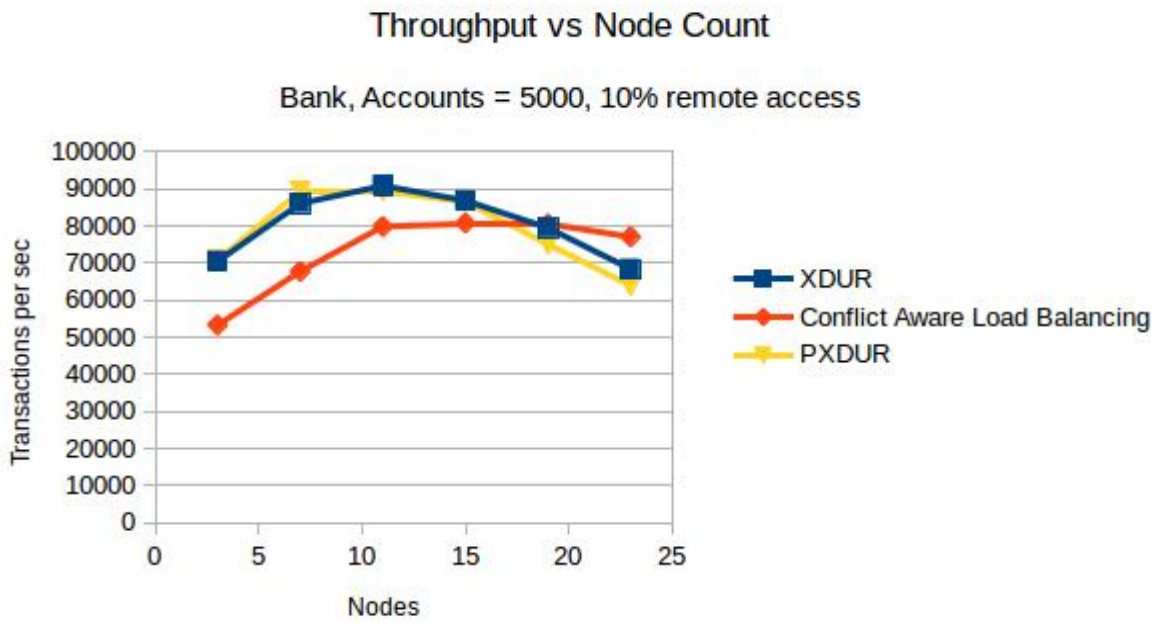

Figure 4.8: Bank 10% Remote Access

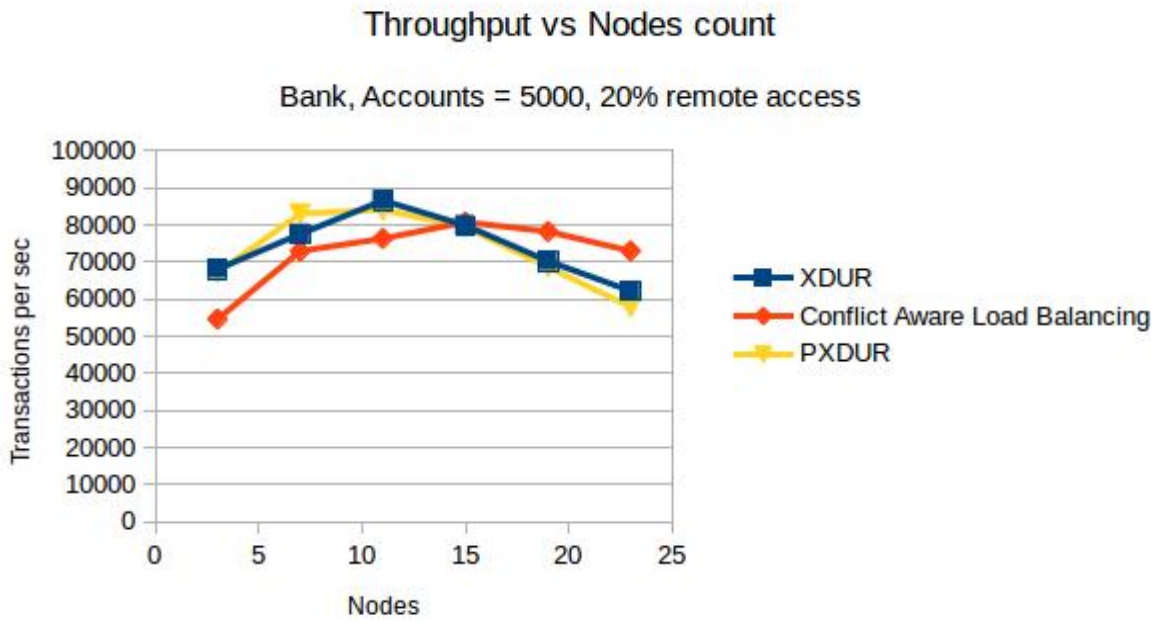

Figure 4.9: Bank 20% Remote Access

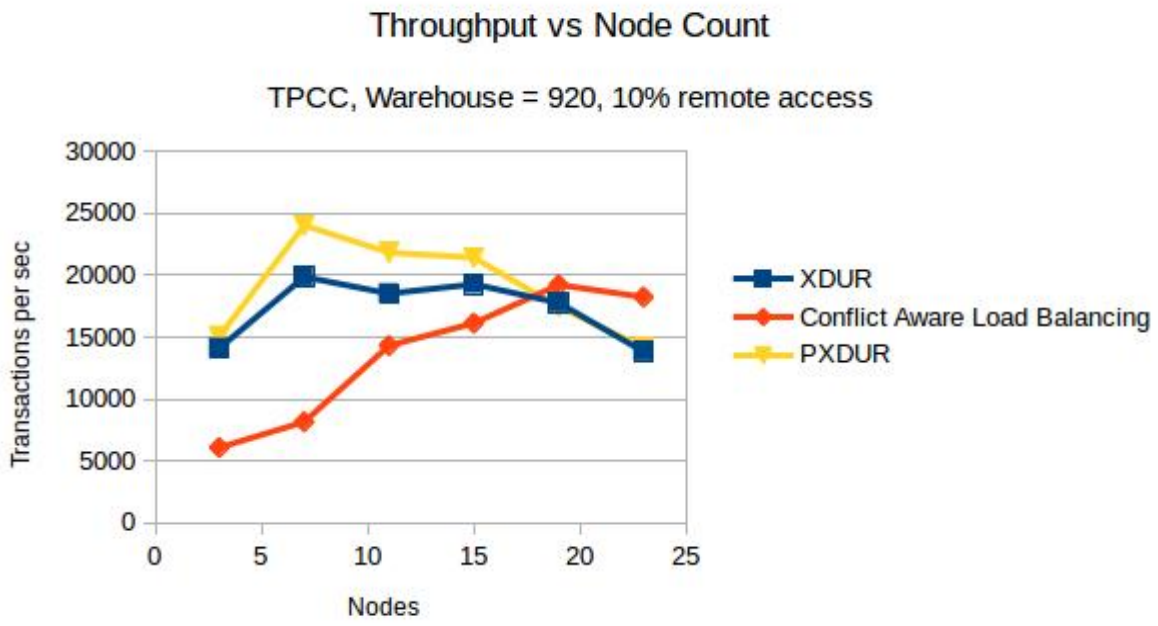

Figure 4.10: TPCC 10% Remote Access

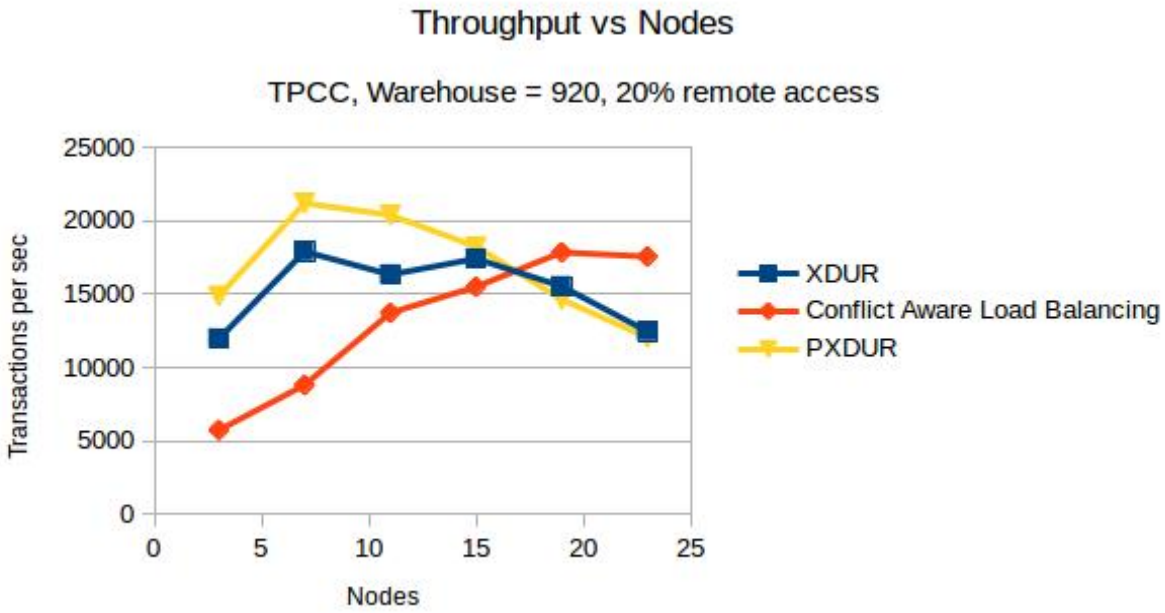

Figure 4.11: TPCC 20% Remote Access

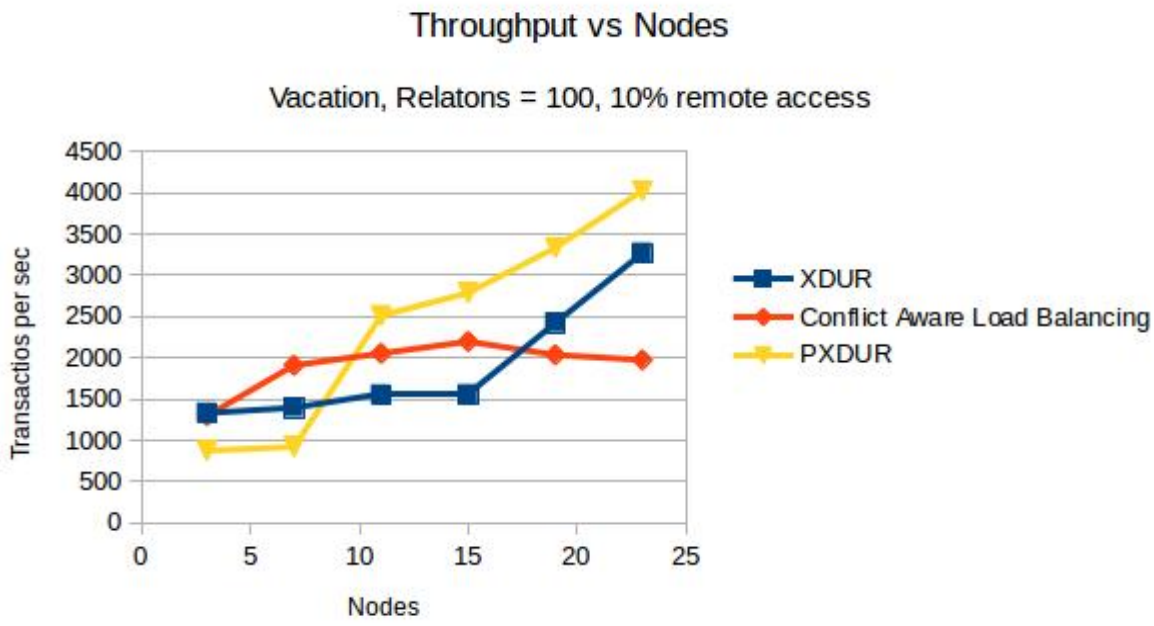

Figure 4.12: Vacation 10% Remote Access

of the three implementations where 10% of transactions are configured to access remote objects. Figure 4.9 gives the same for the case where 20% of the transactions can access remote objects. Both XDUR and PXDUR show a decrease in overall throughput with the increase in remote access. For both the cases, PXDUR and XDUR give similar throughput for 3 – 15 node cases with PXDUR showing a small performance gain for the 7 node case. XDUR outperforms PXDUR for 19 and 23 node cases. The competitor is able to behave faster than both XDUR and PXDUR for the 19 and 23 node cases for 10% remote access scenario and 15, 19 and 23 node count for the 20% remote access scenario. For the 23 node case, the competitor shows a throughput gain of 17% over the next best (XDUR).

