2008

Self-adjusting multi-granularity locking protocol for object-oriented databases

Deepa Saha
University of Windsor

SELF-ADJUSTING MULTI-GRANULARITY LOCKING PROTOCOL FOR OBJECT-ORIENTED DATABASES

by

Deepa Saha

A Thesis
Submitted to the Faculty of Graduate Studies
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada

2008

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-87697-8

Our file Notre référence
ISBN: 978-0-494-87697-8

AUTHOR'S DECLARATION OF ORIGINALITY

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

ABSTRACT

Object-oriented databases have the potential to be used for data-intensive, multi-user applications that are not well served by traditional applications. Despite the fact that there has been extensive research done for relational databases in the area of concurrency control; many of the approaches are not suitable for the complex data model of object-oriented databases. This thesis presents a self-adjusting multi-granularity locking protocol (SAML) which facilitates choosing an appropriate locking granule according to the requirements of the transactions and encompasses less overhead and provides better concurrency compared to some of the existing protocols. Though there has been another adaptive multi-granularity protocol called AMGL [1] which provides the same degree of concurrency as SAML: SAML has been proven to have significantly reduced the number of locks and hence the locking overhead compared to AMGL. Experimental results show that SAML performs the best when the workload is high in the system and transactions are long-lived.

DEDICATION

I dedicate this thesis to my friends and family members for their unconditional love and support which has inspired me to pursue my goals.

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my sincere gratitude to my advisor Dr. J. Morrissey for her guidance and support throughout the course of research without which it would not have been possible to complete this work successfully.

I would like to extend my thanks to my committee members for taking the time to review my work. Their constructive criticism always helped the work to be better. I would like to thank Dr. D. Wu for being the chair in the examination committee.

My sincere gratitude goes to my fabulous sister Beeta and my friends for their support, criticism and suggestions. Special thanks to my friend Debashis Roy for his technical assistance and suggestions with the research work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Numerous large, multi-user, advanced computer applications, like software development, computer aided design (CAD) and manufacturing (CAM), network management, knowledge-based systems, financial forecast, medical informatics, complex scientific and business applications and multi-media information systems involve computations on a large amount of objects with a complex data model. Object-oriented databases (OODBs) can deal with these applications more efficiently while other types of databases might fail.

There are certain benefits that these applications can derive by using an object-oriented database. OODBs handle complex data structures in a way which is similar to that of an object-oriented programming language and hence provides lesser developmental cost and better performance through better integration to the application tier. This in turn reduces both development time and maintenance costs. The main objective of object-oriented database systems is to provide consistent, data independent, secure, controlled and extensible data management services to support the object-oriented model. Storing Java or .NET objects 'just as they are in memory' is the best way to implement a persistence solution. The schema for an object database takes form naturally as per the application objects' persistence. By using an object-oriented database instead on an RDBMS, the overhead of object-relational mapping can be avoided which otherwise would have resulted in an increased demand on resources. Object databases' true zero-administration nature as well as a small footprint, removal of the need for OR mapping tools makes it a good technology for these kinds of applications. Complex cross referencing among objects can be difficult and error-prone to model in a relational database system. Relationships among objects are often dealt with using foreign keys in RDBMSs. So, fetching an object and then fetching objects it references, and then the objects they reference; can result in complicated and difficult-to-maintain code.

Most ODBMSs implement reachability persistence. That means that any object referenced by a persistent object is also persistent. This means a whole lot of objects can

be stored or fetched with a single call. The ODBMS engine handles the details of maintaining the references when objects are stored, and fulfills them when objects are fetched. Highly-connected object structures are not easily translated to "fit" into a relational database and the conversion is often confusing and difficult to maintain. On the other hand, an ODBMS requires no translation of the original structure into a model for the database. If the ODBMS provides programmer control over the depth of reachability persistence, the developer can control whether a whole tree is fetched or stored, a branch is fetched or stored, or individual twigs are fetched and stored. And, again, the integrity of the structure is preserved by the database engine itself. In many cases class structures of an application changes over time, new data members are added or new object relationships needs to be included. As most applications evolve as they age so should the data structures they support. An ODBMS has the capability to adapt to data structure changes more easily than a RDBMS. If a RDBMS is used, the schema might have to be changed to fit the new object structure and then the query code will have to be altered to handle the changes. Some ODBMSs allow the change of the structure of objects "on the fly" and "old" and "new" objects can come together in the same database. If the new object structure has additional fields, reading an old object into the new application simply loads the additional fields with default (i.e., null or zero) values. If the new object structure has fewer fields, reading an old object into the new application skips the now non-existent fields. If ODBMs are used instead of RDBMs then there would be no need of writing translation code to pass data back and forth between row objects fetched from the database and actual objects in an application. This is an important consideration if multiple applications that access the same database have to be maintained but in somewhat different ways. In such a situation, all of the translation codes for all of the different applications have to be synchronized.

