Sapienza Università di Roma

Dottorato di Ricerca in Ingegneria Informatica

XXIV Ciclo – 2011

Performance Models of Concurrency Control Protocols for Transaction Processing Systems

Pierangelo Di Sanzo

Sapienza Università di Roma

Dottorato di Ricerca in Ingegneria Informatica

XXIV Ciclo - 2011

Pierangelo Di Sanzo

Performance Models of Concurrency Control Protocols for Transaction Processing Systems

Thesis Committee

Prof. Bruno Ciciani (Advisor)
Prof. Antonio Puliafito

Reviewers

Prof. Kishor S. Trivedi
Prof. Miklos Telek

Author's address:
Pierangelo Di Sanzo
Dipartimento di Ingegneria informatica, automatica e gestionale
Sapienza Università di Roma
Via Ariosto 25, I-00185 Roma, Italy
e-mail: disanzo@dis.uniroma1.it
www: http://www.dis.uniroma1.it/~disanzo/

Contents

1 Introduction **1**
1.1 The Performance Issue 2
1.2 Performance Modeling of Concurrency Control Protocols 3
1.3 Contribution Overview 4
1.4 Structure of the Dissertation 7

2 Concurrency Control Protocols in Transactional Processing Systems **9**
2.1 Basics 9
2.2 Database-Oriented Protocols 10
2.3 Brief Overview on Software Transactional Memories 12
2.4 Database Transactions vs. Memory Transactions 13
2.5 STM-Oriented Protocols 14

3 Literature Overview **17**
3.1 Database Systems 17
3.2 Software Transactional Memories 19

4 Performance Modeling of MultiVersion Concurrency Control Protocols **23**
4.1 Introduction 23
4.2 An Overview of MultiVersion Concurrency Control Protocol Ensuring Snapshot-Isolation 24
4.3 The Analytical Model 25
4.4 System Model 25
4.4.1 Basic Analytical Model 26
4.4.2 Numerical Resolution 35
4.4.3 Extended Analytical Model 35
4.5 Simulation Model 38
4.6 Model Validation 39

5 Performance Modeling of Concurrency Control Protocol for Software Transactional Memories **43**
5.1 Introduction 43

5.2 System Model and Considerations 45
5.3 The Analytical Model 47
5.3.1 Modeling Approach Overview 47
5.3.2 Thread-level Model 48
5.3.3 Transaction-level model: The Commit-Time Locking Case . 50
5.3.4 Coping with Multiple Transaction Classes 57
5.3.5 Hints on Model Extension for Non-uniform Data Access . . 62
5.4 Simulation model 62
5.5 Validation 63
5.5.1 Single-class Case 66
5.5.2 Multi-class Case 70
5.6 On Removing Exponential Assumptions 71

6 Performance Modeling of Concurrency Control Protocols (CCPs) With Arbitrary Data Access Patterns 75
6.1 Introduction 75
6.2 Effects of Data Access Patterns with SS2PL Protocol 77
6.3 System Model 78
6.3.1 Transaction Model 79
6.3.2 Hardware Resource Model 80
6.4 The Analytical Model 80
6.4.1 Transaction Execution Time 81
6.4.2 Lock Holding Time 81
6.4.3 Data Contention 82
6.4.4 Wait Time 84
6.4.5 Operation Execution Time 85
6.4.6 Numerical Resolution 86
6.4.7 Coping with Multiple Transaction Classes 86
6.5 Model Validation 88
6.5.1 Part-A 89
6.5.2 Part-B 92
6.5.3 Part-C 92
6.6 Analysis of the Sensitivity to Data Access Patterns with Other Protocols 95

7 Conclusions 99

Bibliography 103

Acronyms 111

Chapter 1

Introduction

In many data sharing environments data manipulation leverages on transaction processing. A transaction consists of a set of operations whose processing must provide specific guarantees. These guarantees are defined in terms of properties of transactions, namely Atomicity, Consistency, Isolation and Durability (ACID properties). In IT applications transaction processing is used for the development of various components of different layers of the system architecture. E.g., transactions can be used in a database client application to execute sets of operations on data contained in a database server, or by processes to execute sets of operations on files, as well as in multi-threaded applications to execute sets of memory read/write operations.

Transaction processing relies on the so-called Concurrency Control Protocols (CCPs). A CCP defines a set of rules that allow the system to concurrently execute transactions preserving the desired properties.

Over the last few decades, transaction processing got an important role in many contexts, spanning from enterprise applications to operating systems. As a consequence, its increasing popularity has led to a growing interest in CCPs. Today, CCPs act as core components for the design and implementation of a wide spectrum of applications, as banking, booking, e-commerce, as well as of many low level synchronization mechanisms, as in file system management and, in general, in concurrent programming. For these reasons they are of interest for a lot of IT players. Furthermore, despite the transaction processing is not a new research area, CCPs continue to even more attract the interest of researchers, as evidenced, e.g., by the emergence of transactional memories [1], which today represent an hot topic in the concurrent programming research.

The growing interest in CCPs is also due to a recent trend in computer manufacturing [2]. Over the last decade, the hardware architecture of a majority of computer systems, including the entry level ones, has profoundly changed, moving from

a single-core to a multi-core architecture. On the other hand, the exponential growth of the CPU clock speed has stalled, so that, nowadays, the increase of computing power of a system is mainly due to the increase of the number of CPU cores. As a consequence, single process/threaded applications can no longer take advantage of such a performance gain of hardware architectures. In order to continue to exploit the growth of computing power to boost the performance performance of applications, we need for concurrent applications. This results in a growing care for CCPs.

1.1 The Performance Issue

The appeal of the transactional processing systems is mainly due to the ability to transparently ensure the properties of transactions, requiring the system user only to demarcate the blocks of operations which form a transaction. On the other hand, the concurrency control may have a remarkable impact on the system performance. The latter is a critical issue for the transaction processing systems. Suffice it to say that the transaction response time is one of the most widely used indicators in service level agreements negotiation, as well as the maximum achievable system throughput is used as a foundamental indicator in transaction processing and database system benchmarking [3].

The impact on system performance of a CCP depends on many factors. Basically, a contribution to the performance degradation is due to the dynamics of execution of transactions, which depend on the rules of the protocol. For example, some protocols prevent data conflicts by blocking the execution of a transaction when, while accessing a data item, a (potential) conflict with a concurrent transaction occurs. Conversely, other protocols aborts a transaction (which has to be subsequently restarted) when a conflict with a concurrent transaction is detected. Both blocking and restarting a transaction result in an increment of the transaction response time. The performance degradation is also due to the extra processing time associated with the execution of the code for the protocol implementation, which may require both large data structure management and explicitly memory fence instructions or expensive hardware operations (e.g. compare-and-swap) for process/thread synchronization [1].

The effects on system performance due to the concurrency control are complex to analyze. Infact, the transaction execution dynamics depend on the mix of various factors, as the transaction profiles (including, e.g., the operation types and the accessed data items), the transaction arrival rate, the concurrency level, the processing speed of the system, etc. For example, transactions may experience multiple waiting phases whose durations depend, in turn, on the execution time of the conflicting transactions. Also, a transaction may experience a number of aborts which depends on the

[1] in some systems, as database ones, these costs can be affordable, but in others, as transactional memories, their impact can be remarkable [4].

read/update rate of the set of accessed data items determined by the concurrent transactions. Furthermore, generally, a protocol provides higher performance than others depending on the workload and system features. E.g. some protocols are optimized for read-intensive workloads (as MultiVersion Concurrency Control (MVCC) protocols [5]) and some protocols perform better than others when used in systems with low resources [6].

According to the observations made so far, understanding and/or evaluating the impact of the concurrency control on the system performance are fundamental issues, and, on the other hand, they are non-trivial tasks because of the multitude of involved factors. At any rate, designing, optimizing and tuning transaction processing systems are complex activities which require a deep knowledge of the alternatives and implications associated with the choices of CCPs.

The analysis and the proper understanding of the impact on the system performance of the concurrency control require quantitative approaches. A largely used approach in computer system performance analysis is the model-based one [7, 8]. With this approach the analysis is conducted using an analytical or simulative system model. With respect to the measurement-based approach, which entails direct measurements on the real system, it provides various advantages. Basically, it allows to conduct performance studies by avoiding the burden of building (a component of) the real system or a prototype, as well as by avoiding expensive constructions of test-cases. This can be very valuable, in particular in the early stages of the system design. The model-based approach allows to abstract from undesired effects due to factors which, conversely, could be unmovable when the performance assessment is conducted with a real system. It is an inexpensive approach to explore alternatives, to test new ideas, as well as to analyze, through the composition of models, more complex systems. Obviously, all assumptions and approximations used in the construction of a model, and all its implications and limitations, are crucial aspects to be taken into account in the performance analysis. Finally, the model-based approach must not be considered a completely alternative approach to the measurement-based one, rather they have to be considered complementary.

1.2 Performance Modeling of Concurrency Control Protocols

Analytical modeling and simulation are two common approaches used for the performance analysis of transaction processing systems. In the follow, we discuss some basic aspects which need to be considered when dealing with the design of performance models for CCPs.

A transaction processing system is characterized by a peculiar aspect with respect to a system without data contention [9]. In the latter case, the operation response time

is affected by the queuing and processing delay in accessing hardware resources. In a transaction processing system, the transaction response time is affected by both hardware resources and data contention, and these, in turn, can affect each other. For example, when a transaction T is blocked due to data contention, then T may determine an increment of the data contention probability of the concurrent transactions due to the increase of the lock holding time of locks held by T. Similarly, if T is aborted and restarted, it may determine an increment of the processing time of the concurrent transactions due to the extra resource utilization for the re-processing. The increment of the data contention and/or the higher processing time of the concurrent transactions, in turn, may determine a subsequent increase of probability for T to be blocked or restarted. These factors entail non-trivial dependences between the various system performance indicators (e.g. the transaction response time vs. the transaction contention probability). Furthermore, the predominance of the impact on the system performance of some factors with respect to others also depends on the mix of mechanisms used by the CCP.

A performance models of a CCP has to be an effective tool aimed to analyze and/or understand performance issues related to the concurrency control. The intrinsic complexity of the transaction processing systems imposes to rely on system models where specific assumptions are needed in order to make the analysis feasible. For this reason, the level of abstraction of a model represents a fundamental choice determining its validity. For example, models aimed to perform a qualitative analysis of the transaction execution dynamics could abstract from the actual utilization of the hardware resources. On the other hand, it has been shown that the amount of available hardware resources can determine which type of protocol has the chance to provide the best performance [6]. Accordingly, such models would provide unreliable results if used to conduct a performance comparison study between differet protocols.

The wide diversity characterizing the workloads of applications for transaction processing systems, and the attempt to build models whose validity is not restricted to specific applications, entails the adoption of generic workload models, by the definition of a limited number of parameters determining the workload model configuration space.

Finally, the complex relations existing between the various factors that can affect the system performance lead to use approximation based approaches.

1.3 Contribution Overview

In this dissertation we present performance models of CCPs for transaction processing systems. Primarily, we use an analytical approach. Further, we also use detailed simulation models to evaluate the accuracy of the analytical models we propose, and to analyze some features of the protocols we deal with. We preferred to focus mainly on the analytical approach for two main reasons: (1) analytical modeling can be a

practical approach for building cost-effective computer system performance models and, in particular, (2) the analytical approach enables to quantitatively describe the complex dynamics characterizing the concurrency control, allowing us to analyze and understand existing dependencies between system performance indicators and other system configuration parameters, and to reason about their implications. In this work, we deal with both Database Systems (DBS) and Software Transactional Memories (STMs), which represent traditional and emerging transaction processing systems, respectively.

The first model we present focuses on the MVCC in DBS. The performance modeling of CCPs has been largely conducted in the field of DBS. Anyway, the most of performance analysis work focus on modeling and/or evaluation of protocols considering basic concurrency control strategies and the associated implementation mechanisms (e.g. blocking or restart-oriented lock-based protocols, optimistic timestamp-based protocols). Over the time, the need of performance gain led to the design of new, more complex protocols. Thus, what often happens nowadays is that systems use protocols relying on more complex strategies than those analyzed in performance modeling studies. This entails the need of new efforts in the performance analysis and of new performance evaluation tools. In particular, this is the case of the most used MVCC protocol in Database Management Systems (DBMS), including both commercial and open source ones. It combines different mechanisms, as data versioning, transaction blocking and transaction restart. This mix of mechanisms provides high performance in particular with read-intensive workloads, which characterize many applications which require transaction processing. On the other hand, it determines more complex transaction execution dynamics with respect to other protocols. The literature does not provide analytical performance models which allows us to study and to quantify the effect on system performance of such a mix of mechanisms. We address this lack by providing an analytical performance model tailored for this protocol. To cope with the its complexity, we use a modeling approach wich focuses on the transaction execution dynamics. These are captured by means of a transaction model which represents the transaction execution throught its phases and on basis of the phase transition probabilities. By relying on this transaction model, we incrementally derive a set of analytical expressions to calculate the various probabilities and the other involved quantities. The system performances indicators can be calculated numerically resolving the set of expressions. By this model we can evaluate the expected transaction response time and other indicators, as the data validation failure probability and the data version creation rate, which allow to quantify the impact on system performance associated with the different mechanisms used by the protocol.

We then move to the field of STMs. Very little performance modeling work has been made in this field. Concurrency control in DBMS and in STMs relies on similar concepts, therefore, used methodologies and models for DBS can also provide a valuable support for performance analysis of STMs. Anyway, transaction processing in STMs and in DBS is different in many aspects [10], and this entails differences

concerning both the used CCPs and the choice of appropriate system models. So far, performance analysis of STMs has been essentially conducted with the measurement-based approach. More recently, a few analytical and simulation models have been proposed. In some cases, these models assume simplified system models which do not allow to evaluate time-related performance indicators (e.g. the transaction response time and the system throughput), hence, they enable to study and evaluate protocols only by a limited analysis perspective. In other cases, the focus of the proposed models is shifted to different aspects from the CCPs. We propose a framework tailored for a more comprehensive performance analysis of STMs which overcomes the main limitations of the previous studies. In this framework we deal with the effects on system performance associated with both the CCPs and the dynamics related to the transaction executions by the concurrent threads of an application. Our system model is ispired to a typical STM application where threads are supposed to run on CPU-cores of a multi-core processor system. Threads alternate the execution of transactions, where they perform accesses both on shared and local data, and code blocks where they perform only local computations (e.g. see [11]). To this purpose, we propose a two-layered modeling approach, where a layer captures the dynamics related to the execution of threads, delegating the concurrency control model to another independent layer. These allows us to evaluate the system performance also capturing the effect due to the continuos variation of the concurrency level and to the mix of different transactions in the system. At same time, with our modeling approach, this can be simply obtained by devoloping a CCP model for a fixed, albeit parametric, concurrency level and mix of transactions. Furthermore, the two-layered structure makes the framework feasible to build models for different CCPs. We present an istance of the CCP layer for the case of the Commit-Time Locking (CTL) protocol [12], which is currently used by several STM implementations. The complete instantiation of the framework allows us to evaluate, in addition to the expected system throughput, various indicators, as the transaction abort probabilities throughout the various phases of the transaction execution. The indicators can be evaluated on basis of various system configuration parameters, as the profiles of the transaction classes, the number of concurrent threads, the duration of the different memory operations and the shared memory size.

Finally, we propose a modeling approach which opens a new perspective in the CCPs performance analysis. So far, proposed modeling approaches rely on the assumption according to which the data items accessed by transactions do not depend on the phase of the transaction execution. In other words, the data access sequences of transactions are not considered. Actually, in many applications for transaction processing systems, transactions tend to access data items according to specific sequences (e.g. see TPC-C [3] and TPC-W [13] benchmark applications). We show that performance delivered by CCPs which acquire locks during the transaction execution (i.e. before the commit time, as the Two-phase locking (2PL) [5]), can be strongly affected by such data access patterns. Futhermore, we show that performance models

which ignore these aspects can provide unreliable results when used in performance analysis of applications. Today, the aforesaid types of protocols represent a very large class of CCPs used in transaction processing systems.

To cope with the above-mentioned problem, we propose, by focusing on the Strong-Strict 2PL (SS2PL) protocol, which is the most used version of the 2PL, a modeling approach which enables to capture the effects due to transaction data access patterns where data accesses depend on the transaction execution phases. This approach can be used in the case of both deterministic and probabilistic transaction data access patterns. In addition, we show as this approach is also suitable for applications with different transaction profiles, where each profile is characterized by a different data access pattern. The model we present allows to evaluate the average transaction response time for each transaction profile and various other indicators, as the lock holding time and the transaction waiting time. These indicators are very useful to investigate the effects on the system performance of the data access patterns of applications. Models built with the approach we present can also be used to support the selection of the optimal transaction data access patterns for the design of applications.

The analytical models we present in this dissertation can be coupled with different hardware resource models and can be resolved by iteration. We show how this can be done by using an hardware model typically adopted in performance studies of CCPs.

Finally, all these models can help to answer further typical questions in the area of the transaction processing systems, as: Which is the potential performance bottleneck? How does the system scale up? Which mechanism used by a protocol is the main responsible for the performance loss? Which is the main transaction class responsible for the system overhead?

1.4 Structure of the Dissertation

The rest of this dissertation is organized as follows. In Chapter 2 we provide an overview of the CCPs in the field of both DBS and STMs. A literature overview is presented in Chapter 3. In Chapter 4 we present the performance model of the MVCC protocol. The modeling framework for STMs and the model of the CTL protocol are presented in Chapter 5. In Chapter 6 we present the performance study on the transaction data access patterns and the performance model of the SS2PL protocol. Further, we conclude the Chapter 6 with a brief analysis of the sensitivity to data access patterns of other protocols, including those considered in previous chapters. A concluding discussion is in Chapter 7.

Most of the material contained in this dissertation can also be found in the following

articles:

Pierangelo Di Sanzo, Bruno Ciciani, Roberto Palmieri, Francesco Quaglia, and Paolo Romano. On the analytical modeling of concurrency control algorithms for software transactional memories: The case of commit-time-locking. *Performance Evaluation*, to appear. Available on line: June 2011.

Pierangelo Di Sanzo, Bruno Ciciani, Francesco Quaglia, and Paolo Romano. A performance model of multi-version concurrency control. In *Proceedings of the 16th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 41–50, 2008.

Pierangelo Di Sanzo, Roberto Palmieri, Bruno Ciciani, Francesco Quaglia, and Paolo Romano. Analytical modeling of lock-based concurrency control with arbitrary transaction data access patterns. In *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, WOSP/SIPEW '10, pages 69–78, New York, NY, USA, 2010. ACM.

Pierangelo Di Sanzo, Roberto Palmieri, Paolo Romano, Bruno Ciciani, and Francesco Quaglia. Analytical modelling of commit-time-locking algorithms for software transactional memories. In *Proceedings of the Computer Measurement Group 10th International Conference*. Computer Measurement Group, 2010.

Chapter 2

Concurrency Control Protocols in Transactional Processing Systems

Over the years, the need for performance improvement has conducted researchers and system designers to explore many alternatives and to evaluate different solutions for the problem of concurrency control in transactional processing systems. Today a variety of Concurrency Control Protocols (CCPs) exist. In this chapter we provide an overview of these protocols, focusing on the fields of Database Systems (DBS) and Software Transactional Memories (STMs) [14]. We discuss their peculiarities and provide various examples. Before discussing STMs' protocols, we also provide a brief overview on STMs and point out some basic differences with respect to DBS. We do not focus on protocols for distributed systems because they are out of the scope of this dissertation. Further details about notions we provide in Section 2.1 and 2.2 can be found in [5].

2.1 Basics

A largely used criterion to evaluate the correctness of CCPs is *serializability*. It states that an execution of a set of transactions must have the same effect on data items as some serial execution of the same transactions. Serializability corresponds to the higher isolation level defined by ANSI/ISO SQL Standard [15]. Actually, a such isolation level is not necessary in many applications. For example, in some benchmark applications (e.g. [3]) some transactions require a lower level, as *Read Committed*, which only ensures that transactions do not read updates on data items made by transactions not committed yet. Furthermore, some DBMS do not guarantee serializability

(e.g. Oracle Database [16]). In this case, to enforce serializable executions, the system user has to explicitly add specific instructions within transactions. *Recoverability* is another requirement for the execution of transactions. An execution is *recoverable* if transactions which read values written by another transaction T do not commit before T. This ensures that if T has to be aborted, it can be safely done. However, this is not enough to avoid *cascading aborts*. This phenomenon occurs when a transaction T_i gets aborted and another transaction T_j has already read a value written by T_i. In this case also T_j has to be aborted. Cascading abort can be avoided by preventing transactions from reading values wrote by uncommitted transactions. In this case the execution is said *cascadeless*. Another phenomenon is the following one. When a transaction T_i is aborted, the previous value of each data item updated by T_i has to be restored. If another transaction T_j updates one of this data item before T_i gets aborted, restoring the previous value of this data item entails the loss of the value written by T_j. This can be avoided by preventing transactions from writing values written by uncommitted transactions. This is called *strictness*. Note that strictness implies cascadeless, which, in turn, implies recoverability. All these are orthogonal properties with respect to serializability.

As concerns concurrency control strategies, basically, CCPs can be classified in pessimistic and optimistic ones. In addition, there are a variety of mixed protocols. Pessimistic protocols avoid conflicts between transactions by blocking and/or restarting a transaction before a conflicting operation is executed. To this purpose, conflicts are detected before the execution of the operations. Optimistic protocols do not block or restart transactions before operations, but allow them to be executed as soon as they are requested. The conflict detection is delayed to the end of transaction, i.e. to the commit-phase. In this phase, if a conflict is detected, the transaction is aborted and restarted. These protocols are also called *certification* protocols.

In the follow, we describe various examples of pessimistic, optimistic, and mixed protocols. We start from the field of DBS, and after we move to STMs.

2.2 Database-Oriented Protocols

In DBS, pessimistic protocols are tipically implemented by means of *locks*. Transactions acquire a *shared* lock on a data item before executing a read operation, or an *exclusive* lock before executing a write operation. If an exclusive lock for a data item is held, no other locks can be acquired for the same data item, while, if a shared lock is held, only shared locks can be acquired. If a transaction can not acquire a lock due to another lock held by a concurrent transaction (i.e. a lock *conflict* occurs) then it is blocked and waits until the concurrent transaction releases the lock. The Two-phase locking (2PL) is a protocol where a transaction releases locks only after having acquired all needed locks. The 2PL ensures serializability. The Strong Strict 2PL (SS2PL) [17] is a version of the 2PL where all locks acquired by a transaction

are released only when the transaction terminates or is aborted. The SS2PL ensures serializability and strictness. For these reasons, it is the most used version of 2PL.

Locking protocols which block transactions on conflict are subject to deadlock. In these cases, further mechanisms are needed to avoid this problem. A widely used method is based on timeouts, i.e. a transaction which experiences a lock conflict can waits at most for a fixed time, after which it gets aborted. This is a low cost method, but it can cause the abort of a transaction also if deadlock does not really occur. Another method is based on the wait-graph. This is a directed graph where a node i corresponds to a transaction T_i. An edge from a node i to a node j means that the transaction T_i is blocked due to a lock held by transaction T_j. When a lock conflict occurs the corresponding edge is added to the graph. A cycle in the graph indicates that a deadlock has occurred, and it is solved by aborting a transaction in the cycle. The aborted transaction has to be subsequently restarted. The drawback of wait-graph method is due to the graph management cost. The method based on timeouts is more widely used in DBMS.

Concerning optimistic protocols, the conflict detection can rely on various mechanisms. One of these is the Serialization Graph Test (SGT) Certification. This is based on a serialization graph, i.e. a directed graph where a node i corresponds to a transaction T_i and an edge from a node i to a node j denotes that an operation executed by transaction T_i precedes and has conflicted with an operation executed by transaction T_j. When a transaction executes an operation and a conflict occurs, then an edge is added to the graph. At commit time, if the transaction is within a cycle in the graph, it gets aborted.

The Basic Timestamp Ordering (Basic TO) is a protocol which aborts transactions as soon as a conflict is detected. It allows a transaction T executing an operation only if no conflicting operations have been already executed by other concurrent transactions started after the transaction T. Otherwise T is immediately aborted. This protocol uses timestamps associated with the start of transactions to detect the order of the operations. Basic TO ensures serializability. This version does not provide recoverability, but can be easily specialized to enforce it.

MultiVersion Concurrency Control (MVCC) is a technique which maintains multiple copies, or versions, of a data item. Each version is produced by a write operation. When a transaction reads a data item updated by a concurrent transaction, it is served by using a previous version of the data item. Hence, by maintaining previous versions of updated data items, a reading transaction is never blocked or aborted. An example of such a protocol is the Multiversion Timestamp Ordering (MVTO). Upon a read operation, a transaction T reads the version of a data item produced by the last transaction committed before T started. Upon a write operation of a transaction T on a data item d, if a version of d has already been read by a transaction started after T, then T is aborted. Otherwise a new version of d is created. Finally, a transaction T is committed only after all transactions which have produced versions read by T have been committed. MVTO ensures serializability and recoverability.

Many different protocols can be defined by combining the aforesaid techniques. For example, a common version of MVCC used in DBMS is based on locks. Specifically, transactions acquire an exclusive lock upon a write operation. When a lock is acquired, the write operation is allowed to be executed only if the data item has not been updated by concurrent transactions, otherwise the transaction gets aborted. Versions created by a transaction T become visible to other transactions only after T has been committed. This protocol ensures an isolation level lower than serializability, namely *snapshot-isolation* [18]. This level ensures a transaction reads all data from the same (consistent) snapshot of the database and prevents lost updates. In DBMS which provide snapshot-isolation as highest isolation, the user can add instructions to explicitly acquire locks within transactions to enforce serializability. However, although snapshot-isolation does not provide serializability, this isolation level is considered acceptable for a wide set of applicative contexts. Also, several recent works have provided formal frameworks for the identification of classes of applications where this type of isolation level suffices to ensure serializability [19], and for detecting (and correcting) applications potentially exposed to non-serializable execution histories [20].

2.3 Brief Overview on Software Transactional Memories

STMs [21] are emerging as a highly attractive and potentially disruptive programming paradigm for concurrent applications. The early proposals for Transactional Memories (TMs) architectures date back to 90s [22]. However, the research on this topic has been largely dormant till the 2002, when the advent of multi-core processors made parallel programming exit from the niche of scientific and high-performance computing and turned it into a mainstream concern for the software industry. One of the main challenges posed by parallel programming consists of synchronizing concurrent access to shared memory by multiple threads. Programmers have traditionally used locks, but lock-based synchronization has well-known pitfalls. Simplistic coarse-grained locking does not scale well, while more sophisticated fine-grained locking risks introducing deadlocks and data races. Furthermore, scalable libraries written using fine-grained locks cannot be easily composed in a way that retains scalability and avoids deadlock and data races [23].

By bringing the concept of transaction to parallel programming, STMs allow freeing the programmers from the burden of designing and verifying complex fine-grained lock synchronization schemes. By avoiding deadlocks and automatically allowing fine-grained concurrency, transactional-language constructs enable the programmer to compose scalable applications safely out of thread-safe libraries. With an STM, programmers have just to demarcate code blocks which have to be executed as transactions. The underlying STM layer provides the illusion that transactions are executed in a serial fashion, which allows programmers to reason serially on the

correctness of their applications. Of course, the STM layer does not really execute transactions serially, instead, "under the hood" it allows multiple transactions to execute concurrently by relying on a Concurrency Control Protocol (CCP).

Today research on STMs is very active. Commercial releases of STMs do not exist yet, however many research prototypes (e.g. [12, 24, 25]), as well as prototypes for commercial systems (e.g. [26]) are available.

2.4 Database Transactions vs. Memory Transactions

Some basic differences exist between transactions in DBS (database transactions) and transactions in STMs (memory transactions) [10]. Memory transactions are executed ensuring atomicity, isolation and consistency. Unlike database transactions, they do not ensure durability, but encompass operations reading/writing data only in volatile memory. This also leads to another significant difference. As memory transactions do not require access to persistent storage when data are updated, the execution time is tipically much smaller compared to database transactions. Additionally, memory transactions are mediated by lightweight language primitives (e.g. the *atomic{}* construct) that do not suffer, e.g., of the overheads for SQL parsing and plan optimization typical of database environments. These factors make the memory transactions execution time typically two or three orders of magnitude smaller than database transactions [27], even when considering complex STM benchmarking applications.