Both XDUR and PXDUR allow speculative execution, therefore even one transaction that aborts due to remote contention can trigger a chain of local aborts along the line of transactions reading speculatively committed versions of objects involved in remote contention. For PXDUR, this cascading effect can be more severe as compared to XDUR since PXDUR's parallel nature allows for a higher number of speculative transactions to commit per unit of time therefore a remote abort can abort a potentially larger number of speculatively committed transactions than in XDUR. When contention between nodes is possible, the competitor has two distinct advantages over both XDUR and PXDUR implementations: *1)* The competitor does not forward speculative versions of objects committed by transactions awaiting global order, thus an abort due to remote contention does not propagate to other transactions; *2)* The competitor performs a read-set validation test for transactions committing locally before

they are submitted to the ordering layer. Thus many transactions which are rendered invalid due to remote contention are caught during the local read-set validation without burdening the ordering layer. On the other hand, a transaction aborting due to remote access in the case of XDUR and PXDUR can be only detected during the final read-set validation. These two reasons explain the observation in which the competitor outperforms the XDUR and PXDUR with increasing margins with the increase in the number of remote accesses.

Figures 4.10 and 4.11 depict the performance of PXDUR, XDUR and the competitor for the TPC-C benchmark under the scenarios where 10% and 20% of transactions can access remote objects respectively. In this case, PXDUR outperforms both the XDUR and the competitor for 3 – 15 node cases for both contention scenarios. PXDUR gives significant performance gain over XDUR in both the cases going in the range of 21 – 24%. Similar to Bank, the competitor gives the best throughput for the 19 and 23 node cases and the performance gain is higher for the 20% remote access scenario. The results show that the performance improvement provided by PXDUR due to faster local processing of speculative transactions offsets the performance degradation due to remote contention by a significant margin so as to still provide a performance gain of 20% over the next best implementation. The competitor's performance improvement over the speculative implementations can be explained as in the case of Bank.

Figure 4.12 shows the performance of all the three implementations for the Vacation benchmark under the scenarios where 10% of transactions can access remote objects. Vacation is the only benchmark where remote contention is studied under high contention, hence the difference in trend. In this case, the competitor dominates the other two for 3 and 7 node cases. PXDUR shows poor performance for 3 and 7 node cases and lies at the bottom but it performance picks up as the number of nodes increases and outperforms both XDUR and the competitor for 15 – 23 node cases. This behavior is explained by the fact that the local contention on the system is itself very high and 10% of transactions access objects from remote replicas. Thus a transaction accessing remote object is very likely to conflict with a local transaction on the given remote node. If the total number of nodes is low, the quorum sizes are smaller leading to a lesser load on the network. Due to low network load under low node count, more transactions reach the commit phase early resulting in either retrial (in case of abort) or a new request from the given client (in case of commit). This results in the system running near (or at) its maximum capacity leads to a high number of active transactions in the system. Due to the high contention, a higher number of active transaction entails a higher abort percentage and a lower throughput. On the other hand, as the node count increases, the load on the network increases for the same overall client count. As a result, lesser transactions reach their commit phase per unit of time and the system runs at a lower capacity. This results in lesser number of transactions running in parallel. Since the contention is high, a smaller number of active transactions means lesser number of conflicts, leading to lesser number of aborts and greater number of commits. Thus the system throughput actually goes up with increase in number of nodes. Since the underlying structure of PXDUR is similar to XDUR, also XDUR shows a very similar trend.

The competitor performs the best for 7 and 11 replica cases and its performance does not change significantly with changes in node count. The primary reason for this is the early detection of many remote conflicts as explained for the Bank benchmark.

Chapter 5

TSAsR : Timestamp Based AsR

This chapter presents `TSAsR`, which aims to improve the performance of state-of-the-art Database Management Systems (DBMS) under concurrent access. The eager locking based concurrency control mechanisms in DBMSs tend to be too conservative and do not allow the system to utilize the full potential of multi-core hardware. One way to address this disadvantage is to move towards more optimistic concurrency control methods, which allow the transactions to proceed and detect the inconsistencies later. However, this will require a re-design and overhaul of the concurrency control in DBMSs, which may introduce new overheads. This work proposes a solution that can guarantee serializability by leveraging the weaker isolation levels defined by ANSI/ISO SQL specification [2], already provided by most DBMSs (i.e Read Committed). However, this approach requires the addition of a certifier which can be applied on top of the DBMS's default concurrency control to identify and commit only serializable transactions.

5.1 The Protocol

Time stamp based As Serializable transactions or TSAsR make use of a timestamp based serializability certifier in the AsR framework, which rests on top of the DBMS's concurrency control. Serial-Safety Net (SSN) [51, 52] is an efficient general-purpose certifier, which can enforce serializability on top of various concurrency control schemes. Just like visible reads, SSN does not control database access, instead it tracks dependencies among transactions and prevents those transactions from committing which might close a dependency cycle. SSN requires the underlying concurrency control mechanism to provide at least Read Committed isolation level along with the prevention of lost updates.

SSN works by determining the low and high watermarks of a transaction. SSN provides every transaction with a commit time stamp at the beginning of its commit phase. During the commit SSN determines the timestamp of the transaction's earliest successor and that of the

latest predecessor. The earliest successor of a transaction represents write-anti-dependencies where a transaction overwrites an object read by another transaction but commits first. Such a successor represents a back-edge in transaction dependency graph. The predecessor of a given transaction is a transaction from which the current transaction reads or a transaction which reads the objects overwritten by the current transaction. SSN determines the lowest successor timestamp and the highest predecessor timestamp when a transaction commits. Then it performs a simple exclusion test to decide whether the transaction's commit will result in a cycle in the dependency graph.

As compared to AsR, TSAsR is very effective for the cases where the workload shows at least a moderate contention level. When the transactions are mostly non-conflicting there is no actual contention on the meta-data used (e.g., locks), therefore the additional processing required by the certifier adds to overhead. Considering the extreme case of transactions accessing disjoint parts of the database, they could (in principle) run without locking the accessed objects. In such a scenario, there is no execution schedule that TSAsR allows but the original concurrency control does not. However, as shown in the evaluation study, the limited overhead of TSAsR enables the achievement of similar performance to the original concurrency control in such scenarios. That makes TSAsR a concrete alternative to Serializability in both favorable (with contention) and adverse (without contention) scenarios.

Berkeley DB Java [41], the well-known and widely used open-source DBMS, was utilized as the basis for TSAsR, with the SSN processing added to the transactional operations. The system can be easily configured to run under TSAsR rather than Serializability, which means that programmers can execute all existing applications using this altered Berkeley DB version without any modification to the application source code, counting upon the same guarantees provided by the native Serializable concurrency control. To evaluate TSAsR, two well-known database benchmarks (i.e., TPC-C [8] and TPC-W [12]) and one synthetic benchmark (i.e. Bank) were used. Results confirmed the benefits in high-contention scenarios and demonstrated that TSAsR provided an improvement in throughput of multiple orders of magnitude.

5.1.1 Consistency and Isolation levels

The isolation level of the system defines the level of consistency observed by the application. Isolation defines the point when the changes made by a transactions become visible. In particular, there are four general levels defines as the standard for DBMSs. However there can be other isolation settings as well. Figure 5.1 presents the four isolation levels defined by ANSI/ISO SQL specifications. The isolation levels are represented as circles whose area represents the schedules permitted by the given isolation level. The fact that each circle lies within the previous circle and encloses lesser area is indicative of the lesser number of schedules allowed as the isolation level is tightened.

The outermost circle represents `Read Uncommitted` isolation level. At this level, there are

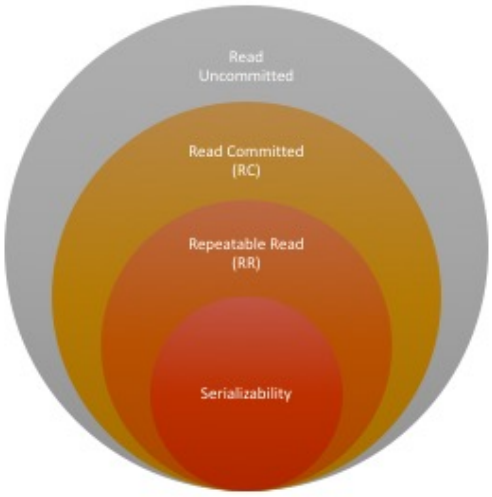

Figure 5.1: DBMS isolation levels

essentially no constraints on transactions. There is no explicit synchronization between transactions. Thus a transaction can read the data modified by another transaction even before the writer transaction commits. This isolation level allows `dirty reads`, which can occur when a transaction reads the data that is not permanent. For example a modification made by another transaction, which aborts in future.

The next inner circle represents the `Read Committed (RC)` isolation level. Under RC, transactions keep write locks on the data they modify. The locks are only released when a transaction commits or aborts. Thus no transaction can read the data written by a running transaction thereby preventing dirty reads. RC allows `non-repeatable read` and `lost write` anomalies. Since a transaction does not lock the object it reads, it is possible that an object read earlier by a transaction may be modified if a transaction re-reads it at a later time. This is called a `non-repeatable read`. Similarly a transaction may want to write an object it read earlier and it is possible that the object may have been modified between the read and write. Since the transaction will not read the new version of the object at the time of write, its modification will essentially mean that the previous update never happened at all. This is called a `lost write` or `lost update` anomaly.

The next inner circle represents the `Repeatable Read (RR)` isolation level. It prevents both non-repeatable read and lost write by taking read-locks on objects read, for the transaction's

lifetime. The read-locks can be shared to increase concurrency while the write-locks are exclusive. The only anomaly permitted by RR isolation level is `phantom read`. This anomaly can only occur during range queries when insertions are possible. Since locks are taken on individual objects, another transaction could insert a record in a range being read by another transaction.

The innermost circle represents the highest isolation level `Serializability`, which makes the transactions operate in isolation. Serializability prevents phantom reads, however different database implementations employ different methods to provide serializability. Each higher isolation level automatically prevents the anomalies prevented by lower levels.

5.1.2 Serial Dependency Graphs

As a transaction accesses shared object, serial dependencies are generated which constrain its place in the global partial order of transactions. Serial dependencies can take two forms:

Ti T : T accessed a version created by Ti, thus T must be serialized after Ti. This is a `read-write` dependency.

T Tj : T read a version that Tj overwrote, thus T must be serialized after Tj. This is a `read-anti-dependency`.

A read implies a dependency on the transaction that produced the read version, and an anti-dependency on the transaction that will overwrite the read version. A write implies a dependency on the transaction that wrote the previous version and on all those readers who read the new version. The relation T U represents a serial dependency of either case i.e both types of dependencies. T is the predecessor of U and U is the successor of T. The set of all serial dependencies between committed transactions forms the edges in a directed graph G, whose vertices are committed transactions and whose edges indicate required serialization ordering relationships. When a transaction commits, it is added to G, along with any edges involving previously committed transactions. T may also have po-tential edges to uncommitted dependencies, which will be added to G if/when those transactions commit.

We define a relation for G, such that Ti Tj means Ti is ordered before Tj along some path through G (i.e., Ti . . . Tj). We say that Ti is a predecessor of Tj (or equivalently, that Tj is a successor of Ti). When considering potential edges, we can also speak of potential successors and predecessors. These are transactions for which the potential edges (along with edges already in G) require them to be serialized after (or respectively before) T . A cycle in G produces Ti Tj Ti , and indicates a serialization failure because G then admits no total ordering. The SSN algorithm depends upon the relationship between the partial order of transactions defined by G and their total order defined by their commit times. The commit time of a transaction T is a monotonically increasing time stamp c(T) which is given to T when it starts the commit phase. An edge in G is a forward-edge when the predecessor committed first in real time and back-edge when the successor committed first. A forward

edge can be a read/write dependency while a back-edge is always a read anti-dependency.

5.1.3 Cycle prevention in SSN

In addition to the commit timestamp c(T) of transaction T, SSN associates T with two other timestamps: $\Pi(T)$ and $\eta(T)$, which are respectively the low and high watermarks used to detect conditions that might indicate a cycle in the dependency graph G, if T is committed. We define $\Pi(T)$ as the commit time of Ts oldest successor U reached through a path of back edges. $\Pi(T)$ can be computed only from the immediate successors of T in G, without traversing the whole graph. If a predecessor of T exists say transaction U, such that $c(T) <= \Pi(T)$, then it is possible that transaction U could be both a predecessor and a successor of T, since U committed before T's earliest successor. Thus if transaction T is committed, it can lead to a possible cycle in G. This check $c(T) <= \Pi(T)$ is called an exclusion window check. The determination of exclusion window can be further simplified by noting : 1) Only those predecessors needed to be considered which committed before T. 2) Among those predecessors, only the one with the highest time stamp needs to be considered for the exclusion window check. $\eta(T)$ represents transaction T's high watermark or the highest commit stamp among the T's predecessors. The exclusion window check finally becomes : $\eta(T) <= \Pi(T)$.

5.2 System Overview

Before going into the lower-level details for processing transactional executions, we must first give a brief overview of typical database structures. In abstraction, a database index is generally seen as a hash map, where the keys are IDs of some comparable variable type that is used to retrieve the associated data. In actuality, many implementations of databases utilize a B+ Tree [6, 42] . There may be multiple entries and divisions utilizing the object keys in each node of the tree, to allow efficient searching. At the bottom of the tree is a list of all items currently in the structure, sorted in ascending order. The nodes above the list encompass sub-sets of the data.