While programming using Object-oriented languages, if an ODBMS is used, the access is the same for all applications as the objects being fetched and stored are being manipulated in the same way. Again if the application is factored properly then in a change in the class structure means only a change to a single library. They were created to handle large and complex data that relational databases could not and hence it is believed that they have the potential to be applied widely.

In recent years, there has been much interest in object-oriented databases for advanced database systems and because of that; the performance of object-oriented databases has become a significant issue. Past researchers mainly focused on single-user performance issues but in a multiple user and applications environment, their performance issues have not been studied thoroughly. However, to be accepted widely, object-oriented databases will have to have a good performance in an environment with multiple users and/or multiple applications. The problems of concurrency control in a relational database, for example: the lost update problem and the uncommitted dependency problem, remain in object-oriented database. In addition to that, the complexity of the object-oriented data model and the object-oriented database paradigm introduces some unique challenges in concurrency control when multiple applications use the same object-oriented database.

In order to control the concurrent accesses from multiple transactions, the object-oriented database system uses a variety of concurrency control techniques. Many of the current object-oriented databases use concurrency control techniques that are borrowed from relational database systems. They were not modified to match the performance requirements of an object-oriented application as they are quite different from the relational database applications. Depending on the implementation of the concurrency control protocol, there could be considerable performance differences among object-oriented systems. One of the main techniques used for concurrency control is based on the concept of locks. In the case of object-oriented applications' concurrency control, they could involve a huge number of objects and thus require significant overhead for the locks used for each object. These applications might operate on these objects for a long time and be interactive in nature and so the transactions involved are usually long-lived. Holding locks for a long time is problematic for object-oriented databases as most of them use a navigational mode of access. Severe performance degradation can occur if locking a wrong granularity is chosen. Some granularity protocols as well as a multi-granularity protocol and an adaptive multi-granularity protocol are available for improving performance and reducing the overhead incurred but they all posses some weaknesses. This thesis proposes a self-adjusting multi-granularity protocol called SAML to improve the degree of concurrency of object-oriented databases

that can be achieved while alleviating certain locking overhead issues that is often encountered.

The rest of the thesis is organized as follows. CHAPTER II briefly discusses some basic concepts and provides a literature review of related work done in the field of database concurrency control protocols. CHAPTER III contains a detailed description of the design and development of the proposed SAML protocol. The major components such as the granularity units, lock request protocols, lock compatibility matrix and lock de-escalation are explained clearly. CHAPTER IV deals with the simulation of the proposed protocol. The simulation methodology and models are illustrated and CHAPTER V presents the results of experimental studies using different parameters. The entire thesis work has been summarized in CHAPTER VI along with some suggestions and directions for future research in this area.

CHAPTER II

BACKGROUND AND LITERATURE REVIEW

This chapter focuses on object-oriented databases and reviews some basic concepts of these databases. A brief review on the work that has been done in the field of concurrency control of relational databases as well as object-oriented databases is presented. The concept of granularity locking used for concurrency control of object-oriented database is addressed elaborately.

2.1. Object-Oriented Database

"An object database management system (ODBMS, also referred to as object-oriented database management system or OODBMS), is a database management system (DBMS) that supports the modelling and creation of data as objects" [2].