Another difference concerns the isolation level required for memory transactions. Serializability is considered largely sufficient as isolation level in DBS. However, with serializability inconsistent data values can be read by transactions that will be subsequently aborted. It has been shown that the effects of observing inconsistent states can be much more severe in STMs than in DBS [28]. In STMs, in fact, transactions can be used to manipulate program variables whose state directly affects the execution flow of user applications. As such, observing arbitrarily inconsistent memory states (as it is allowed, for instance, by the optimistic CCPs used in DBS) could lead applications to get trapped in infinite loops or in exceptions that could never be triggered in any sequential execution. This is not the case for DBS, where transactions are executed via interfaces with precisely defined (and more restricted) semantic (e.g. with SQL interfaces), and are executed in a sandboxed component (the DBMS) which is designed not to suffer from crashes or hangs in case the concurrency control allowed observing inconsistent data snapshots. For these reasons memory transactions require a higher isolation level than serializability, namely *opacity* [29]. The latter, in addition, prevents all transactions (also transactions that will be subsequently aborted) from seeing inconsistent values of data items. As we will see in the following, most protocols used in STMs rely on the so-called *read validation* [30] to provide opacity.

2.5 STM-Oriented Protocols

Essentially, CCPs for STMs are based on combined techniques. Most of them provide opacity. Generally, in STMs implementations for object oriented languages (e.g. JAVA) a data item corresponds to a memory object (object-based STMs), while in other languages (e.g. C language) it corresponds to a memory word (word-based STMs). In STMs which use locks, one or more data items can be mapped to a single lock. This entails smaller memory structures needed for lock management. On the other hand, this can lead to false conflicts when two transactions access two different data items mapped with the same lock. The number of data items associated with a single lock can be a design choice or, as in some STMs, it can be adaptively changed at run-time. To simplify the discussion, in the following we assume that a data item is mapped with a single lock. In the rest of this section we provide some examples of STMs which use different protocols.

TL2 [12] is a STM which uses exclusive locks. Upon a write operation the accessed data item is not immediately updated, but the new value is stored in a private buffer of the thread executing transaction. Upon a read operation it is checked if the accessed data item has not been updated by another transaction after the reading transaction started. This check is said *validation*. Furthermore, it is checked if the associated lock is not held by another transaction. If one check fails then the reading transaction gets aborted. At commit time a transaction T tries to acquire an exclusive lock on each data item to update. If a lock is held by another transaction then T is aborted, otherwise, after the lock acquisition phase, all data items read by T are validated again. If one of these data items is not valid then T is aborted, otherwise all data items are updated and all locks are released. Protocols which acquire lock at commit-time are called Commit-Time Locking (CTL) protocols.

TinySTM [24] uses a protocol similar to protocol used in TL2. They differ in the lock acquisition time, namely TinySTM acquires the lock before executing the write operation. Furthermore, TinySTM uses other optimizations, e.g. as hierarchical locking to reduce the cost of validation and dynamic tuning of some configuration parameters.

Also mixed locking techniques can be used, e.g., as in SwissTM [31]. In this STM each data item has a *write* lock and a *read* lock. A write lock prevents other transactions from writing, but not from reading. A read lock prevents other transactions from reading. Upon a write operation a write lock is acquired. At commit time a transaction acquires also the read lock of each data item to be updated. Both read and write locks are released after the transaction commits or abort. This mixed technique allows to detect conflicts between write operations as soon as possible, but delays conflict detection between read and write operations at commit time. Furthermore, SwissTM uses a mechanism which, in case of conflict, favors long transactions by aborting shorter ones.

DSTM [32] is an STM which uses the concept of *ownership* of a data item. An

ownership is exclusive but revocable. Upon a write operation a transaction T_i acquires an exclusive ownership of the data item. If another transaction T_j executes a write operation on a data item owned by T_i, T_j can wait a while, but eventually T_j acquires the ownership, possibly determining the abort of T_i if it has not yet released the ownership. On a read operation a transaction T checks if the accessed data item and all data items previously read are still valid. If yes, T executes the read operation, otherwise T is aborted. At commit time a transaction validates again all read data items. If the validation fails the transaction gets aborted, otherwise all data items are updated and ownerships are released.

Finally, multiversion protocols have also been used in STM, as in JVSTM [33]. It has been designed for the Java language. JVSTM uses the so-called *versioned* boxes to store versions of data items, which are shared java object. On read operation a transaction reads the version contained in the last box made visible before the start of the transaction. On write operation the value to write is stored within a box created by the transaction. Transactions execute the commit operation in mutual exclusion by acquiring an exclusive global lock. After the lock acquisition, all boxes of updated data items are made visible to other transactions and the lock is released.

A still open debate concerns progress guarantees which CCPs for STMs have to provide. In locking protocols running transactions can be affected also by threads which execute transactions and are not running. For example, if a thread is suspended (e.g. preempted by operating system) while it is executing a transaction that holds an exclusive lock, all other running transactions which try to acquire the lock can't make progress. Non-blocking protocols are those protocols which can provide progress guarantees. *Obstruction-freedom* [34] is the weakest progress guarantee of non-blocking protocols. Obstruction-freedom ensures that if a thread runs by itself for long enough (including when other threads are suspended) then it makes progress. Obviously, also all transactions executed by the thread are guaranteed to be obstruction-free. Roughly speaking, if a transaction is executed by itself long enough to complete, it eventually successfully commits. Obstruction-free protocols guarantee that a deadlock does not occur. *Lock-freedom* [35] provides stronger progress guarantees than obstruction-freedom. It guarantees that if threads run long enough then at least one thread makes progress. The first work which introduced TMs describes a transactional memory with a protocol which guarantees lock-freedom. Lock-free protocols guarantee that livelock does not occur. Finally, *Wait-freedom* [35] provides the strongest progress guarantees. It ensures that if threads run long enough then all threads make progress. Wait-free protocols guarantee that starvation does not occur.

Despite progress guarantees may seem desirable, today there is not a common agreement on their effectiveness in STMs [36]. The STMs we presented in this chapter provide different progress guarantees. E.g. DSTM provides obstruction-freedom. The version of JVSTM we presented provides lock-freedom for only-read transaction, while a more recent version is completely lock-free. The other STMs we dis-

cussed above do not provide obstruction-freedom.

Chapter 3

Literature Overview

In the literature a number of publications which cope with the performance analysis of CCPs exist. Most of them focus on DBS. Less work has been done in the field of STMs. In this chapter we provide a literature overview focusing on the performance modeling. We start from the DBS, and after we move to the field of STMs.

3.1 Database Systems

The work done the field of DBS includes both simulation and analytic approaches. Most of analytical studies use simulation to validate the analytical models. Early studies encompass locking protocols. Afterwards also optimistic protocols have been considered. E.g. static locking protocols (i.e. where transactions predeclare all lock requests before starting) have been studied by simulation in [37] and [38], and analytical models have been proposed in [39] and [40]. Dynamic locking (as the SS2PL described in Section 2.2) has been analyzed by simulation in [38] and [41], and by an analytical approach in [42]. Optimistic concurrency controls have been analyzed by analytical models in [43] and by simulation in [44] and [45].

The majority of studies rely on system models based on the following assumptions. The database contains a fixed number of data items. A data item can represent, e.g., a record, a page or an entire table. Transactions perform *n* operations uniformly distributed over the whole set of data items. Some studies assume the presence of data skew, e.g by considering hot spot using the *b*-*c* rule (i.e. a fraction of *b* operations access a fraction of *c* data items in the database [42]). In other studies data items are partitioned in different sets, and fixed-sized subsets of transaction operations access different data item sets [9]. In studies where both read and write data accesses are considered, each data access is a read/write with a fixed probability (e.g. in [46]).

In some studies, to model a workload with more transaction profiles, transactions are grouped into different classes, where the number of operations and the read/write probability depend on the transaction class [9]. A constant number of transactions in the system is assumed in some models (e.g. [42]). In other works transactions are assumed to be generated by a fixed number of users s (closed system model) [46, 6]. In some studies a 'think time' exponentially distributed with a fixed average value is considered between transactions executed by users [47]. In closed system models the transaction throughput is used as system performance indicator. When s is assumed to be large, an open system model is considered, and transaction arrivals are modeled as a Poisson process with fixed average arrival rate (open system model) ([48, 9]). In this case the system performance is evaluated through the average transaction response time. As concerns the hardware resources, CPU and disk are considered. Two different models for the hardware resources have been used. The first one is a simple model where resources are assumed to be 'infinite', i.e. a transaction never waits for a CPU or I/O request ([44, 41]). In this model transactions interfere due to data contention, but they do not compete for hardware resources. In the second model a number of CPU and/or disks are explicitly accounted for (e.g. see [47]). If a resource is serving a transaction then the arriving transactions are queued. Typically, the CPU and I/O service demands associated with each operation depend on the operation type (read/write), and they are assumed to be fixed or exponentially distributed random variables. CPU and disks are modeled as a queuing network, where a transaction operation involves CPU and disk requests. In some studies the presence of a memory buffer is assumed ([49, 9]). A subset of data items are contained in the buffer and are replaced according to a policy (e.g. least-recently used). With a memory buffer, when a transaction accesses a buffered data item the I/O time is not considered.

A contribution aimed to explore the effects of the hardware resources on the CCP is provided in [6]. In this work the authors highlight as most of the previous results were seemingly contradictory, and they presented a simulative study aimed to evaluate pessimistic and optimistic protocols. The main result of this work is that, in environments with limited resources, pessimistic protocols perform better, because they tend to prevent further resource utilizations by blocking conflicting transactions. Conversely, with low resource utilizations, so that further wasted work can be tolerated, optimistic protocols are preferable. Finally, the authors claimed that the seeming contradictions of previous studies were due to the differences in the underlying assumptions concerning the hardware model.

An analytical model based on a recursive solution has been proposed in [50]. The work addresses the case of shared and exclusive locks with multiple transaction classes. A performance study encompassing a restart-oriented protocol is described in [51]. In this work a method to improve the response time based on *volatile savepoints* is considered. A savepoint allows to reduce the wasted processing time when a transaction is restarted. The ideal distribution of checkpoint over the transaction lifetime is evaluated. Other analytical works have been presented in [52, 47, 53], for

which, most of results have been surveyed in [47]. In the latter work a queuing network model for hardware resources is presented and an approximated analytical approach for the dynamic locking is considered. Furthermore, the work revises models for optimistic and mixed protocols, and some possible improvements are proposed.

A mean value analysis methodology aimed to model lock-based and optimistic protocols is presented in [9]. It brings together various results by different previous studies of the authors. On basis of probabilistic assumptions, the proposed methodology allows to obtain simple approximate expressions for evaluating various probabilities and other quantities which characterize the dynamics of the execution of a transaction (e.g. the conflict probability, the mean wait time on conflict, and the average number of transaction restarts). The provided expressions allow to build approximate models, which can be coupled with a hardware model and can be resolved via iterations.

Finally, few publications addressed the evaluation of the performance of MVCC protocols, and they are mostly based on simulative approaches (e.g. [54]). Analytical models have been proposed in [55, 56]. The objective of these studies was to provide an analysis of the storage cost for maintaining data items, by relying on the evaluation/prediction of the space occupancy for the different versions of the data items vs, e.g., the data update frequency.

3.2 Software Transactional Memories

As concerns STMs, the wide majority of existing performance studies focus on the evaluation of STMs implementations. Among these, some studies compare different STMs prototypes (e.g., [12, 24, 31]), while other studies focus on the assessment of alternative design choices [25]. Some studies are aimed to evaluate adaptive policies [57], and finally, other studies use STMs implementations to evaluate alternative conflict detection and validation strategies (e.g. [30]). A very limited number of studies rely on model-based approaches. A simulation study has been presented in [58]. The authors propose a simulation model to analyze the performance with three protocols, namely a pessimistic protocol and two optimistic ones, with write buffering and with in-place memory updates, respectively. The pessimistic protocol relies on the typical shared/exclusive lock mechanism and transactions are aborted on lock conflict. Conversely, with the optimistic protocols transactions can perform write operations without checking if a concurrent transaction has already read the accessed data item, and read operations require data validation. With write buffering, when a transaction executes a write operation, it stores the new value locally, and the value is made visible at commit time. With in-place memory update, the new value is immediately stored in shared memory, and an undo log is used to restore the previous value of the updated data item if the transaction gets aborted. By using the simulation model and synthetic workloads, the study encompasses the evaluation of three performance

indicators, i.e. the mean number of restarts per transaction, the mean number of steps executed and the mean number of locks held by a transaction. The authors show as in their tests optimistic protocols perform better in terms of mean number of restarts.

The studies based on analytical are presented below. The same authors of the above-mentioned simulation study also proposed an analytical approach [59]. They provide an analytical framework for STM systems where the same protocols as in the simulation study are considered. Subsequently, they extended the framework for the case of optimistic protocol with lazy locking [60]. The framework is based on an absorbing discrete-time Markov Chain [61] which models the execution of a transaction. The system model assumes a fixed number k of active transactions in the system, each one executing N read/write accesses uniformly distributed on L data items, with P_w as probability for an access to be a write. Given a protocol, the outcome of the analytical model depends exclusively on the aforesaid four parameters. The framework allows to evaluate the same performance indicators as the simulation model proposed by the authors. The analytical models have been validated by performing comparative tests with the output of a discrete event simulator. In these studies the performance indicators are evaluated with respect to the probability P_w and the number of concurrent transactions N. Further, the authors present some results calculated by means of the analytical model. They confirm the advantages of the optimistic protocols, unless P_w assumes very large or very small values. The study presented in [62] proposes two analytical models to compare the performance of the typical lock-based approach for the execution of a critical section and of a simple version of the CTL protocol (see Section 2.5). In the case of the CTL protocol, the authors consider transactions which speculatively access the critical section. At commit-time, if a concurrent transaction has committed, then the committing transaction aborts and restarts, otherwise successfully commits. The system model consists of N processors witch execute N threads, each one repeatedly issuing critical sections/transactions. All critical section/transaction executions are assumed to access to the same memory location protected by a unique global lock. The duration of a critical section/transaction is exponentially distributed. Concerning the transaction model, the durations of the abort phase and the commit phase are not considered. A queuing based model have been used for both critical sections and transactions. Both the models have been validated by simulation. According to the results obtained with models, the typical critical section approach generally outperforms the transaction approach, while with low contention they are comparable. The same authors subsequently proposed other two works. Also in these works they use a queuing-based approach. In the first work [63] they consider a system with a fixed number of treads, each one executing on a processor. Threads execute a number of different transaction types according to a given distribution probability. The transaction types differ in the number of checkpoints executed. While executing a checkpoint, a transaction may be aborted. The execution of transactions is modeled by means of a continuous-time Markov Chain [61], where a state is represented by the number of active transac-

tions and by the number of checkpoints executed by the first transaction that will commit. A transition occurs when a new transaction starts, commits, aborts or executes a checkpoint. The conflict detection is simply assumed to be lazy or eager (see Section 2.5) depending on the number of checkpoints of transactions. Read and write accesses are not differentiated and the conflicting probability between transactions is considered to be fixed and to be an input parameter for the model. In the tests performed by the authors, the conflict probability has been evaluated by experimental measurements on a real system. The model has been validated against a real system by using STAMP benchmark [11] and by comparing the average transaction response time predicted by the analytical model. In the second work [64] the authors propose a similar approach. Unlike the previous approach, in this model a state of the Markov Chain is represented by the number of active transactions and by the number of transactions that will commit. A transition occurs when a new transaction starts, commits or aborts. Also in this work read and write accesses are not differentiated. The conflict detection relies on checkpoints and the conflict probability is calculated according to the size of the overlapped sets of data items accessed by concurrent transactions. Also this model has been validated by using STAMP benchmark.

Chapter 4

Performance Modeling of MultiVersion Concurrency Control Protocols

4.1 Introduction

Several DBMS rely on MVCC protocols (Section 2.2). These protocols are also largely used in other data management systems (as JBossChache [65]), and are gaining ground in STMs (e.g. JVSTM [33]). The exploitation of multiple data versions allows the system to immediately serve, via a version of the accessed data item, a read operation, which with other protocols could entail a delay or an abort of the reading transaction in case of conflict. This approach improves the level of concurrency between transactions and, mainly, it makes multiversion protocols especially suitable for a read-intensive workload. Such a kind of workload is representative of several applications, as most of the Web-based ones.

In Section 2.2 we also discussed the most widely used MVCC protocol in DBMS. This protocol provide the snapshot-isolation level and it has been adopted in both mainstream proprietary and open source DBMS (e.g. Oracle Database [16] and PostgreSQL [66]). In this chapter we address the performance modeling of the MVCC and we present an analytical performance model tailored for the aforesaid protocol. The model we propose is able to capture the transaction execution dynamics due to the mix of mechanisms used by this protocol and allows to evaluate both the main performance indicators (e.g. average transaction response) and other specific indicators

for this protocol (e.g. the version check failure probability, the frequency of creation of versions, the lock holding time). In our analysis we consider a level of abstraction which makes the model independent of some specific aspects of systems. In particular, the model is independent of the specific policy adopted by the system to retrieve data item versions (e.g. explicit storing, as in PostgreSQL, or dynamic regeneration of the required version via rollback segments, as in Oracle Database). This makes the model suitable for a variety of version management mechanism implementations.

The model has been validated via a simulation study. The used simulation model explicitly mimics the dynamics of the transaction executions in a database system where the transction execution is regulated by the considered MVCC protocol.

The remainder of this chapter is structured as follows. In Section 4.2 we provide a description of the protocol rules. The analytical model is presented in Section 4.3, where we first provide a basic version of the model, and after we present an extended version copying with multiple transaction classes and non-uniform data access. Finally, we present a model validation study in Section 4.6.

4.2 An Overview of MultiVersion Concurrency Control Protocol Ensuring Snapshot-Isolation

The protocol we are analyzing combines different concurrency control techniques, namely data versioning, transaction blocking and transaction restart. In the follow we describe more in depth the protocol and the basic implementation mechanisms.

Each transaction in the system is associated with a so-called *Start-Timestamp*, whose value is set when the transaction starts. This value is used to determine the set of transactions that are concurrent with T. In particular, this set is formed by the transactions that are active when Start-Timestamp is set for T, plus the transactions with timestamp greater than Start-Timestamp. When a transaction T tries to write a data item x that has not yet been accessed by this same transaction, *version check* is performed to determine whether no concurrent transaction that wrote x has already been committed. In the positive case, version check is said to have failed, and T is immediately aborted. Otherwise, T tries to acquire an lock on x, which can lead to a wait phase in case the lock is currently held by any other active transaction T'. In the latter case, if T' is eventually committed, then T gets aborted in order to avoid the so called *lost update phenomenon* [18]. After the lock acquisition, T is allowed to create a new version of x. If T wants to read/write a data item x previously written during its execution, the version of x just created by T is immediately supplied. Instead, a read operation on a data item x not previously written by T is served by accessing the version of x that has been committed by the most recent transaction not concurrent with T. In this way all read operations are never blocked and do not cause transaction abort. When T commits or aborts, all the acquired locks are released. In case of

commit, all the data item versions created by T become visible to other transactions.

4.3 The Analytical Model

4.4 System Model

We consider an open system model where transactions arrive according to a Poisson Process. A transaction consists of a begin operation, which is followed by a number of read or write operations, each one accessing a single data item, and finally by a commit operation. Begin, write and commit operations are assumed to require a mean number of CPU instructions denoted with nI_b, nI_w and nI_c, respectively. CPU instructions to support read accesses are modeled in a slightly more complex way, as a reflection of the fact that a read access can require traversing the history of data item versions to retrieve the correct one. This is modeled by assuming for a read access a baseline of a mean number of nI_r^F CPU instructions, plus a mean number of nI_r^V CPU instructions for each traversed version. In the case of transaction abort, we assume the execution of a mean number of nI_a CPU instructions. Also, the transaction is rerun after a randomly distributed back-off time with mean value $T_{backoff}$. When a read or write operation is performed, if the accessed data item is not in the buffer then a disk access occurs. Each disk access is assumed to require a fixed latency $t_{I/O}$. The CPU is modeled as an M/M/k queue, where k is the number of CPUs, each of which is assumed to have a processing speed denoted as $MIPS$.

We first present a basic version of the analytical model, relying on the following additional assumptions: (1) transactions belong to a unique class with a mean number of N_w write operations and N_r read operations per transaction and with an arrival rate λ, (2) transactions perform accesses uniformly distributed over the whole set of D data items. These assumptions will be then removed while presenting an extended version of the analytical model.

From this point onwards, all the assumptions we make in this chapter are finalized only to the construction of the analytical model. They are not considered in the simulation model we used in the validation study.

In our analytical model we ignore the effects on performance of transaction aborts and restarts due to deadlocks. Previous studies on locking protocols (e.g., [67], [42]) have shown that these effects are negligible with respect to the data contention effects. Furthermore, given that the above-mentioned studies deal with the 2PL protocol, assuming that deadlock does not occur reveals even more realistic in case of MVCC since it does not use locks on read operations, hence further reducing the deadlock probability.

Finally, we assume that the system is stable and ergodic.

4.4.1 Basic Analytical Model

Transaction Execution Model

We assume that each transaction is formed by an interleaving of read/write operations such that the are N_r reads uniformly mixed with N_w writes. This choice is motivated by the fact that a transaction never aborts while executing a read operation. Thus, in order to evaluate the average transaction response time, it is important the average number of read operations executed by a transaction, including the operations executed by the aborted runs of the transaction. The execution of a transaction is modeled through a directed graph. Figure 4.1 shows an example for a transaction with $N_w = 2$. Each node represents a state of a transaction execution corresponding to a specific phase of the transaction lifetime. The label of an arc from a node p to a node q represents the transition probability from state p to state q. If the label is omitted, than the transition probability is intended to be 1. Obviously, the sum of all transition probability values for outgoing arcs from a node must be 1. The states labelled with *begin*, *commit* and *abort* are used to model the execution of the respective operations. Instead, as for read/write accesses to data items, we use a different state labelling approach to denote the corresponding phases. Considering that the sequence of N_r read operations performed by a transaction is uniformly distributed across the N_w write operations, we assume to have a write operation after executing $N_r^S = N_r/(N_w + 1)$ read operations (see Figure 4.1). According to this rule, state $\hat{0}$ represents the phase in which the initial N_r^S read operations are performed before the first write access, and states $\hat{i}$ (with $1 \leq i \leq N_w$) represent phases in which a write operation has been issued, followed by a mean number of N_r^S read operations.

According to the MVCC description provided in Section 4.2, when a write operation needs to be carried out, version check is performed. If version check for the i-th write fails, the transaction is aborted. The corresponding state transition probability is denoted as $P^I_{A,i}$. (The related arc starts from state $\widehat{i-1}$ and ends to state *abort*.) On the other hand, if version check succeeds (this occurs with probability $1 - P^I_{A,i}$) a wait phase for lock acquisition occurs with probability P_{cont}, corresponding to the probability that a lock is being held by another transaction.

Note that, by assumption (2) in Section 4.4, P_{cont} is independent of the accessed data item. Thus, the probability of transition from state $\widehat{i-1}$ to state $\tilde{i}$ can be expressed as $P_{w,i} = (1 - P^I_{A,i})P_{cont}$. On the other hand, the probability that a lock is immediately granted after version check is $1 - P_{cont}$. Thus, the probability of transition from state $\widehat{i-1}$ to state $\hat{i}$ is $P_{E,i} = (1 - P^I_{A,i})(1 - P_{cont})$. A transaction in a waiting state $\tilde{i}$ gets aborted with probability $P^C_{A,i}$, which we will subsequently evaluate. When a read/write operation is executed, the accessed data item might be already available in the buffer pool, otherwise a disk access is needed. We denote with P_{BH1} the expected buffer hit probability. However, as suggested in [9], in order to provide a more accurate evaluation of the effects of buffer hits in case of transaction restart, a differ-

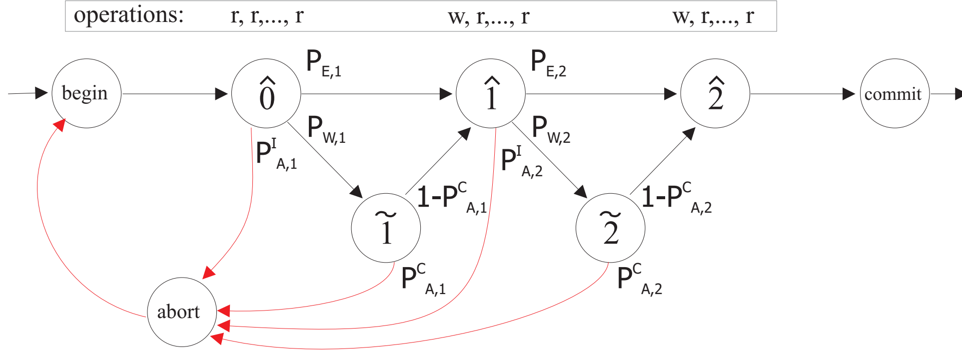


Figure 4.1: Base Transaction Execution Model.

ent value of the expected buffer hit probability P_{BH2} is considered when, in a rerun, the transaction accesses a data item already accessed prior to the abort. Both P_{BH1} and P_{BH2} are intended as input parameter for our model, whose value will reflect specific choices for what concern buffer pool size and related replacement policies.

According to the previous considerations, the graph modeling transaction execution is extended as in Figure 4.2. Specifically, the graph is partitioned into $N_w + 1$ subgraphs $G_0, G_1, \ldots, G_w$. Subgraph G_0 represents the first transaction run, for which we consider P_{BH1} as the buffer hit probability for all read/write operations. Subgraph G_k (with $1 \leq k \leq N_w$) represents reruns of the transaction executed when a previous run has already accessed all data items before the k-*th* write, and then has been aborted. Hence, in the subgraph G_k we use P_{BH2} as the buffer hit probability before the k-*th* write, while P_{BH1} is used as the buffer hit probability for subsequent data accesses. For example, referring to Figure 4.2, if the transaction aborts in state $\tilde{1}$ of subgraph G_0, the subsequent run is represented by subgraph G_1, where P_{BH2} is the buffer hit probability for all read operations occurring up to the 1-*st* write.

In the extended graph, we use the subscript '$_{ki}$' to label arcs of subgraph G_k. Hence, we have $P_{W,ki} = (1 - P^I_{A,ki})P_{cont}$ and $P_{E,ki} = (1 - P^I_{A,ki})(1 - P_{cont})$.

Transaction Response Time

When a new transaction starts it will require a number N of (re)runs, depending on the number of experienced aborts, to commit. We denote with N_{G_k} the expected number of times that a run described by subgraph G_k is (re)started before the transaction commits. For a run associated with a generic subgraph G_k, we denote with $\hat{P}_k(i)$ the probability to reach state $\hat{i}$ (i.e. the transaction does not abort before). This probability value iteratively depends on the probability to reach state $\widehat{i-1}$, thus

$$\hat{P}_k(0) = 1,$$

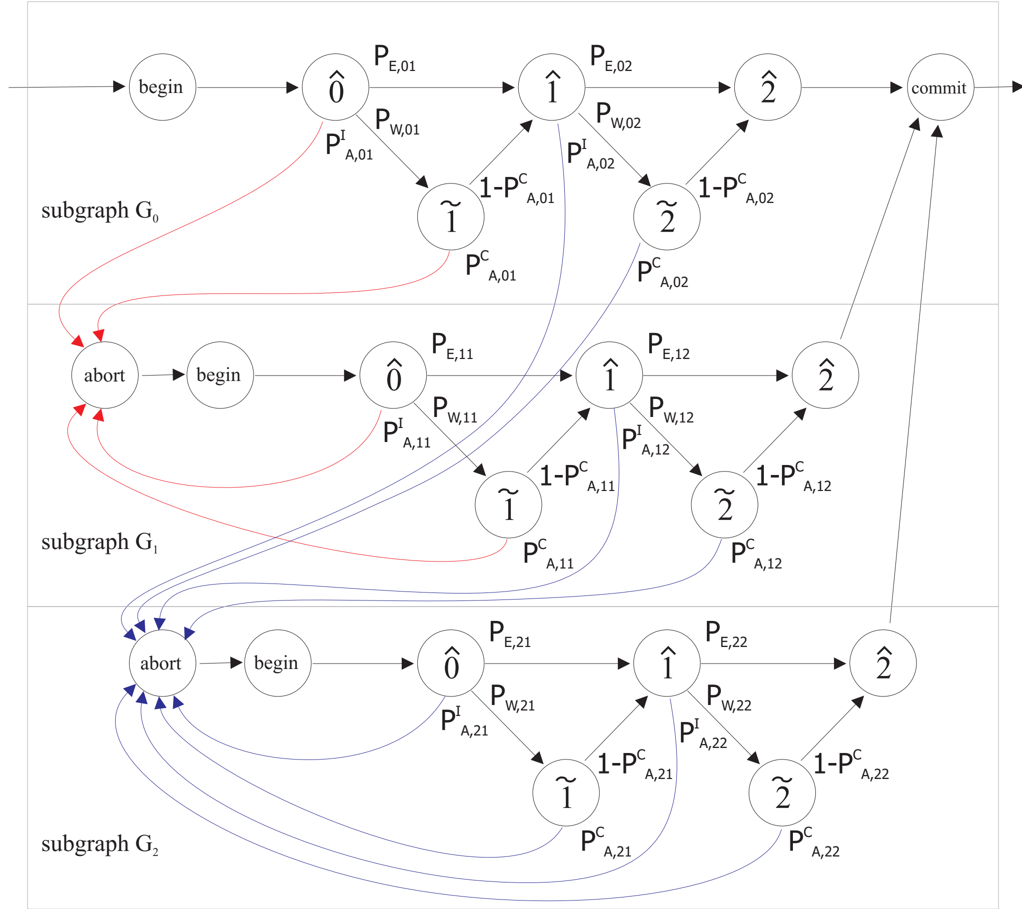


Figure 4.2: Transaction Execution Model.

$$\hat{P}_k(1) = \hat{P}_k(0)(1 - (P^I_{A,k1} + P_{W,k1}P^C_{A,k1})),$$

and, for a generic state $\hat{i}$,

$$\hat{P}_k(i) = \hat{P}_k(i-1)(1 - (P^I_{A,ki} + P_{W,ki}P^C_{A,ki})).$$

Finally, by construction, $\hat{P}_k(commit) = \hat{P}_k(N_w)$.