For transactions to access objects in the tree, they must navigate via the nodes using latches, which are short-term synchronization points that are allocated to threads to prevent others from reaching their position. In terms of the database, transactions will latch hand-over-hand, securing the next node while they still hold the current one, which will prevent other transactions from passing and potentially reaching the same data earlier than they do. Once transactions get to the bottom of the tree, they may read the data encompassed by the current slot they are looking at, or may add data to an empty slot in that leaf nodes storage.

5.2.1 TSAsR without range queries

Transactions in TSAsR start at Read Committed (RC) isolation which prevents dirty reads. In order to do so, write locks are placed on items when a transaction modifies them, thus preventing concurrent transactions to access them until the lock holder commits or aborts. RC does not prevent lost updates which is essential for SSN's working. Thus additional functionality is added to the write operation to prevent the lost updates.

The basic protocols of SSN require space and computation linearly proportional to a transactions footprint. SSN requires meta-data tracking across object versions. It was originally designed for a multi-version system. When applied to a system where objects have a single version, additional arrangements need to be made to simulate object versions. Constant space is required to store the meta-data representing each version. SSN summarizes dependencies between transactions using various timestamps that correspond to commit times. The timestamps are maintained in the shared object versions without a need to remember the committed transactions that influenced them. SSN supports early detection of exclusion window violations, aborting the transaction immediately if the arrival of a too-new (too-old) potential predecessor (successor) dooms it to failure.

The SSN protocol requires some meta-data involves some notations which are described here. For a transaction txn, following quantities are required:

- `txn.cstamp:` It represents the transaction end time. It is a unique time stamp which a transaction acquires when it begins commit. It is denoted as c(txn)
- `Txn.pstamp:` This is the predecessor high watermark or the timestamp of the txn's latest predecessor. It is also denoted as $\eta(txn)$
- `Txn.sstamp:` This is the successor low watermark or the timestamp of the txn's earliest successor. It is also denoted as $\Pi(T)$.

For an object obj, following meta-data is defined:

- `Obj.cstamp:` It represents the object creation time. It is the timestamp of the latest writer which installed the current version of the object. It is denoted as c(Obj).
- `Obj.pstamp:` This is the version access stamp. It represents the timestamp of the latest transaction that accessed the given object. It is denoted as $\eta(Obj)$.
- `Obj.sstamp:` This is the version successor timestamp. It is also denoted as $\Pi(Obj)$.

Suppose a transaction T created a version O and transactions R and W respectively read and over-wrote O. Then we can define $c(O) = c(T)$, $\Pi(O) = \Pi(T)$ and $\eta(O) =$ max timestamp among the transactions who read O and committed before T. In this case it is c(R).

Concurrent hash maps are attached to each individual database in an application, where the maps connect the primary keys of data objects to individual `time stamp blocks (TSBlock)`. `pStamps` and `sStamps` are two concurrent hash maps which simulate the multi version objects. Object versions are represented by commit stamps of the transactions which write them. Whenever a transaction which writes an object commits, a new element is added in

both the maps with the commit stamp serving as the key. The pstamp or the sstamp of the new object version is updated as per the SSN protocol described later. Thus a get/update operation on the sStamps or pStamps map is always done for a particular commit stamp (or a particular object version) which acts as the lookup key. The `readers` set contains the transaction IDs of the active readers. The `writer` field contains the Id of the active writer. The `cStamp` field contains the commit stamp of the last writer. The `locked` field is used for locking the TSBlock during updates. Each transaction calculates its pStamp and sStamp from the objects it has accessed and checks for the exclusion. Transactions need to determine their respective low and high watermarks at the time of commit. These watermarks are determined by looking at corresponding time stamps of the objects read and written. Thus, every transaction context has a read-set and a write-set which record the objects the transaction reads and writes respectively. These sets are implemented as hashmaps where the key is the object's Id and the value is the commit timestamp of the version accessed. It is important to store the commit stamps as they help in prevention of `lost write anomalies` as described later.

The modified transactions can be better understood through the read, write and commit operations described in Algorithm 1,2 and 3 respectively.

We begin by observing the `read` operation. The variables required from the application are the transaction performing the operation `Tx` and the key representing the object `Objkey`. First, the transaction must retrieve the meta-data (i.e., the TSBlock) associated with the object, or it must create a new one if the object does not exist (line 4.2). In order to avoid the anomalies arising from concurrent access of the shared objects, transactions must lock the TSBlock for mutual exclusion (line 4.3). Next, the transaction updates its pstamp to be the maximum of txn.pstamp and the object's commit stamp. If the object's current version's sstamp is valid, transaction's sstamp is updated to the minimum of txn.sstamp and object's sstamp(line 4.6) . The transaction marks itself in the object's readers array and the object is added to the transaction's read-set. An exclusion check is also done to see if the current transaction needs to be aborted (lines 4.11 - 4.13). If Tx passes the exclusion check,actual database read takes place (line 1.15). The TSBlock tsb is unlocked after the read.

Next we observe the `write` operation. Similar to the read operation, the TSBlock tsb is retrieved and locked at the beginning. This is followed by a check for `lost update` (line 5.5 - 5.12).If Tx is writing an object X it read previously, it is possible that some other transaction Ty may have overwritten the read version causing the version read by Tx to be obsolete. If Tx goes on to write X without retrieving the updated version, it will seem that Ty never wrote the object X. To avoid this, the write operation ascertains that if the transaction Tx is going to write an object it read previously, the version it read is the current version of the object. This is done by comparing the object's commitstamp stored in Tx's readset with the object's current commitstamp (tsb.cstamp). If a mismatch occurs, a lost update is indicated and Tx aborts. Otherwise, Tx's pStamp is updated to the maximum of the Tx.pstamp and the object's pstamp (line 5.14). The transaction marks itself as the current writer of the object. If the object is already in the transaction's read set, it is removed

and added to the write-set(line 5.18). Similar to the read,the exclusion check is done to see if the current transaction needs to be aborted (lines 5.19 - 5.21). If Tx passes the exclusion check,actual database write takes place (line 5.15). The TSBlock tsb is unlocked after the database write.

Algorithm 4 Read Operation

```
1:  Procedure : Read Operation
    Input: Tx, Objkey
2:  tsb = getOrInsert(Objkey,new TSBlock)
3:  tsb.lock()
4:  Tx.pstamp = max(Tx.pstamp,tsb.cstamp)
5:  if (tsb.sStamps[tsb.cstamp] is invalid) then
6:      Tx.sstamp = min(Tx.sstamp,tsb.sStamps[tsb.cstamp])
7:  end if
8:  Add Objkey to Tx.Readset
9:  Add Tx.Id to tsb.readers
10: ▷ Check for exclusion
11: if ( Check_Exclusion(Tx) = false) then
12:     Tx.abort()
13:     Return null
14: else
15:     Database Obj = db.get(objKey)
16:     tsb.unlock()
17:     Return Obj
18: end if
```

Algorithm 5 Write Operation

```
1: Procedure : Write Operation
   Input: Tx, Obj
2: tsb = getOrInsert(Objkey,new TSBlock)
3: tsb.lock()
4: ▷ Check for lost updates
5: if (Tx.Readset contains Objkey) then
6:     ▷ Get the commit stamp of the object's version read by Tx
7:     RSCstamp = Tx.Readset.get(Objkey)
8:     if ( tsb.cstamp ≠ RSCstamp) then
9:         ▷ The version in the read-set does not match the current version,thus abort Tx
10:        Tx.abort()
11:    end if
12: end if
13: tsb.writer = Tx.Idt
14: Tx.pstamp = max(Tx.pstamp, tsb.pStamps[tsb.cstamp].pstamp)
15: if (Tx.Readset contains Objkey) then
16:    Tx.Readset.remove(Objkey)
17: end if
18: Tx.Writeset.insert(Objkey)
19: if ( Check_Exclusion(Tx) = false) then
20:    Tx.abort()
21:    Return
22: else
23:    db.put(objKey)
24:    tsb.unlock()
25:    Return
26: end if
27: Return
```

Finally we talk about the `commit` operation where the crux of the algorithm lies. The parallel commit operation in SSN consists of two phases: 1) The `pre-commit phase`. This phase involves the determination of $\pi(Tx)$ and $\eta(Tx)$. The two watermarks are finalized to perform the exclusion test ; 2) The `post-commit phase`. This phase occurs after the transaction has passed the exclusion test. Tx then starts the post-commit phase to finalize creation of new versions it wrote and timestamps of existing versions it read.

Finalizing $\Pi(Tx)$: To finalize $\Pi(Tx)$ a transaction needs to determine its successor's timestamp which was earliest to commit. To accomplish this, transaction Tx iterates over the read-set to determine the minimum sstamp among the objects it has read (lines 6.5 - 6.19). It is possible that for an object Obj in Tx.Readset, there may be parallel writer Ty who is also in its pre-commit phase and has taken a commit stamp earlier than Tx. Such a transaction will alter the Obj.sstamp and can also affect $\Pi(Tx)$. Therefore Tx will have to

wait for Ty to finish (commit or abort). If Ty commits, Tx then updates Tx.sstamp with Ty.sstamp to finalize its successor low-watermark (line 6.14). The writer field in an object's TSBlock stores the Tid of the current writer. It can be used the get the current writer's context from the transaction table, so that it can be monitored.

Finalizing $\eta(Tx)$: To finalize $\eta(Tx)$, a transaction needs to determine the timestamp of its latest predecessor committed before itself. Tx iterates over its write-set to determine the maximum Obj.pstamp among the objects in its write-set. An object Obj could only have at most one successful overwriter . Before Tx enters pre-commit, however, multiple concurrent readers could have read Obj during Tx s lifetime. Thus, Tx must wait for all the parallel readers which have acquired a lower commit timestamp to finish. Once a parallel reader commits, Tx updates its pstamp with the reader's pstamp. To accomplish this, the transaction's write-set is traversed and concurrent readers are monitored using the TiD from the the object's readers array (lines 6.24 - 6.33). A point to be noted is that Obj.pstamp or Obj.sstamp for read/write-set objects are obtained through their respective TSBlocks as shown in read and write operations. Here the TSBlocks are not locked.

Once $\pi(Tx)$ and $\eta(Tx)$ are finalized, the exclusion test is performed to check if it is safe to commit Tx. Once the transaction has passed the exclusion test (i.e $\eta(Tx) > \pi(Tx)$), it begins the post commit phase . Tx tries to update the pstamp of its read-set objects. For every object in its read-set, it calls UpdatepStamp operation on the object's TSBlock. In UpdatepStamp operation, if the given object version's pstamp is less than Tx.cstamp, object's pstamp is updated (lines 6.44 - 6.49). Since tsb.pstamps is a concurrent data structure, no locking is needed. For the write-set objects, Tx simply updates the sstamp of every object to Tx.sstamp and adds a new entry to the object's tsb.pStamps and tsb.sStamps structures. The TSBlock needs to be locked for these operations to avoid conflict with parallel readers or writers.

Algorithm 6 Commit Operation

```
1: Procedure : Parallel Commit Operation
   Input: Tx
2: ▷ Get the next time stamp
3:  Tx.cstamp = next_timestamp()
4: ▷ Finalize  pi(Tx)
5: for (Objkey in Tx.Readset) do
6:     tsb = get(Objkey,new TSBlock)
7:     RScStamp = Tx.Readset.get(Objkey)
8:     TxWriter = tsb.Writer
9:     if ( TxWriter ≠ null ∧ TxWriter.cstamp < Tx.cstamp) then
10:        while ( TxWriter.status == INFLIGHT) do
11:            Spin
12:        end while
13:        if ( TxWriter.status = COMMIT) then
14:            Tx.sstamp = min(Tx.sstamp,TxWriter.sstamp)
15:        else
16:            Tx.sstamp = min(Tx.sstamp, tsb.sstamps[RScStamp].sstamp)
17:        end if
18:    end if
19: end for
20: ▷ Finalize  eta(Tx)
21: for (Objkey in Tx.Writeset) do
22:    tsb = get(Objkey,new TSBlock)
23:    WScStamp = Tx.Writeset.get(Objkey)
24:    for ( TxReader in tsb.readers) do
25:        if ( TxReader.cstamp < Tx.cstamp) then
26:            while ( TxReader.status == INFLIGHT) do
27:                Spin
28:            end while
29:            if ( TxReader.status = COMMIT) then
30:                Tx.pstamp = max(Tx.pstamp,TxReader.pstamp)
31:                ▷ In case some reader was missed
32:                Tx.pstamp = max(Tx.pstamp, tsb.pstamps[WScStamp].pstamp)
33:            end if
34:        end if
35:    end for
36: end for
37: ▷ Exclusion check
38: if ( Check_Exclusion(Tx) = false) then
39:    Tx.abort()
40:    Return
41: end if
42: Tx.status = COMMITTED
```

```
43: ▷ Post commit phase
44: for (Objkey in Tx.Readset) do
45:     tsb = get(Objkey,new TSBlock)
46:     RScStamp = Tx.Readset.get(Objkey)
47:     if ( Tx.cstamp > tsb.pstamps[RScStamp].pstamp) then
48:         tsb.pstamps[RScStamp].pstamp = Tx.cstamp
49:     end if
50: end for
51: for (Objkey in Tx.Writeset) do
52:     tsb = get(Objkey,new TSBlock)
53:     WScStamp = Tx.Writeset.get(Objkey)
54:     tsb.sStamps.[WScStamp].sstamp = Tx.sstamp
55:     tsb.pStamps[Tx.cstamp].pstamp = Tx.cstamp
56:     tsb.cstamp = Tx.cstamp
57: end for
58: Return
```

Chapter 6

TSAsR: Evaluation

This chapter describes the experimental evaluation of TSAsR followed by the discussion on the experimental findings. TSAsR is implemented upon the BerkeleyDB Java database [41]. The testbed used is a 144 core machine which has 8 Intel Xeon processors, each with 18 cores running at 2.30GHz, and 128 GB of RAM. In order to remove the overhead caused by thread migration across sockets, we use a single socket consisting of 36 cores for the experiments. We vary the thread count from 1 to 32 and monitor the system throughput. Three commonly used transactional application benchmarks : TPCC [8], TPCW [12] and Bank were used to evaluate TSAsR. TPCC and TPCW are on-line transaction processing benchmarks which represent complex workloads. Bank is used because it is lightweight and presents a different workload type. In order to present more insightful and fair results, the evaluation disabled the possibility of having the phantom read anomaly. The reasoning for this decision is because phantom reads are tightly related to the presence of range queries and insert operations performed by the application. With phantom reads disabled, the strongest and sufficient isolation level provided is Repeatable Read; otherwise, it is clearly Serializability.