According to Won Kim [3], the lead developer of one of the earliest commercial object-oriented databases (ORION), an object-oriented database is "a database system which directly supports an object-oriented data model". A data model is a logical organization of real world entities or objects, their relationship and constraints on them and a data model capable of capturing the object oriented concept is an object oriented data model. An object oriented database is a collection of objects and these objects' behaviour and relationship is defined according to object oriented concepts. An object in an object database is comparable to an object in application memory. In most object databases, there are language bindings that allow the usage of persistent objects in applications. The database schema itself is created using an object definition language, which defines the object classes that can be stored and their relationships. The main objective of OODBMS is to provide consistent, data independent, secure, controlled and extensible data management services to support the object-oriented model. They were created to handle voluminous and complex data that relational databases could not. Object oriented concepts facilitate a rich data model for the next- generation database applications like CAD/CAE/CASE/CAM systems, knowledge-based systems and multi-media information systems.

According to ODMG2.0 (The Object Data Management Group), an object database management system is "a database management system that integrates database system capabilities with object-oriented programming language capabilities" The Object Oriented Database Manifesto [4] specifically lists the following features as mandatory for a system to support before it can be called an OODBMS; complex objects, object identity, encapsulation, types and classes, class or type hierarchies, overriding, overloading and late binding, computational completeness, extensibility, persistence , secondary storage management, concurrency, recovery and an ad hoc query facility.

2.2. A Brief History

During the early to mid-1970s, object-oriented database management systems were used as a database support for graph-structured objects. In the late 1980s object databases started to evolve, but they acquired wide usage only in a few markets; namely telecom, scientific and financial applications. Some outstanding research projects of that time included Encore-Ob/Server, EXODUS, IRIS, ODE, ORION, Vodak and Zeitgeist. The ORION project has more published papers than any of their contemporary projects. Some of the early commercial products incorporate Gemstone, Gbase, and Vbase. In early to mid-1990s additional commercial products like ITASCA, Jasmine, Matisse, Objectivity/DB, ObjectStore, ONTOS, O_2, POET, and Versant Object Database came into existence. Some of these products still remain on the market. These early commercial OODBs were integrated with various languages: GemStone used Smalltalk, Gbase used LISP, whereas Vbase used COP. Most of 1990s, C++ was used widely in the commercial object database management market. Vendors started to use Java in the late 1990s and more recently, C# (C# is a multi-paradigm programming language that encompasses functional, imperative, generic and object-oriented programming disciplines; It is developed by Microsoft as part of the .NET initiative). 2004 onwards object databases have experienced an exponential growth period when open source object databases emerged. OODBs have benefited from the popularity of object oriented languages like Java and C#. These databases were commonly affordable and user-friendly as they were entirely written in OOP languages like Java or C#. Db4o

(db4objects) and Perst (McObject) are two of these databases. In recent times another open source object database Magma (written in Squeak) has been in development.

Object databases have been accepted as a solution for the object-relational (OR) impedance mismatch: a set of conceptual and technical difficulties which are often encountered when a relational database management system is being used by a program written in an object-oriented programming language or style; particularly when objects and/or class definitions are mapped in a straightforward way to database tables and/or relational schema. This is undoubtedly the biggest problem faced in the era of object-oriented programming. The wide use of OOP languages like Java and .NET has made the dilemma even more crucial. As a result of this, object databases are getting recognition as a complement but not replacement for solving OR mismatch effectively.

Besides applications areas like engineering and spatial databases, as well as scientific research areas like high energy physics, molecular biology, object databases are being used in embedded systems as well as real-time systems. They are providing embeddable persistence solution in devices, on clients, in packaged software, in real-time control systems and to power websites. Now, they are typically being used as an embedded database but most vendors do have a network version available and clients can access a running server via TCP/IP.

2.3. Object-Oriented Database Definitions

The data models of relational databases and object oriented database are considerably different and by databases, people often mean the relational database. But the increasing use of object-oriented languages has made people look for an alternative. So it is vital to understand the main concepts of OODBMS and how they have emerged from the object data model.

- Object: Real world entities that are of some meaning are referred to as Objects. An object consists of a set of values for its attributes. The values of the attributes of an object constitute the state of the object. The domain of an attribute is the class to which the values of the attribute belong; the domain of an attribute may be any class

including a primitive class, for example, integer, string, etc. Again, an attribute may have a single value or a set of values from its domain. So these values can also be objects, thus forming relationships among objects.

- Object Identifier: This is a system-wide unique identifier that an object is associated with. This acts as a reference to the object and is used to indicate other objects to which it is related. Thus, an object identifier makes a convenient way of navigating through a complex network of objects. As object identifiers are the only means of accessing objects, most of the existing object-oriented applications use navigational model of computation. Nevertheless, there are also other declarative methods of manipulating data in object-oriented databases.