N_{G_k} can be calculated considering that it iteratively depends on the number of runs described by subgraphs G_j, with $j < k$, and, for each run, on the probability to reach states $\hat{j}$ or $\tilde{j}$ and that the transaction is subsequently aborted. Further, it depends on the probability for the transaction to be aborted before to reach the state $\hat{k}$ during the execution of a run described by the subgraph G_k. Hence, we have $N_{G_0} = 1$ and, for $1 \leq k \leq N_w$, N_{G_k} can be calculate as

$$N_{G_k} = \hat{P}_k(k) \sum_{j=0}^{k-1} N_{G_j} \hat{P}_j(k-1)(P^I_{A,jk} + P_{W,jk} P^C_{A,jk}) + N_{G_k}(1 - \hat{P}_k(k)).$$

Simplifying previous equation we get

$$N_{G_k} = \frac{1}{\hat{P}_k(k)} \sum_{j=0}^{k-1} N_{G_j} \hat{P}_j(k-1)(P^I_{A,jk} + P_{W,jk} P^C_{A,jk}).$$

We denote with R_{begin}, $R_{k\hat{i}}$, $R_{k\tilde{i}}$, R_{com} and R_{abt}, respectively, the mean residence time for states *begin*, $\hat{i}$, $\tilde{i}$, *commit* and *abort*. On average, runs represented by subgraph G_k spend, R_{begin} time in state *begin*, plus time in other states, according to the probability for these states to be reached. Hence we get

$$\hat{R}_{ki} = \hat{P}_k(i) R_{k\hat{i}},$$

$$\tilde{R}_{ki} = \hat{P}_k(i-1) P_{W,ki} R_{k\tilde{i}},$$

$$R_{k\,com} = \hat{P}_k(commit) R_{com},$$

$$R_{k\,abt} = (1 - \hat{P}_k(commit)) R_{abt}.$$

State $\hat{0}$ of each subgraph is always visited in each run, thus $\hat{R}_{k0} = R_{k\hat{0}}$. Therefore, the mean run execution time, for a run represented by subgraph N_{G_k}, is

$$R_{G_k} = R_{begin} + \hat{R}_{k0} + \sum_{i=1}^{N_w} (\hat{R}_{ki} + \tilde{R}_{ki}) + R_{k\,com} + R_{k\,abt.}$$

The mean transaction response time is

$$R_{tx} = \sum_{k=0}^{N_w} N_{G_k} R_{G_k}.$$

Lock Holding Time

A lock is acquired when visiting each state $\hat{i}$ (with $1 \leq i \leq N_w$), and is released at end of the run. If the run terminates with transaction commit, then all its locks are released upon completion of the phase associated with the state *commit*. Instead, if the run terminates with transaction abort, then the locks are released upon entering the state *abort*. To simplify, as in other models for locking protocols [9], we assume lock release in case of abort as an instantaneous action, which does not contribute to lock holding time. Hence, locks are held by a transaction in the time interval between the acquisition and either the start of the abort phase, or the end of the commit phase. Using the expressions previously defined for the mean time spent in each state, the

mean lock holding time for the i-*th* acquired lock in a generic run represented by subgraph G_k can be expressed as

$$T_{H,ki} = \sum_{j=i}^{N_w} \hat{R}_{ki} + \sum_{j=i+1}^{N_w} \tilde{R}_{ki} + R_{k\,com}.$$

Hence, the mean lock holding time for the i-*th* acquired lock, evaluated across all the (re)runs of the transaction, can be expressed as

$$T_{H,i} = \sum_{k=0}^{N_w} N_{G_k} T_{H,ki},$$

and the mean lock holding time is

$$T_H = \frac{1}{N_w} \sum_{i=1}^{N_w} T_{H,i}.$$

Lock Contention Probability

As already hinted, due to assumption (2) in Section 4.4, the lock contention probability P_{cont} is uniform across all the data items, thus being independent of the specific accessed data. Given that transactions arrive according to a Poisson Process, we also use this assumpion for the lock arrivals. Hence, the lock contention probability can be expressed as the expected data utilization factor, namely

$$P_{cont} = \frac{\lambda N_w T_H}{D}.$$

Lock Waiting Time

Now we evaluate $R_{k\tilde{i}}$, namely the average wait time experienced by a transaction T when tries to acquire a lock held by any other active transaction T'. We remark that if T' successfully commits then T gets aborted. $R_{k\tilde{i}}$ corresponds to the average residence time in state $\tilde{i}$ of subgraph G_k. We consider the approximation in which at most one transaction is queued for lock acquisition on whichever data item. This assumption is considered also in other studies on 2PL protocol (e.g. [53]). Note that in our case this approximation is further supported by the fact that, differently from 2PL, in this MVCC protocol if a transaction T' commits, then any transaction T waiting for a lock held by T' gets immediately aborted. Hence, if T' commits, T needs to wait for the completion of at most one transaction. We approximate $R_{k\tilde{i}}$ as the mean residual time required by T' to terminate the current run (with either commit or abort), evaluated at the time of conflict occurrence, namely when T enters

state $R_{k\tilde{i}}$. The probability that, at the time of conflict occurrence, T' is executing a run modeled by subgraph G_k is

$$P_{cont,k} = \frac{N_{G_k} T_{H,k}^{Tot}}{T_H^{Tot}},$$

where

$$T_{H,k}^{Tot} = \sum_{i=1}^{N_w} T_{H,k,i},$$

and

$$T_H^{Tot} = \sum_{k=0}^{N_w} N_{G_k} T_{H,k}^{Tot}.$$

Thus, at conflict time, the probability values for T' to be in states $\hat{i}$ and $\tilde{i}$ (with $1 \leq i \leq N_w$) within subgraph G_k are

$$P_{cont,k\hat{i}} = \frac{\hat{R}_{ki}}{T_{H,k}^{Tot}} i,$$

$$P_{cont,k\tilde{i}} = \frac{\tilde{R}_{ki}}{T_{H,k}^{Tot}} (i-1),$$

and, finally, the probability for T' to be in state *commit* is

$$P_{cont,k\,com} = \frac{R_{k,com}}{T_{H,k}^{Tot}} N_w.$$

Now we introduce the conditional probability $\hat{P}_k(j|i)$ to reach state $\hat{j}$ during a (re)run associated with subgraph G_k, given that state $\hat{i}$ (with $i \leq j$) has already been reached during that same run. For $j = i$ we have

$$\hat{P}_k(j|i) = 1,$$

and, for $j > i$, we have the following iterative expression

$$\hat{P}_k(j|i) = \hat{P}_k(j-1|i)(1 - (P_{A,k\,i+1}^I + P_{W,k,i+1} P_{A,k,i+1}^C)).$$

If, at conflict time, T' was executing in state $\hat{i}$ (with $1 \leq i \leq N_w$), then we calculate the residual lock holding time as

$$\tilde{R}_{k\hat{i}} = a_{k\hat{i}} \hat{R}_{k,i} + B_{k\hat{i}}.$$

In the above equation $a_{k\hat{i}} \hat{R}_{k,i}$ is the average residual time of T' in the state $\hat{i}$ (remember that the mean residence time is $\hat{R}_{k,i}$) and $B_{k\hat{i}}$ is the additional time to terminate the

current run given that T' has reached state $\hat{i}$. Note that $a_{k\hat{i}}$ depends on the distribution of the residence time in the state $\hat{i}$. This distribution is affected by the buffer hit probability. In fact, if the accessed data item is not in the buffer then the I/O time predominates over the CPU time. Conversely, if the data item is in the buffer then the distribution depends on the distribution of the CPU service time. We remember that the I/O time is assumed to be fixed. Hence, as proposed in [9], we approximate the residual time by setting $a_{k\hat{i}} = 2(1 - P_{BH1})$ for $1 \leq k \leq N_w$ and $1 \leq i < k$, and $a_{k\hat{i}} = 2(1 - P_{BH2})$ for $1 \leq k \leq N_w$ and $k < i \leq N_w$.
As concerns $B_{k\hat{i}}$, we have

$$B_{k\hat{i}} = \sum_{j=i+1}^{N_w} \hat{P}_k(j|i)(R_{k\hat{i}} + P_{W,ki} R_{k\tilde{i}}) + \hat{P}_k(N_w|i)R_{com}.$$

Similarly, if at conflict time T' is in state $\tilde{i}$ (with $2 \leq i \leq N_w$) we have

$$\tilde{R}_{k\tilde{i}} = b\tilde{R} + B_{k\tilde{i}},$$

where

$$B_{k\tilde{i}} = \sum_{j=i}^{N_w} \hat{P}_k(j|i)R_{k\hat{i}} + \sum_{j=i+1}^{N_w} \hat{P}_k(j|i)(P_{W,ki} R_{k\tilde{i}}) + \hat{P}_k(N_w|i)R_{com}.$$

In this case we approximate the average residual time $bB_{k\tilde{i}}$ by using $b = 1$ (as in the case of exponential distribution service time).
Finally, if at conflict time T' is executing in state *commit*, we have

$$\tilde{R}_{com} = cR_{com}.$$

Also in this case we approximate $\tilde{R}_{com}$ by using $c = 1$
Overall, we express $R_{k\tilde{i}}$ as

$$R_{k\tilde{i}} = \sum_{k=0}^{N_w} P_{cont,k}(\sum_{i=1}^{N_w} P_{cont,k\hat{i}} \tilde{R}_{k\hat{i}} + \sum_{i=2}^{N_w} P_{cont,k\tilde{i}} \tilde{R}_{k\tilde{i}} + P_{cont,k\,com} \tilde{R}_{com}).$$

Version Check Failure Probability

Version check for transaction T upon write access to data item x fails if a concurrent transaction wrote x and committed. As we assumed the system to be stable, the rate of commit events is equal to the transaction arrival rate λ. By approximating commit events occurrence as a Poisson Process, for assumption (2) in Section 4.4,

we have that version check failure probability corresponds to the probability that the requested data item has been updated by at least one concurrent transaction during the time period from the startup of transaction T and the data access instant. Hence, the version check failure probability $P^I_{A,ki}$ while performing the i-*th* write during an run modeled by subgraph G_k can be expressed as

$$P^I_{A,ki} = (1 - \exp(-\frac{\lambda N_w}{D}\vec{R}_{ki})),$$

where $\vec{R}_{ki}$ is time between the startup of T and version check occurrence. This time can be evaluated as

$$\vec{R}_{ki} = \sum_{j=0}^{i-1}(R_{k\hat{\imath}} + P_{W,ki}R_{k\,\widetilde{i+1}}).$$

Version Access Cost Model

Existing implementations of multiversion concurrency control rely on different approaches for the management of data item versions. Some products (e.g. Oracle Database [68]), explicitly store only the most recent committed data item versions, so to reduce space usage, and exploit the information stored in the DBMS log to reconstruct data pages when an older data item version is required. Instead, other products use explicit version storing (e.g. PostgreSQL [66]). Given that our aim is to provide an analytical model independent of specific implementation issues, we model the cost of a read operation as $nI^F_r + nI^V_r N^V_{read}$, where N^V_{read} is the number of backward traversed data item versions in order to retrieve the correct one. With this approach, further implementation dependent management costs (e.g. garbage collection cost) could be modeled as additional workload on hardware resources, which we neglect in the present analysis for simplicity.

In order to solve the previous read cost model, we now evaluate the average number of backward traversed versions for each read operation in state $\hat{\imath}$ of whichever subgraph G_k, namely $N^V_{read,ki}$.

Given assumption (2) in Section 4.4, committed versions of a data item are born with an approximated rate $\sigma = \lambda N_w/D$. Denoting with $\Delta T_{s,ki}$ the time interval between transaction startup and the arrival in state $\hat{\imath}$ of subgraph G_k, we can then approximate $N^V_{read,ki}$ as

$$N^V_{read,ki} = \Delta T_{s,ki}\,\sigma.$$

Note that this value corresponds to the average number of versions committed during the time interval $\Delta T_{s,ki}$. Using $\vec{R}_{ki}$ previously introduced, we approximate $\Delta T_{s,ki}$ as

$$\Delta T_{s,ki} = \vec{R}_{ki} + R_{k\hat{\imath}}/2.$$

Hardware Resource Model

The CPU load (number of instructions) due to the execution of a run represented by subgraph G_k is

$$C_k = nI_b + N_r^S(nI_r^F + nI_r^V N_{read,k\,0}^V) + nI_{vc} +$$
$$+\hat{P}_k(i)\sum_{i=1}^{N_w-1}(nI_w + N_r^S(nI_r^F + nI_r^V N_{read,ki}^V) + nI_{vc}) +$$
$$+\hat{P}_k(N_w)(nI_w + N_r^S(nI_r^F + nI_r^V N_{read,k\,N_w}^V)) +$$
$$+\hat{P}_k(commit)nI_c + (1 - \hat{P}_k(commit))nI_a.$$

where we denote with nI_{vc} the average number of CPU instructions to perform version check. Note that version check occurs in states $\hat{i}$ (with $0 \leq i \leq N_w - 1$). The CPU utilization can be expressed as

$$\rho = \frac{\lambda \sum_{k=0}^{N_w}(N_{G_k} C_k)}{k\ MIPS}$$

We denote with $p[queuing]$ the wait probability for CPU requests, which can be easily computed by leveraging classical queuing theory results on M/M/k queues [69]. Then, defining $\gamma = 1 + p[queuing]/(k(1 - \rho))$, we can evaluate the average response time for each state of the graph as

$$R_b = \gamma \frac{nI_b}{MIPS},$$
$$R_{com} = \gamma \frac{nI_c}{MIPS},$$
$$R_{abt} = \gamma \frac{nI_a}{MIPS},$$
$$R_{k\hat{i}} = \gamma \frac{N_r^S(nI_r^F + nI_r^V N_{read,ki}^V) + nI_w + nI_{vc}}{MIPS} + T_{IO} G_{ki}$$

where $nI_w = 0$ for $i = 0$, $nI_{vc} = 0$ for $i = N_w$, and G_{ki} is expressed as

$$G_{ki} = N_r^S P_{BH1}$$

for k=0 and i=0,

$$G_{ki} = N_r^S P_{BH2}$$

for $1 \leq k \leq N_w$ and $i = 0$,

$$G_{ki} = N_r^S P_{BH2} + P_{BH2}$$

for $1 \leq k \leq N_w$ and $i = k$,

$$G_{ki} = (N_r^S + 1)P_{BH2}$$

for $1 \leq k \leq N_w$ and $1 \leq i < k$,

$$G_{ki} = (N_r^S + 1)P_{BH1}$$

for $1 \leq k \leq N_w$ and $k < i \leq N_w$.

4.4.2 Numerical Resolution

The proposed model, analogously to, e.g., those in [70, 52, 9], can be solved via an iterative approach. Once assigned numerical values to all parameters described in Section 4.4 and to nI_{vc}, P_{BH1} and P_{BH2}, and once the initial values of all other probabilities are set equal to 0, all model parameters can be evaluated via the provided equations, and can be used as the input for the next iteration. We have experimentally observed that, if the chosen initial values define a stable system, then the computation converges in a few iterations.

4.4.3 Extended Analytical Model

We provide in this section an extension of the model, which is able to handle both variable length transactions and non-uniform data access. In practice, this means removing assumptions (1) and (2) in Section 4.4.

Variable Length Transactions

We adopt a transaction clustering approach based on the average number of operations executed by transactions within a same class. Specifically, transactions with a similar number of read and write operations are grouped into a class C^{rw}, where r and w identify the corresponding number of expected reads and writes.

Further we denote with R and W two sets of integers, which are used to list the average number of read and write operations of different classes. Thus, for each C^{rw}, $r \in R$ and $w \in W$. We denote with $X = \{(r, w)\}$ the set of all (r, w) pairs characterizing the workload, hence $|X|$ is the total number of classes. A transaction belongs to class C^{rw} with probability P^{rw}_{TC}, thus the average arrival rate of transactions of class C^{rw} is $\lambda^{rw} = \lambda P^{rw}_{TC}$. Now we redefine some parameters appearing in the basic model in order to capture the presence of transaction classes. To this end, we use the superscript '^{rw}' to denote the parameter redefinition for each class C^{rw}. We have

$$P^{rw}_{W,ki} = (1 - P^{I,rw}_{A,ki})P_{cont},$$

where $P^{I,rw}_{A,ki}$ is version check failure probability for a transaction of class C^{rw}, and

$$\hat{P}^{rw}_k(i) = \hat{P}^{rw}_k(i-1)(1 - (P^{I,rw}_{A,ki} + P^{rw}_{W,ki}P^{C}_{A,ki})).$$

The expected number of runs whose execution is represented by subgraph G_k for a transaction of class C^{rw} is

$$N^{rw}_{G_k} = \frac{1}{\hat{P}^{rw}_k(k)} \times$$

$$\times \sum_{j=0}^{k-1} N_{G_j}^{rw} \hat{P}_j^{rw}(k-i)(P_{A,jk}^{I,rw} + P_{W,jk}^{rw} P_{A,jk}^{C}).$$

The mean times spent in the different states by a transaction of class C^{rw} in a run are the following

$$\hat{R}_{ki}^{rw} = \hat{P}_k^{rw}(i) R_{k\hat{i}}^{rw},$$

$$\tilde{R}_{ki}^{rw} = \hat{P}_k^{rw}(i-1) P_{W,ki} R_{k\tilde{i}},$$

$$R_{k\,com}^{rw} = \hat{P}_k^{rw}(commit) R_{com}^{rw}$$

and

$$R_{k\,abt}^{rw} = (1 - \hat{P}_k^{rw}(commit)) R_{abt}^{rw}.$$

The mean execution time for a run modeled by subgraph G_k is

$$R_{G_k}^{rw} = R_{begin} + \hat{R}_{k0}^{rw} + \sum_{i=1}^{w} (\hat{R}_{ki}^{rw} + \tilde{R}_{ki}^{rw}) + R_{k\,com}^{rw} + R_{k\,abt}^{rw}.$$

and the mean transaction response time for class C^{rw} is

$$R_{tx}^{rw} = \sum_{k=0}^{w} N_{G_k}^{rw} R_{G_k}^{rw}.$$

Concerning lock holding time equations in Section 4.4.1, the corresponding expressions for transactions of class C^{rw} are

$$T_{H,ki}^{rw} = \sum_{j=i}^{w} \hat{R}_{ki}^{rw} + \sum_{j=i+1}^{w} \tilde{R}_{ki}^{rw} + R_{k\,com}^{rw},$$

$$T_{H,i}^{rw} = \sum_{k=0}^{N_w} N_{G_k}^{rw} T_{H,ki}^{rw},$$

and

$$T_H^{rw} = \frac{1}{w} \sum_{i=1}^{w} T_{H,i}^{rw}.$$

Contention probability against transactions of class C^{rw} can be expressed using the average transaction arrival rates for the different classes, that is

$$P_{cont}^{rw} = \frac{\lambda^{rw} w T_H^{rw}}{D},$$

thus the average contention probability becomes

$$P_{cont} = \sum_{(r,w)\in X} P_{cont}^{rw}.$$

Version check failure probability for a transaction of class C^{rw} is therefore

$$P^{I,rw}_{A,ki} = 1 - \exp(-\frac{\lambda w_{avg}}{D}\vec{R}^{rw}_{ki}),$$

where

$$\vec{R}^{rw}_{ki} = \sum_{j=0}^{i-1}(R^{rw}_{k\hat{i}} + P^{rw}_{W,ki}R^{rw}_{k\,\widetilde{i+1}})$$

and

$$w_{avg} = \sum_{(r,w)\in X}\frac{\lambda^{rw}w}{|X|}.$$

The average lock waiting time becomes a weighted average across the waiting times caused by transactions of different classes. Hence

$$R_{k\tilde{i}} = \frac{1}{P_{cont}}\sum_{(r,w)\in X}P^{rw}_{cont}R^{rw}_{k\tilde{i}},$$

where $R^{rw}_{k\tilde{i}}$ is the residual execution time of transactions specialized for each single class. Finally, to evaluate the average number of accessed versions for read operations, since committed versions of a data item are generated with an average rate

$$\sigma = \sum_{(r,w)\in X}\frac{\lambda^{rw}w}{D},$$

we have for read operations by a transaction of class C^{rw} the following expression

$$N^{V,rw}_{read,ki} = \Delta T^{rw}_{s,ki}\,\sigma,$$

where

$$T^{rw}_{s,ki} = \vec{R}^{rw}_{ki} + R^{rw}_{k\hat{i}}/2.$$

Expressions for the hardware resource model in Section 4.4.1 still hold when considering per class parameters.

Non-uniform Data Access

We now consider non-uniform data access probability. For each data item $x \in D$ we denote as $P_D(x)$ the corresponding data access probability. For simplicity, we consider in this section fixed length transactions, even though, in a similar manner to what was done in the previous section, it is possible to consider several transaction classes characterized by different lengths.

Differently from the uniform access case, the contention probability depends on the accessed data item. Data item x is locked for an approximated average time fraction $\lambda P_D(x) N_w T_H$, which we denote with $P_{cont}(x)$. Thus contention probability is

$$P_{cont} = \sum_{x \in D} P_D(x) P_{cont}(x) = \sum_{x \in D} P_D^2(x) \lambda N_w T_H.$$

To evaluate version check failure probability we note that committed versions of data item x are born with an average rate $\sigma(x) = \lambda P_D(x) N_w / D$, thus

$$P^I_{A,ki} = \sum_{x \in D} P_D(x)(1 - \exp(-\frac{\sigma(x)}{D} \vec{R}_{ki})),$$

where $\vec{R}_{ki}$ is the same as in Section 4.4.1. Also, the average number of accessed data item versions depends on the data access distribution. Hence, similarly to what done in Section 4.4.1 we have

$$N^V_{read,ki}(x) = \Delta T_{s,ki}\, \sigma(x).$$

Therefore, the average number of accessed versions for a read operation in state $\hat{\imath}$ of subgraph G_k is

$$N^V_{read,ki} = \sum_{x \in D} P_D(x) N^V_{read,ki}(x).$$

4.5 Simulation Model

The simulation model we used for validating the analitical model has been implemented on a discrete-event simulation platform. It is inspired to a general architecture of a database system and is similar to models used in other simulation studies (e.g. [46, 71]).

The model simulates an open system. It contains the following simulation objects:

- Workload Generator (WG);
- Transaction Manager (TM);
- Concurrency Control Manager (CCM);
- Buffer Manager (BM).
- CPU;
- Disk.

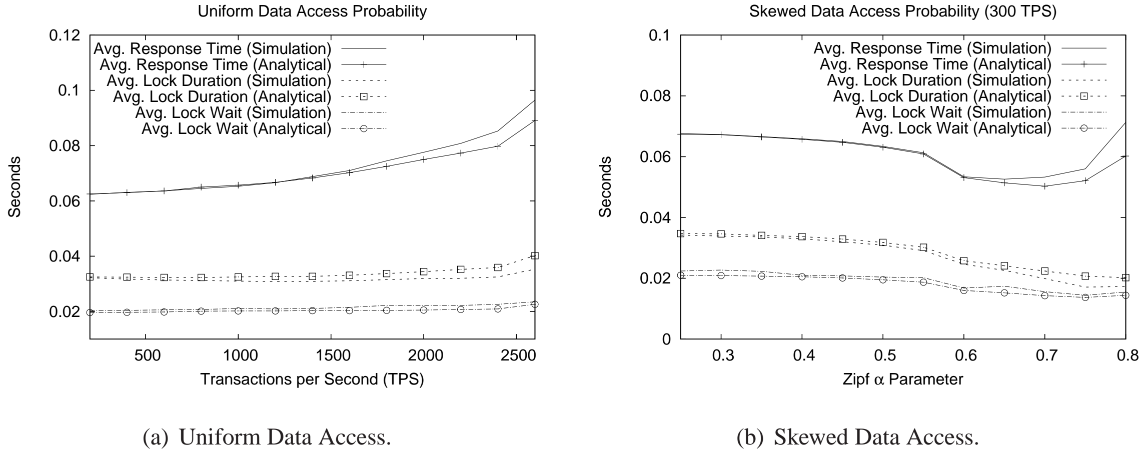
The WG is in charge of generating transactions according to an exponential distribution. In the case of multiple transaction classes, to generate a new transaction, the WG selects the transaction class on basis of the associated probability. For each operation of a transaction, it randomly selects the accessed data item and the type of operation according the read/write probability. The selections rely on a (pseudo) random number generator. The TM receives transaction execution requests from the WG and manages the execution of the transaction operations. The CCM acts according to the rules of the MVCC protocol as considered in this study. Furthermore, the CCM keeps a transaction wait-for-graph [5] for detecting transaction deadlocks. A deadlock is resolved by aborting the arriving transaction, which is resubmitted by the TM after a back-off phase. The BM is in charge of managing the buffer space. When an access to a data item is requested, if it is in the buffer space then the request is immediately served, otherwise the BM sends a request to the DISK. When the DISK receives the request for a data item, the load entails a fixed delay, whereupon the DISK notifies the BM of the completion of the request. The BM uses the Least Recently Used replacement policy. The CPU have k cores with a common queue. When a processing request is received it is enqueued, and after a number of instructions associate with the processing request are executed then the CPU notifies of the completion of the request .

A transaction is executed according the following model (to simplify here we assume that deadlock does not occur). The WG sends a *transaction execution* request to the TM, together with the sequence of operations to be executed by the transaction. The TM receives the request and sends a *begin processing* request to the CPU. When the request has been processed the CPU notifies the TM of the completion. Then the TM sends an *operation execution* request for the first operation of the transaction to the CCM. When the CCM allows to execute the operation, it sends the request to the BM, where the request is possibly blocked for waiting a page load from the disk. Hence an *operation execution* request is sent to the CPU. When the CPU notifies the BM of the completion of the request, the BM, in turn, notifies if the completion of the request to the TM. At this point the TM can execute the next operation. When all operations of the transaction have been completed, the TM sends a *commit processing* request to the CPU. When the TM is notified of the completion, it sends a *commit* request to the CCM, which releases all locks acquired by the transaction and notifies the completion to the TM.

4.6 Model Validation

In this section we present a simulation study aimed at evaluating the accuracy of the proposed analysis.

We consider a number of 10000 data items, and a buffer pool having size equal to the 20% of the data set of the database. Concerning the number of instructions



(a) Uniform Data Access.

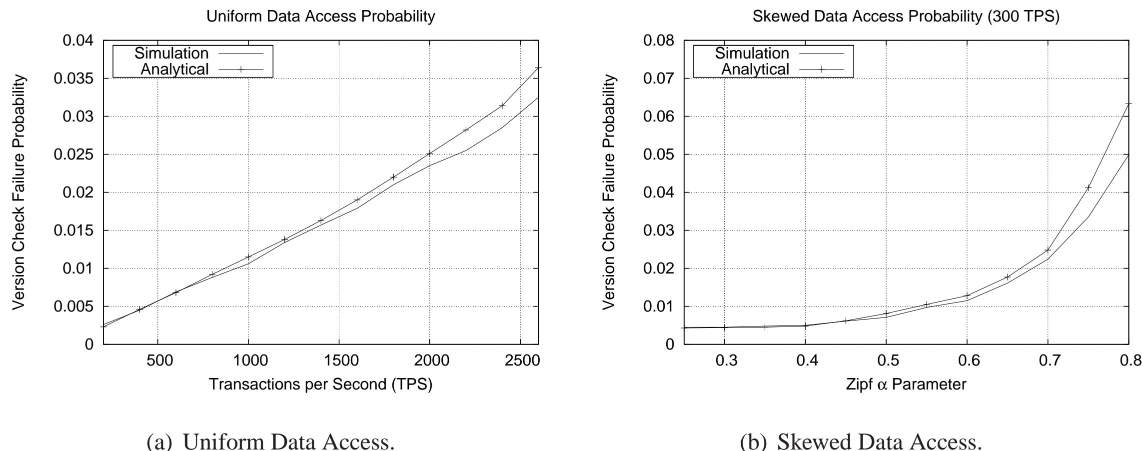
(b) Skewed Data Access.