Unmodified BerkeleyDB and original AsR [38] were selected as competitors. From here, we shall refer `unmodified BerkeleyDB` as `BerkeleyDB`. Since the evaluation does not involve range queries, BerkelyDB was configured to use `repeatable read RR` isolation level. While both AsR and TSAsR were configured to start at `Read Committed RC` isolation level with upgrades after three aborts. We analyze the behavior of TSAsR under three contention levels: low, medium and high. Each differs from the other with respect to the total number of shared objects available. Table 6.1 summarizes the configuration used.

As a general trend, the Berkeley DB system does not show great scalability even in no-contention cases. Results show only a slight increase in the system throughput with the increase in thread count. The system seems to saturate quickly due to some limited internal resources causing additional overhead. As the number of threads using the database increases, the time to read or write an item extends, even if contention between threads is

Table 6.1: TSAsR Configurations used for each benchmark.

Application	**Low contention**	**Medium contention**	**High contention**
TPCC (warehouses)	10	20	40
TPCW (Items)	50	150	400
Bank (Accounts)	500	1000	2000

very low.

Protocols like AsR and TSAsR find their best usage when there is some amount of contention in the system. With little or no contention the locking performed by transaction will have no conflict while there is additional processing overhead associated with the certification mechanism in TSAsR. This trend is also reflected from the evaluation results. For high and medium contention cases, TSAsR routinely outperforms both BerkeleyDB and AsR by a significant margin, especially for higher thread counts (12 and above). However both AsR and BerkeleyDB outperform TSAsR for smaller thread counts (1 – 4). At lower thread counts, BerkeleyDB benefits from the generally low contention on the locks in the system. Among TSAsR and AsR, TSAsR has a high commit overhead owing to the fact that a committing transaction needs to lock the meta-data of the objects it modifies. In comparison, AsR's visible read overhead is much lower for smaller thread counts. This helps AsR outperform TSAsR for lower thread counts. Lastly, the performance of both AsR and TSAsR goes down with the increase in thread count. This could be attributed to the fact that the certification overhead of both AsR and TSAsR tends to increase linearly with thread count.

6.0.1 TPCC

TPC-C [8] is a larger benchmark simulating stock warehouses with item orders and deliveries. The benchmark is initialized by creating a set number of warehouses, allocating districts, customers, items, and other information to each one individually. There are five main transactional operations that can be performed: i) placing new warehouse orders; ii) checking the status of orders; iii) updating system information and customer balances with order payment and confirmation; iv) checking the stock levels of warehouses and refilling as necessary; and, v) delivering the items required to fulfill batches of orders. The creation of new orders and the payment processing are the most frequent among the operations. The operational profile ensures that 92% of the transactions executed are write transactions. The contention in the benchmark can be altered by decreasing the number of warehouses available while increasing the number of user threads.

Figures 6.1, 6.2 and 6.3 represent the high, medium and low contention scenarios respectively. All the plots show that the throughput for BerkeleyDB decreases sharply as the number of

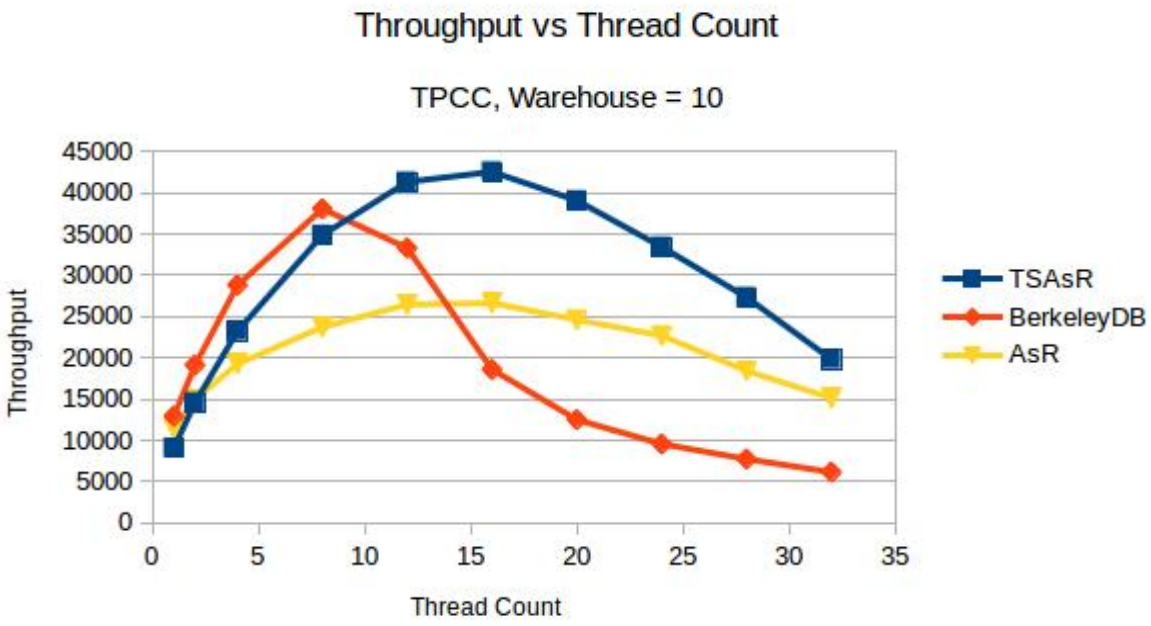

Figure 6.1: TPCC High contention

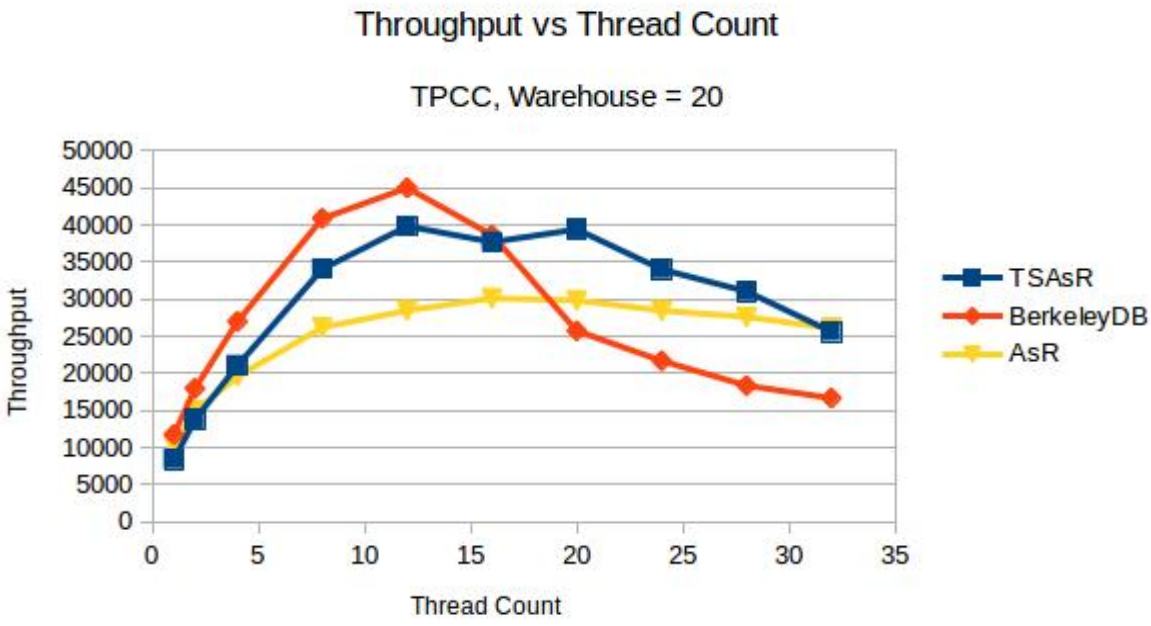

Figure 6.2: TPCC Medium contention

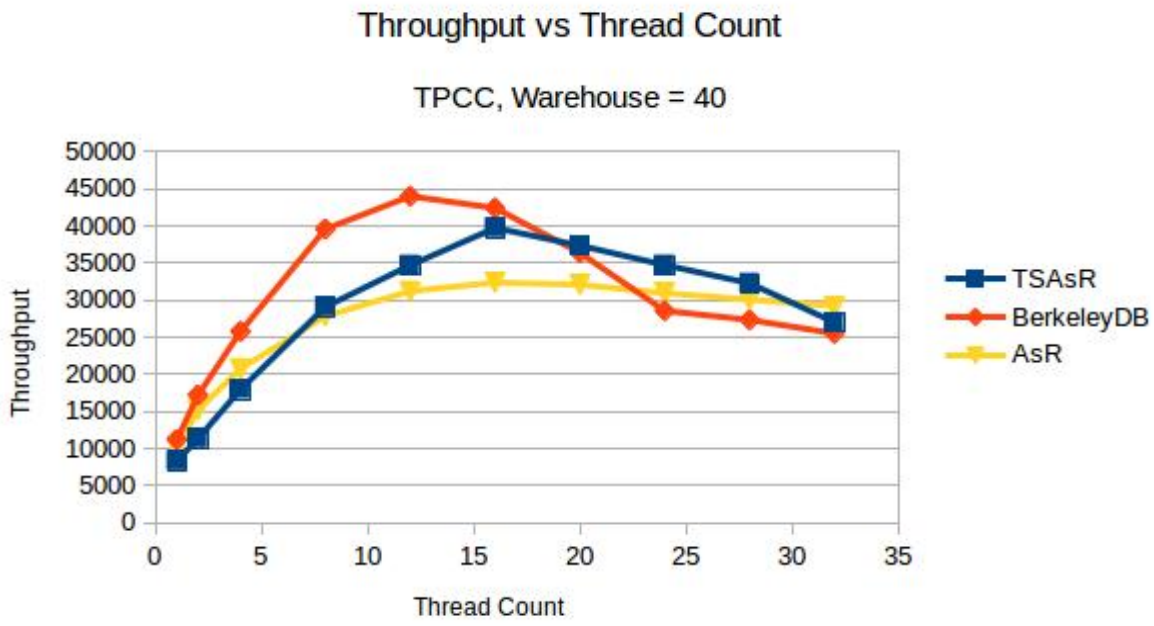

Figure 6.3: TPCC Low contention

threads increases beyond a limit, and it goes down at a slower rate with the increase in the number of warehouses. Both AsR and TSAsR show similar trends, with AsR only outperforming TSAsR for very low (1–4) or very high (32) thread counts. The figures show that the benefits of TSAsR over BerkeleyDB are visible only when there is atleast moderate contention in the system. The performance of BerkeleyDB tends to improve with the decrease in contention i.e when number of warehouses is high or number of user threads is low.

The high contention scenario shown in Figure 6.1 has the warehouse count fixed at 10. BerkeleyDB gives the best performance for low thread counts (1 – 8). However, as the thread count crosses the warehouse count, TSAsR starts to dominate. TSAsR gives the best performance in the for thread count 12 – 32. The best speedup is obtained around 16–24 threads. For the 20 thread case, TSAsR gives a speedup of more than 3X over BerkeleyDB and outperforms AsR by a margin of 60%.

The moderate contention scenario (Figure 6.2) operates with the warehouse count 20. As expected, BerkeleyDB's performance improves with the decrease in contention. BerkeleyDB dominates the other two for thread counts 1 – 16. TSAsR dominates the others for thread counts 20 and above. BerkelyDB's performance falls below both AsR and TSAsR as the thread count increases beyond 20. TSAsR dominates the other two for thread counts 20 – 28. The best improvement margin for TSAsR occurs for 20 threads case, where it outperforms AsR by 31% and BerkeleyDB by 56%.