- Class: All objects that share the same set of attributes belong to the same class. An object belongs to a class and is an instance of that class. A class consists of attributes as well as methods, which define the behaviour of a class. The concept of class directly captures the concept of instance-of relationship between an instance and the class to which it belongs. In most systems, an instance belongs to one class only. But some object-oriented databases use the concept of multiple-inheritance which makes it possible for an instance to logically belong to more than one class.

- Class Inheritance: In any object-oriented system, a class may have any number of subclasses. However, some systems allow a class to have only one superclass and in this case, the class inherits attributes and methods from only one class. This phenomenon is known as single inheritance. On the contrary, some systems allow a class to have any number of super classes and the class inherits attributes and methods from more than one superclass; this is called multiple-inheritance. Together, the classes in a system form a hierarchy called the class hierarchy. It captures the generalization relationship between a class and its direct and indirect subclasses and captures the IS-A relationship between a class and its superclass/es.

2.4. Concurrency Control

A transaction is a unit of a program execution that accesses and possibly updates various data items [5]. These updates of data in a database have to be done in such a way so that concurrent executions and failures of various forms do not result in an inconsistency in the database. Transactions are required to have the ACID properties: atomicity, consistency, isolation and durability. Atomicity ensures that either all of the effects of a transaction are reflected in the database or none of them are. Consistency ensures the same consistent state of a database is maintained before and after a transaction has been executed. Isolation ensures that concurrently executing transactions are isolated from each other in a way that each of them gets the impression that no other transaction is executing concurrently with it. Durability ensures that even on an event of a system failure, the updates of a committed transaction is not lost.

Though isolation is one of the fundamental properties of a transaction, it may not be preserved any longer while several transactions are being executed concurrently. In order to maintain the isolation property, the system has to control the interaction among the concurrent transactions. Concurrency control protocols are used to attain this control.

2.4.1. Concurrency Control in Relational Databases

When multiple transactions are trying to access the data, conflicts among the data accesses is unavoidable and it is the concurrency control mechanism's responsibility to resolve these conflicts. If isolation property of a database cannot be protected, then the consistency of a database cannot be maintained either. In the 1970's researchers first worked on the concurrency control mechanisms and since then a number of variations and improvements have been done on them ever since. All of these mechanisms use the concept of serializability to guarantee the consistency of a database, i.e. all these mechanisms make sure that the schedule is serial. According to the concept of serializability, any schedule that is produced by the concurrent processing of a set of transactions to be an effect equivalent to a schedule produced when these transactions are run serially in some order. A few of these protocols are lock-based protocols, timestamp-based protocols and validation based protocols.

In lock-based protocols, serializability is maintained by letting the data items to be accessed in a mutually exclusive manner. Data items can only be accessed if the transaction puts a lock on that item. *A Lock is a database system object associated with a database object that prevents undesired operations of other transactions by blocking them.* There are two basic modes of locking: shared and exclusive. If a transaction obtains a shared lock on a data item, then it can read the data item but cannot write. On the contrary, if a transaction has obtained an exclusive lock on a data item then it can both read and write the item. Every transaction requests for an appropriate mode of lock on a data item, depending on the types of operations that it wants to perform on that data. The protocol follows a set of rules indicating when a transaction may lock and unlock the data items. Deadlock is very common in this protocol. Deadlock is a state when two or more transactions are waiting for the other to release a resource, or more than two processes are waiting for resources in a circular chain. Locks are the most common mechanism for maintaining consistency is a database system [4]. One commonly used protocol is called two-phase locking. Two-phase locking has two phases, namely the growing phase and a shrinking phase. In the growing phase locks are accumulated which is followed by shrinking phase where locks are released. So a lock request is always followed by a lock release. If for any reason, this rule is violated during the lifetime of a transaction, then that transaction has to be aborted. Two-phase locking also has the problem of encountering deadlocks. One modification of two-phase locking is called strict two-phase locking. In this protocol the locking is done in two phases and all exclusive mode locks obtained by a transaction be held until that transaction commits. Another variety of two-phase locking is called rigorous two phase locking. It requires all locks to be held until the transaction commits. Another modification of two-phase locking is conservative two phase locking where transactions obtain all the locks they need before the transactions begin. This is to ensure that a transaction that already holds some locks will not block other transactions' waiting for other locks. Conservative two-phase locking can prevent deadlocks. Two-phase locking is known as a pessimistic method of concurrency control as it requires locks to be obtained before data can be accessed.