Figure 4.3: Latency Results.

required for the different phases of the execution of a transaction (e.g. the begin phase and the data item read phase), we have used in both the simulator and the analytical model values compliant with those used in the studies presented in [72, 9]. However, our experiments are carried out considering more modern hardware. Specifically, for both simulation and analytical model, the database system is assumed to be hosted by a 8-CPU machine with processor speed equal to 1GHz.

In a first set of observations, we aimed to verify the accuracy of the basic analytical model, namely the one relying on the hypotheses of single transaction class and uniform data access. For this setting, we report in Figure 4.3(a) the transaction execution latency, the lock waiting time and the lock holding time (lock duration) vs the transaction arrival rate. Transactions of the unique class perform an expected amount of 20 data item accesses, with 20% of them being write operations. By the plotted results, we can see that the model provides a very good accuracy when comparing its latency prediction and the simulation outputs. Slight discrepancies between analytical and simulative data can be observed for transaction workload close to the system saturation point (i.e. on the order of 2500 transactions per second). To further observe the behavior of the model, we plot the probability of version check failure and the expected number of transaction (re)runs required for successful completion in Figure 4.4(a) and in Figure 4.5(a), respectively. By the plotted results we have again very good compliance between analytical and simulative values, unless for workload close to the saturation point.

In a second set of experiments, we have considered non-uniform data access, so to evaluate the accuracy of the model extension provided in Section 4.4.3. We have focused on a single transaction class, with data access pattern ruled by a Zipf distribution function with parameter α. For this setting, we have fixed the transaction

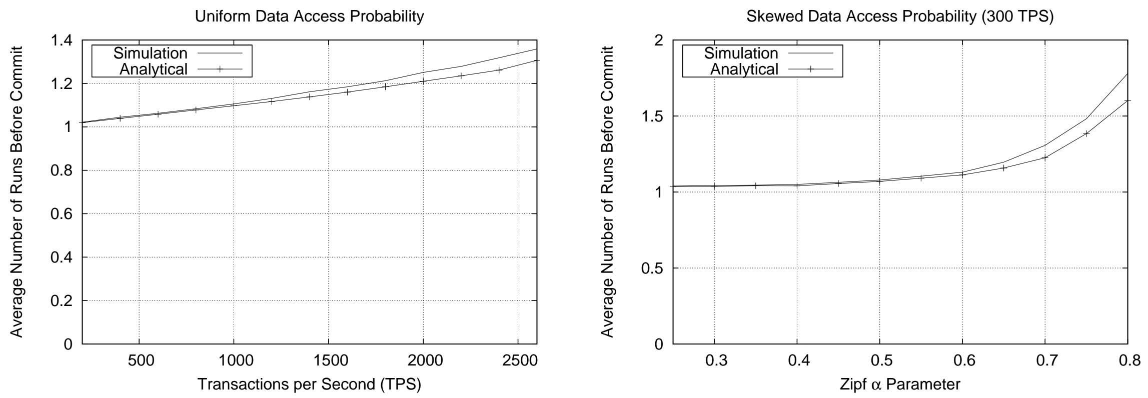


(a) Uniform Data Access. (b) Skewed Data Access.

Figure 4.4: Version Check Failure Probability.

workload (at 300 transactions per second - TPS) and we have varied the value of α in between 0.3 and 0.8 (as in the classical range observed for data access skew in Web contexts [73, 74]). The results for latencies, version check failure probability and expected number of (re)runs are reported, respectively, in Figures 4.3(b), 4.4(b) and 4.5(b). As for the transaction execution latency, compared to the uniform data access case, it shows a non-monotonic trend in the skewed data access case. This is due to the mixed effects of both increased buffer hit and increased contention as the parameter α of the Zipf distribution grows. The effects show different balances while α gets increased so monotonic behavior is not guaranteed. However, also in this case the analytical model provides results well matching the simulative data. An increased discrepancy (compared to the uniform data access case) is observed near the saturation point (which is reached for α values close to 0.8). This is mainly due to the fact that, as the skew increases, the probability for a transaction to abort because of an access to a highly popular data item correspondingly increases. The subsequent re-execution of such transactions leads, in its turn, to an overall increase of the skewness of the initially assumed data access distribution, namely $P_D(x)$. It is our intention to enable the model to capture this phenomenon in a future work.

Finally, we have considered uniform data access but differentiated transaction classes. This has been done to evaluate the accuracy of the extension of the analytical model provided in Section 4.4.3. For this setting, we have considered 8 different transaction classes, with different length in terms of requested data items, spanning from about 20 up to 40 accessed data items, and with different percentages of read vs write operations. In Figure 4.6, we report the expected execution latency for 4 of the considered classes, as evaluated via both the analytical model and the simulator. Again, we observe a very good compliance between analytical and simulative data.



(a) Uniform Data Access. (b) Skewed Data Access.

Figure 4.5: Average Number of Runs Before Commit.

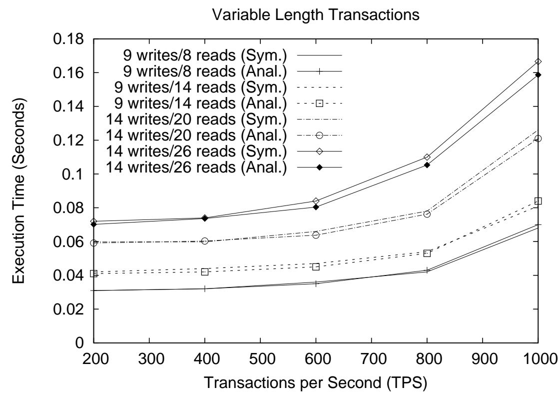


Figure 4.6: Latency Results for Variable Length Transactions (Uniform Data Access Case).

Chapter 5

Performance Modeling of Concurrency Control Protocol for Software Transactional Memories

5.1 Introduction

In Section 2.4 we presented some basic differences between transactions in DBS and in STMs. These differences have an impact on concurrency control. In fact, the CCPs commonly adopted in database environments are not likely to fit some typical requirements of STMs. For example, we discussed the opacity, which is considered an adequate isolation level for STMs, and is actually provided by many STMs prototypes. Typical optimistic protocols for DBMS do not prevent transactions doomed to abort from seeing inconsistent data item values, hence they are not able to guarantee opacity. Let us give another example. We said that STMs are characterized by fine grained volatile memory operations and the transaction execution time is typically two or three orders of magnitude smaller than in database environments. As a consequence, if database oriented locking protocols, where transactions are tipically forced to wait on lock conflicts, are blindly ported to STMs environments and are actuated on top of operating system supported mutex and/or semaphores, they would induce excessive overhead and non-negligible thread (re-)schedule delay. E.g., if a thread executing transaction is descheduled on lock conflict, the context-switching cost may be much larger then the transaction execution cost, hence the transaction may experience an unacceptable execution time. At the same time, being STMs optimized for

tightly-coupled multi-core/processor systems, if the thread is not descheduled on lock conflict, the system performance may be penalized because of lower CPU utilization.

The wide set of database oriented performance analysis results represent an important reference point for the performance modeling of STMs. However, according to the aforesaid observations, they can not be representative of a comprehensive performance study for the STMs.

So far, the field of STMs has been very little explored by the performance modeling community. In Chapter 3 we discussed the analytical performance models proposed in the literature. Some of these models suffer from some limitations, while others do not focus on the concurrency control. In the work presented in [62] a simple scenario is assumed, where transactions execute the commit operation serially by acquiring a unique global lock. Read and write operations, as well as interleaved operations, are not considered. The analytical framework proposed in [59] assumes a fixed number of active transactions in the system. Actually, in real STM-based applications, threads alternate the execution of transactions and non-transactional code. Furthermore, the framework abstracts over time by describing the execution of a transaction as a sequence of steps whose duration is left unspecified. For these reasons it is not aimed to evaluate the time-related performance metrics, such as the system throughput and the transaction response time. The analytical models presented in [63] and [64] are targeted to the prediction of time-related performance metrics, but they do not focus on the CCP. In fact these works do not consider a specific protocol. In the former work, the transaction conflict probability is assumed as input to the model. In the latter, the conflict probability is simply estimated on the basis of the overlapping sets of data items accessed by transactions.

In this chapter we focus on the concurrency control in STMs. We propose a new modeling approach well-suited for STMs applications. In particular, we provide a general modeling framework which overcomes the main lacks of the previous works we discussed above. We use a two-layered modeling approach. A thread-level model predicts the system performance as a function of the degree of concurrency within the system, independently of the specific CCP adopted by the system. The latter aspect is instead assigned to the transaction-level model, which can be specialized on the basis of a specific protocol. We provide an instantiation of the transaction-level model for the case of the Commit-Time Locking (CTL) protocol. We presented this protocol in Section 2.5, when we spoke about TL2 Software Transactional Memory (STM) [12]. In Section 5.3.3 we provide a more detailed description of the protocol and of the implementation mechanisms. The complete instantiation of the model allows to evaluate the main performance metrics, as the system throughput and the average transaction execution time, and additional specific metrics, as the abort probability for a transaction for each transaction execution phase. All the metrics can be evaluated with respect to various parameters, as the number of concurrent threads, the size of the data item set, and different workload configuration parameters. The model has been validated against simulation results obtained considering workloads configura-

tions inspired to the widely used STAMP benchmark [11].

The remainder of this chapter is structured as follows. The analytical modeling methodology, with the specific model instantiation for the CTL protocol, are provided in Section 5.3. In the same section, we describe an extension of the model to cope with multiple transaction classes, and, further, we provide some hints to extend the model to cope with non-uniform data access. In Section 5.4 we describe the simulation model. The evaluation study is presented in Section 5.5. Finally, in Section 5.6 we provide some hints on how to relax some modeling assumptions.

5.2 System Model and Considerations

We consider an STM application with a number of k active threads. Each thread executes on a distinct CPU-core. This choice is motivated by the fact that generally in STM applications, as concerns the parallelism, preventing the execution of a number of transactions larger than the number of available cores is an effectiveness approach to boost performance [36]. In fact, this can minimize the number of context switchings and reduce the frequency of conflicts, keeping the high CPU utilization [1].

Threads alternate the execution of transactions and non-transactional code blocks. A non-transactional code block is formed by a sequence of machine instructions which we denote as *ntcb*. While executing a non-transactional code block a thread performs only local computation, namely it does not access the shared data. Each transaction starts with a *begin* operation, then executes a number of transactional operations (namely, either *read* or *write* operations) on shared data items and finally ends by issuing a *commit* operation. Overall, after the *begin* operation and after each transactional operation the thread executes a code block, denoted as *tcb*. Also in this code block the thread does not access shared data. The thread model a the transaction model are depicted in Figure 5.1.

We denote the expected time required by a thread to execute the *begin*, *read*, *write* and *commit* operations with t_{begin}, t_{read}, t_{write} and t_{commit}, respectively. Furthermore, we denote the expected duration of *tcb* and *ntcb* as t_{tcb} and t_{ntcb}, respectively. Whenever a transaction is aborted, an *abort* operation is executed, whose handling has an expected duration t_{abort}. After experiencing an abort, a transaction is temporarily held in a back-off state for a time interval whose average value is denoted as $t_{backoff}$, at the end of which a new run of the transaction starts. We assume that *ntcb* and the back-off interval have an exponentially distributed duration. Possible extensions of the model to cope with cases where t_{ntcb} and $t_{backoff}$ represent the mean of generic distributions will be discussed in Section 5.6.

[1] We remark that in STMs, when a conflict occurs, tipically it is preferable that the transaction restarts, instead of descheduling the executing thread to free up the CPU-core.

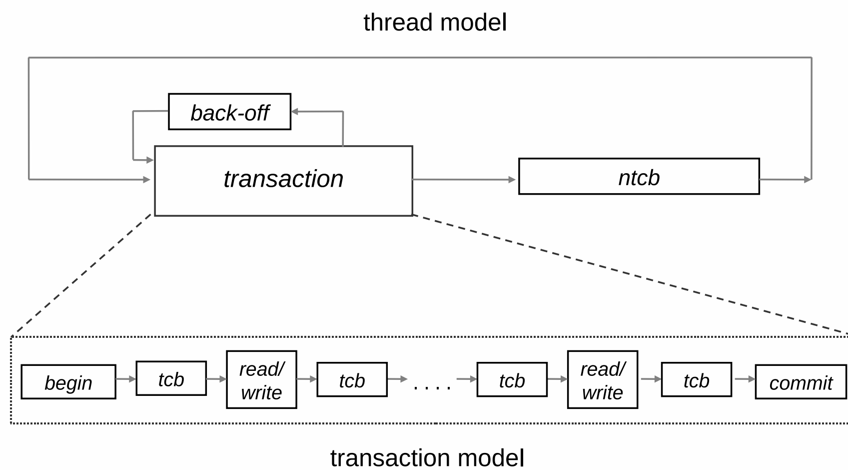


Figure 5.1: Thread model and transaction model.

All the durations defined above are assumed as input to the model. Note that, all of them, except $t_{backoff}$, are affected by both the speed of the underlying hardware platform and the internals of the underlying STM system. The choice of capturing the above times through ad-hoc input parameters enhances the flexibility of our model for two main reasons. (i) It allows the model to be employed for what-if analysis aimed at forecasting the performance for diverse scenarios and/or workloads. As an example, the model can be used to assess the performance of STM-based applications when deployed on different hardware platforms (which might give rise to different machine instruction patterns) or when changing the internals of the underlying STM system. (ii) It allows the model to be easily coupled with an hardware resource model by resolving the final model through iteration (in the same fashion that we did in the models presented in chapters 3 and 4). Due to the latter reason, in this work we do not care of the hardware resource model.

5.3 The Analytical Model

5.3.1 Modeling Approach Overview

As discussed above, we logically structure our model in two distinct parts, each one capturing complementary aspects of the execution dynamics of STM-based applications. The first part of the model, which we name thread-level model, is presented in Section 5.3.2. It allows to determine how the various threads in the system alternate among the following three phases (see figure 5.1) :

(i) execution of a non-transactional code block,

(ii) execution of an STM transaction,

(iii) blocked in back-off.

By allowing the determination of the probability distribution of the number of threads in each of these three phases, this layer of the model can be used to output standard performance metrics such as throughput and the average transaction execution time. This part of the model is de-facto oblivious of the specific algorithm used by the STM to regulate concurrency, over which it abstracts via two key input parameters: (a) the mean run execution time of a transaction (independently of its final outcome) and (b) the commit probability for a run of a transaction, given a number $i \in [1, k]$ of threads concurrently executing transactions. Instead, these parameters are computed by what we refer to as transaction-level model. The latter modeling component is focused on capturing proper dynamics associated with the specific conflict detection and resolution schemes adopted by the STMs, assuming a constant, albeit parametric, number of threads simultaneously executing transactions.

By decoupling the modeling of the dynamics associated with thread alternation among the various phases from the modeling of the concurrency control algorithm, our two-layered modeling methodology provides the below reported benefits:

1. It simplifies the modeling stage of the concurrency control algorithm, delegated to the transaction-level model. In fact with this approach, as we will show, the transaction-level model does not require to explicitly consider dynamic variations of the number of threads concurrently executing transactions, but it only requires to provide performance predictions under the assumption that exactly i threads are concurrently executing transactions. Then, it will be the responsibility of the thread-level model to exploit the independent performance forecasts associated with different values of i in order to generate the final performance predictions.

2. It allows seamless replacement of the model of the CCP with alternative ones, either targeting different protocols and/or relying on different modeling approaches.

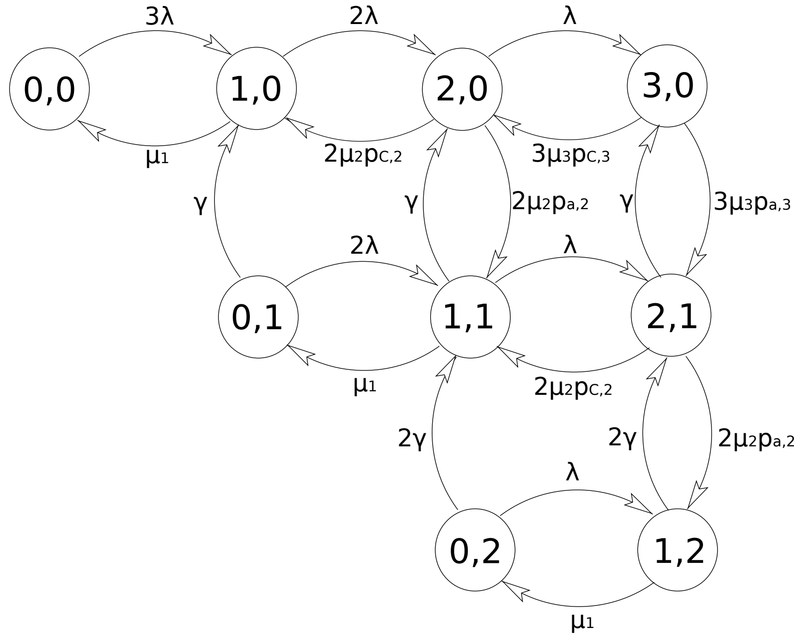


Figure 5.2: State transition diagram of the CTMC for $k = 3$.

5.3.2 Thread-level Model

We model the execution of all threads via a Continuous Time Markov Chain (CTMC) [69]. Each state of the CTMC is marked and identified by a couple of integers (i, j) representing, respectively, the number of threads running transactions and the number of threads in back-off. Since the total number of threads in the system is equal to k, the only admissible states in the CTMC are those for which the corresponding (i, j) pair respects the constraint $i + j \leq k$. Note that in each state $k - i - j$ threads are executing non-transactional code block.

We denote with $\lambda = \frac{1}{t_{ntcb}}$ the rate according to which a thread executes a non-transactional code block (in between two transactions). Furthermore we denote with μ_i and $p_{c,i}$, respectively, the execution rate of the runs of transactions (independently of whether a run gets aborted or committed) and the probability for a run of a transaction to successfully commit, assuming that there are i threads concurrently running transactions (namely when the system is in the state (i, j), with $0 \leq j \leq k$). Note that give a state (i, j), μ_i and $p_{c,i}$ depend only on the running transactions in the state. For each i, the thread-level model takes μ_i and $p_{c,i}$ as input parameters from the transaction-level model.

We can now list the rules defining the transition rates of the CTMC:

- for $i + j < k$, the transition rate from state (i, j) to state $(i + 1, j)$, associated

with the run of a new transaction after a thread completed a non-transactional code block, is equal to $\lambda \cdot (k - i - j)$;

- for $i > 0$, the transition rate from state (i, j) to state $(i - 1, j)$, associated with the commit event of a run of a transaction and the subsequent activation of a non-transactional code block, is equal to $i \cdot \mu_i \cdot p_{c,i}$;

- for $i > 0$, the transition rate from state (i, j) to state $(i-1, j+1)$, associated with the abort event of a run of a transaction and the start of the back-off period, is equal to $i \cdot \mu_i \cdot p_{a,i}$;

- for $j > 0$, the transition rate from state (i, j) to state $(i+1, j-1)$, associated with the termination of back-off periods and a new run of a transaction, is equal to $j \cdot \gamma$, where $\gamma = \frac{1}{t_{backoff}}$.

We exclude state $(0, k)$ as a possible one since, (i) the CTMC characterizing our model does not express state transitions where multiple transactions get simultaneously aborted due to (mutual) conflicts, and (ii) adopting whichever literature STM concurrency control algorithm, if a single thread is currently executing a transaction then it cannot be aborted. An example of the CTMC for the case of three threads (namely $k = 3$) is depicted in Figure 5.2.

As typically expected in any real system, assuming for any state where $i \in [1, k]$ that $\mu_i > 0$, $p_{c,i} \neq 0$ and $p_{c,i} \neq 1$ (the cases of $p_{c,i} = 0$ or $p_{c,i} = 1$ express, respectively, a pathological scenario with no possibility of transaction progress and a trivial scenario entailing no data contention), the CTMC is irreducible [61]. Thus the steady-state probability vector $\mathbf{v}$ can be computed by using following equations:

$$\mathbf{v} \cdot Q = \mathbf{0} \tag{5.1}$$

$$\sum_{(i,j) \in S} v_{(i,j)} = 1 \tag{5.2}$$

where Q is the infinitesimal generator matrix of the CTMC and S is the set of states of the CTMC. Assuming that we are provided with μ_i and $p_{c,i}$ values ($\forall i \in [1, k]$), we can evaluate the system throughput τ as the sum of the transaction commit rates in the different states, weighted according to the probability for the system to be in each state (i, j)

$$\tau = \sum_{(i,j) \in S'} v_{(i,j)} \cdot i \cdot \mu_i \cdot p_{c,i} \tag{5.3}$$

(where S' is the subset of S containing any state where $i > 0$). The overall transaction commit and abort probabilities, denoted as p_c and p_a, can be accordingly evaluated, using the expressions

$$p_c = \frac{\sum_{(i,j)\in S'} v_{(i,j)} \cdot p_{c,i}}{\sum_{(i,j)\in S'} v_{(i,j)}} \tag{5.4}$$

and

$$p_a = (1 - p_c) \tag{5.5}$$

5.3.3 Transaction-level model: The Commit-Time Locking Case

In this section we introduce an analytical model of CTL protocol, focusing on the version implemented within the TL2 STM layer [12]. This version is considered as one of the best performing concurrency control algorithms for typical STM workloads. We start by overviewing such a target version of the CTL protocol, and then we move to the presentation of its analytical model.

Algorithm Overview

CTL protocol acquires locks at commit-time, and locks only involves written data items. This choice enhances concurrency with respect to conventional lock-based schemes by, e.g., avoiding to block transactions issuing a write operation on a data item that has already been read/written by a concurrent transaction. Given the absence of read-locks, consistency is ensured via a validation mechanism used to notify transaction T, which speculatively read a data item x, about the fact that x was overwritten by some concurrent transaction T' preceding T in the commit order. To this end, a versioning scheme is employed which associates a timestamp value with each data item, referred to as Write-Version-Clock (WVC). The generation of WVC values relies on a unique Global-Version-Clock (GVC), which is read by any transaction upon startup, and is atomically increased upon transaction commit. The updated value is used as the new WVC value for all the data items written by the committing transaction. Manipulation of the GVC typically relies on a Compare-and-Swap (CaS) operation directly exploiting atomic sequences of machine instructions (e.g., via the LOCK prefix in IA-32 compliant processors). In other words, each transaction updates the GVC as an acyclic, one shot operation, which does not require software spin-locking for accessing the corresponding critical section. Hence, any delay in the access to the GVC is only related to the underlying firmware protocol used to support the atomicity of the machine instruction pattern implementing the CaS. When validating a transaction against a read data item x, two actions are performed:

1) it is checked whether there is a write-lock being held on x (which implies that another transaction has written x and is currently within its commit phase);

2) it is checked whether the current timestamp associated with x is greater than the timestamp read by the transaction upon starting up (which indicates that some concurrent transaction has overwritten x and has already been committed).

If one of the previous checks fails, the transaction gets aborted. This validation scheme is used upon read operations and, as we shall discuss below, also at commit time. Accordingly, the *opacity* property [29] is guaranteed, which ensures that the snapshot observed by any transaction (including transactions that are eventually aborted), is equivalent to the one that would have been observed according to some serial execution history. As discussed in [29], this property is crucial since for several categories of STM-based applications, transactions observing an inconsistent snapshot may be trapped within infinite loops, or may even cause the application program to crash (e.g., due to an invalid memory reference). As far as write operations are concerned, in CTL they are buffered within a private workspace until the commit phase. When a transaction attempts to commit, it first acquires the write-locks for all the data items within its write-set. If any of these lock acquisitions fails (due to lock holding by some other transaction), the transaction is aborted. Otherwise, the transaction increments the GVC and tries to validate all the data items it has within its read-set (according to the aforementioned validation procedure). If the validation fails for at least one item within the read set, the transaction gets aborted. If no abort occurs, the data within the write-set are copied back to their original memory locations, updating their WVCs with the value of the GVC. All the acquired locks are released at the end of the commit phase, or upon the abort. By the above description, we have that a read operation on a data item that was previously written by the transaction gives rise to an access to the transaction private workspace. Thus it is not subject to the previously depicted read validation mechanism. In other words, the validation mechanism is used for read operations associated with any data item that has not already been accessed in write mode by the transaction.

Analytical Model

In order to simplify the discussion, we present the analytical model in an incremental fashion. We start by presenting the model considering that:

- all the transactions encompass the same amount n of operations;
- the accesses (both in read or write mode) to the shared data items are uniformly distributed.

Model extensions to cope with multiple transaction profiles and non-uniform accesses will then be discussed in Section 5.3.4 and Section 5.3.5.

In our analytical model, we rely on the following assumptions:

- the sequence of read operations issued on shared data items form a Poisson process;
- the arrival of transaction commit events form a Poisson process.

A discussion on how to relax the above assumptions will then be provided in Sections 5.6

As already discussed, the transaction-level model computes the rate of the runs of transactions $\mu_i = 1/r_{t,i}$ (where $r_{t,i}$ is the average run execution time) and the transaction commit probability $p_{c,i}$ under the assumption that there are i threads simultaneously processing transactions, with $1 \leq i \leq k$. We analyze the case $i = 1$ and $i \neq i$ separately.

If $i = 1$, a single thread is currently executing transactional code, thus no data conflict can arise. This also means that the currently executed transaction can not be aborted and it follows that $p_{c,1} = 1$. Therefore, for the average transaction execution time we have that

$$r_{t,1} = t_{begin} + n \cdot t_{op} + (n+1)t_{tcb} + t_{commit} \tag{5.6}$$

where t_{op}, namely the average time to execute an access operation on a shared data item, is equal to

$$t_{op} = t_{read}(1 - p_{write}) + t_{write} \cdot p_{write} \tag{5.7}$$

where we denote with n the number of transactional operations on shared data items within a transaction, with p_{write} the probability that the access is in write mode, and with $(1 - P_{write})$ the probability that the access is in read mode.

As already discussed in Section 5.3.3, if the transaction accesses a data item x in write mode, producing a new version, any subsequent read on x by the same transaction will return the previously written version, retrieving it from the transaction private workspace. Analogous considerations apply for subsequent writes over the same data item x, which will simply update the copy of x buffered within the private workspace. Hence read/write operations issued on previously updated data items are simply not taken into account by the parameter n. On the other hand the cost of the corresponding accesses within the private workspace is encapsulated in t_{tcb}.

By the previous notation, we have that

$$n_{write} = n \cdot p_{write} \tag{5.8}$$

is the average number of shared data items accessed by the transaction in write mode, and

$$n_{read} = n \cdot (1 - p_{write}) \tag{5.9}$$

is the average number of read operations occurring on distinct shared data items that were not already accessed by the transaction in write mode.

For $i \neq 1$ we proceed as follows. Once fixed i, we use a procedure that iteratively recalculates the values of $p_{c,i}$ and $r_{t,i}$. Upon starting the iterative procedure, the initial values can be selected as $p_{c,i} = p_{c,i-1}$ and $r_{t,i} = r_{t,i-1}$ for commodity. The output values by an iteration step are used as the input values for the next step. We conclude the iterative procedure as soon as the corresponding input and output values for $p_{c,i}$ and $r_{c,i}$ differ by at most an ϵ. In all the configurations that we have experimented, using ϵ=1%, the procedure has always converged in at most fifteen iterations.

In each iteration step the following set of parameters, captured by our model, are re-evaluated:

- $p^{o}_{a,l}$, namely the probability for a transaction to abort while executing its lth operation due to validation fail (recall that a transaction can abort while executing an operation only if the operation is a read);

- p_{alc}, namely the probability for a transaction to abort at commit time due to lock contention experienced in the commit-time lock acquisition phase;

- p_{avf}, namely the probability for a transaction to abort at commit time due to validation failure of its read-set.

In order to model these parameters, we consider that the expected system state *seen* by any of the i active transactions is determined by the activities associated with the other $i-1$ transactions currently within the system. Thus we use the following approach.