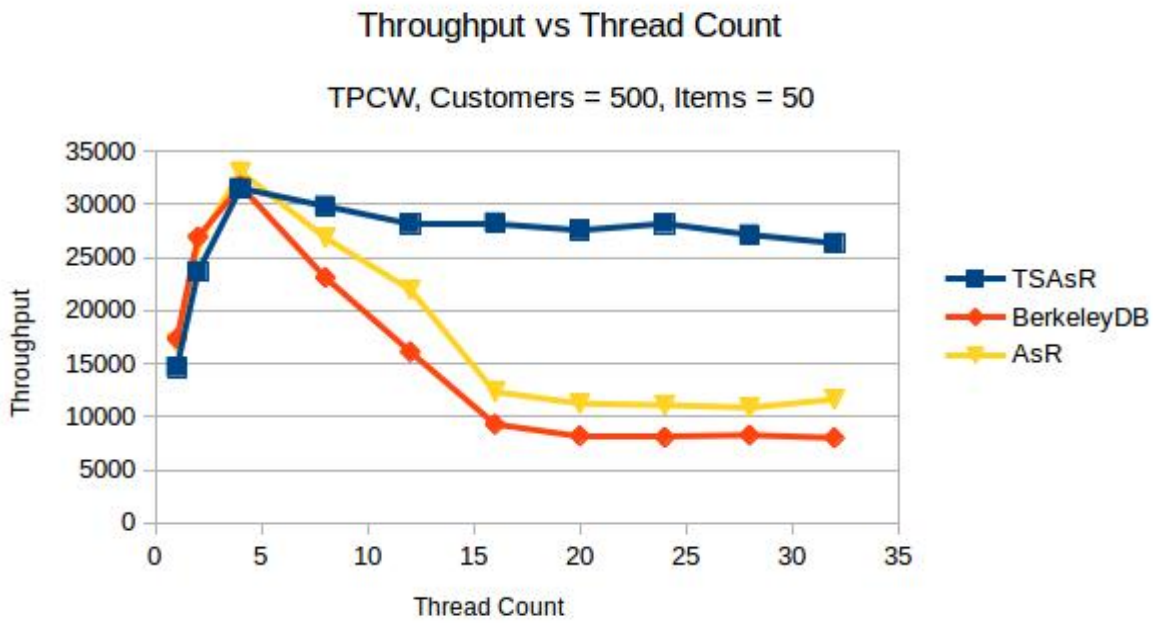

Figure 6.4: TPCW High contention

In the low contention case, (Figure 6.3), BerkeleyDB dominates the other two upto thread count 16. The margin of improvement is better than the high and medium contention cases, but that is to be expected. Its performance again dips below the other two for thread counts above 24, but the rate of performance decrease is much lower as compared to previous cases. TSAsR dominates the others for thread counts 20 – 28. The best improvement margin occurs at thread count 24 where TSAsR outperforms BerkeleyDB by a margin of 20% and AsR by 13%.

6.0.2 TPCW

TPC-W [12] is a benchmark that emulates a commerce website, and is similar to TPC-C but in a broader sense. There are 14 different types of transactions, that function as a user or maintainer navigating the website, looking at and purchasing objects, performing maintenance, etc. In terms of data, there are 8 different types of objects representing customers, items, orders, and so on. Transactions are completed one-by-one and leave a given state in the system in order to influence the next operations performed. For instance, if a customer just observed some search results, that state influences the chances to add the item(s) to their cart or to return to the query page or home page.

The parameters that influence the systems contention, besides the number of threads (i.e., active users and administrators), are the number of items available as well as the number of

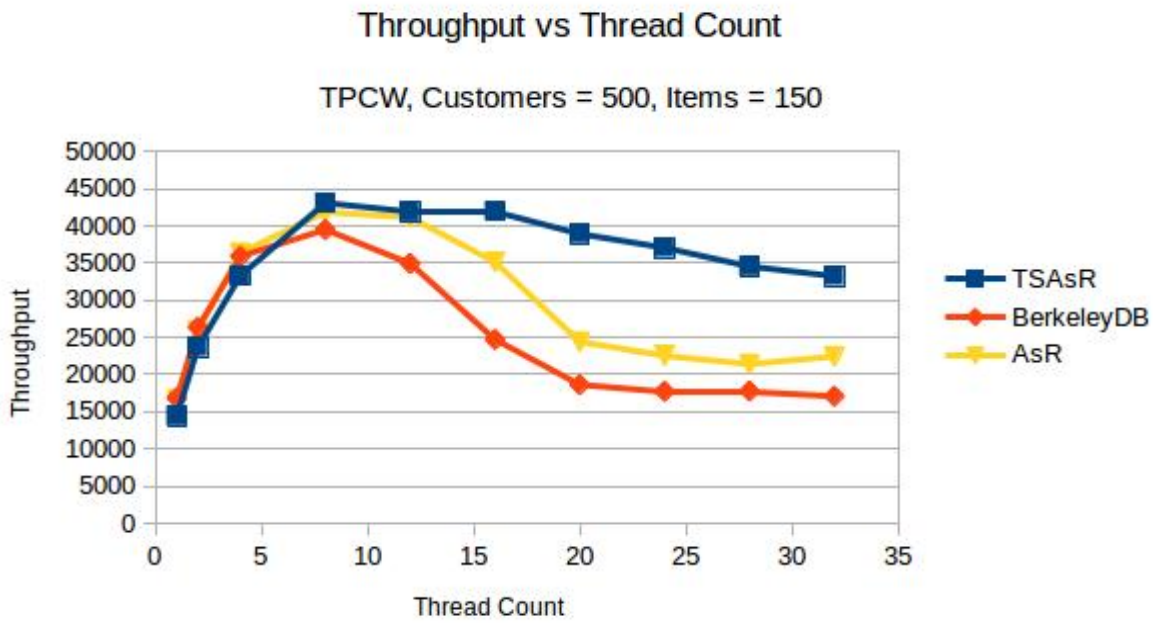

Figure 6.5: TPCW Medium contention

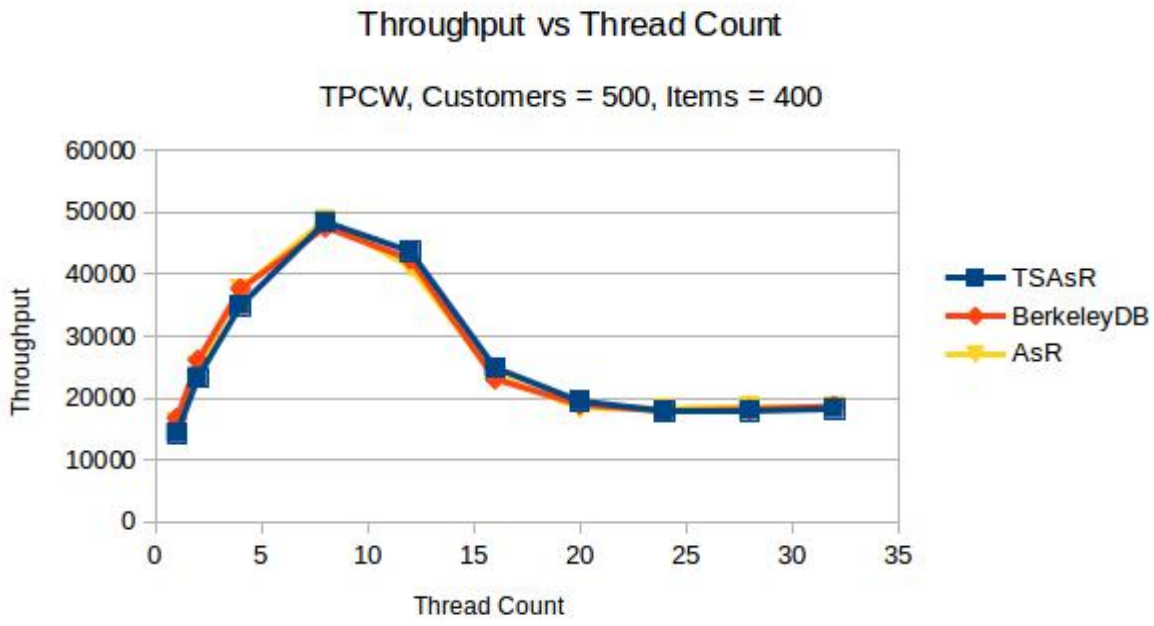

Figure 6.6: TPCW Medium contention

customers who have accounts with the website. One thread, until some endpoint is reached, is meant to simulate one customer or one administrator performing their necessary operations. The customer count does not represent the number of active threads. In this evaluation we fix the customer count to 500 and vary the item count as 50, 150 and 400 to represent the high, medium and low contention cases respectively.

Figures 6.4, 6.5 and 6.6 represent the high, medium and low contention cases for the TPCW benchmark, respectively. The trend is similar to TPCC in some respects. For all the cases, BerkeleyDB outperforms TSAsR for lower thread counts, while TSAsR tends to dominate for higher thread counts. However, the margin of improvement for BerkeleyDB in low thread count cases over TSAsR is much less as compared to TPCC. AsR seems to follow BerkeleyDB's trend however its performance degradation is lower with the increase in thread count as compared to BerkeleyDB.

The high contention scenario shown in Figure 6.4 has the item count fixed at 50. Both BerkeleyDB and AsR slightly outperform TSAsR for thread counts $1 - 4$. However, as the thread count increases beyond 8, TSAsR shows clear benefits. Both BerkeleyDB and AsR show significant performance degradation over thread count 8, but TSAsR showes only a gradual decrease outperforming the others by a significant margin. For the 24 thread case, TSAsR gives a speedup of 3.5X over BerkeleyDB and 2.5X over AsR.

The moderate contention scenario (Figure 6.5 operates with the item count 150. The trends are similar to the high contention case, but both BerkeleyDB and AsR show lower degradation rates. The degradation sets in later as well, occurring beyond thread count 12. TSAsR again gives the significant performance improvement over BerkeleyDB and AsR for thread counts 16 and above. It gives a speedup of 2.0X over BerkeleyDB and 1.5X over AsR for the 20 thread case.

In the low contention case, (Figure 6.6, all the three implementations tend to coincide, not showing any significant performance difference. This case serves to show that TSAsR's performance does not very significantly from the unmodified system even under low contention, when BerkeleyDB's performance does not degrade with increase in thread count.

6.0.3 Bank

The Bank benchmark is a simple implementation, and consists of only two main operations: balance check, which opens an account observes its value; and transfer, which withdraws money from one account and deposits it into another. In these tests, a write-intensive workload was used to show the general difference between the meta-data processing and traditional eager-lock based synchronization. The contention is managed by changing number of bank accounts available along with the number of user threads.

Figures 6.7, 6.8 and 6.9 represent the high, medium and low contention cases for the Bank benchmark, respectively. Like the earlier cases, all the plots show that the throughput for