Another method for determining the serializability order of transactions is to select an ordering among transactions in advance. Timestamping uses this concept.

Though it was originally designed for distributed database systems, many centralized databases use it too [5]. Timestamping has the potential to solve the problem of deadlocks during locking-based protocol. In this method, the database system assigns a unique number to each transaction, called a timestamp, before the transaction starts its execution. In this method, a transaction's timestamp is equal to the value of the clock when the transaction enters the system and a logical counter is used the value of the counter is the value is equal of a transaction's timestamp when it enters the system. The timestamps of the transactions determine the serializability order. Conflicting transactions are always processed in timestamp order and as no transaction ever waits, there is no deadlock either. Timestamping is also a pessimistic protocol in that it forces a wait or rollback whenever a conflict is detected, even though the schedule might be conflict serializable [5].

A concurrency control scheme incurs overhead of code execution due to possible delays of transactions. Again, in cases where a majority of transactions are read-only, the conflicts among transactions might be low. So, in these cases, even if there is no concurrency control scheme, the transactions might leave the system in a consistent state. With this assumption validation-based protocols try to use a method that imposes less overhead but as we do not know in advance which transactions are going to be in conflict, it uses a monitoring scheme to predict that. In this scheme the actual writes take place only after the transaction issuing the write has committed. A validation test is used to check whether the serializability is maintained till the end or not. This is an optimistic concurrency protocol as transactions are executed optimistically assuming they will be able to finish execution and validate at the end.

Until now, this discussion mentioned only protocols that work on a single data item. However, it is often useful to access a set of data items as a single unit as this allows fewer locks to be used. The multiple granularity locking protocol [7] aims to present a concurrency control protocol that minimizes the number of locks while accessing sets of objects in a database. The data items are organized in a tree structure and the smaller data items are contained in larger ones. The root of the tree represents the entire database. When a transaction locks a node, its nodes are also locked implicitly as

well. Locks can be shared or exclusive and a third kind of lock called intention lock has been introduced which works as a lock for all ancestors of a node that is explicitly locked. They have also defined a compatibility matrix to resolve lock conflicts. This protocol has the potential of increasing concurrency and decreasing overhead.

2.4.2. Concurrency Control in Object Oriented Databases

When database functionality was combined with object-oriented concepts then Object-oriented databases became the ideal information repository that is shared by multiple users and, multiple applications on different platforms. Though many of the techniques of traditional concurrency control can be carried over to object oriented databases but the model for transactions supported in conventional database is not suitable for long-duration transactions. Supporting concurrency control in an OODB is more difficult compared than in a relational database because of some issues; for example, the semantics of methods, nested method invocation and referentially shared objects. In addition to that, conventional approaches for transaction management do not work well on OODBMS because of the complex objects and complex transactions OODB deal with. In OODBMS, the methods represent the behavioral aspects of objects and exploiting the semantics of these methods can help in eradicate data contention problem. New semantics in object-oriented databases need to be used to improve concurrency. There is still a lack of efficient concurrency control in OODBMS.

While designing a concurrency control protocol for object-oriented databases, many researchers did adopt some concepts from the existing protocols for relational databases, some re-implemented existing protocols like two-phase locking and time-stamping and made them suitable for object-oriented databases. Some other protocols were modified according to the nature of object-oriented databases to better serve their needs. Semantic-based locking, versioning [7] and simple hierarchical locking schemes are some examples of that. Most of the hierarchical schemes are some variations of the multi-granularity locking [7].