When a transaction successfully commits, an average number n_{write} of write-locks are first acquired, and then released after read-set validation and write-back phases. Actually, the duration of the lock acquisition and release phases are typically negligible with respect to the duration of validation and write-back phases (recall that, during lock acquisition, transactions do never block, even if they experience contention). Hence, for simplicity, we assume lock acquisition and release to be instantaneous and to occur, respectively, at the beginning and at the end of the commit phase. Also, if a transaction is aborted, no real rollback operation is required for undoing the effects of the corresponding write operations since transaction write-sets are reflected to memory only in case of succesful commit attempts. Thus, to simplify, we ignore the cost of aborts when we evaluate the average lock holding time, by assuming that if a transaction successfully completes the lock acquisition phase, it holds the locks for an average time equal to t_{commit}.

Let us now compute the probability for a transaction to abort while executing a read operation on a shared data item x, given that it finds the corresponding write-lock currently busy. For this case to be possible, there must exist another transaction that has written x, that is currently in its commit phase and that has successfully acquired the write-locks for all the data items in its write-set. Given that we are assuming uniformly distributed accesses to distinct data items within a transaction, it follows

that the probability for a committing transaction to have a specific data item within its write-set is n_{write}/d. Exploiting the aforementioned assumption of Poissonianity of the arrival process of read operations, we can rely on the PASTA property (Poisson Arrivals See Time Averages) [75] to compute the probability to incur in a raised write-lock during a read operation as

$$p_{lock} = l_r \cdot t_{commit} \cdot \frac{n_{write}}{d} \quad (5.10)$$

where l_r is the rate according to which the remaining $i-1$ transactions in the system successfully execute the write-lock acquisition phase. This rate can be evaluated as follows

$$l_r = \frac{1}{r_{t,i}} \cdot (p_{c,i} + p_{avf}) \cdot (i-1) \quad (5.11)$$

where p_{avf} is the probability for a transaction to abort during the read-set validation phase. Such a transaction contributes anyway to the lock-acquisition rate since read set validation occurs after write-locks are acquired at commit time over any written data item. We will evaluate p_{avf} later in this subsection.

Now we determine the probability $p^o_{a,l}$ for a transaction T to abort while executing the l-th operation. The rate u_r at which a data item is updated by transactions is equal to

$$u_r = c_r \cdot \frac{n_{write}}{d} \quad (5.12)$$

where c_r expresses the rate at which the other $i-1$ transactions successfully commit, and can be evaluated as

$$c_r = \frac{1}{r_{t,i}} \cdot p_{c,i} \cdot (i-1) \quad (5.13)$$

Upon the l-th operation by transaction T, the average time $t_{b,l}$ elapsed since T started its execution can be expressed as $t_{begin} + t_{tcb} \cdot l + t_{op} \cdot (l-1)$. As we are assuming that the arrival of transactions to the commit phase forms a Poisson process, the probability $p^o_{u,l}$ for a read (executed as the l-th operation of T) to access a shared data item that has been updated by some successfully committing transaction after T started can be expressed as

$$p^o_{u,l} = 1 - e^{-u_r \cdot t_{b,l}} \quad (5.14)$$

In the above expression, in order to avoid overcomplicate the model, we decided not to capture the case of repeated transactional read operations on the same data item. In this case, in fact, the invalidation window for a data item x would no longer correspond to the time elapsed since the beginning of the transaction (namely $t_{b,l}$), but would be equal to the (average) time elapsed since the last occurrence of a read on x. Clearly, the error introduced by this modeling choice depends on the actual frequency of occurrence of repeated read operations on the same data item during the same transaction. On the other hand, the model captures faithfully the effects of a

frequent optimization technique (possibly implemented at the compiler level), which allows sparing subsequent read operations issued within the same transaction on the same data item from the cost of validation. To this end, it is sufficient to copy the values read from the shared transactional memory to thread local variables, and to redirect subsequent reads on these data items (within the same transaction) towards the thread local variables. Note that, since with this optimization subsequent read operations on a data item do not target the shared transactional memory, they do not even need to be accounted for while computing the value of the parameter n.

We can now evaluate the probability for a transaction to abort during the execution of its first operation (i.e., when l=1), namely $p^o_{a,1}$ as

$$p^o_{a,1} = (1 - p_{write}) \cdot (p_{lock} + (1 - p_{lock}) \cdot p^o_{u,1}) \tag{5.15}$$

Since the abort of a transaction T during its l-th operation (where $2 \leq l \leq n$) implies that T did not abort during its previous $l - 1$ operations, it follows that

$$p^o_{a,l} = p^o_{na,l} \cdot (1 - p_{write}) \cdot (p_{lock} + (1 - p_{lock}) \cdot p^o_{u,l}) \tag{5.16}$$

where $p^o_{na,l}$ is the probability of not aborting until the completion of the $(l - 1)^{th}$ operation. For this last probability we have

$$p^o_{na,1} = 1 \tag{5.17}$$

and

$$p^o_{na,l} = (1 - p^o_{a,l-1}) \cdot p^o_{na,l-1} \tag{5.18}$$

In equations (5.15)-(5.16) we have assumed that the event of finding a write-lock raised on the shared data item by a concurrent transaction currently attempting to commit, and the event that the same data item was previously updated by a different (already committed) concurrent transaction are independent. Overall, independence is related to that these events belong to commit time activities across distinct transactions.

The probability p_{alc} for a transaction T to abort at commit time due to lock contention while acquiring the write-locks can be evaluated as follow. T can experience contention while requesting the lock on a data item x only if, at the time in which T starts its commit phase, some other transaction that has written x has successfully completed its lock acquisition phase, and is still executing the commit procedure. Considering that T aborts only if *at least one* of the data items in its write-set is locked, then, as in [9], we approximate this last probability, namely p_{wlc}, with an upper bound value, that is

$$p_{wlc} = 1 - (1 - p_{lock})^{n_{write}} \tag{5.19}$$

Thus we have

$$p_{alc} = p^o_{na,n+1} \cdot p_{wlc} \tag{5.20}$$

where we recall that $p^{o}_{na,n+1}$ is the probability for a transaction not to be aborted until the completion of its nth operation, that is until it enters its commit phase. Consequently, the probability p^{la}_{na} for a transaction not to be aborted during its execution and to succeed in its commit-time lock acquisition phase is equal to

$$p^{la}_{na} = p^{o}_{na,n+1} \cdot (1 - p_{wlc}) \tag{5.21}$$

Let us now show how we can evaluate p_{avf}, namely the probability for a transaction T to abort at commit time due to validation failure for its read-set. The validation fails if at least one data item x belonging to the read-set of T has the corresponding write-lock raised by another transaction, or if a new version of x has been committed after the validation executed by T during its last read operation on x. We denote with $p^{r}_{u,l}$ the probability that the shared data item accessed in read mode at the lth operation by T has been updated after the last validation (occurred upon the corresponding last read operation on x). We calculate this probability as follows

$$p^{r}_{u,l} = 1 - e^{-u_r \cdot t_{v,l}} \tag{5.22}$$

where $t_{v,l}$ is the elapsed time since the original validation, that is

$$t_{v,l} = (t_{tcb} + t_{op}) \cdot (n - l + 1) + t_{commit} \tag{5.23}$$

Analogously to what we did in equation (5.16), we evaluate the abort probability due to failure in the validation of the data item associated with the lth transactional access of T as follows

$$p^{r}_{a,l} = p^{r}_{na,l} \cdot (1 - p_{write}) \cdot (p_{lock} + (1 - p_{lock}) \cdot p^{r}_{u,l}) \tag{5.24}$$

where $p^{r}_{na,1} = 1$ and, for $l > 1$, $p^{r}_{na,l} = (1 - p^{r}_{a,l-1}) \cdot p^{r}_{na,l-1}$. Then, we can express p_{avf} as

$$p_{avf} = p^{la}_{na} \cdot p_{rvf} \tag{5.25}$$

where

$$p_{rvf} = \sum_{l=1}^{n} p^{r}_{a,l} \tag{5.26}$$

Finally, successful commit probability for the case of i active threads can be evaluated as

$$p_{c,i} = p^{la}_{na}(1 - p_{rvf}) \tag{5.27}$$

The average execution time of a transaction $r_{t,i}$ can now be computed as the sum of the average time for a transaction to reach a different execution phase, weighted with the probability for the transaction to abort exactly during that phase. Let us denote with

- $t_{a,l}$ the average duration of a transaction that aborts during its l-th operation, that is:

$$t_{a,l} = t_{begin} + l \cdot (t_{tcb} + t_{op}) + t_{abort} \tag{5.28}$$

- $t_1 = t_b + t_{tcb} + t_{abort}$ the average duration of a transaction that aborts during its commit phase due to contention while acquiring locks for the data items in its write-set, where

$$t_b = t_{begin} + n \cdot (t_{tcb} + t_{op}) \tag{5.29}$$

- $t_2 = t_b + t_{tcb} + t_{commit} + t_{abort}$ the average duration of a transaction that aborts during its commit phase due to failure in validating its read-set;

- $t_3 = t_b + t_{tcb} + t_{commit}$ the average duration of a transaction that successfully commits.

Overall, the average transaction execution time can be expressed as

$$r_{t,i} = \sum_{l=1}^{n} (p^o_{a,l} \cdot t_{a,l}) + p_{alc} \cdot t_1 + p_{avf} \cdot t_2 + p_c \cdot t_3 \tag{5.30}$$

Now let us evaluate the time t_{GVC} spent by any committing transaction while updating the GVC. We consider this time logically included in t_{commit}, thus t_{commit} is the sum of two terms, namely t'_{commit} and t_{GVC}, where t'_{commit} is the time to execute all the other operations, distinct from GVC manipulation, during the commit phase. As explained in Section 5.3.3, the atomic operations required for the update of GVC typically rely on firmware level protocols offered by modern SMP and/or multi-core machines. Assuming fairness by these protocols vs different CPU/core requests, we model the delay for performing an atomic increment of the GVC, denoted as t_{GVC}, by means of an M/D/1 queue [69] with service rate $\mu = \frac{1}{t^{inc}_{GVC}}$ (where t^{inc}_{GVC} expresses latency for the updating machine instructions, once the firmware has granted access to the corresponding critical section) and arrival rate $\beta = l_r$ (note that the increment of the GVC is performed by any transaction that successfully acquired all the requested locks). According to this modeling approach, t_{GVC} corresponds to the residence time within the M/D/1 queue, namely

$$t_{GVC} = (1 + \frac{\rho}{2 \cdot (1 - \rho)}) \cdot t^{inc}_{GVC}, \tag{5.31}$$

where $\rho = \frac{\beta}{\mu}$.

5.3.4 Coping with Multiple Transaction Classes

In this section we extend the analytical model by considering the case of q different transactional classes, associated with different transaction profiles. The number of

operations executed by a transaction of class m, with $m \in [1, q]$, is denoted by n^m, and each operation is a write operation with probability p^m_{write}. Hence $n^m \cdot (1 - p^m_{write})$ expresses the total amount of distinct transactional read accesses. A thread executes a transaction of class m with probability P^m. Also, t^m_{commit} and t^m_{abort} are the expected commit time and abort time for a transaction of class m, respectively.

Multi-class Thread-level Model

For q transactional classes, the state of the CTMC can be identified by $2q$ integers $(i_1, .., i_q, j_1, .., j_q)$ where i_m and j_m (with $m \in [1, q]$) represent the number of threads running transactions of class m and the number of threads in backoff due to an abort of a transaction of class m, respectively. Note that $i_1 + .. + i_q + j_1 + .. + j_q \le k$ for each state of the CTMC.

For any state $(i_1, .., i_q, j_1, .., j_q)$, the average transaction execution rate and the transaction commit probability for a transaction of class m depend on the mix of active transactions in that state. Thus we denote them as $\mu^m_{i_1,..,i_q}$ and $p^m_{c,i_1,..,i_q}$, respectively. Also, the abort probability for a transaction of class m while residing in state $(i_1, .., i_q, j_1, .., j_q)$ is denoted as $p_{a,i_1,..,i_q} = 1 - p_{c,i_1,..,i_q}$.

The rate according to which a thread executes a new transaction of class m is $\lambda_m = P^m/t_{ntcb}$. The rules defining the transition rates from any two states of the CTMC are the following:

- for $i_1 + ... + i_q + j_1 + ... + j_q < k$, the transition rate from state $(i_1, ..., i_m, ..., i_q, j_1, ..., j_q)$ to state $(i_1, ..., i_m + 1, ..., i_q, j_1, ..., j_q)$, associated with the activation of a run of a transaction of class m is equal to $\lambda_m(k - i_1 - ... - i_q - j_1 - ... - j_q)$;
- for $i_m > 0$, the transition rate from state $(i_1, ..., i_m, ..., i_q, j_1, ..., j_q)$ to state $(i_1, ..., i_m - 1, ..., i_q, j_1, .., j_q)$, associated with a successful commit event of a transaction of class m is equal to $i_m \mu^m_{i_1,...,i_q} p^m_{c,i_1,...,i_q}$
- for $i_1 + ... + i_m + ... + i_q \ge 2$ and $i_m \ge 1$, the transition rate from state $(i_1, ..., i_m, ..., i_q, j_1, ..., j_m, .., j_q)$ to state $(i_1, ..., i_m - 1, .., i_q, j_1, .., i_m + 1, ..., j_q)$, associated with an abort event of a transaction of class m is equal to $i_m \mu^m_{i_1,...,i_q} p^m_{a,i_1,...,i_q}$
- for $j_m > 1$, the transition rate from state $(i_1, ..., i_m, ..., i_q, j_1, ..., j_m, ..., j_q)$ to state $(i_1, ..., i_m + 1, ..., i_q, j_1, ..., j_m - 1, ..., j_q)$, associated with the termination of a back-off period of an aborted transaction of class m is equal to $\gamma \cdot j_m$.

We can evaluate the stead-state probability vector v for the CTMC as we made in Section 5.3.2 for the case of single transaction class. Hence, the execution rate τ_m of transactions of class m can be expressed as

$$\tau_m = \sum_{(s') \in S'} v_{s'} \cdot i_m \cdot \mu^m_{s''} \cdot p^m_{c,s''} \tag{5.32}$$

where we used s' in place of $i_1, .., i_q, j_1, .., j_q$ and s'' in place of $i_1, .., i_q$, and where S' is the subset of S containing any state where $i_m > 0$. The overall system throughput is

$$\tau = \sum_{m=1}^{q} \tau_m \tag{5.33}$$

The commit probability for a transaction of class m is

$$p_c^m = \frac{\sum_{(s')\in S'} v_{s'} \cdot p_{c,s}^m}{\sum_{(s')\in S'} v_{s'}} \tag{5.34}$$

Multi-class Tread-level Model for CTL

Fixed a configuration of active transactions $i_1, .., i_q$, the thread-level model is in charge of evaluating for each transactional class m the rate of the runs of transactions $r^m_{t,i_1,..,i_q}$ and the transaction commit probability $p^m_{c,i_1,..,i_q}$. As for the single-class models, if there is just one active transaction, that is $i_m = 1$ and $i_w = 0$ for each $w \neq m$, the average transaction execution time of the transaction of class m is

$$r^m_{t,i_1,..,i_q} = t_{begin} + n^m \cdot t_{op} + (n^m + 1)t_{tcb} + t^m_{commit} \tag{5.35}$$

where t^m_{op}, namely the average time to execute an access operation on a shared data item for a transaction of class m, is equal to

$$t^m_{op} = t_{read}(1 - p^m_{write}) + t_{write} \cdot p^m_{write} \tag{5.36}$$

When the number of active transactions is greater that one we use the same iterative approach as in Section 5.3.3, by stopping the iterations when two consecutive values of the commit probability for transactions of each class m (if $i_m \geq 1$) differ by at most an ϵ. Also, in what follows we use the same assumptions and considerations as in Section 5.3.3.

When a transaction of class m is active, its concurrent transactions are:

- i_x active transactions of each other class x such that $x \neq m$ and $i_x \geq 1$;
- $i_m - 1$ active transactions of the same class m, if $i_m \geq 2$.

At the start of each iterative step we evaluate the following parameters. The lock rate associated with transactions of each class x, expressed as

$$l^x_r = \frac{1}{r^x_{t,i_1,..,i_q}} \cdot (p^x_{c,i_1,..,i_q} + p^x_{avf}) \tag{5.37}$$

where p^x_{avf} is the probability for a transaction of class x to abort during the read-set validation phase. The probability for a transaction of class m to find a write-lock

raised while issuing a read operation, which is expressed as

$$p_{lock}^{m} = \sum_{x=1,x\neq m}^{q} l_r^x \cdot i_x \cdot t_{commit} \frac{n^x \cdot p_{write}^x}{d} + l_r^m \cdot (i_m - 1) \cdot t_{commit} \frac{n^m \cdot p_{write}^m}{d}. \quad (5.38)$$

The commit rate associated with transactions of class x, which is expressed as

$$c_r^x = \frac{1}{r_{t,i_1,..,i_q}^x} \cdot p_{c,i_1,..,i_q}^x \quad (5.39)$$

Finally, the update rate by concurrent transactions of a transaction of class m, which is expressed as

$$u_r^m = \sum_{x=1,x\neq m}^{q} c_r^x \cdot i_x \frac{n^x \cdot p_{write}^x}{d} + c_r^m \cdot (i_m - 1) \frac{n^m \cdot p_{write}^m}{d}. \quad (5.40)$$

After solving the previous equations, we evaluate in each iterative step all the parameters we list below. The probability $p_{u,l}^{o,m}$ for a read operation, executed as the l-th operation of a transaction T of class m, to access a data item that has been updated by some successfully committing transaction after T started, which can be expressed as

$$p_{u,l}^{o,m} = 1 - e^{-u_r^m \cdot t_{b,l}^m} \quad (5.41)$$

where $t_{b,l}^m$ is the average elapsed time since the validation performed on the data item upon the read access by the transaction of class m, which can be evaluated the same way as the single-class case.

The probability to abort while executing the 1-st operation on a shared data item for a transaction of class m, expressed as

$$p_{a,1}^{o,m} = (1 - p_{write}^m) \cdot (p_{lock}^m + (1 - p_{lock}^m) \cdot p_{u,l}^{o,m}), \quad (5.42)$$

and the probability to abort while executing the l-th operation with $l \geq 2$ for a transaction of class m, expressed as

$$p_{a,l}^{o,m} = p_{na,l}^{o,m} \cdot (1 - p_{write}^m) \cdot (p_{lock}^m + (1 - p_{lock}^m) \cdot p_{u,l}^{o,m}) \quad (5.43)$$

where $p_{na,l}^{o,m}$ is the probability of not aborting until the completion of the $(l-1)^{th}$ operation, for which we have

$$p_{na,1}^{o,m} = 1 \quad (5.44)$$

and

$$p_{na,l}^{o,m} = (1 - p_{a,l-1}^{o,m}) \cdot p_{na,l-1}^{o,m} \quad (5.45)$$

The contention probability during write-lock acquisition phase for a transaction of class m can be then approximated as

$$p^m_{wlc} = 1 - (1 - p^m_{lock})^{n^m \cdot p_{write}} \tag{5.46}$$

Hence the probability for a transaction of class m to abort at commit time due to write-lock contention is

$$p^m_{alc} = p^{o,m}_{na,n^m+1} \cdot p^m_{wlc} \tag{5.47}$$

The probability for a transaction of class m not to be aborted during its execution and to succeed in its commit-time lock acquisition phase is

$$p^{la,m}_{na} = p^{o,m}_{na,n^m+1} \cdot (1 - p^m_{wlc}) \tag{5.48}$$

The probability that a data item in the read-set of a transaction belonging to class m, which is accessed at the l-th transactional operation, has been updated when T executes the read-set validation can be expressed as

$$p^{r,m}_{u,l} = 1 - e^{-u^m_r \cdot t^m_{v,l}} \tag{5.49}$$

where $t^m_{v,l}$ is the elapsed time since the original validation, which can again be computed teh same way as for the single-class case.

Thus the abort probability due to failure in the validation of the lth data item within the read-set can be evaluated as follows

$$p^{r,m}_{a,l} = p^{r,m}_{na,l} \cdot (1 - p^m_{write}) \cdot (p^m_{lock} + (1 - p^m_{lock}) \cdot p^{r,m}_{u,l}) \tag{5.50}$$

where $p^{r,m}_{na,1} = 1$ and, for $l > 1$, $p^{r,m}_{na,l} = (1 - p^{r,m}_{a,l-1}) \cdot p^{r,m}_{na,l-1}$. Hence, the probability for a transaction of class m to abort during the read-set validation phase can be expressed as

$$p^m_{avf} = p^{la,m}_{na} \cdot p^m_{rvf} \tag{5.51}$$

where

$$p^m_{rvf} = \sum_{l=1}^{n^m} p^{r,m}_{a,l} \tag{5.52}$$

Finally we can evaluate the probability of successful commit when residing within state

$(i_1, .., i_q)$ as

$$p^m_{c,i_1,..,i_q} = p^{la,m}_{na}(1 - p^m_{rvf}) \tag{5.53}$$

For brevity we do not detail the equations for the evaluation of average transaction execution time and t_{GVC} because they can be simply derived by using the same approach we have show at the end of Section 5.3.3. In fact, the evaluation of the average transaction execution time for a transaction of class m can be done by using the already provided set of equations, by substituting the parameter values that depend on the specific transactional class with the ones we calculated in this section. Regarding

the evaluation of t_{GVC}, by using the approach discussed in Section 5.3.3, we have just to evaluate λ as the sum of the lock rate due to all the active transactions across the different classes, namely

$$\lambda = \sum_{m=1}^{q} l_r^m \cdot i_m \qquad (5.54)$$

5.3.5 Hints on Model Extension for Non-uniform Data Access

By relying on the approach in [9], which has been proposed for the case of concurrency control algorithms in database systems, our model could be extended to cope with non-uniform data accesses. We provide hints on how the extension could be realized in this section.

The proposed approach considers the whole set of d shared data items as grouped in s disjoint subsets, possibly exhibiting different cardinalities. The set of n operations executed by a transaction are grouped in s different subsets, possibly exhibiting different cardinalities, where each operation accesses a data item belonging to a different data subset. The accesses executed on each subset of data items by a transaction are uniformly distributed over the subset.

Different subsets of data items exhibit different access frequencies. As a consequence, the probability to find a lock raised on a data item and the data item update rate are different for each specific subset of data items. To evaluate them for a given subset we can use the same equations (5.10) and (5.12) by considering, in place of n, only the subset of operations executed by the transactions on that specific subset of data items. Consequently, the subsequent equations, expressing the abort probability for a transaction, can be determined by considering the probability of finding the lock raised and the data item update rate as differentiated for each subset, and then weighting the corresponding effects by the fraction of operations executed on the specific subset.

5.4 Simulation model

The simulation model has been implemented on a discrete-event simulation platform. It simulates a closed system with k concurrent threads which access shared data through an STM layer. The model incorporates k Thread (TH) simulation objects and an STM simulation object. Each TH simulates a thread which alternates the execution of transactions and non-transactional code blocks. The STM regulates the concurrency on basis of the rules of the considered CTL protocol. It keeps a list of d data items, and for each data item it keeps an *update* timestamp and a lock. Furthermore, the STM keeps an integer value to simulate the GVC mechanism.

When a TH has to execute a new transaction, the latter is created by selecting (relaying on pseudo-random number generation) the transaction class, the data items

to be accessed and, for each data item, the access mode (write or read). Then the TH sends a *begin* request to the STM. Upon a *begin* request the STM reads the value of the GVC and assigns this value to the *begin* timestamp of the transaction. After a time t_{begin} the STM notifies the TH of the completion of the request. Then the TH waits for an exponential time with mean t_{tcb} (in order to simulate the execution of a *tbc*) and sends to the STM an *operation execution* request for the first operation of the transaction. Upon this request the STM checks if the operation is a read or a write. In the former case it checks if the accessed data item is not locked and if the *update* timestamp of the requested data item has a value less than the *begin* timestamp of the transaction. If at least one check fails, then the transaction gets aborted. In this case, the STM, after a time $t_{read}+t_{abort}$, notifies the TH of the abort event. Otherwise the STM, after a time t_{read}, notifies the TH of the completion of the operation. In the case of write operation the STM simply notifies the TH of the completion of the operation after a time t_{write}. When the TH receives the notification of the completion of an operation, it continues by simulating the execution of the next *tbc* and after it moves to the next operation. When all operations of a transaction and the last *tbc* have been executed, the TH sends a *commit* request to the STM. The latter executes the commit operation as follows. If at least one data items in the write-set of the transaction is locked by another transaction then the STM notifies, after a time t_{abort}, the TH of the abort event, otherwise it acquires all locks. In the latter case the STM continues the commit operation by incrementing the GVC and then by validating the read-set, i.e. by checking if at least one *update* timestamp of the data items in the read-set has a value greater than the *begin* timestamp of the transaction. If the validation fails then the STM releases all previously acquired locks and notifies the TH of the abort event after a time $t_{commit}+t_{abort}$. Otherwise the STM updates all the *update* timestamps of the locked data items with the new value of the GVC, releases all acquired locks and notifies the TH of the commit event after a time t_{commit}. When the TH receives such a notification, it waits for a exponential time with mean t_{ntbc} (in order to simulate the execution of a *ntbc*) and then it moves to the execution of the next transaction. When the TH is notified of an abort event, it waits for an exponential time with mean $t_{backoff}$ and then re-executes the aborted transaction by sending to the STM the new *begin* request.

5.5 Validation

In this section we provide the results of an evaluation study aimed to verify the accuracy of the proposed modeling methodology, and of the presented CTL model. The study is based on the comparison between some key performance parameters determined via our analytical model and the corresponding values as obtained by means of the simulation model described in Section 5.4. The simulation results were obtained by repeating a number of independent runs (with different initial seeds for

the random generators) until the amplitude of the 90% confidence intervals on the throughput (committed transactions per second) became smaller than 10% of the average throughput value.

The workload parameters for this study have been selected on the basis of measurement and tracing activities, carried out for the STAMP benchmark [11]. To this end, we have exploited an implementation of TL2 which we have instrumented to trace the data access pattern and the costs associated with the corresponding operations, as well as the internal operations performed by the STM layer. Measurements have been carried out using a quad-core 2.4 GHz machine equipped with 4 GB of RAM and running the Suse Linux operating system (kernel 2.6.17).

In our study we focus on two of the applications included in the STAMP benchmark, namely Intruder and Vacation. Intruder is a signature-based network intrusion detection system which processes network packets in parallel via a user-tunable number of threads that concurrently update two main data structures, namely a FIFO queue and a self-balancing tree. In this benchmark, each thread spends around 33% of the time executing transactional code, and generates relatively short transactions, belonging to three different classes (capture, reassembly, and detection), the 90% percent of which exhibit a read plus write set made of up to n = 71 items, 30% of which are accessed in write mode. Based on our measurements, we set $t_{tcb} = 0.5\mu sec$, $t_{ntcb} = 5\mu sec$ and $t_{commit} = 2\mu sec$.

Vacation, on the other hand, implements an on-line transaction processing system emulating a travel reservation system. The system is implemented as a set of trees that keep track of customers and their reservations for various travel items. Client threads perform a number of sessions, each one enclosed in a coarse-grained transaction (compared to Intruder), which are again differentiated into three classes (reservations, cancellations, and updates), all interacting with the travel system's data layer. In this application, client threads spend almost all their execution time (92%) executing transactions, the 90% percent of which exhibit a read plus write set made of up to n = 200 items, 12% of which are accesses in write mode. Based on our measurements, we set $t_{tcb} = 0.2\mu sec$, $t_{ntcb} = 5\mu sec$ and $t_{commit} = 5\mu sec$

In addition to the above parameters, we used our tracing facility to determine also the following set of parameters: $t_{begin} = 0.2\mu sec$, $t_{read} = 0.25\mu sec$, $t_{write} = 0.2\mu sec$, $t_{abort} = 1\mu sec$. Finally, the back-off period, $t_{backoff}$, was set to $2\mu sec$.

By the above description, both the selected benchmark applications entail multi-class transactions. Hence the tracing process and the related outcomes have been used in a differentiated manner depending on whether the target is the validation of the single-class or the multi-class model.