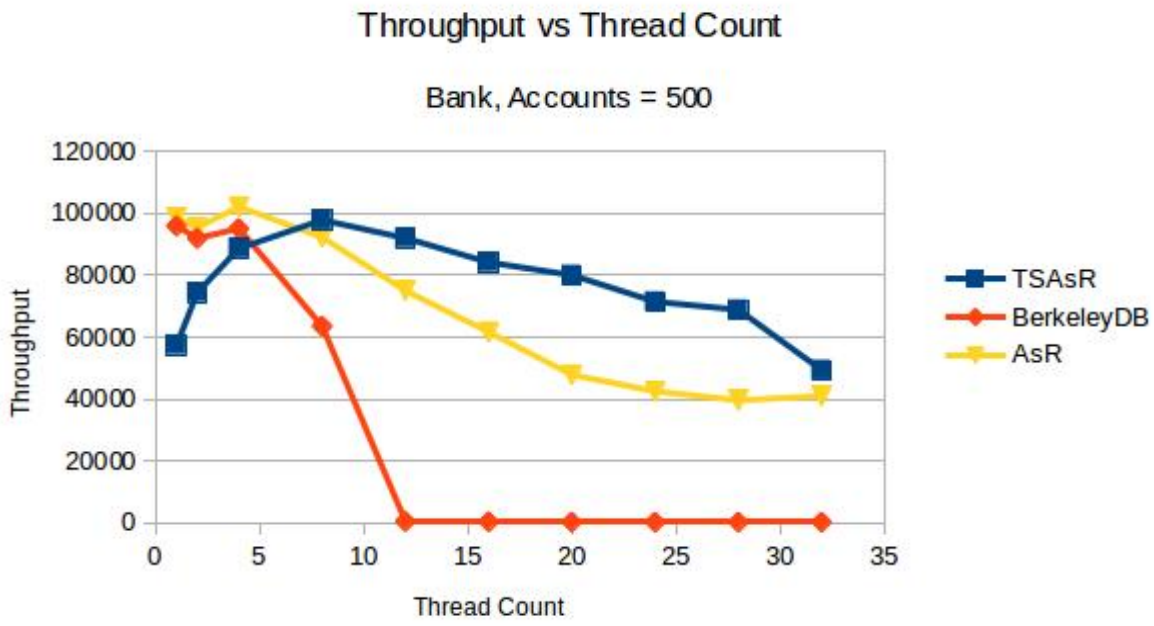

Figure 6.7: Bank High contention

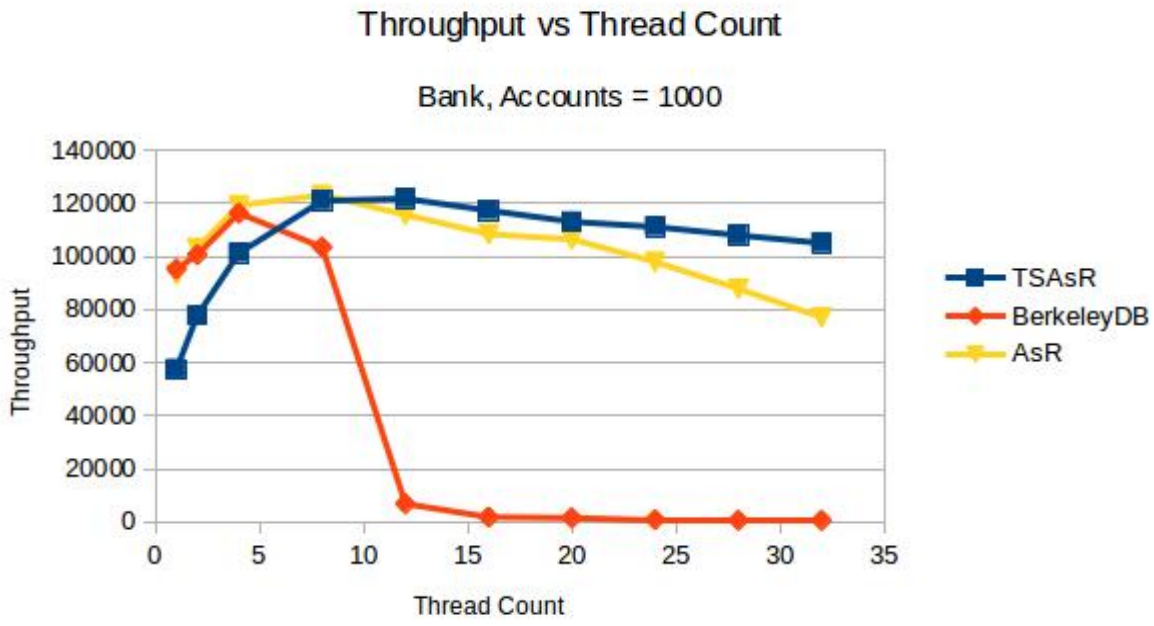

Figure 6.8: Bank Medium contention

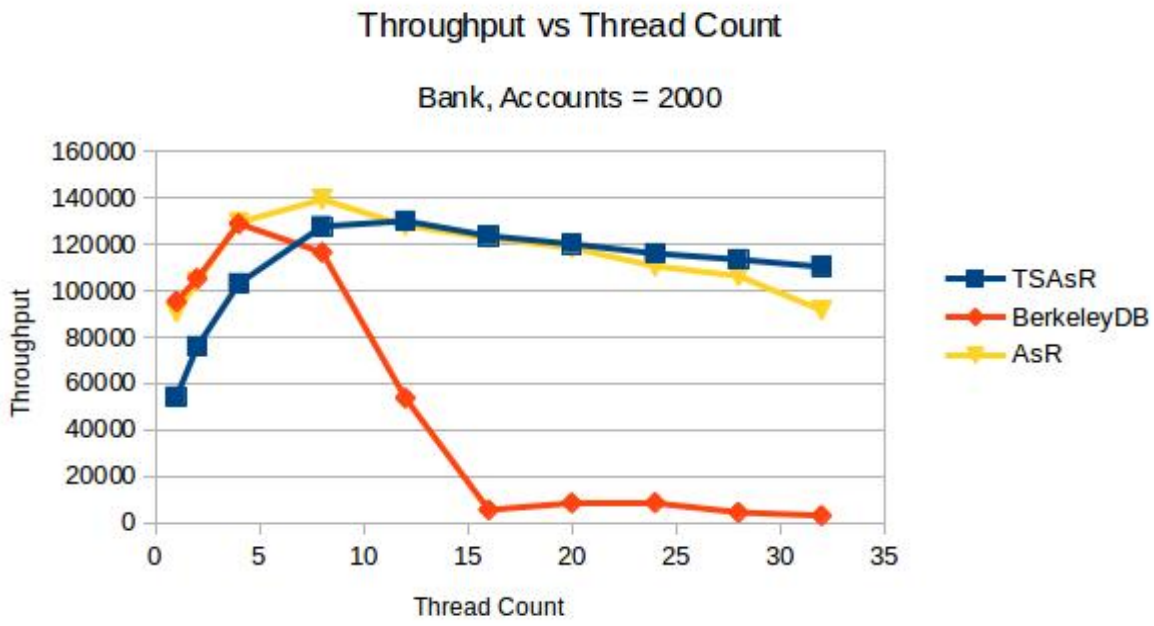

Figure 6.9: Bank Low contention

BerkeleyDB decrease sharply as the number of threads increases beyond a certain value. However, for Bank benchmark, BerkeleyDB's performance degradation is much higher as compared to TPCC and TPCW. It shows an increase in performance for thread counts 1–4, but there is a very sharp decrease in performance beyond thread count 8. For the low contention case, BerkeleyDB holds till thread count 12. AsR tends to outperform TSAsR and sometimes even BerkelyDB for the low thread count cases. It is possible that AsR's smaller visible-read overhead for low thread count may have be a sweet spot with Bank's lightweight transactions.

For the high contention case (Figure **??**, 500 Accounts), AsR and BerkeleyDB dominate TSAsR for thread counts 1–4. AsR is the best of the lot in this range and slightly dominates BerkeleyDB. TSAsR outperforms the other two for thread counts 12 and above. The biggest performance gain over the next best competitor AsR is about 66% – 69% for 20 and 24 threads.

For the medium contention scenario (Figure **??**, 1000 Accounts), the trend is similar to the previous case. In this case, TSAsR starts to dominate from thread count 12 and onwards however, the performance improvement is smaller as compared to the high contention case. The best improvement margin for TSAsR over the next best AsR is about 35% for the 32 thread case.

In the low contention scenario (Figure **??**, 2000 Accounts), BerkeleyDB's performance shows

a sharp drop after the thread count of 12. BerkeleyDB and AsR show similar performance for thread counts 1 – 4, after which BerkeleyDB's performance starts to fall. TSAsR catches up with AsR for thread count 12 and starts to dominate the three thereafter. The best performance improvement achieved by TSAsR over AsR is 20%, for the 32 thread case. The speedup of both AsR and TSAsR over BerkeleyDB for any thread count above 12 is order of magnitudes in all the case for Bank benchmark. This is attributed to the sharp degradation suffered by BerkeleyDB at higher thread counts.

Chapter 7

Run-time environment for verified distributed systems : Verified JPaxos

In this chapter we present Verified JPaxos, which involves the development of a run-time for a Multipaxos based distributed system that can be easily formalized. Multipaxos is a single leader paxos [26] based algorithm which tries to run several instances under a designated leader. First, the Mutipaxos algorithm is specified in HOL. Isabelle/HOL's code generator is used to generate Scala [40] code implementing the system model. The HOL generated Scala code represents the I/O automata based upon the Multipaxos specifications. The actual implementation details of the HOL modeling is out of the scope of this work. Here we assume that the HOL model for the algorithm is already developed and we intend to develop a run-time which utilizes the HOL generated scala code. The actual proving of the HOL generated code is also orthogonal to this work.

The run-time consists of Java and Scala classe and is added on top of the HOL generated code, which can drive the I/O automaton. The run-time interacts with the network layer and the service which runs on the server node. The service represents the user application running on the system. The run-time takes client requests from the service, batches them to increase performance, and utilizes the distributed algorithm to order them. Once ordered, the run-time notifies the service which can take appropriate action. The HOL generated code produces Multipaxos messages which need to be sent to other nodes. The run-time takes the responsibility of sending the messages to the correct destination nodes. It ensures no packets are duplicated or lost by using the TCP protocol for data transfer between nodes. The run-time also houses the failure detector [15] which is responsible for initiating the leader selection phase, if the current leader is suspected.

The run-time is based upon the design of JPaxos [25]. The main reason for this is performance. JPaxos is optimized for high performance and introduces features like batching to improve it further. A run-time based upon the same design principles as JPaxos is expected to provide similar performance benefits. The HOL generated system specification integrated

with the run-time forms a functional unit of a distributed system which can be deployed over a node cluster to create a fault tolerant service.

7.1 Isabelle

Isabelle [39] is a generic proof assistant which allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus. Isabelle provides formalization of mathematical tools and formal verification which finds its major application in proving the correctness of computer hardware, software, computing algorithms and protocols. HOL (Higher order logic) [50, 37] is currently the best developed Isabelle object-logic. HOL supports functional programming and it can be used describe system properties and also makes it easy to verify their correctness.

7.2 Jpaxos

Jpaxos [25] is a full-fledged high performance Java implementation of state machine replication based on Paxos [26]. It tends to bridge the gap between the theoretical and practical aspects of Paxos algorithm. JPaxos adds features like managing a finite replicated command log, ensuring that replicas stay close together, crash recovery, efficient utilization of the network and processor cores to increase system performance and stability. JPaxos uses Multipaxos [26] as the distributed algorithm. The advantage of Multipaxos over classical Paxos is that Multipaxos selects a leader for multiple instances. This is in contrast to Paxos which involves a leader selection phase leading to two extra communication steps per instance. Parallel instances and batching are two key features which also contribute towards the improvement of performance.

Batching involves accumulation of client requests in a batch before it is sent for ordering. Thus instead of initiating a Paxos instance per client request, this feature several client requests to be ordered per Paxos instance. This also reduces the network traffic as number of messages is reduced. JPaxos supports multiple instances running simultaneously. Higher number of instances means more requests batches getting ordered per second leading to performance enhancement.

JPaxos is parallelized to some extent so as to scale with the increase in the number of CPU cores. The large amount of shared states and asynchronous communication required in JPaxos make it easier to implement the system as a single thread, sidestepping all concurrency. However, such a system does not scale. Thus JPaxos is a hybrid system where the main event queue which triggers state transitions based on received messages, is single threaded. However, it communicates with some other modules through queues which have listener threads waiting on them. Thus, when it comes to concurrency, JPaxos tends to

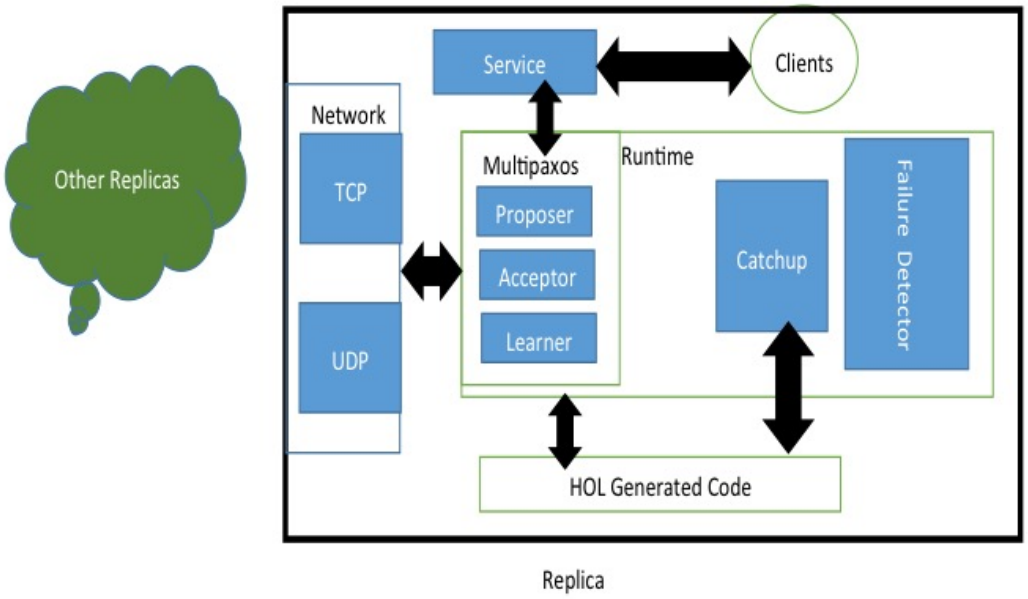

Figure 7.1: Block diagram of a node

compromise between implementation complexity and scalability.

JPaxos is a well documented practical implementation of Multipaxos. It has been studied well over past few years and has acceptable performance level. Therefore, the design principles used in the implementation of JPaxos can act as a guideline for producing an efficient run-time for Paxos based algorithm. These were the reasons due to which we selected JPaxos's run-time as the starting point. The run-time follows the JPaxos, as the event queue is single threaded and there are listener threads which wait for specific events. The state transitions occurring in JPaxos code are mirrored by the code generated from HOL specifications.

7.3 System Architecture

Figure 7.1 presents a block diagram of the system. The crux of the system is the system specification part which is modeled in HOL and the corresponding run-time code is generated in scala. This code represents the systems state and models state changes. In the context of Multipaxos, the HOL generated part specifies the initial state of the system. It consists

of functions which are called when a Multipaxos message is received. These functions (or handlers) take the corresponding Multipaxos message and the system's current state as input. They model the state change as a response to the message in accordance with Multipaxos specification and return the new state along with any messages which need to be sent to other replicas as a consequence of the state change.