To avoid the potential data contention, most researchers have looked into the object structure and operation semantic while designing an appropriate concurrency

control mechanism. That is why; the researchers started exploiting the rich semantics of object-oriented data model to achieve better performance for OODBMS. Various researchers have taken up either the transaction semantics or the data semantics for SCC in OODBMS. A technical report was published [8] which summarized the use of application and data semantics to optimize concurrency. They have explained three models of transaction approach: first of which is the compatibility set approach and the second one is constraint-based approach. The compatibility set based approach [9] is one of the earliest efforts of using transactions semantics, whereas the constraint-based approach has the capability of handling nested transactions in OODB. The latter makes use of patterns to express correctness constraints to allow higher concurrency. The other approach called the data approach contains two models. The first of them redefines a transaction as a sequence of typed operations on objects and the second one uses a serial dependency concept. A semantic-based concurrency control scheme for OODB is presented in [11]. In this paper, they have used the semantics of methods and the conflict between lower level operations or methods was ignored due to the commutativity of methods invoked at the higher level in nested method execution. They require a lock on an object if a method or operation is invoked on the object. These locks are converted to retained locks at the end of a sub-transaction. If a top-level transaction commits, all the locks held are released.

A semantic two-phase locking protocol for OODB is presented in [12]. They consider nested method invocations in their work. They also exploit semantics of methods for better concurrency. They have allowed any two methods to communicate with each other if application programmers think their execution orders not important and this is done by using semantics of methods. Thus, by taking semantics into consideration, higher concurrency can be achieved. The only condition for this to work was that, the semantically communicating methods should be executed atomically. The one drawback of this method is the big overhead which incurs while using locking for each atomic operation. A method and an algorithm which is capable of real-time scheduling of the SCC in OODB were initiated in [13]. They describe their work as the realization of scheduling of concurrency control based on a combination of databases and operating systems. To avoid the worst case, they have made the early invoked transaction wait for a

long time to avoid deadlock in the scheduling of the Object-oriented Database systems. A locking scheme which is the same as the locking scheme of [12] was proposed in [14] as they also consider semantics of methods and RSO. In their method, called Enhanced Semantic Locking, they argue semantics can be provided at the discretion of the application programmer in methods. On the other hand, in order for ESL to support RSO, they adopt "in-place" conflict resolution policy. According to this, lock modes are not associated with methods. Commutativity of methods is determined when methods invoke shared sub-objects at the same time. This scheme requests a lock whenever a read or write atomic operation is invoked. However, ESL is different from [12] in that lock conversion for retained lock is prohibited.

A locking-based concurrency control method was introduced in [14]. It deals with three important issues in object-oriented databases: semantics of methods, nested method invocation and referentially shared objects. In their proposed scheme, locks are required for the execution of each of the methods instead of atomic operations. Also, a way of automating commutativity of methods is provided. One shortcoming of this method is that they have not considered inheritance hierarchy. The semantics of multi-level transactions was exploited in [16], in the environment of linear hash structures to increase concurrency. They have designed a three-tier client/server layered architecture and an object-oriented implementation of multi-level transactions accessing linear hash structures. Multi-level transactions enhance the concurrency in the linear hash structure and handle transaction aborts.

Multi-granularity locking [7] was originally developed to reduce locking overhead in relational databases. According to this protocol, the database is organized in the form of a hierarchy and locking a higher level means implicitly locking the lower levels. In case of a relational database, the hierarchy is formed by database, relation and row where the dataset is the root and row is the leaf. The designers of earlier object-oriented databases, for example, ORION and O2 followed the same principle and created protocols [3] by replacing database hierarchy with simple class and object hierarchy. These protocols have used strict two phase locking for these protocols but the rich

semantics contained in object-oriented databases have not been exploited in these methods.

A performance evaluation of three multi-granularity locking options for concurrency control was done in [17]. The three protocols taken into consideration were: non-class hierarchy locking, class granularity locking and class hierarchy locking protocol. All of these protocols used strict two phase locking. In class granularity locking, two types of objects can be locked: instance object and class object. Each instance requires a separate lock but locking a class means implicitly locking all instances of the class. In the class hierarchy protocol, apart from instances and classes, a class hierarchy can be a locking unit too. Setting a lock on a class hierarchy would mean implicit locks on all subclasses of that class. In non-class hierarchy locking, locks have to be set on every object to be accessed. Their experimental result indicates that class granularity locking is the better approach in terms of lock overhead in most of the cases that they have considered. However, they have assumed that the option chosen for locking cannot be changed during a transaction's execution. Although they have locks the objects dynamically but the type of object that will be locked was chosen statically before the transactions began.