To validate the single-class model, we configured the simulator to generate durations of the above mentioned timing activities based on exponential distributions. On the other hand, the validation of the multi-class version of the model, which captures more in detail the execution dynamics of the STM system, has been performed by replaying within the simulator the exact timing of actions as logged in the execution

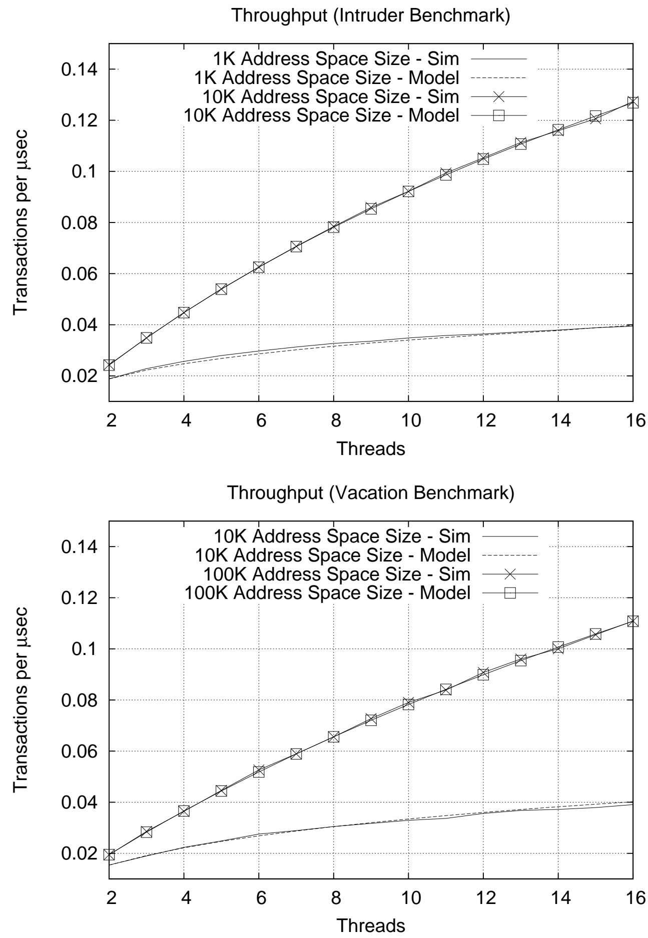


Figure 5.3: Throughput.

traces.

As for data accesses, the simulator generates them according to a uniform distribution across the total number of d data items/memory words (in compliance with the assumptions of our analytical model). The parameter d is treated as an independent parameter of the validation study. Note that, once fixed the number of threads, variations of d allow to capture settings with differentiated levels of contention, which, in their turn, determine different transactions' abort probabilities. Clearly, higher levels of data contention are achieved when the memory is configured with lower values of d, since transactional memory accesses by the threads are distributed on a smaller number of distinct memory words. We consider different values for the parameter d, associated, respectively, with reduced and increased values of the benchmarks' data-

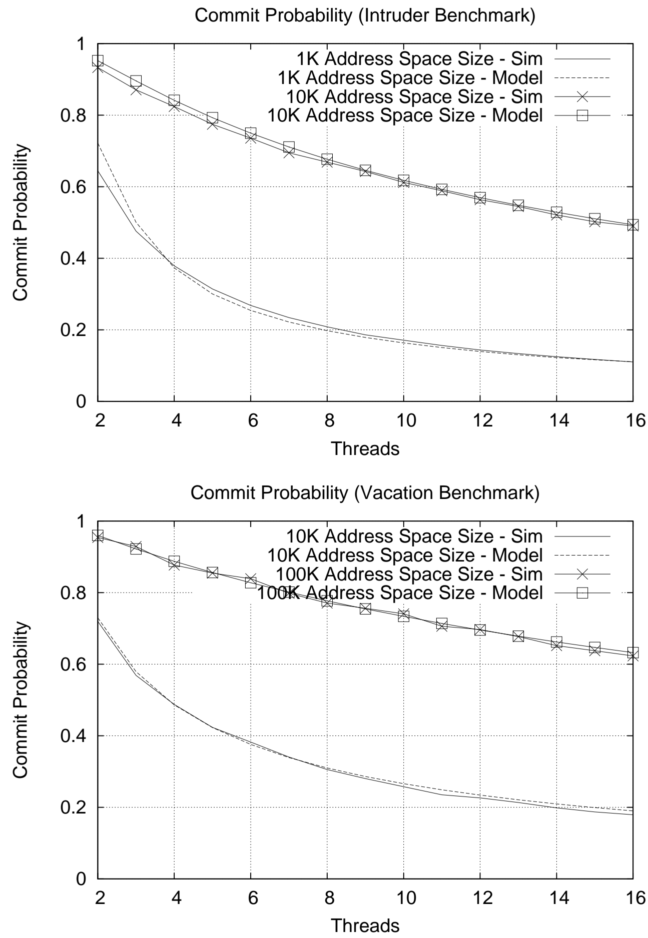


Figure 5.4: Commit probability.

set size according to the indications provided in [11]. Specifically, for Intruder, we set *d* to 1,000 and 10,000, whereas, for Vacation, we set *d* to 10,000 and 100,000.

5.5.1 Single-class Case

The comparison between analytical and simulation results is based on the following four parameters: (A) the system throughput (Figure 5.3), (B) the commit probability (Figure 5.4), (C) the mean execution time evaluated over each single run of transactions, independently of whether the run is committed or aborted (Figure 5.5) and (D) the likelihood of each of the possible causes of transaction abort (Figure 5.6).

The plots in Figure 5.3 and Figure 5.4 point out the accuracy of the presented

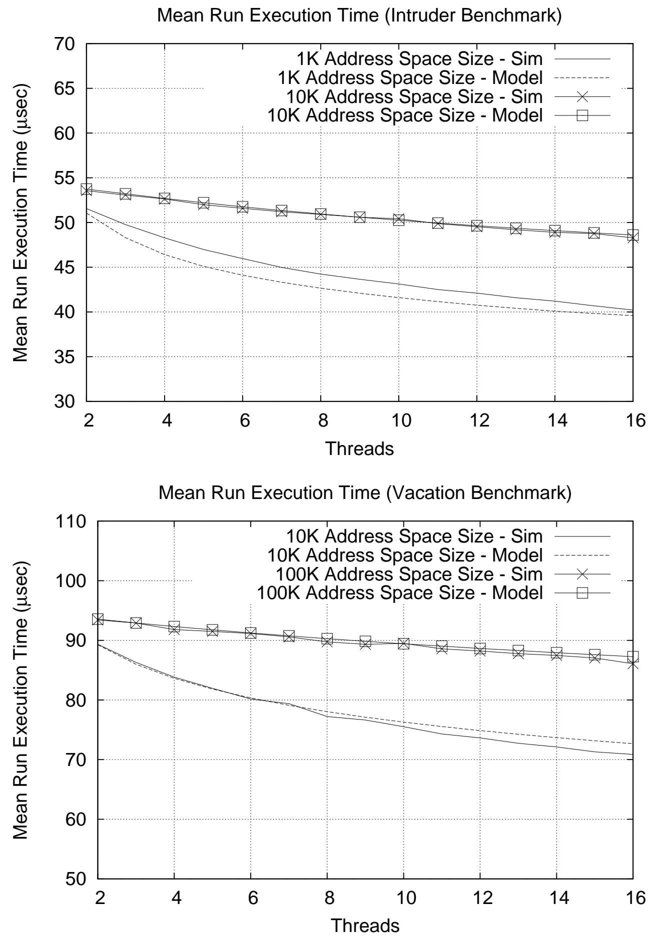


Figure 5.5: Mean run execution time.

analytical model, highlighting how analytical and simulation results coincide across the whole considered region of the parameters space, namely low vs high number of threads, as well as large vs small address space. In Figure 5.4, in correspondence with the lower value of d, we can appreciate the accuracy of the analytical model even in high contention scenarios (namely, for very reduced values of the transaction commit probability). As for Figure 5.5, we remark how, when considering the case of smaller address spaces, the relatively high contention probability often leads transactions to be early aborted (i.e., as soon as the first conflicting memory reference is issued), thus contributing to a reduction of the mean value for the run execution time. (Recall that the mean run execution time is evaluated over both committed and aborted run instances.) On the other hand, we observe an increase of the mean

run execution time in the configuration with larger address space, where the weight of aborted run instances becomes lower. Note that, due to the aforementioned early abort phenomenon, the variance of the mean run execution time grows in high contention scenarios. The above phenomenon, and their effects on the observed mean value, are correctly captured by our analytical model with very limited error, which is an additional support of the high accuracy of our analytical approach. The only exception is represented by the case of the Vacation benchmark when configured to use the smaller address space. In this case, the accuracy of the analytical model in predicting the mean run execution time is in fact subject to a slight deterioration as the number of threads increases. We argue that this is imputable to the fact that the Vacation benchmark comprises transactions whose execution latency is (on average) significantly longer than the Intruder benchmark. This leads to an increase of the variance of the run execution time and to a corresponding amplification of the model's prediction error.

In Figure 5.6 we evaluate the accuracy of the analytical model in predicting the different causes of aborts for the transactions. Specifically, we set the number of threads to eight and report: (i) the probability for a transaction to abort during its execution before reaching the commit phase (recall that this can only happen due to a validation failure during a read operation), denoted as $p_{a,ex} = 1 - p^{o}_{na,n+1}$ (see Equations (5.16-5.18)); (ii) the probability for a transaction to abort in the commit phase during the writeset lock acquisition, namely p_{wlc} (see equation (5.19)); (iii) the probability for a transaction to abort in the commit phase due to read-set validation failure, namely p_{rvf} (see equation (5.26)). Also in this case we observe that the accuracy of the proposed analytical model is very good for the scenarios in which the benchmarks are configured to use the larger datasets. On the other hand, with smaller datasets, namely the ones associated with very high contention rates (note that the probability of abort is around 0.7 and 0.8 in these scenarios), there is a slight degradation of the analytical model accuracy. We argue that this is imputable to the fact that the error introduced by assuming a Poisson Process for the arrival of transaction to the commit phase, which remains negligible at low/medium contention levels, shows an increasing trend at very high contention levels. This phenomenon is confirmed by the plots in Figure 5.7, where we evaluate the goodness of this assumption in different workload scenarios by contrasting the empirical density functions of the transaction interarrival time to the commit phase, as computed by the simulator, and the exponential distribution functions whose average value has been computed via the analytical model. More in detail, the plots on the left side of Figure 5.7 have been obtained by considering moderate contention scenarios obtained by selecting, for each benchmark, the largest address spaces and degree of concurrency equal to eight, that give rise to probability of abort on the order of 20% and 35% for Vacation and Intruder, respectively. On the other hand, the plots on the right side of Figure 5.7 are associated with a very high (and, arguably, somewhat pathological in practice) contention scenario, in which we select for each benchmark the smallest address

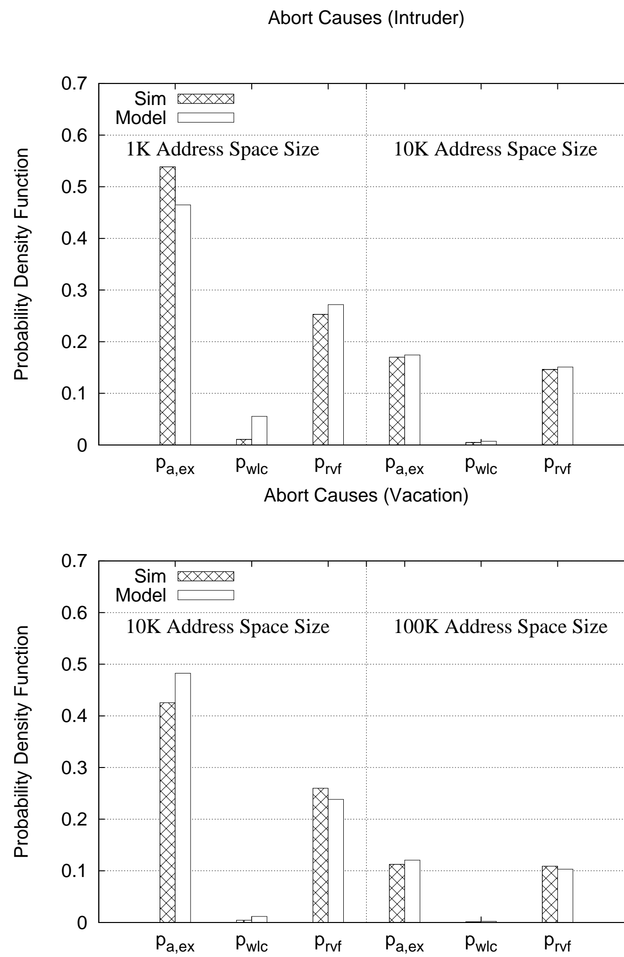


Figure 5.6: Abort causes.

spaces and degree of concurrency equal to eight, that give rise to probability of abort on the order of 70% and 80%, for Vacation and Intruder, respectively. The reported results clearly highlight that, up to medium contention levels, there is an excellent match between the empirical and analytical distributions, thus confirming the validity of the Poissionianity assumption for the commit phase arrival in case the timing of actions natively associated with the transactions follows exponential distributions. The right side plots, conversely, highlight a higher discrepancy between the empirical and analytical density functions in very high contention scenarios.

However, it is interesting to highlight that the degradation of the goodness of the poissionianity assumption leads to a (slight) increase of the model's error only when predicting some internal state variables, such as the likelihood of the various abort

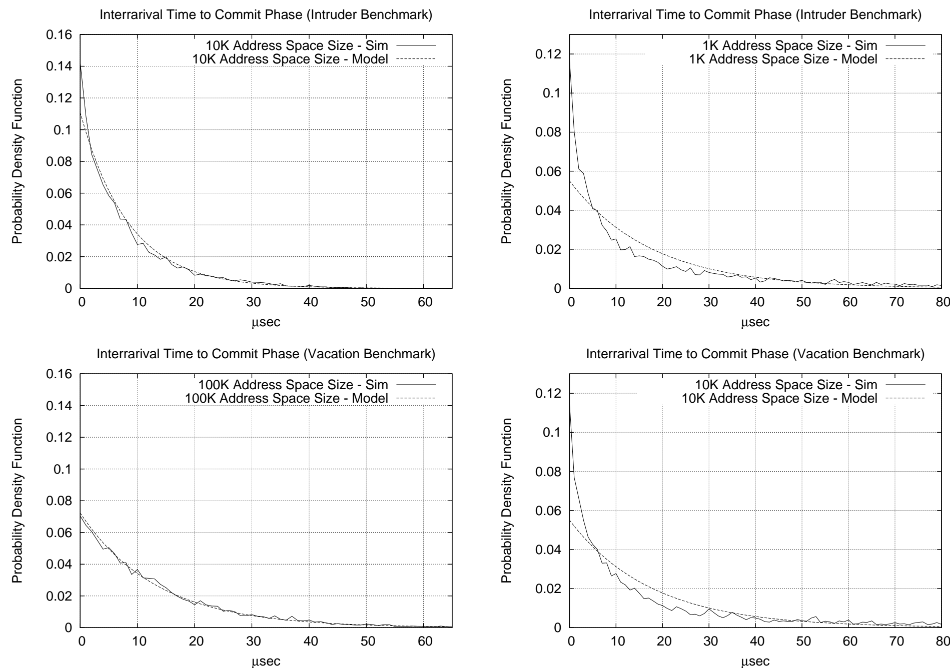


Figure 5.7: Distribution of the transaction interarrival times to the commit phase.

causes. On the other hand, the model's accuracy in predicting external performance metrics, such as throughput and commit probability, remains very high across every analyzed workload, even those associated with very high contention rate (see Figure 5.3 and Figure 5.4).

5.5.2 Multi-class Case

In this section we validate the variant of the analytical model capturing multi-class transaction profiles. To this purpose, the timing of accesses to shared memory data items has been simulated by replaying the execution traces of the Vacation benchmark. On the other hand, we used the reduced data set size selected for this benchmark (i.e., 10,000 data items) in order to stress the accuracy of the model when considering non-minimal contention scenarios. The parameters characterizing this workload are summarized in Table 5.1.

By the results shown in Figure 5.8 and 5.9, we have that the analytical model again shows a very good match vs simulative results. In particular, for each individual class the performance indicators are evaluated by the model in a very accurate manner while increasing the number of threads.

Parameter	Class 1	Class 2	Class 3
Transaction Class Probability (P^m)	0.898	0.047	0.056
Transaction Class Length (n^m)	154	57	121
Write Probability per Class (p^m_{write})	0.046	0.117	0.080

Table 5.1: Parameters used for the multi-class study (Vacation benchmark)

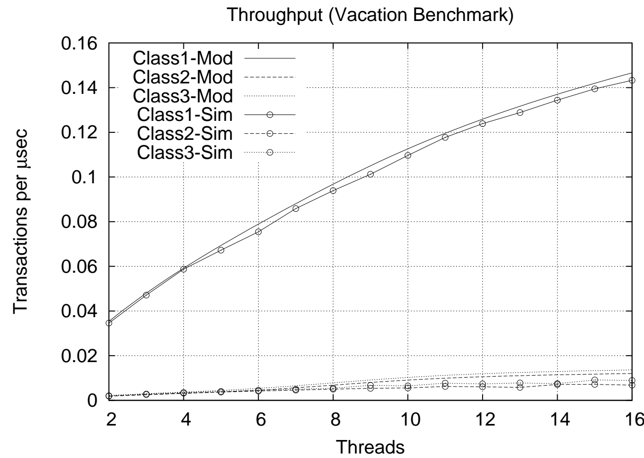


Figure 5.8: Analytical vs Simulative Results for the Multi-class Scenario.

5.6 On Removing Exponential Assumptions

In the analytical model presented in this chapter we have exploited the assumption of exponential distribution of several random variables. In this section we discuss how our modeling approach could be extended to relax some of these assumptions.

As for the thread-level model in Section 5.3.2, the reliance on a CTMC representation maps onto exponential assumptions for the times with which i) transactions exit from their back-off period following an abort event, and ii) the execution of a non-transactional code block is completed. If one want to consider generic, but known, distributions of these quantities, the CTMC could be replaced with other random process, e.g. a Semi-Markov process [76] or a Markov Regeneative process [77, 78]. At this point one should rely on solution techniques to calculatate the steady-state probability vector.

For what concerns the transaction-level model, we exploited the assumption that the arrivals of the transaction commit events form a Poisson process to compute $p^o_{u,l}$ in equation (5.14) and $p^r_{u,l}$ in equation (5.22). Further, we exploited the assumption

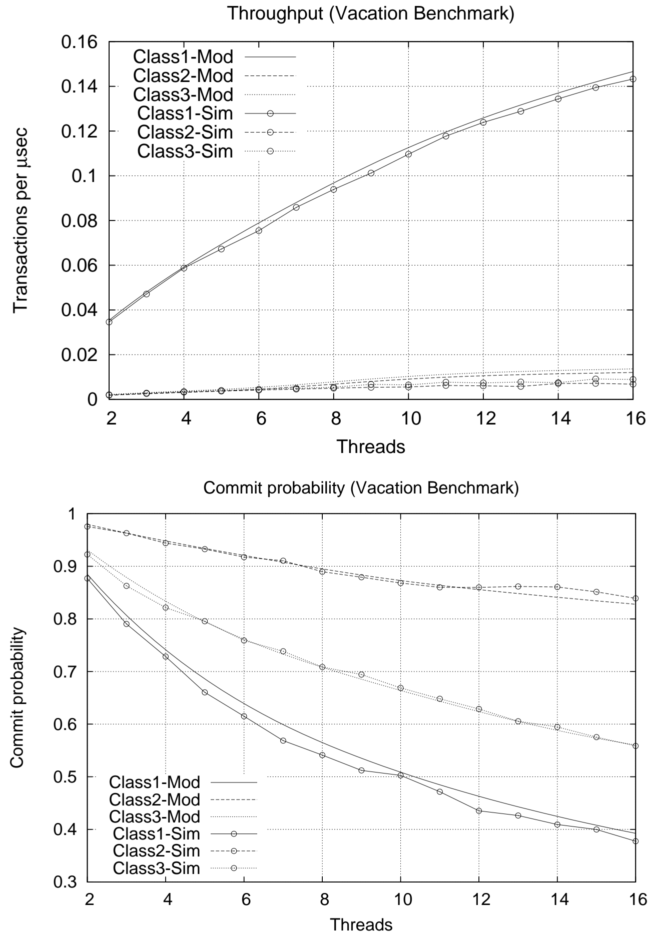


Figure 5.9: Analytical vs Simulative Results for the Multi-class Scenario.

that the arrivals of read operations on shared data items form a Poisson process to derive the expression of p_{lock} in equation (5.10).

As for equations (5.14) and (5.22), they could be extended to account for arbitrary distributions of the transaction arrivals to the commit phase. We could in fact write them as

$$p^{o}_{u,l} = \Phi(t_{b,l}, u_r) \tag{5.55}$$

$$p^{r}_{u,l} = \Phi(t_{v,l}, u_r) \tag{5.56}$$

where $\Phi(t, \eta), t \in (0, \infty)$ expresses the generic cumulative distribution function of the arrival process to the commit phase, having as $\eta = r_{t,i} = 1/E[\Phi(t, \eta)]$ its average

arrival rate, and u_r could be computed as before using equation (5.12) and equation (5.13).

More problematic would be, instead, relaxing the assumption that the arrivals of read operations form a Poisson process. In equation (5.10), in fact, we exploited directly the PASTA property [75] of Poisson arrival processes to compute the probability of finding a write-lock busy during a read on a data item x . However, if one were to assume that the arrival process of read operations on x formed a generic renewal process, one should explicitly account for the dynamic of interleaving between the arrival process of read operations on x and the stochastic process associated with the arrival of transactions that updated x to the commit phase. This would require determining the conditioned probability that, given an arbitrarily small interval $[t-h, t]$, there is a transaction T that is locking the data item x during its commit phase given that a transaction T' issues a read on x in the same time interval, or more formally:

$$lim_{h \to 0} Pr\{X(t-h) = 1 | N(t-h) \geq 1\} \tag{5.57}$$

where $X(t)$ expresses the number of transactions (that updated x) to be in the commit phase at time t and $N(t)$ is the counting process associated with the arrival of read operations (on x).

Chapter 6

Performance Modeling of CCPs With Arbitrary Data Access Patterns

6.1 Introduction

The development of analytical performance models of CCPs relies on the use of system workload models. The workload model strongly affects the performance model development. As we discussed in Section 3, tipically used workload models represent the transaction profiles in terms of various parameters, as the number of operations, the read/write probability and the distribution of accesses on the set of data items. On the other hand, they abstract from some (more detailed) features which differentiate the profile of workloads of transactional applications. This level of abstraction is generally considered adequate for the purpose the proposed performance models are intended. In fact, mainly, it allows to symplify the construction of effective models which, simultaneously, do not prevent from both (1) analyzing the dynamics related to the specific CCPs by allowing a quantitative evaluation of system performance indicators in a large workload configuration space, and (2) understanding the motivations behind performance provided by different CCPs under the same workload profiles. On the other hand, the choice of such a level of abstraction restricts the range of validity of the performance models. In particular, when the analysis focuses on more detailed workload profiles, e.g. considering a specific class of applications, the workload model could be unable to provide an adequate representation of them.

In this chapter we address the aforesaid issue by considering a tipical assumption used in workload models which may remarkably affect the accuracy of the analytical performance models of CCPs. Basically, in previous performance modeling studies it is assumed that a transaction accesses data items according to some probability distribution which does not depend on the phase of the transaction execution (i.e. the probability distribution is considered to be the same for each operation executed by the transaction). This suggests some observations. In many applications, transactions access the data items, or sets of data items, according to specific patterns. Let's consider an example. In a warehouse application a transaction which creates an order of a customer in the database may execute the following actions: reading the list of products ordered by the customer, checking the customer address, updating the order information and updating the availability of the products in stock. This could lead to the execution of the following sequence of operations: reading from the Cart table, reading from the Customer table, updating the Orders table and, finally, updating the Products table. We note that this entails a specific data access pattern to be executed by the transaction, where each database table is accessed depending on the execution phase of the transaction. This feature is very common in many transactional applications (e.g. see TPC-C [3] and TPC-W [13] benchmark applications).

In particular, the question which arises from the aforesaid observations is the following: If the presence of specific transaction data access patterns is not considered in performance models of CCPs, can these models be considered reliable when a more realistic workload is considered?.

The results of a simulation study we conducted, which we show later in this chapter, clearly showed as the performance provided by locking protocols which acquire locks before to execute operations can be very sensitive to the variation of transaction data access patterns. Specifically, if we consider two equal workloads, except the sequence of accessed data items by the transactions, the performance can remarkably change. As a consequence, performance models which do not take into account such an aspect could provide unreliable results.

To cope with this problem, we propose an analytical modeling approach for these types of locking protocols which allows to build performance models capable to capture the effects on system performance of the transaction data access patterns. Specifically, we consider the case of the Strong-Strict 2PL (SS2PL) protocol (see Section 2.2), as in our study it showed a high sensitivity to the access patterns, and we provide a model tailored for it. We recall that the SS2PL is one of the most used protocol in (commercial) database systems. The aforesaid types of protocols are also largely used in STMs.

The analytical model we present allows to evaluate the main system performance indicators, as the expected transaction execution time and the throughput saturation point. Furthermore, it allows to evaluate other indicators which can be used to study more in deep the impact of the transaction data access patters, as the mean lock holding time and the transaction wait time when a lock conflict occurs. The model

can be coupled with different hardware resource models and can be resolved via an iterative technique.

The accuracy of model has been evaluated via simulation. Our validation study relies on both (i) Synthetic workload descriptions (e.g. in terms of machine instructions for specific operations within a transaction), some of which analogous to those used for the validation of previous analytical results coping with SS2PL, and (ii) A workload we derived by abstracting the main features of the transaction profiles specified by the TPC-C benchmark [3].

The rest of this chapter is structured as follow. In Section 6.2 we present some results of the simulation study we conducted which show the sensitivity to the transactions data access patterns of the SS2PL protocol. In Section 6.3 we describe the system model we used to build the analytical model, which is presented in Section 6.4. In Section 6.5, we present a model validation study. Finally, in Section 6.6, we present further results of our simulation study on sensitivity to the transactions data access patterns for other protocols, including those we considered in previous chapters of this dissertation, i.e. the MVCC and the CTL.

6.2 Effects of Data Access Patterns with SS2PL Protocol

Intuitively, in locking protocols as SS2PL, the presence of trends in the sequence of data items accessed by the transactions can have a significant impact on the distribution of locks' duration. Consider, for instance, two sets of data items, say X and Y, which are always the first, respectively the last, ones to be accessed within a transaction. Being the duration of the locks held on the data items of set X longer than the duration of the locks on the tuples of set Y, it follows that the probability of contention on the data items of set X will be much higher than the probability of contention on the data items belonging to set Y. This may have an impact on the conflict probability of transactions, hence, consequently, it may lead to a non-minimal impact on the system performance.

We present the results of a simulation test where we reproduced the workload used in another performance analysis work for the SS2PL [9]. The comparison with the results of another previous work allowed us also to validate our simulation model. In this test we considered two different transaction profiles. In the first transaction profile (which we name phase-independent), the accesses are uniformly distributed across the whole set of items, independently of the transaction execution phase. Furthermore, transactions execute 15 data accesses in write mode. This profile reproduces the transaction profile used in the aforesaid work. In the second transaction profile (which we name phase-dependent) transactions execute, as in the first one, 15 data accesses in write mode. However the accessed data items varies across different transaction execution phases. Specifically, the whole set of items is partitioned into 5 equally sized, non-overlapping sets $\{S_1, \ldots, S_5\}$ (which might be seen as represen-

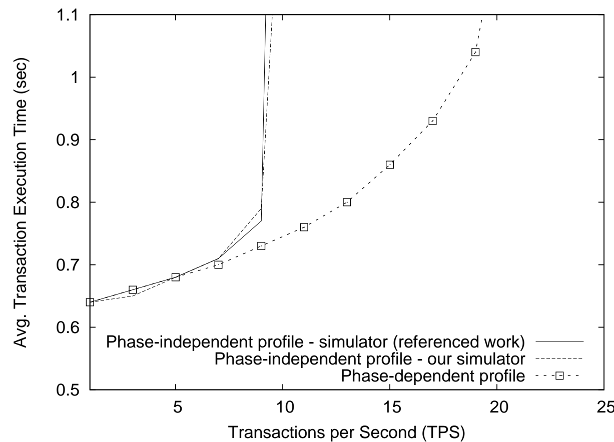


Figure 6.1: Performance comparison with phase-independent and phase-dependent profiles

tative of, e.g., distinct database tables). In the first phase the transactions perform three accesses, uniformly distributed in the set S_1. In the second phase they perform three accesses uniformly distributed in the set S_2. Hence transactions continue in this fashion, until they complete 15 data accesses. All the other workload and system configuration parameters are equal to those used in the referenced work. In Figure 6.1 we plotted the average transaction execution time for the workload composed by transactions with the phase-independent profile as evaluated by both the simulation model used in the referenced work and by our simulator. The curves are quite overlapped, demonstrating the validity of our simulation model vs the simulation model used in the referenced work. Furthermore, in the same figure we plotted the average transaction execution time evaluated by means of our simulation model for the workload composed by transactions with phase-dependent profile. The results show clearly as the difference is remarkable, demonstrating the sensitivity of the SS2PL to the transaction access patterns.