7.3.1 System State

The system state consists of variables which describe a node in Multipaxos algorithm. It contains information like node's Id, the current ballot number (depending upon the leader's Id), pending instances etc. It also stores the command log which is used to keep track of the finished and running instances and to perform catchup and recovery.

7.3.2 Run-time

The HOL generated system describes the initial state and outlines the functions to cause state changes as per the modeled algorithm. The run-time is required to call these functions to effect state changes so as to constitute a running Multipaxos system. The run-time links all the components of a replica as shown in Figure 7.1. The run-time interacts with the HOL code to affect state changes, it communicates with the service to get the client requests and send back the replies. It interacts with the network layer to communicate with other nodes in the system. It also houses components like the failure detector and catchup handler.

The run-time consists of both Java and scala classes. This is done in part to reduce code complexity and in part to enhance performance. The part of run-time code which deals directly with HOL generated scala code and handles the system state is implemented in scala. This is done to provide straight forward communication with scala code. These classes include the Proposer, Acceptor and Learner modules and some handlers for additional features. The Java part of the code is responsible for dealing with the network, service and implementing the failure detector [15] and many other utilities. It inherits most of its functionality from JPaxos code.

The main event handling is implemented in Java and comprises of a dispatcher queue and a thread monitoring the queue. The main event handler is responsible for executing the algorithm steps whenever a Paxos message is received. The receiver thread monitoring the network traffic deserializes every received message and calls the appropriate listener. The listener then creates the corresponding event and pushes the event on dispatcher queue. The main event handler which monitors the dispatcher queue retrieves the event and executes it by calling the appropriate module. For example, when a Propose message is received by a node, the receiver thread invokes the `Propose` listener which creates an `Accept` event and puts it on the dispatcher queue. When the main event handler sees it, it calls the Acceptor

module to handle the message. The acceptor module is implemented in scala and calls the underlying HOL generated code to effect the state changes. The HOL code produces the new state and the `OnAccept` messages to be sent to all the nodes. These messages are given to the sender thread to send over the network.

Batching of client requests is implemented to enhance performance. The size of the batch and the average waiting time are configurable quantities. The client module sends its requests via an interface to the proposer which then batches these requests. Once the batch size has reached its limit or the stipulated time period is over, the batch is sent as the "Value" in a Propose message. The non-leader replicas forward their requests to the leader. The committed requests are sent back to the service.

7.3.3 Additional features and Handlers

The HOL generated code provides run-time APIs through handlers. There is a generic message handler which handles the algorithm messages (for example propose, accept etc). The caller module invokes the generic handler with the message and system state, the handler then invokes the appropriate function depending upon the message received. However there are other handlers which implement some additional features like maintaining the commit order and helping other replicas with catchup.

The commit handler takes care of the commit order when multiple instance are running in parallel. If multiple instances are running in parallel it is possible due to network traffic that on a given node, a later instance gets a majority and decided earlier than a preceding instance. In such a case,if the service does not handle this out of order arrival, it should only be notified when they are no holes in the committed instances log. This is especially true for DUR type services whose correctness may depend upon the order in which the requests were executed. The commit handler monitors the log for currently running requests and makes sure that the service receives committed instances in the correct order.

The catchup handler is used to provide catchup information to the nodes lagging behind. It is possible in the course of Multipaxos algorithm, that one or more nodes may lag behind due reasons like a temporary network failure. Such a node may have missed several instances while the other nodes make progress. Whenever a node receives a message, a check is performed to see if the difference in the current message's instance and the last instance committed on the node crosses a pre-configured window, a catchup is triggered. The node which lags behind requests the other nodes for the information about the instances it has missed through a catchup message. The catchup message contains the requesting node's latest committed instance number. When a node receives this message,the catchup handler is invoked. This handler returns the part of the log containing the instance information from the last instances listed in the catchup request to the latest instance committed by the receiving node.

7.3.4 Network behavior

The Paxos implementation uses stable TCP (Transmission Control Protocol) connections to communicate with other nodes. TCP channels are created to communicate with all the other nodes when each node is initialized. There is one TCP channel per node pair. TCP is preferred over UDP here because of its more reliable nature. TCP ensures that the data reaches the destination in correct order. It keeps track of the data reception through acknowledgements and retransmits if a packet is lost. It takes care of packet disassembly and re-ordering on its own. Thus TCP alleviates some of uncertainty from the distributed protocol therefore relieving the protocol from handling some of the network related issues. This reduces the complexity of distributed algorithm's run-time.

UDP (Used datagram protocol) is also used by the system's failure detector. The failure detector comes into play when a leader is suspected. When the protocol is running smoothly, every node keeps receiving messages from the leader. These messages are also counted as heartbeat messages which indicate that the leader is correct. However if the leader were to crash, other nodes do not receive any message from it. After a stipulated time period, each node starts suspecting the leader. Now the failure detector kicks in and sends suspect messages to other nodes.

7.3.5 Message serialization/deserialization and redundancy

Many verification frameworks tend to use automated serialization solutions like the Java's object serialization. However, we have opted for per-object serialization. Every message class passed over the network has a serialization/deserialization functionality associated with it. This makes the serialization/deserialization process more efficient as it closely tailored to the structure of the object being serialized.

However, there is an overhead associated with this part as well. The HOL generated code requires the Multipaxos messages as input and generates the next step messages as output. Since HOL does not have serialization/deserialization mechanisms, equivalent messages need to be generated in Java for communicati on over network. Thus a message generated from the HOL generated scala code is first converted to the equivalent Java message, which is then serialized and sent over the network. Similarly a received message is first deserialized to create the Java message. An equivalent scala message is created which is compatible with HOL generated code. This introduces redundancy in the system and the overhead can increase with the increase in the number of messages.

7.3.6 Service

The service is provided by the user of the replicated system. It represents the module which is replicated. In this work, the service is responsible for taking the requests from the client, getting the requests ordered by calling the distributed algorithm, executing the request and notifying the client once its request has been executed.

7.4 Evaluation

To evaluate Verified Multipaxos we used a service layer based on PaxosSTM [55]. We used original JPaxos and PSync as competitors. LastVoting was used as the consensus algorithm for Psync while JPaxos used the same PaxosSTM based service layer as Verified Multipaxos. We used the Amazon EC2 cluster as testbed using upto 15 nodes. Each node is equipped with a quad socket, where each socket hosts an Intel Xeon , 8-core, 2.3 GHz CPU. The memory available is 30GB and the network bandwidth is about 10 Gbps.

To evaluate the run-times we needed to measure the ordering capability of each run-time's distributed algorithm. The transactional benchmarks generally used to evaluate replicated systems require some processing overhead at the service layer, therefore the throughput does not reflect the true ordering potential of the distributed algorithm. To address this, we implemented a synthetic benchmark, which incurs minimal processing cost at the service layer. The benchmark puts a random integer in the transaction's R/W set and commits every transaction after the ordering is done without any certification overhead.

The clients were put locally on each node. In order to avoid changing the total load on the system with the change in number of nodes we kept the total number of clients fixed and balanced the clients among the nodes. The best results were found for an overall client count of 1200. To load the system, the commands are injected in a closed loop i.e the clients are notified after the requests are ordered. Each request carried a payload of 40 bytes. Batching was employed to improve performance. The batch size used was 12000 Bytes, thus each request batch contained 300 individual client requests. The proposer node creates request batches and other nodes forward their client requests to the leader. The performance is measured in terms of the number of request batches ordered per second.Each data-point represents the average of 10 measurements.

Figure 7.2 presents the performance of the synthetic benchmark for the three systems under consideration. Number of requests per second is the quantity monitored by varying the number of nodes. In practice, the system performance initially tends to increase with the increase in node count till a limit is reached. Afterwards, the performance degrades with the increase in node count. For Verified Multipaxos and PSYNC [9], best performance is achieved at 5 nodes, while JPaxos being more resilient peaks at 7 nodes. This trend is to be expected as for a lower node count, the ordering layer and the network are not saturated.

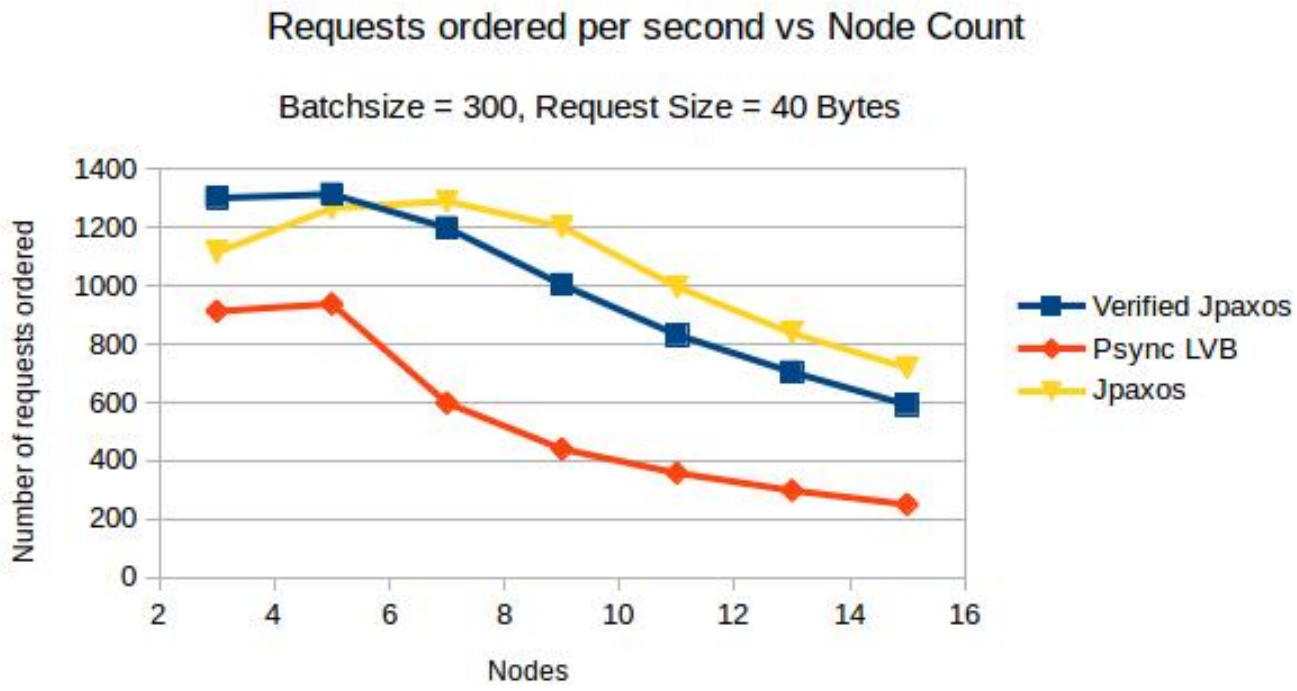

Figure 7.2: Performance of Verified JPaxos, PSYNC Last Voting and JPaxos for the synthetic benchmark

The number of requests ordered in these cases is determined by the number of requests the clients can submit and process per unit time which in-turn is dependent upon the node's computing power. In this setup, the total number of clients in the system is kept fixed, thus an increase in the number of nodes leads to a lesser number of clients per node. The load on individual node decreases with increase in node count. As a result, if the ordering layer and the network are not saturated, an increase in the number of nodes leads to an increase in performance. However, as we increase the number of nodes, the ordering layer's overhead increases due to the increase in quorum size, higher number of messages exchanged, longer broadcast phases, network saturation etc. Thus increasing the number of nodes after a limit tends to degrade the performance.

For 3 and 5 nodes cases, Verified Multipaxos gives the best performance. For all other cases JPaxos tends to dominate followed by Verified Multipaxos. PSYNC's performance is the least of the three for all the cases. For 3 nodes, Verified Multipaxos shows a speed-up of 16% over Jpaxos and 42% over PSYNC. For the 5 node case the speed-up decreases to about 3% – 4% over JPaxos. For 7 – 15 nodes, JPaxos dominates Verified Multipaxos. The difference in performance is 7.5% for the 7 node case and increases to about 20% for all the higher node cases. Verified Multipaxos still outperforms PSYNC in all these cases giving a speedup of 2.0x – 2.5 x.

The reason for this trend is the fact that in case of Verified Multipaxos, the HOL generated scala state houses all the data structures necessary to specify the system state including the command log. Many data structures included in the state increase linearly with the increase in node count. Unlike JPaxos, whose performance has been studied over a long time and which is optimized to increase performance, the scala state used by Verified Multipaxos still has a lot of scope of optimization to scale well with the system. Another reason which can lead to performance degradation with the increase in thread count is the message redundancy which is explained in Section 7.3.5. The number of messages per instances grows with the increase in the number of nodes. This overhead can also contribute to the performance degradation with increase in node count. However, our speedup over PSYNC for all cases, justifies the use of JPaxos based run-time for the easily verifiable scala back-end.

Chapter 8

Conclusion and Future Work

8.1 Thesis Summary

In this thesis, we present three main research contributions. The first is PXDUR, which is a high performance fault-tolerant transactional system that uses Deferred Update Replication (DUR) approach. PXDR enhances the DUR protocol through two optimizations : speculative forwarding of shared objects from locally committed transactions awaiting total order; skipping read-set validation phase during commit when it is safe to do so.