The multi-granularity model for object-oriented databases has be to quite different from that of relational database. A multi-granularity concurrency control [18] was proposed which took some features of object databases like class hierarchy, composite object hierarchy and schema evaluation into account while designing. This method was inspired by Gray's protocol [7] and it tried to accommodate the concept of composite objects as a logical lockable granules for locking which was not addressed by Gray. In their protocol, locking in schemas and locking in instances were developed separately and then they were integrated. For instance locking, composite objects were distinguished from primitive objects. They have also proposed a dual-queue scheduling for improving concurrency. With some examples, they have proved that their multi-granularity protocol for object-oriented database works delivers a better degree of concurrency than the protocol used in ORION.

AMGL [5] is a concurrency control protocol for object-oriented databases. which is adaptive to the transaction requirement. The ability to use different lock granularity units dynamically at run time enables this protocol to reduce lock overhead in some situations compared to the conventional locking protocols. They have used three lockable granules namely, instance, class and class hierarchy and by using lock escalation and de-escalation, these lock granules can be adjusted to suit the requirements of a transaction dynamically. As no application specific information is needed for this protocol, it can be employed in any object-oriented databases. They have taken composite objects as well as schema modification into consideration. The locks that this protocol uses are soft locks and firm locks. A soft lock is similar to an intention lock with the added power of being able to keep track of the information about underlying objects covered by the lock. This information is used in lock escalation and de-escalation process. It also is a notification to all ancestor objects that the object is locked. A firm lock is like a regular lock with the added advantage of lock escalation and de-escalation being performed on that. AMGL was compared against instance granularity locking, class granularity locking and multi-granularity locking. A series of experiments [1] were found and found out that their protocol is the best choice amongst those protocols when the lock conflicts are not high. Again, in terms of locking overhead, in most cases, AMGL was even worse than instance granularity locking.

CHAPTER III

DESIGN AND DEVELOPMENT OF SAML

SAML is a Self-Adjusting Multi-granularity Locking protocol that is proposed to improve the concurrency control of object-oriented databases. It aims to maximize parallelism of transactions while keeping the locking overhead as low as possible. This protocol facilitates choosing appropriate locking granularity according to the requirements of the transaction. This protocol uses three levels of granularity for locking, namely instance lock, class lock and class hierarchy lock. SAML uses lock de-escalation which enables the lockable granules to be adjusted automatically during the lifetime of a transaction. This optimistic locking protocol uses conservative two-phase locking to avoid deadlocks.

This chapter presents the development of the SAML protocol. First, determining the appropriate lockable granules has been discussed and then the lock types, modes as well as the lock compatibility matrix have been designed. After that, the main components of SAML and the locking rules are developed and discussed. The chapter concludes with a discussion on LG graph and the usage of lock de-escalation in SAML.

3.1. Lockable Granularity Units

The size of the objects that can be locked is known as the granularity unit. These granularity units are used for ensuring consistency. The finer the granularity, the greater would be the potential for parallelism. But lock overheads are increased in this case. On the other hand, a coarse granularity unit would lead to fewer locks and as a result there will be fewer overheads in testing, setting and maintaining the locks. This can lead to less concurrency as the active transactions may hold lock on more resources than required, which in turn can result in the rejection of a lock request by other transactions. Hence, it would be more beneficial if the database management system provides a variety of granularity units depending on the different requirements of different transactions.

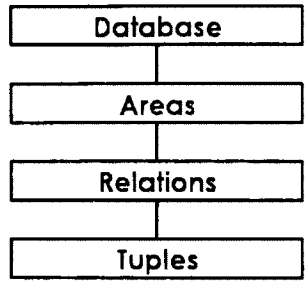

Figure 3-1. Lock hierarchy in RDBMS

In relational databases, a simple lock hierarchy is followed to organise the lockable granularity units or the resources. In relational databases, granularity units can be records, files, fields etc. Figure 3-1 demonstrates the basic set of resources in a Relational Database. The lines between these resources represent the containment relationship. Every database has one or more areas and each area contains one or more relations. Again, each relation contains many tuples. In RDBMS, despite the fact that all the resources are in a parent-child relationship, they are independent from each other in terms of their content. That means, even though some relations of an area are changed, those relations will still be contained in that area. While each of these nodes in the hierarchy can be locked, the children of those nodes are also locked implicitly.