6.3 System Model

In this section we present the system model we assume to build the analytical model. We consider a transactional system with a set of I items, each of which represents an unit of data that can be accessed by an operation within a transaction (e.g a tuple or a set of tuples in a table of a database). Transactions are processed according to the SS2PL protocol. We assume an open system model. This choice is motivated by the fact that open models are more suited for scenarios with a large number of users (like

in, e.g., transactional applications over the Internet).

6.3.1 Transaction Model

We assume a workload characterized by a single transaction profile. This is specified in terms of data items accessed and locality of the accesses during the lifetime of the transaction. Transactions arrive according to a Poisson process with arrival rate λ. On the basis of these assumptions we present a basic version of the analytical model with one transaction class. In Section 6.4.7 we present a model extension where we consider multiple transaction classes to cope with workloads with differentiated transaction profiles.

Each transaction consists of a begin phase, which is followed by a number of M operations, each one accessing in read or write mode a single data item, and finally by a commit phase. According to the SS2PL, to execute a read operation, a transaction has to obtain a shared lock on the target data item, while, for write operations, exclusive locks are needed. Each operation might entail a wait (block) phase in case the requested lock is currently unavailable. During the initial and commit phases a mean number of nI_b and nI_c CPU instructions, respectively, are executed. Also, the execution of an operation is assumed to require a mean number of nI_o CPU instructions. In case the access to a data item causes a buffer miss, a time $t_{I/O}$ is needed to fetch the data from disk. Finally, for simplicity, we do not explicitly model the I/O delay associated with the commit phase (e.g. the transaction log write onto stable storage). Anyway, given our disk model, this delay would only entail an additional latency term in the expression of the transaction execution time.

To cope with access patterns executed by the transactions, we represent the transaction access pattern by means of a $I \times M$ matrix (which we name access matrix) denoted by A. Element $A_{i,k}$ expresses the probability that the kth operation of the transaction accesses the ith data item. Note that the sum of each column of A must be equal to 1. Further we represent with a vector W, with $|W| = M$, the type of the access, namely read or write, performed by the transaction in the different phases of its execution. Specifically, W_k (resp. $1 - W_k$) is probability that the kth operation is a write (resp. read) operation. The access matrix A and the vector W are the building block allowing our model to capture the transaction execution history, and its effects on performance, in terms of locality variation in different phases of its execution.

For the sake of clarity, let us now consider an example transaction T characterized by a simple access pattern on a small database consisting solely of $I = 4$ different data items. Let us assume that T carries out 2 data accesses, respectively a read and a write operation. The read operation accesses to the data item 1, whereas the write operation is targeted, with equal probability, to the data item 2 or 3. Based on these assumptions, we can describe the data access pattern of T through the following access matrix A and vector W:

$$A = \begin{pmatrix} 1 & 0 \\ 0 & 0.5 \\ 0 & 0.5 \\ 0 & 0 \end{pmatrix}$$

$$W = (0, 1)$$

As a further example, the matrix can be instantiated in a way that the access to a specific item j is always prevented up to a given phase f of the transaction execution (this can be done by setting to 0 all the elements $A_{i,k}$ with $i = j$ and $k \leq f$). This, in its turn, captures scenarios where, e.g., items inside a given table of a database are always accessed after operations on other tables have been already executed.

6.3.2 Hardware Resource Model

In accordance with typical assumptions in previous analytical studies (e.g. [79, 9]), we assume a transactional system with an underling hardware system where the CPU is modeled as an M/M/k queue. k is the number of CPU-Cores, each of which has processing speed denoted as $MIPS$ (measured in terms of million instructions per second). The disk has a fixed I/O delay denoted as $t_{I/O}$. Anyway, we underline that our focus is on the effects of data accesses and contention on logical resources, not on physical resources. In fact, the contribution we provide is orthogonal to the assumed model for the underlying physical system, given that our model for logical resources' contention can be actually coupled with different models for physical resources.

6.4 The Analytical Model

On the basis of the transaction model described in Section 6.4.1, a transaction can be modeled through a direct graph (see Figure 6.2), where the nodes represent different states of the transaction execution and the arcs represent state transitions. A label on an arc from a node p to a node q represents the transition probability from state p to state q. If the label is omitted, then the transition probability is intended to be 1. States labelled with *begin* and *commit* represent the initial and commit phases respectively, while the state labelled with $\hat{k}$ represents the execution of the kth operation, and, finally, the state labelled with $\tilde{k}$ represents a waiting phase (due to lock contention) preceding the kth operation. We denote with $P_{W,k}$ the probability that the requested data item by the kth operation is currently locked.

In the following we make some assumptions that we consider in our analytical model. Other assumptions will be made in the next sections. We remark that all the

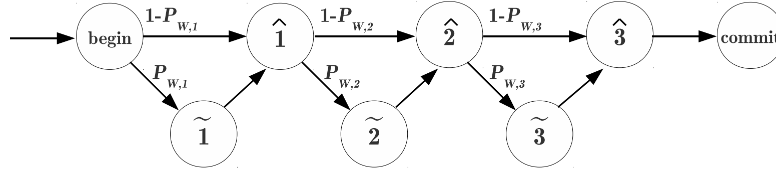


Figure 6.2: Transaction model.

assumptions we will make from this point will be considered only for the construction of the analytical model, but not in the simulation model.

With locking protocols, a transaction can be aborted by the deadlock manager, however we ignore deadlock related aborts in our analytical model since previous studies (e.g., [67], [42]) have shown as their effects on the final perceived performance are negligible with respect to the data contention effects, therefore they can be considered not relevant in performance analysis.

We assume a buffer hit probability P_{BH} when a data item is accessed. Actually, we do not explicitly model the buffering policy and the related effects since several models have already been proposed to cope with the evaluation of hit probability vs the item popularity, see, e.g. [80], which is orthogonal to our study. Hence, P_{BH} will be considered as an independent parameter in our study.

Finally, we assume the system to be stable and ergodic.

6.4.1 Transaction Execution Time

According the transaction model we have presented above, we denote with R_{begin} and R_{commit} the times spent in states *begin* and *commit* respectively, and with $\hat{R_k}$ and $\tilde{R_k}$, where $1 \leq k \leq M$, the times spent in states $\hat{k}$ and $\tilde{k}$, respectively. The evaluation of these times is presented in the following sections. The mean transaction execution time can be evaluated as the sum of the average times spent in each state, that is:

$$R_{tx} = R_{begin} + \sum_{k=1}^{M} (\tilde{R_k} + \hat{R_k}) + R_{commit},$$

6.4.2 Lock Holding Time

The wait phase experienced by a transaction for lock acquisition on a given data item depends on the average lock holding times of transactions preceding T in the lock

access queue on that item. In our model we explicitly capture the fact that accesses to data items can occur at different phases of a transaction. Hence, if data item a is typically the first one to be accessed by transactions, and data item b is normally the last one to be accessed, then the average lock holding time on item a will be significantly longer than the lock holding time on item b.

We evaluate the lock holding time for each data item, and how it is affected by the transaction access pattern, by exploiting the access matrix A. Specifically, if data item is accessed by a transaction at the kth operation, then it gets locked up to the end of the execution of the commit phase. Hence, the lock holding time for the access to a data item at the kth phase can be expressed as:

$$D_k = \sum_{j=k}^{M} \hat{R}_j + \sum_{j=k+1}^{M} \tilde{R}_j + R_{commit}.$$

We know that the probability to access data item i at the kth transaction phase is expressed as $A_{i,k}$. Hence, the average lock holding time for data item i can be evaluated as:

$$Th_i = \frac{\sum_{k=1}^{M} A_{i,k} D_k}{\sum_{k=1}^{M} A_{i,k}},$$

where the sum at denominator is due to the fact that the average lock holding time must be evaluated by considering only the transactions for which an access to data item i actually occurs.

6.4.3 Data Contention

The arrival rate of read accesses towards the ith data item can be expressed as:

$$\lambda_{read,i} = \lambda \sum_{k=1}^{M} A_{i,k}(1 - W_k),$$

while for write accesses we have:

$$\lambda_{write,i} = \lambda \sum_{k=1}^{M} A_{i,k} W_k.$$

We recall that if the data item is requested by a write operation and it is locked (in either shared or exclusive mode) then a lock conflict occurs, the transaction is blocked and the write operation is enqueued. On the other hand, if the data item is requested by a read operation then a lock conflict occurs only if the item is locked in exclusive mode. Hence only in this case the transaction is blocked and the read operation is enqueued. To cope with the determination of data contention and transaction wait time without overly increasing the complexity of the our model, we built

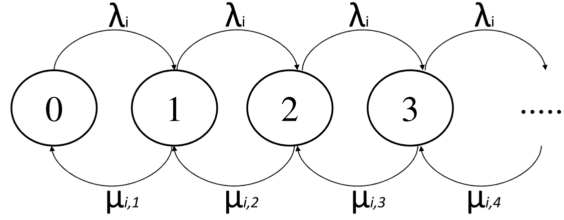


Figure 6.3: Markov chain for data item *i*.

a simple approximate queue model to capture the read-write lock conflict dynamics. By the results of the validation study, the used approximations are adequate for our purpose. We recall that in typical scenarios of database system applications modeled by means of an open system, with protocols which block transactions on lock conflict, such as the SS2PL, the system reaches the saturation point when the lock contention probability accessing a data item is still relatively low. E.g., in tests presented in [9] the saturation point is reached when the lock contention probability is lower than 10 percent. In all the tests we present in Section 6.5 the saturation point is reached when the lock contention probability is, on average for all data items, lower than 15 percent. Hence, approximate solutions which provide a good accuracy with low data contention level are generally considered adequate. For example, in some studies (e.g. [53]) the proposed analytical models are based on the assumption that the number of queued transactions waiting for a lock on the same data item is at most equal to one.

We have modeled the lock contention on each single data item *i* as a birth-death process [69] with fixed arrival rate, equal to

$$\lambda_i = \lambda_{read,i} + \lambda_{write,i}$$

and variable service rate $\mu_{i,j}$ (see Figure (6.3)), where j corresponds to the number of standing requests for data item *i* in the corresponding state of the Markov chain. For each single data item *i* the value $\mu_{i,j}$ depends on the interleaving of read and write requests observed in state j. We approximate $\mu_{i,j}$ with its average value, calculated as follow. If in state j the top standing request for lock access is a write request, then $\mu_{i,j}$ is equal to $\frac{1}{Th_i}$. In fact, since the exclusive write lock delivered to the write request blocks any other standing request, then the item is reserved for the write request, whose expected locking time is Th_i. On the other hand, if the top standing request is a read request, all the other standing read requests, if any, can be concurrently served. In the latter case, if in state j there are $l \leq j$ standing read requests, then we have $\mu_{i,j} = \dfrac{l}{Th_i}$. Overall, denoting with $P_{read,i}$ and $P_{write,i}$ the probability that an incoming

access request is a read request or a write request, respectively, we have

$$P_{read,i} = \frac{\lambda_{read,i}}{\lambda_i}$$

and

$$P_{write,i} = \frac{\lambda_{write,i}}{\lambda_i}.$$

We approximate the probability for the top request in state j to be a write request (respectively a read request) with $P_{write,i}$ (respectively $P_{read,i}$).

Thus we have

$$\mu_{i,j} = \frac{1}{Th_i}(P_{write,i} + ((P_{read,i} \sum_{k=1}^{j-1} kP_{read,i}^{k}) + jP_{read,i}^{j}))$$

When a write access occurs, a conflict is raised if the target data item i is locked either in shared or in exclusive mode. Thus we can model the contention probability for a write access $PW_{write,i}$ on data item i as the sum of the probabilities to stay in any of the states j, with $j > 0$, of the Markov chain, which is equal to $1 - P_0$ (where P_0 is the probability to be in state 0 of the Markov chain). Hence, from queuing theory [69], we have

$$PW_{write,i} = 1 - \frac{1}{1 + \sum_{k=1}^{\infty} \prod_{j=0}^{k-1} \frac{\lambda}{\mu}_{i,j+1}},$$

By the formula it can be noted that $PW_{write,i} \leq 1$ only if the sum at the denominator converges to a finite value. Given that $\mu_{i,j} \geq \frac{1}{Th_i}\ \forall j > 0$, the condition $\frac{1}{Th_i} > \lambda_i$ for every data item i in the transactional system is sufficient to ensure that the contention probability for any write access is less than 1, thus representing a stability condition for the system.

To evaluate the contention probability of a read access we recall that a conflict can occur only if the data item is locked in exclusive mode. Hence, the contention probability can be evaluated as the fraction of time during which the data item is locked in exclusive mode. This time fraction corresponds to the utilization of the data item vs write accesses. Thus we have

$$PW_{read,i} = \lambda_{write,i} Th_i.$$

6.4.4 Wait Time

When an incompatible lock is found on the currently required data item, the transaction experiences a wait time t, which corresponds to the time spent in state $\tilde{k}$ (see Figure 6.2), with k being the index of the operation causing the conflicting access.

The wait time depends on the data being requested (i.e. on the amount of currently standing access requests for that data item), not on the value of k.

We firstly evaluate the average waiting time for a transaction in case of a conflict on a specific data item i, which we denote as $R_{wait,i}$. After we evaluate the average waiting time experienced in each state $\tilde{k}$ (with $1 \leq k \leq M$), which we denote as $\tilde{R}_k$. The latter value will depend on the transaction access matrix A, which expresses, for each operation, the likelihood of access to each specific item. $R_{wait,i}$ can be evaluated through the aforementioned Markov chain associated with data item i. In particular, the average amount of standing accesses is

$$N_i = \sum_{j=1}^{\infty} jP_j,$$

where P_j is probability to stay in state j of the Markov chain. When a conflict occurs upon data access, if no other access requests to the same item are currently queued, the wait time corresponds to residual lock holding time. On the other hand, in case other access requests are currently queued for lock acquisition, a further delay occurs due to lock holding on that item by transactions associated with the queued requests, that is on average Th_i for each one. Thus, given that a conflict has occurred, we have

$$R_{wait,i} = \left(\frac{N_i}{1 - P_0} - 1\right) Th_i + L_i,$$

where:

$$L_i = \frac{\sum_{k=1}^{M} A_{i,k} D_k^2}{2 \sum_{k=1}^{M} A_{i,k} D_k}$$

and represents the normalized residual lock duration, depending on the different durations evaluated on the basis of the access pattern. Now, through $R_{wait,i}$, by exploiting the access matrix A, we have

$$\tilde{R}_k = \sum_{i=1}^{I} A_{i,k} R_{wait,i} \left[PW_{read,i}(1 - W_k) + PW_{write,i} W_k\right]$$

6.4.5 Operation Execution Time

Times spent by a transaction in states *begin*, $\hat{k}$, with $1 \leq k \leq M$, and *commit* can be evaluated by exploiting the model of the underlying hardware resources, which has been provided in Section 6.3.2. The CPU load for the execution of a transaction is

$$C_{cpu} = nI_b + M \cdot nI_o + nI_c$$

and from queuing theory we get for the CPU utilization the following expression:

$$\rho = \frac{\lambda \cdot C_{cpu}}{k \cdot MIPS}.$$

Denoting with $p[queuing]$ the wait probability for a request in an M/M/k queue [69], and defining γ as

$$\gamma = 1 + p[queuing]/(k(1-\rho)),$$

we can evaluate the execution times R_{begin} and R_{commit} of states *begin* and *commit* respectively as:

$$R_{begin} = \gamma \frac{nI_b}{MIPS}$$

and

$$R_{commit} = \gamma \frac{nI_c}{MIPS}.$$

Execution times of states $\hat{R}_k$ further depend on buffer hit probability and I/O delays. Using the notation in Section 6.3, we have

$$\hat{R}_k = \gamma \frac{nI_o}{MIPS} + P_{BH} \cdot t_{I/O}$$

for each k such that $1 \leq k \leq M$.

6.4.6 Numerical Resolution

The model can be solved via an iterative procedure. After assigning the values to hardware configuration parameters (e.g the CPU power) and transactional system parameters (e.g. the access matrix) the value 0 has to be assigned to the parameters $R_{wait,i}$, $PW_{read,i}$ and $PW_{write,i}$ (with $1 \leq i \leq I$). Then the other model parameters can be evaluated via the provided equations, using the results as the input for the next iteration. The desired computational accuracy can be fixed by defining a value ϵ specifying the maximum difference between values obtained by two consecutive iterations (e.g. if R_n is the transaction execution time at iteration n, then the computation can be stopped when the condition $R_n - R_{n-1} \leq \epsilon$ becomes true).

This iterative approach has been used also in other pre-existing studies on performance models of CCPs (e.g, [70, 81, 52, 9]). As it has been done in these studies, we have empirically observed that it converges in a few iterations in all tests we carried out, provided that the input assignment defines a stable system.

6.4.7 Coping with Multiple Transaction Classes

In this section we show how our model can be employed in scenarios where the workload entails different transaction profiles (or classes). We denote as C the number of

the different transaction classes, each of which can be represented through a specific transaction model featured as the one described in 6.4.1. We use the following notation:

- Vector $\overline{M}$, with $|\overline{M}| = C$, for which each element, denoted by M^c, with $1 \leq c \leq C$, represents the number of operations of a transaction of class c.

- Vector $\overline{A}$ of matrix elements, with $|\overline{A}| = C$, for which each element, denoted by A^c, with $1 \leq c \leq C$, is the access matrix of a transaction of class c.

- Vector $\overline{W}$, with $|\overline{W}| = C$, for which each element, denoted by W^c, with $1 \leq c \leq C$, is the vector representing the write probabilities for operations of a transaction of class c.

Further the transaction arrival rate for class c is denoted by λ_c.

The accuracy level while describing a workload with differentiated transaction profiles according to the previous notation can be tuned in accordance to the requirements of the performance analysis the end-user is carrying out. Roughly speaking, the more the identified transaction classes, the more accurate the workload description. As an extreme, each plausible access pattern could be associated with a specific class in such a way to describe the variation of the transaction locality over the data items in a deterministic manner. In this case each access matrix will be characterized by columns having a single element equal to 1, and all the other elements equal to 0. As it will be clear by the below description of the modifications to the model equations in case of multiple classes, a large number of classes will only entail an increased amount of computation power for the iterative model solving procedure. In general, if transactions are composed by a fixed number of predefined statements, as in, e.g., a lot of three-tier Web based applications, to obtain a good compromise we suggest to model the workload using a single class for each predefined transaction pattern.

With more transaction classes, some of the previously introduced equations must be rewritten in order to consider parameter dependency on the access pattern and the arrival rate of each class. For simplicity, we only show the final shape of these equations without explicitly repeating intermediate modeling steps, which are anyway intuitive once the model for the case of single transaction class has been analyzed.

The transaction execution time for class c is

$$R^c_{tx} = R^c_{begin} + \sum_{k=1}^{M^c} (\tilde{R}^c_k + \hat{R}^c_k) + R^c_{commit},$$

where we added the superscript c to the parameters introduced in Section 6.4.1 to emphasize that each of them is related to class c. The average lock holding time for

data item i becomes

$$\frac{\sum_{c=1}^{C} \lambda^c \sum_{k=1}^{M^c} A_{i,k}^c (\sum_{j=k}^{M^c} \hat{R}^c{}_j + \sum_{j=k+1}^{M^c} \tilde{R}^c{}_j + R_{commit}^c)}{\sum_{c=1}^{C} \lambda^c \sum_{k=1}^{M^c} A_{i,k}^c},$$

The arrival rates of read and write accesses towards the ith data item become

$$\lambda_{read,i} = \sum_{c=1}^{C} \lambda^c \sum_{k=1}^{M^c} A_{i,k}^c (1 - W_k^c),$$

and

$$\lambda_{write,i} = \sum_{c=1}^{C} \lambda^c \sum_{k=1}^{M^c} A_{i,k}^c (W_k^c).$$

In the end, we can rewrite the equation of $\tilde{R}_k$ for each transaction class as

$$\tilde{R}_k^c = \sum_{i=1}^{I} A_{i,k}^c R_{wait,i} [PW_{read,i}(1 - \overline{W}_k^c) + PW_{write,i} \overline{W}_k^c]$$

6.5 Model Validation

We evaluated the accuracy of the analytical model via a set of differentiated tests based on output comparison vs the results obtained by using discrete-event simulation. The simulation model we used is similar to the model we presented in section 4.5, except the Concurrency Control Manager, which follows the rules of the SS2PL, and the Workload Generator, which generates the operations of the transactions on basis of the access pattern of the transaction class.

In this section we present a set of tests we carried out in relation to three scenarios characterized by diverse workload configurations and system parameters. This section is structured as follow. In Part-A we show the results of a validation test where we reproduced the test we described in Section 6.2. In Part-B we consider a synthetic workloads which induces noticeable effects on lock contention across different transaction classes. In Part-C we finally provide validation results for the case of transaction workloads derived by abstracting the main features of the well known TPC-C benchmark [3].

#Items	1000
#CPUs	5
CPU Speed	10 MIPS
$t_{I/O}$	0.035s
#Accesses x Xact (M)	15
P_{write}	100%
P_{BH}	0.27
nI_b	150000
nI_o	20000
nI_c	250000
Access Distribution - Phase i (Phase Independent Workload)	Unif. in [1,1000]
Access Distribution - Phase i (Phase Dependent Workload)	Unif. in $[1+\lfloor\frac{i-1}{3}\rfloor\cdot 200,(\lfloor\frac{i-1}{3}\rfloor+1)\cdot 200]$

Table 6.1: Parameters settings for Part-A (as in [9]).

#Items	100000
#CPUs	8
CPU Speed	2000 MIPS
$t_{I/O}$	0.004 ms
P_{BH}	0
P_{write}	20%

Table 6.2: Parameters settings for Part-B.

6.5.1 Part-A

We start by showing the results of the validation test where we reproduced the workload with the phase-dependent transaction profile describe in Section 6.2. All system configurations parameters have been set according to the original test we inspired to (i.e. the test in [9]). In Figure 6.4 we plotted the average transaction execution time as evaluated by our analytical model and by simulation. As we noted in Section 6.2, with the phase-dependent profile the transaction response time remarkably changes due to the effects of access locality variations across different transaction execution phases. The results show as our model well captures this phenomenon.

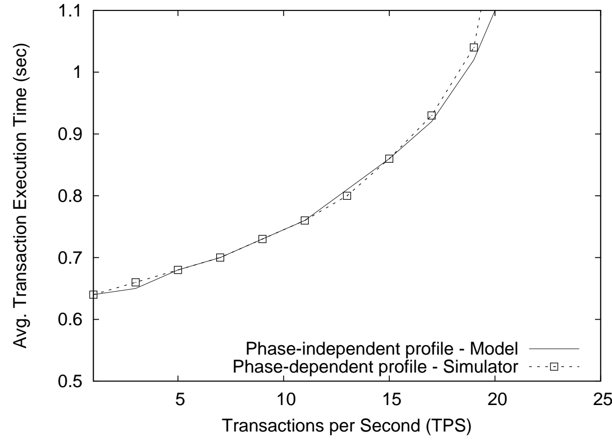


Figure 6.4: Performance comparison for both independent and phase-dependent profiles (Part-A).

	#Accesses (M)	Access Distribution - Phase $i \in$[1,M]
Profile P_1	20	Uniform in $[1+\lfloor\frac{i-1}{4}\rfloor\cdot 20000,(\lfloor\frac{i-1}{4}\rfloor+1)\cdot 20000]$
Profile P_2	8	Uniform in $[1+\lfloor\frac{i-1}{4}\rfloor\cdot 20000,(\lfloor\frac{i-1}{4}\rfloor+1)\cdot 20000]$
Profile P_3	8	Uniform in $[1+(\lfloor\frac{i-1}{4}\rfloor+3)\cdot 20000,(\lfloor\frac{i-1}{4}\rfloor+4)\cdot 20000]$

Table 6.3: Synthetic workload (Part-B).

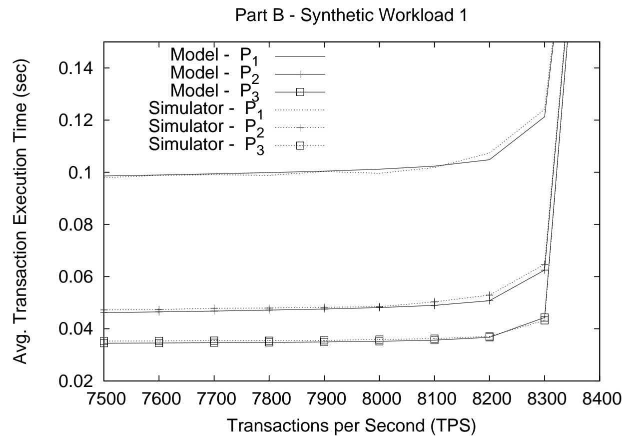


Figure 6.5: Transaction execution time (Part-B).

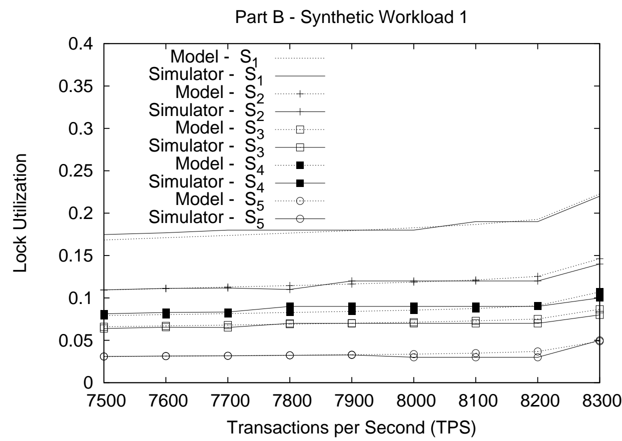


Figure 6.6: Lock utilization (Part-B).

6.5.2 Part-B

We now focus our experimental study on the evaluation of the accuracy of our model in a more complex scenario characterized by multiple transaction profiles and highly skewed phase-dependent data access distributions. Further, compared to the study in the previous section, we consider system parameters representative of more modern platforms (e.g. an increased number of CPUs/Cores and increased processor/disk speed) and applications (e.g. an increased amount of items inside the transactional system). The detailed parameter settings adopted for this study are reported in Table 6.2. As the last preliminary consideration, this time the value of P_{BH} (which would depend on the specific object replacement policy) has been set to 0. (Recall that our analysis is orthogonal to modeling approaches for buffer replacement policies and related hit/miss effects vs the item popularity.)

In this study data items are grouped in 5 contiguous sets (logically equivalent to, e.g., database tables) which we again refer to as $\{S_1, \ldots, S_5\}$. Also, the probability of access in write mode is set equal to 20%. The workload (see Table 6.3) entails three different transaction profiles P_1, P_2 and P_3, with identical arrival rates, and the following access patterns. For class P_1, the pattern is similar to the phase-dependent pattern of Part-A of our study, with the only variation that the number of accesses is equal to 20, and 4 accesses per set are executed before moving to the subsequent set. Transactions of class P_2 perform 4 accesses to the set S_1 and then other 4 accesses to the set S_2 (for a total of 8 accessed items). Similarly, transactions of profile P_3 perform 4 accesses to the set S_4, and 4 subsequent accesses to the set S_5. In every transaction profile, the 4 accesses in each set are uniformly distributed over the whole items in that set. The results for this workload (see Figure 6.5) show a good matching between simulation and analytical values for all the three transaction classes. For this same workload, we also show (see Figure 6.6) a comparison between the lock utilization values for each of the 5 sets as predicted by both the simulation and the analytical model. Beyond confirming the matching between simulation and analytical results, these plots highlight an interesting feature of our model. Specifically, its ability to capture data contention dynamics with single data item granularity makes it capable to predict the performance effects due to the specific organization of the transactional logic (such as the order of the accesses to different data sets within different phases of a transaction). As an example, Figure 6.6 highlights that the accesses to the set of items S_1 represent the system bottleneck.

6.5.3 Part-C

We conclude this section by providing the validation results for a test where we used a workload profile reflecting relevant features of a standard benchmark for transactional systems, namely TPC-C [3]. The item tables' population and layout (see Table 6.4) have been configured by setting the number of warehouses (which represent an

Table Name	# Items	Table ID
WAREHOUSE	500	tb0
DISTRICT	1000	tb1
CUSTOMER	15000	tb2
STOCK	500000	tb3
ITEM	100000	tb4
ORDER	1000	tb5
NEW-ORDER	1000	tb6
ORDER-LINE	1000	tb7
HISTORY	1000	tb8

Table 6.4: TPC-C tables' population.