PXDUR inherits the idea of speculative forwarding from an existing work XDUR [1]. This functionality is used to alleviate local contention on individual nodes. PXDUR enhances this approach by making the execution multi-threaded. PXDUR uses a concurrency protocol to allow speculation in parallel. PXDUR further improves the system's performance, by optimizing the commit phase. This optimization helps to identify when it is safe to skip the read-set validation through a low cost operation, which is based upon the idea of identifying when a transaction originating on a particular node can abort during the commit phase.

Like other DUR based systems, PXDUR finds its sweet spot when the transaction access pattern is fully partitioned. Here, PXDUR can use speculation to achieve high benefits, as it can commit a large number of local transactions through speculative forwarding. A lack of remote conflicts means that all locally committed transactions will not be aborted after total order. Such a scenario also allows PXDUR to utilize the full benefit of commit phase optimization as it is always safe to skip the rad-set validation. PXDUR approach can be used to enhance DUR systems and provide the system with the flexibility of using the optimal number of execution threads as per the system requirements.

As seen through different benchmarks, PXDUR provides significant speedup over other similar protocols for low and medium contention scenarios using parallelism. On high contention workloads, it leverages the speculative forwarding and gives performance comparable to sin-

gle thread execution.

The second contribution presented is TSAsR or Timestamp based As-Serializable Transactions. It is an adaptable transaction processing system which can be applied to an existing DBMS . It provides the serializability isolation level utilizing the DBMS's relaxed concurrency control levels. TSAsR attaches external meta-data to shared objects and utilizes it to substitute the DBMS's internal eager- locking approach with simpler, more optimistic serializability certification method. TSAsR can alleviate the drawbacks involved with the conservative locking protocols present in DBMSs. It associates a unique timestamp with each transaction and applies it to the object versions read and written by the given transaction. It uses these timestamps to determine whether committing a transaction will close a cycle in transaction depndency graph without actually generating and traversing the full graph.

TSAsR tries to ascertain each transaction's latest predecessor timestamp and and earliest successor timestamp at the time of commit. For a given transaction, the predecessors are either those transactions which wrote the objects which the current transaction read; or those transactions which read the objects the current transaction overwrote. The successors are those transactions which have a write anti-dependency on the current transaction. The successors overwrote the objects read by the current transaction and committed first. TSAsR determines these timestamps through the meta-data associated with the shared objects. It performs a simple exclusion check to ascertain whether the given transaction's commit can potentially close a dependency graph cycle. If so, the transaction is aborted.

The evaluation with three benchmarks reveals that TSAsR handles the moderate and high contention scenarios much better than the DBMS's traditional lock-based synchronization primitives thus exposing their limitation in handling high degree of concurrency. It also shows significant performance improvement over existing visbile-read based AsR implementation.

Verified JPaxos constitutes the third contribution which involves the development of a high performance run-time for an I/O automaton representing the Multipaxos algorithm. The run-time forms a part of the larger effort to develop high performance SMR based systems which lend themselves easily to verification. The I/O automaton in question is generated from the Multipaxos specification in HOL (Higher Order Logic) which is easy to verify. The HOL specification of Multipaxos and its subsequent verification is out of this work's scope. We start with the I/O automaton which is exported to scala from the HOL specifciation. The run-time's design is based upon JPaxos which is a high performance Java based Multipaxos implementation.

The run-time drives the scala based automaton by giving it appropriate inputs and utilizing its output as per the specifications of Multipaxos. It also communicates with other nodes and the service layer. It takes the client requests from the service, submits the request batches to the protocol for ordering and notifies the service, once the requests are ordered. It takes Multipaxos messages generated by the scala based I/O automaton and delivers it to the correct nodes. It makes use of TCP protocol to ensure that network issues like packet

duplication or packet loss does not take place.

The evaluation suggests that Verified Multipaxos achieves significant performance benefit over another domain specific language aimed at easy verification of distributed protocols while being not much lower than the original JPaxos.

8.2 Conclusions

The three contributions are significant as they touch different areas of transactional processing systems and enhance such systems by optimizing the targeted aspects. PXDUR shows that there is a significant scope of improving the performance of DUR based systems by addressing local contention and the improving the commit rate. It proves that speculative forwarding and parallelism are viable directions of improving the performance of such systems. TSAsR explores the space of performance improvement in centralized DBMSs under highly concurrent access. It proves that the approach taken by TSAsR, which involves leveraging of relaxed isolation levels provided by the DBMS combined with a lightweight serializability certifier, can lead to significant performance benefits especially when contention is higher. Verified JPaxos proves that it is possible to develop SMR based algorithms which lend themselves easily to verification without compromising the performance.

They all are simple and high performing solutions with a high degree of adaptability. All of these contributions can be easily integrated into different systems without requiring much effort on the user side. They are also flexible with respect to applications. different applications need minimal configuration to use these systems.All the contributions outlined have wide usage potential, are widely adaptable and can be configured to run upon variety of systems and with a variety of applications.

Therefore, PXDUR, TSAsR and Verified JPaxos are efficient high performance solutions which fulfill their aim of optimizing transactional systems.

8.3 Future Work

There are a number of extensions applicable to the solutions presented in this thesis which provide the viable directions of future work. PXDUR achieves high performance for fully partitioned systems, however it has performance limitations when there are cross-partition accesses. New methods are required which can uphold the system's performance when transactions can have significant cross-partition access. based route.

TSAsR can be implemented in different systems to test its scalability under different environments. The current database system used does not scale well as the number of threads increase thereby limiting TSAsR's performance as well. Here TSAsR was used on a cen-

tralized database, however it can be deployed on distributed systems too. TSAsR can be expanded to use isolation levels like Snapshot Isolation increasing its deployment possibilities. TSAsR can also be expanded to support multi-version systems.

The run-time developed in Verified JPaxos can be easily expanded to support other Paxos based protocols. Its scalability can be tested by deploying it with more SMR algorithms. Another way of unlocking its full potential could be enhanced efficiency in the HOL specification over which it operates.

Bibliography

[1] B. Arun, S. Hirve, R. Palmieri, S. Peluso, and B. Ravindran. Speculative client execution in deferred update replication. In *Proceedings of the 9th Workshop on Middleware for Next Generation Internet Computing*, MW4NG '14, pages 5:1–5:6, New York, NY, USA, 2014. ACM.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 1–10, New York, NY, USA, 1995. ACM.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[4] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 729–738, New York, NY, USA, 2008. ACM.

[5] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, pages 35–46, Sept 2008.

[6] D. Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[7] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[8] T. Council. TPC-C benchmark. 2010.

[9] C. Drăgoi, T. A. Henzinger, and D. Zufferey. Psync: A partially synchronous language for fault-tolerant distributed algorithms. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 400–415, New York, NY, USA, 2016. ACM.

[10] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, Nov. 1976.

[11] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, June 2005.

[12] D. F. García and J. García. Tpc-w e-commerce benchmark evaluation. *Computer*, 36(2):42–48, Feb. 2003.

[13] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd. Probe: A thousand-node experimental cluster for computer systems research. volume 38, June 2013.

[14] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[15] R. Guerraoui and A. Schiper. Genuine atomic multicast in asynchronous distributed systems. *Theor. Comput. Sci.*, 254(1-2):297–316, Mar. 2001.

[16] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 1–17, New York, NY, USA, 2015. ACM.

[17] S. Hirve, R. Palmieri, and B. Ravindran. Archie: A speculative replicated transactional system. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 265–276, New York, NY, USA, 2014. ACM.

[18] S. Hirve, R. Palmieri, and B. Ravindran. Hipertm: High performance, fault-tolerant transactional memory. In *Proceedings of the 15th International Conference on Distributed Computing and Networking - Volume 8314*, ICDCN 2014, pages 181–196, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[19] S. Hirve, R. Palmieri, and B. Ravindran. *HiperTM: High Performance, Fault-Tolerant Transactional Memory*, pages 181–196. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[20] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, Aug. 1978.

[21] D. Kaloper-Meršinjak, H. Mehnert, A. Madhavapeddy, and P. Sewell. Not-quite-so-broken tls: Lessons in re-engineering a security protocol specification and implementation. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 223–238, Washington, D.C., Aug. 2015. USENIX Association.

[22] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: Language support for building distributed systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 179–188, New York, NY, USA, 2007. ACM.

[23] T. Kobus, M. Kokocinski, and P. T. Wojciechowski. Hybrid replication: State-machine-based and deferred-update replication schemes combined. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pages 286–296, July 2013.

[24] T. Kobus, M. Kokocinski, and P. T. Wojciechowski. Hybrid replication: State-machine-based and deferred-update replication schemes combined. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, ICDCS '13, pages 286–296, Washington, DC, USA, 2013. IEEE Computer Society.

[25] J. Kończak, N. Santos, T. Żurkowski, P. T. Wojciechowski, and A. Schiper. JPaxos: State machine replication based on the Paxos protocol. Technical Report EPFL-REPORT-167765, Faculté Informatique et Communications, EPFL, July 2011. 38pp.

[26] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, pages 133–169, 1998.

[27] L. Lamport. Distributed algorithms in tla (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 3–, New York, NY, USA, 2000. ACM.

[28] L. Lamport. The pluscal algorithm language. In *Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing*, ICTAC '09, pages 36–60, Berlin, Heidelberg, 2009. Springer-Verlag.

[29] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

[30] C. Li, J. a. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the choice of consistency levels in replicated systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 281–292, Berkeley, CA, USA, 2014. USENIX Association.

[31] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 265–278, Hollywood, CA, 2012. USENIX.

[32] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 137–151, New York, NY, USA, 1987. ACM.

[33] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[34] P. J. Marandi, C. E. Bezerra, and F. Pedone. Rethinking state-machine replication for parallelism. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 368–377, June 2014.

[35] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, Sept. 1992.

[36] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, ii. *Inf. Comput.*, 100(1):41–77, Sept. 1992.

[37] O. A. Mohamed, C. A. Muñoz, and S. Tahar, editors. *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*. Springer, 2008.

[38] J. Niles, Duane F. Improving performance of highly-programmable concurrent applications by leveraging parallel nesting and weaker isolation levels. Master's thesis, Virginia Polytechnic Institute and State University, 2015.

[39] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[40] M. Odersky and al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL, Lausanne, Switzerland, 2004.

[41] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley db. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, pages 43–43, Berkeley, CA, USA, 1999. USENIX Association.

[42] A. Oracle. Berkeley db java edition architecture executive overview, 2006.

[43] R. Palmieri, F. Quaglia, and P. Romano. Osare: Opportunistic speculation in actively replicated transactional systems. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*, pages 59–64, Oct 2011.

[44] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Distrib. Parallel Databases*, 14(1):71–98, July 2003.

[45] S. Peluso, J. Fernandes, P. Romano, F. Quaglia, and L. Rodrigues. Specula: Speculative replication of software transactional memory. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, pages 91–100, Oct 2012.

[46] V. Rahli. Interfacing with proof assistants for domain specific programming using eventml. Presented at the 10th International Workshop on User Interfaces for Theorem Provers, 2012.

[47] S. Revilak, P. O'Neil, and E. O'Neil. Precisely serializable snapshot isolation (pssi). In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 482–493, Washington, DC, USA, 2011. IEEE Computer Society.

[48] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990.

[49] D. Sciascia, F. Pedone, and F. Junqueira. Scalable deferred update replication. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[50] K. Slind and M. Norrish. A brief overview of HOL4. In Mohamed et al. [37], pages 28–32.

[51] T. Wang, R. Johnson, A. Fekete, and I. Pandis. The serial safety net: Efficient concurrency control on modern hardware. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, DaMoN'15, pages 8:1–8:8, New York, NY, USA, 2015. ACM.

[52] T. Wang, R. Johnson, A. Fekete, and I. Pandis. Efficiently making (almost) any concurrency control mechanism serializable. *CoRR*, abs/1605.04292, 2016.

[53] M. Widenius and D. Axmark. *Mysql Reference Manual*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 2002.

[54] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 357–368, New York, NY, USA, 2015. ACM.

[55] P. T. Wojciechowski, T. Kobus, and M. Kokocinski. Model-driven comparison of state-machine-based and deferred-update replication schemes. In *Proceedings of the 2012 IEEE 31st Symposium on Reliable Distributed Systems*, SRDS '12, pages 101–110, Washington, DC, USA, 2012. IEEE Computer Society.

[56] P. T. Wojciechowski, T. Kobus, and M. Kokocinski. Model-driven comparison of state-machine-based and deferred-update replication schemes. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, pages 101–110, Oct 2012.

[57] Y. Yuan, K. Wang, R. Lee, X. Ding, J. Xing, S. Blanas, and X. Zhang. Bcc: Reducing false aborts in optimistic concurrency control with low cost for in-memory databases. *Proc. VLDB Endow.*, 9(6):504–515, Jan. 2016.

[58] V. Zuikevičiūtė and F. Pedone. Conflict-aware load-balancing techniques for database replication. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, pages 2169–2173, New York, NY, USA, 2008. ACM.