Phase	P_0 (47%)	P_1 (45%)	P_2 (4%)	P_3 (4%)
0	(R),tb0	(R),tb0	(R),tb2	(R),tb6
1	(R),tb1	(R),tb1	(R),tb5	(W),tb6
2	(W),tb1	(R),tb2	(R),tb7	(R),tb5
3	(R),tb2	(W),tb0		(W),tb5
4	(W),tb5	(W),tb1		(R),tb7
5	(W),tb6	(W),tb2		(W),tb7
6	(R),tb4	(W),tb8		(R),tb2
7	(R),tb3			(W),tb2
8	(W),tb3			
9	(W),tb6			

Table 6.5: Abstracted TPC-C transaction profiles (classes).

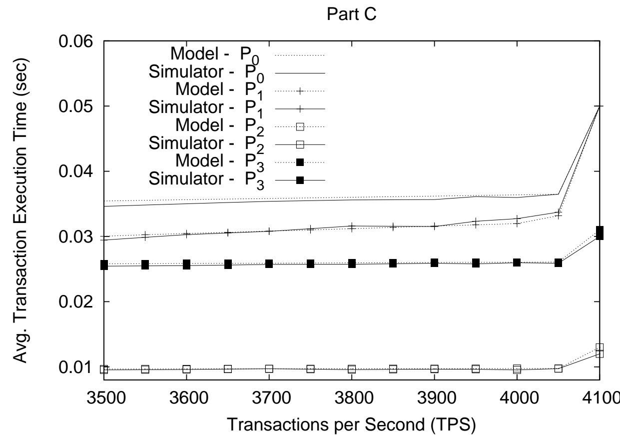


Figure 6.7: Simulation and analytical results for the abstracted TPC-C workload.

explicit scale parameter for TPC-C) to 500. The only variation is related to the scaling of the size of the tables which are accessed via select statements using intervals of keys. This choice is motivated by the fact that such select statements would lead to K read operations, as modeled in our approach. Therefore the scaling has been done in order to provide a fair modeling approach for select (i.e. read) statements operating at different granularity values (single key vs interval of keys).

The characterization of the transaction access patterns is based on the TPC-C workload modeling carried out in [82]. Table 6.5 reports, for each transaction profile and for each transaction execution phase, the accessed item table and the corresponding access mode (read, denoted as (r), or write, denoted as (w)). We consider only 4 of the 5 different transaction classes identified in [82], since one of them, namely the Stock-level transaction, does not impose any isolation guarantee, hence not triggering any concurrency control mechanism at all (whose modeling is the focus of this work). The remaining model parameters (characterizing, e.g., the available hardware resources) are not reported as they are unchanged with respect to Section 6.5.2.

By the results in Figure 6.7, we can observe that our model well fits the simulation output. As for previous cases, the matching can be observed for each single transaction profile included in the workload. These results confirm the high accuracy of our analytical performance model even in case of complex and diverse workloads.

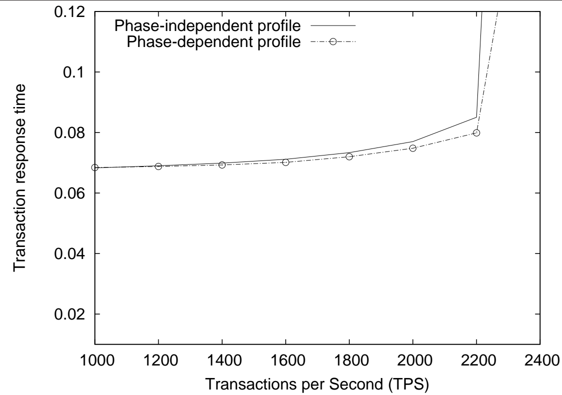


Figure 6.8: Performance comparison with phase-independent and phase-dependent profiles with the MVCC protocol.

6.6 Analysis of the Sensitivity to Data Access Patterns with Other Protocols

We conclude this chapter discussing the results of a simulation study we conducted to evaluate the sensitivity to transaction data access patterns of the other protocols we deal with in this dissertation, namely the MVCC and the CTL protocols. In addition, we also consider the results of another test we conducted with a protocol which can be considered, as concerns the locking mode, the counterpart of the CTL. This protocol uses the eager-locking. i.e., unlike the CTL protocol, a lock is acquired before to execute the write operation.

We start with the MVCC protocol. We used the same workload with the phase-independent transaction profile and the same workload with the phase-dependent transaction profile as in the test described in Section 6.2. The average transaction response time with respect to the transaction arrival rate for both workloads is shown in Figure 6.8. We can see that there is a very slight difference between the transaction response time of the two different profiles, and it becomes a little pronounced near the saturation point.

Now we consider the CTL protocol and the protocol which uses the eager-locking. In all tests we evaluated the throughput with respect to the number of concurrent threads.

In Figure 6.9 we show the results obtained by using the two workloads as in the tests for the MVCC protocol. The plot on the top side is related to CTL protocol. The results show as the CTL protocol appears insensitive to the data access patterns of the phase-dependent transaction profile. The plot on the bottom side is related to the protocol which uses the eager-locking. The results show as with eager-locking

the sensitivity grows.

To test the sensitivity of the CTL also in a more complex scenario, we used another workload profile with three transaction classes. In this workload the transactions of class P_1 have a profile equal to the transactions of class P_1 of the workload with the phase-independent transaction profile used in the former test. The transactions of class P_2 sequentially access the data items belonging to the sets S_1 and S_2. The transactions of class P_3 sequentially access the data items belonging to the sets S_4 and S_5. The throughput obtained with this workload has been compared with the throughput obtained with a workload with the same transaction classes, except that the accesses were independent of the transaction phases. The results are shown in Figure 6.10. On the top side we plotted the throughput for one transaction class [1] for both workloads. Furthermore, we also plotted the commit probability for all classes on the bottom side of the same figure. We can note that also in this test the CTL protocol appeared practically insensitive.

Summarizing the results of the tests we presented above, we can observe that the sensitivity to the transaction data access patterns mostly depends on the concurrency control strategy. Basically, it depends on the locks acquired by the transaction and by the time when they are acquired during the transaction execution. The sensitivity of the SS2PL protocol, which uses both exclusive and shared locks, and they are acquired before executing an operation, is remarkable. The sensitivity is less noticeable with the MVCC protocol. We recall that the MVCC protocol we considered uses exclusive locks as the SS2PL protocol, but never blocks a read operation. Therefore, read operations are not affected by the locks held by concurrent transactions. This aspect can explain the different sensitivity shown by this protocol with respect to the SS2PL, where, conversely, read operations are affected by exclusive locks acquired by the concurrent transactions. Finally, the CTL, which is the most optimistic protocol we considered in our analysis, appeared to be insensitive. We recall that, as the MVCC protocol, the CTL also uses exclusive locks, but they are acquired only at the commit phase. For these reasons, the lock holding time is almost the same for all locks acquired by a transaction, independently from the acquisition order. This aspect can explain the insensitivity of the CTL.

In conclusion, the results suggest as the modeling methodology we proposed in this chapter could also be usefull to improve the accuracy of peformance models for other lock-based protocols. On the other hand, the improvement maily depends on the locking stategy established by the protocol.

[1] The throughput is equal for all classes because transactions are generated with the same probability for all classes.

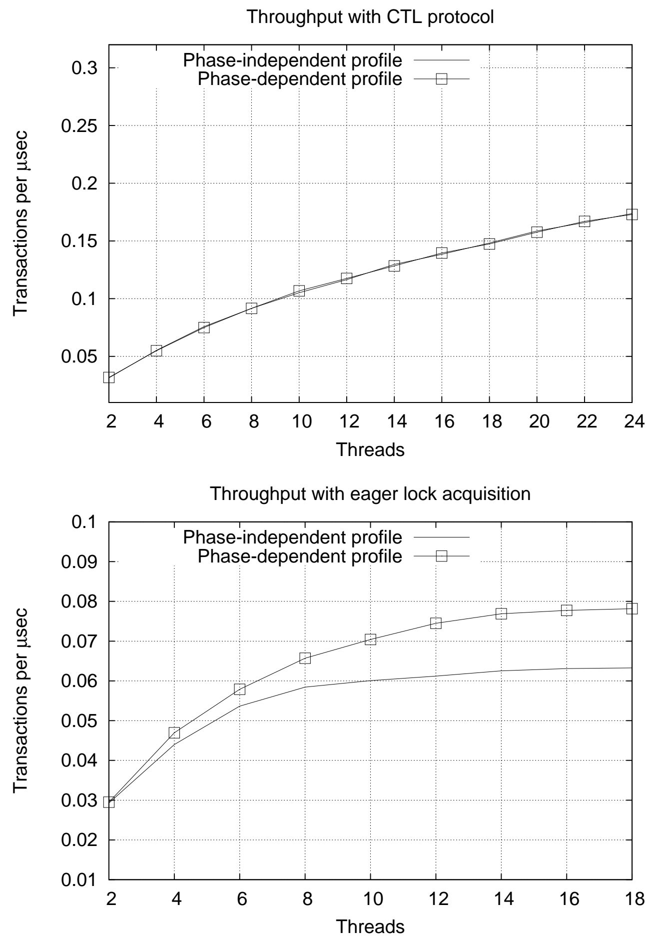


Figure 6.9: Performance comparison with phase-independent and phase-dependent transaction profiles.

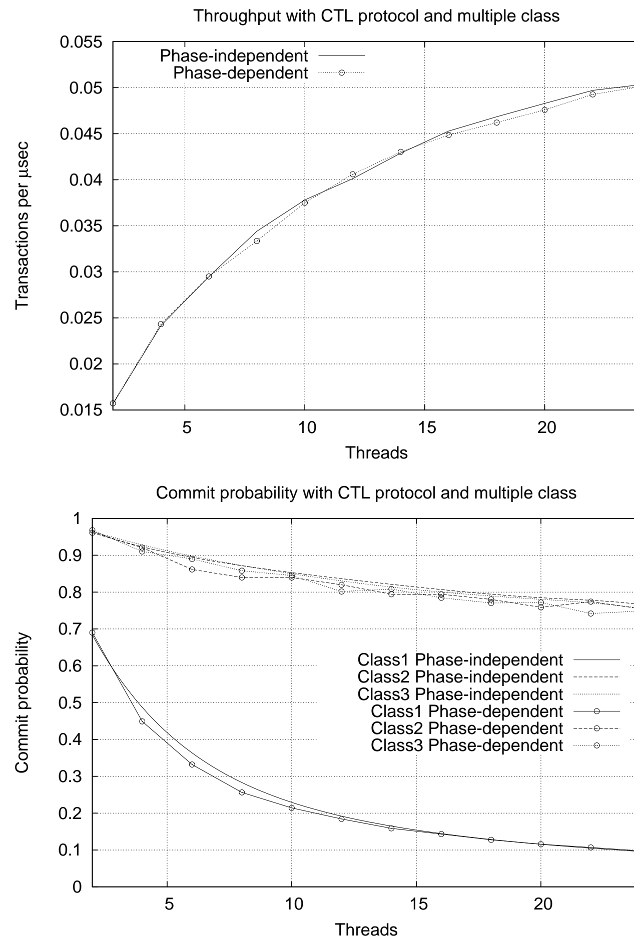


Figure 6.10: Performance comparison with phase-independent and phase-dependent transaction profiles with multiple classes

Chapter 7

Conclusions

Analytical modeling is an effective approach for building computer system performance models. It enables to describe such systems in a quantitative manner and let us to build valuable tools to analyze and understand complex dynamics characterizing them. In this dissertation we dealt with the analytical perfomance modeling of CCPs for transaction processing systems. We focused on two kinds of systems in which the concurrency control plays a key role, namely the DBS and STMs. With our work we contributed to both the development of new models of CCPs and the development of new modeling approaches. We addressed the performance modeling of the MVCC by developing the first analytical model of one of the most used MVCC protocol in DBS. At the base of this model there is an approach which focuses on a model of transaction execution which allows to capture the transaction execution dynamics due to the mix of mechanisms used by the protocol. In the field of STMs we proposed a new analytical framework for building performance models of STM systems. This framework overcomes previous proposals as it allows to conduct a more comprehensive performance study, providing the ability to evaluate various performance indicators, including the analysis of both the transaction execution dynamics and the execution of concurrent threads. This is made possible by the two-layered structure of the framework, which also symplifies the development of models for different CCPs. Leveraging on this framework, we also built a performance model for the case of the CTL protocol, currently used by many STMs. Finally, we proposed a new modeling approach which allows us to analyze the performance of CCPs by a new perspective. We showed as CCPs have different sensitivity to the sequences of data items accessed by transactions. In particular, a largely used class of locking protocols have an high sensitive. This is an aspect which has not been considered in previous performance modeling works, and we showed that analytical models which do not capture the effects due to such data access patterns can provide unreliable

results when used for the performance analysis of applications. To cope with this problem, we proposed a new modeling approach allowing us to capture the effect on the system performance due to arbitrary transaction data access patterns, including the cases with multiple transactions profiles.

In all models we proposed, we used a modeling approach which allowed us to abstract from the specific implementation details of the protocols. Indeed, models we presented focus on that aspects that can have meaningful implications on the system performance by the perspective of the concurrency control. At the same time, with our approach, parameters depending on other factors, e.g. the operation processing time or the buffer hit probability, can be taken into account by both considering them as input to the models and coupling the models with performance models of other system components. This approach provides flexibility, in particular when we want to use a model for the performance analysis with different implementations of a protocol, or when a protocol is used in different systems. In addition, it provides high modularity, allowing us to use a model for building larger system performance models by means of composition of models. Beyond these motivations, we think that the previous aspects are also very important because the field of application of the CCPs is not limited to the systems we considered in this dissertation.

Interesting improvements and extensions of the work we presented are the following.

Concerning the model of the MVCC protocol, we discussed in Section 4.6 the loss of accuracy of the model when the data access skew increases. This is due to the modeling assumption according to which the actual data access distribution is considered the same of data accesses of arriving transactions. When the data access skew becames very high, the actual mix of data accesses is warped by the restarts of transactions accessing many highly popular data items. We think that a possible solution to be evaluated relies on representing workloads with high data access skew by means of multiple transaction classes, so that the data access skew becames lower within each transaction class. Then, an approach based on transaction clustering, as we discussed in Section 4.4.3, could be used.

The STM performance modeling framework we proposed provides an easy way to build CCPs models, thereby also being an useful tool to conduct performance comparison studies of STM protocols. Hence, a natural extension of this work is the development of further models of such protocols. An interesting improvements concerns the thread-level model according the observations in Section 5.6, where we discussed the removal of some assumptions used to build the CTMC.

Finally, as regards the modeling approach for arbitrary transaction data access patterns, it would be interesting to evaluate this approach also for protocols which use mixed concurrency control mechanisms. In the simulation study we presented in 6.6 we showed that also the MVCC protocol we considered in this dissertation and the protocol which uses the eager-locking are sensitive to the data access sequences. As both these protocols use locking and read validation, they could be a

further benchmark for this approach.

We conclude observing that in our work we addressed two main issues we consider very important for the current state of art in the field CCPs performance modeling. The first one is associated to the advancement of the state of art in the field of the concurrency control. Over the time, new CCPs have been proposed in order to fit even more both the performance requirements of applications and the transaction processing requirements of new systems. Today, many systems use protocols based on more complex strategies than those used in the past. Despite the effort made in the field of performance modeling, existing studies do not always provide performance analysis tools suitable for new protocols and systems. The second one is associated to the system models tipically used in CCPs performance modeling studies. Indeed, they use system models which adequately represent quite generic scenarios, and do not take into account the implications related to some basic features of applications which can remarkably affect the system performance. We dealt with the first aforesaid issue in our modeling studies of both the MVCC protocol and the STMs. In the case of the MVCC protocol, we considered a widely used protocol in modern DBMS which is based on a mix of concurrency control mechanisms which improves the performance in read-intensive scenarios. In the case of the STMs, we considered a lock-based protocol which optimistically acquires locks, but it also uses data validation in order to provide an isolation level which well fits the transaction processing requirements of the new STM systems. We dealt with the second aforesaid issue in the modeling studies of both the STMs and the transaction data access patterns. In these studies we made further steps towards a more comprehensive performance analysis of real applications. In the STM modeling framework, we included in the analysis also the effects due to the variation of the concurrency level in the system together with the mix of transactions with different profiles. Finally, in the last contribution, we extended the analysis by including the effects due to a tipical feature of the data access patterns of applications. All these factors provide the ability to extend the field of application of models including the analysis of more realistic scenarios. We think that the lack of approaches which allow us to perform an analysis by a more realistic, application-oriented, perspective is one of the weaknesses which have to be further addressed in the field of analytical performance modeling of CCPs. For these reasons it represents an important direction for the future work.

Bibliography

[1] N. Shavit and D. Touitou. Software transactional memory. In *Proc. of the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 204–213, 1995.

[2] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[3] Transaction Processing Performance Council. Tpc benchmark^{TM} c.

[4] Michael F. Spear, Maged M. Michael, Michael L. Scott, and Peng Wu. Reducing memory ordering overheads in software transactional memory. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 13–24, Washington, DC, USA, 2009. IEEE Computer Society.

[5] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[6] Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency control performance modeling: alternatives and implications. *ACM Trans. Database Syst.*, 12:609–654, November 1987.

[7] Kishor Trivedi, Boudewijn Haverkort, Andy Rindos, and Varsha Mainkar. Techniques and tools for reliability and performance evaluation: Problems and perspectives. In Gunter Haring and Gabriele Kotsis, editors, *Computer Performance Evaluation Modelling Techniques and Tools*, volume 794 of *Lecture Notes in Computer Science*, pages 1–24. Springer Berlin / Heidelberg, 1994.

[8] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Software performance modeling: state of the art and perspectives. Technical report, Dipartimento di Informatica, Università Ca' Foscari di Venezia and MIUR Sahara Project TR SAH/001, 2003.

[9] Philip S. Yu, Daniel M. Dias, and Stephen S. Lavenberg. On the analytical modeling of database concurrency control. *J. ACM*, 40:831–872, September 1993.

[10] Pascal Felber, Christof Fetzer, Rachid Guerraoui, and Tim Harris. Transactions are back—but are they the same? *SIGACT News*, 39:48–58, March 2008.

[11] Chi C. Minh, Jaewoong Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *IISWC '08: Proceedings of the IEEE International Symposium on Workload Characterization*, pages 35–46, Seattle, WA, USA, 2008.

[12] O. Shalev D. Dice and N. Shavit. Transactional locking ii. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.

[13] Transaction Processing Performance Council. Tpc benchmark^{TM} w.

[14] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010.

[15] Ansi x3.135-1992, american national standard for information systems - database language - sql, 1992.

[16] Oracle Database Web Site: http://www.oracle.com/us/products/database/index.html.

[17] Yoav Raz. The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource mangers using atomic commitment. In Li-Yan Yuan, editor, *18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings*, pages 292–312. Morgan Kaufmann, 1992.

[18] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick 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.

[19] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30:492–528, June 2005.

[20] Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan. Automating the detection of snapshot isolation anomalies. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 1263–1274. VLDB Endowment, 2007.

[21] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[22] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21:289–300, May 1993.

[23] Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha. Unlocking concurrency. *Queue*, 4:24–33, December 2006.

[24] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 237–246, New York, NY, USA, 2008. ACM.

[25] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 253–262, New York, NY, USA, 2006. ACM.

[26] Microsoft STM.NET Web site: http://msdn.microsoft.com/en-us/devlabs.

[27] Roberto Palmieri, Francesco Quaglia, Paolo Romano, and Nuno Carvalho. Evaluating database-oriented replication schemes in software transactional memory systems. In *IPDPS Workshops*, pages 1–8. IEEE, 2010.

[28] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM.

[29] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM.

[30] Michael F. Spear, Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Conflict detection and validation strategies for software transactional memory. In *Proceedings of the 20th Intl. Symp. on Distributed Computing (DISC)*, pages 179–193, 2006.

[31] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. *SIGPLAN Not.*, 44:155–165, June 2009.

[32] Maurice Herlihy, Victor Luchangco, and Mark Moir. Software transactional memory for dynamic-sized data structures. pages 92–101. ACM Press, 2003.

[33] João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63:172–185, December 2006.

[34] Maurice Herlihy. Obstruction-free synchronization: Double-ended queues as an example. In *In Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 522–529. IEEE Computer Society, 2003.

[35] Keir Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, February 2004.

[36] Robert Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006.

[37] Daniel R. Ries and Michael R. Stonebraker. Locking granularity revisited. *ACM Trans. Database Syst.*, 4:210–227, June 1979.

[38] Daniel R. Ries and Michael Stonebraker. Effects of locking granularity in a database management system. *ACM Trans. Database Syst.*, 2:233–246, September 1977.

[39] Alexander Thomasian and In Kyung Ryu. A decomposition solution to the queueing network model of the centralized dbms with static locking. In *Proceedings of the 1983 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '83, pages 82–92, New York, NY, USA, 1983. ACM.

[40] Y. C. Tay, R. Suri, and N. Goodman. A mean value performance model for locking in databases: the waiting case. In *Proceedings of the 3rd ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '84, pages 311–322, New York, NY, USA, 1984. ACM.

[41] Wen-Te K. Lin and Jerry Nolte. Communication delay and two phase locking. In *Proceedings of the 3rd International Conference on Distributed Computing Systems (ICDCS)*, pages 502–507, 1982.

[42] Y. C. Tay, Nathan Goodman, and R. Suri. Locking performance in centralized databases. *ACM Trans. Database Syst.*, 10:415–462, December 1985.

[43] Rakesh Agrawal and David J. Dewitt. Integrated concurrency control and recovery mechanisms: design and performance evaluation. *ACM Trans. Database Syst.*, 10:529–564, December 1985.

[44] Peter Franaszek and John T. Robinson. Limitations of concurrency in transaction processing. *ACM Trans. Database Syst.*, 10:1–28, March 1985.

[45] Michael J. Carey and Michael Stonebraker. The performance of concurrency control algorithms for database management systems. In *Proceedings of the 10th International Conference on Very Large Data Bases*, VLDB '84, pages 107–118, San Francisco, CA, USA, 1984. Morgan Kaufmann Publishers Inc.

[46] N B Al-Jumah, H S Hassanein, and M El-Sharkawi. Implementation and modeling of two-phase locking concurrency control - a performance study. *Information Software Technology*, 42(4):257–273, 2000.

[47] Alexander Thomasian. Concurrency control: methods, performance, and analysis. *ACM Comput. Surv.*, 30:70–119, March 1998.

[48] Alexander Thomasian. Performance evaluation of centralized databases with static locking. *IEEE Trans. Softw. Eng.*, 11:346–355, April 1985.

[49] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[50] Alexander Thomasian and In Kyung Ryu. A recursive solution method to analyze the performance of static locking systems. *IEEE Trans. Softw. Eng.*, 15:1147–1156, October 1989.

[51] Alexander Thomasian. Checkpointing for optimistic concurrency control methods. *IEEE Trans. on Knowl. and Data Eng.*, 7:332–339, April 1995.

[52] In Kyung Ryu and Alexander Thomasian. Analysis of database performance with dynamic locking. *J. ACM*, 37:491–523, July 1990.

[53] Alexander Thomasian and In Kyung Ryu. Performance analysis of two-phase locking. *IEEE Trans. Softw. Eng.*, 17:386–402, May 1991.

[54] Michael J. Carey and Waleed A. Muhanna. The performance of multiversion concurrency control algorithms. *ACM Trans. Comput. Syst.*, 4:338–378, September 1986.

[55] Arif Merchant, Kun lung Wu, Philip S. Yu, and Ming syan Chen. Performance analysis of dynamic finite versioning for concurrent transaction and query processing. In *IEEE Transactions on Knowledge and Data Engineering*, pages 103–114, 1992.

[56] Arif Merchant, Kun-Lung Wu, Philip S. Yu, and Ming-Syan Chen. Performance analysis of dynamic finite versioning schemes: Storage cost vs. obsolescence. *IEEE Trans. on Knowl. and Data Eng.*, 8:985–1001, December 1996.

[57] Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Adaptive software transactional memory. In *Proceedings of the 19th International Symposium on Distributed Computing*, Cracow, Poland, Sep 2005. Earlier but expanded version available as TR 868, University of Rochester Computer Science Dept., May2005.

[58] Armin Heindl and Gilles Pokam. Modeling software transactional memory with anylogic. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, Simutools '09, pages 10:1–10:10, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[59] Armin Heindl and Gilles Pokam. An analytic framework for performance modeling of software transactional memory. *Comput. Netw.*, 53:1202–1214, June 2009.

[60] Armin Heindl and Gilles Pokam. An analytic model for optimistic stm with lazy locking. In *Proceedings of the 16th International Conference on Analytical and Stochastic Modeling Techniques and Applications*, ASMTA '09, pages 339–353, Berlin, Heidelberg, 2009. Springer-Verlag.

[61] Kishor S. Trivedi. *Probability and statistics with reliability, queuing and computer science applications*. John Wiley and Sons Ltd., Chichester, UK, 2nd edition edition, 2002.

[62] Zhengyu He and Bo Hong. On the performance of commit-time-locking based software transactional memory. In *Proceedings of the 2009 11th IEEE International Conference on High Performance Computing and Communications*, pages 180–187, Washington, DC, USA, 2009. IEEE Computer Society.

[63] Zhengyu He and Bo Hong. Modeling the run-time behavior of transactional memory. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, pages 307 –315, aug. 2010.

[64] Xiao Yu, Zhengyu He, and Bo Hong. An analytical model on the execution of transactional memory. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2010 22nd International Symposium on*, pages 175 –182, oct. 2010.

[65] JBoss Cache Web Site: http://www.jboss.org/jbosscache.

[66] PostgreSQL Web Site: http://www.postgresql.org/.

[67] J.N. Gray, P. Homan, R.L. Obermarck, and H.F. Korth. *A straw man analysis of probability of waiting and deadlock*. Research report // IBM Research Division. IBM Corp., 1981.

[68] The PostgreSQL Global Development Group. Postgresql 8.2.6 documentation.

[69] Leonard Kleinrock. *Queueing Systems*, volume I: Theory. Wiley Interscience, 1975. (Published in Russian, 1979. Published in Japanese, 1979. Published in Hungarian, 1979. Published in Italian 1992.).

[70] Bruno Ciciani, Daniel M. Dias, and Philip S. Yu. Analysis of concurrency-coherency control protocols for distributed transaction processing systems with regional locality. *IEEE Trans. Softw. Eng.*, 18:899–914, October 1992.

[71] Michael J. Carey and Michael Stonebraker. The performance of concurrency control algorithms for database management systems. In *Proceedings of the 10th International Conference on Very Large Data Bases*, VLDB '84, pages 107–118, San Francisco, CA, USA, 1984. Morgan Kaufmann Publishers Inc.

[72] Ming-Syan Chen, Philip S. Yu, and Tao-Heng Yang. On coupling multiple systems with a global buffer. *IEEE Trans. on Knowl. and Data Eng.*, 8:339–344, April 1996.

[73] M. Arlitt and T. Jin. A workload characterization study of the 1998 world cup web site. *Network, IEEE*, 14(3):30–37, 2000.

[74] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenke. Web caching and Zipf-like distributions: evidence and implications. In *INFOCOM '99: Proceedings of the Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 126–134. IEEE, March 1999.

[75] R. W. Wolff. Poisson arrivals see time averages. *Operations Research*, 30:223–231, 1982.

[76] A. A. Tomusyak. Computation of ergodic distribution of markov and semi-markov processes. *Cybernetics and Systems Analysis*, 5:80–84, 1969. 10.1007/BF01070666.

[77] R. Fricks, A. Puliafito, M. Telek, and K. S. Trivedi. Markov renewal theory applied to performability evaluation. *Modeling and Simulation of Advanced Computer Systems: Applications and Systems*, pages 193–236, 1996.

[78] Vidyadhar G. Kulkarni. *Modeling and analysis of stochastic systems*. Chapman & Hall, Ltd., London, UK, UK, 1995.

[79] Asit Dan. *Performance analysis of data sharing environments*. MIT Press, Cambridge, MA, USA, 1992.

[80] Bruno Ciciani, Francesco Calderoni, Andrea Santoro, and Francesco Quaglia. Modeling of qos-oriented content delivery networks. In *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 341–344, Washington, DC, USA, 2005. IEEE Computer Society.

[81] Bruno Ciciani, Daniel M. Dias, and Philip S. Yu. Dynamic and static load sharing in hybrid distributed-centralized database system. *Comput. Syst. Sci. Eng.*, 7:25–41, January 1992.

[82] Scott T. Leutenegger and Daniel Dias. A modeling study of the tpc-c benchmark. *SIGMOD Rec.*, 22:22–31, June 1993.

Acronyms

2PL Two-phase locking

DBMS Database Management Systems

CCP Concurrency Control Protocol

CTL Commit-Time Locking

DBS Database Systems

MVCC MultiVersion Concurrency Control

SS2PL Strong-Strict 2PL

STM Software Transactional Memory