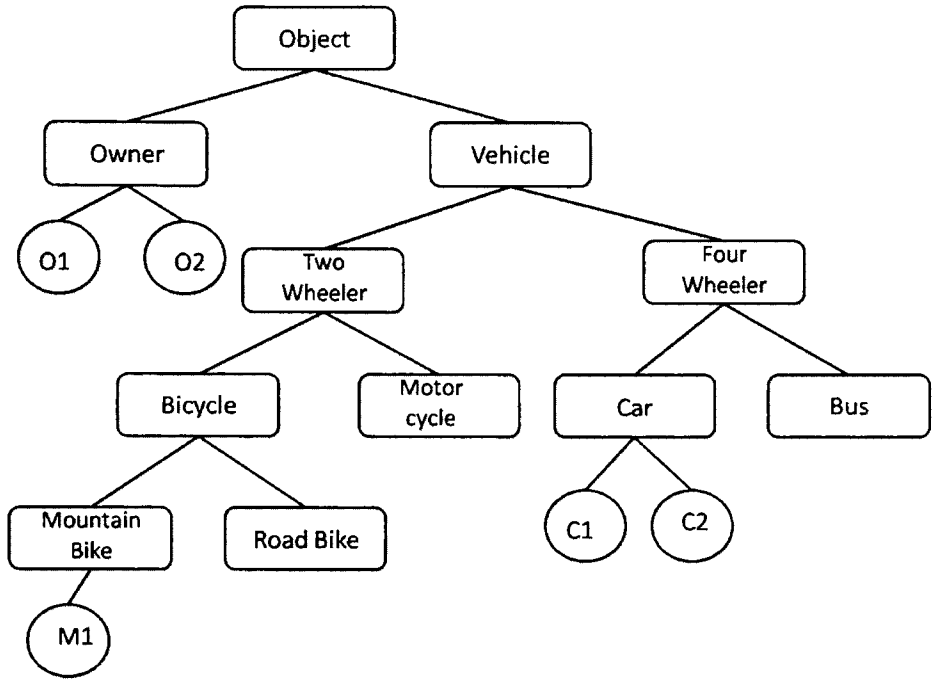

Figure 3-2. An Object Oriented Database

In an object-oriented database, the information contained in the data model is not application dependent. Moreover, this kind of database has a rich semantic data model. This information can be used for concurrency control by determining the appropriate granularity units for locking. Figure 3-2 is an example of an object-oriented database. There is only one root super class in this database and it represents all the objects of the database. All the classes are subclasses of the object class. The object class has two children subclasses: owner and vehicle. The other subclasses are inherited from these two classes. For example, here bicycle is a subclass of two-wheeler which in turn is a subclass of class vehicle. So this relationship is generalization-specialization relationship. All the classes may have one or more instances too. For example, C1, C2 are instances of Car, whereas M1 is an instance of mountain bike.

As shown in Figure 3-2, there are three kinds of resources that can be used as lockable granularity units to increase the concurrency in object-oriented databases. SAML uses these resources namely instance, class and class hierarchy as the granularity units for the protocol. Though SAML locks the coarse most granule and keeps on reducing the granule size on the event of a conflict, it should be noted that it is not

similar to a relational database as changing the content of one resource could potentially affect another resource on a higher level of object oriented databases.

3.2. Explicit Lock and Implicit Lock

SAML proposes some lock modes for the three above discussed granularity units which has been discussed in section 3.4. Apart from that, two additional types of locks are used.

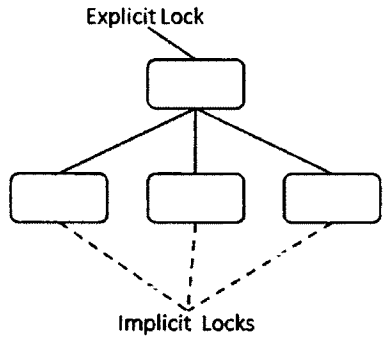

Figure 3-3. Explicit and Implicit Locks

A lock on recourse of the database can either be explicit or implicit. An explicit lock is a lock that is explicitly requested by and is granted to a transaction directly. An implicit lock is derived from an explicit lock. Implicit locks are the locks that need to be acquired, in either direction of the database hierarchy, for explicit locks. In the proposed SAML protocol, the term implicit lock has been used to represent the lock which propagates down the hierarchy as a result of the lock on the current node. This implies that the modes of the implicit locks will be the same as the explicit lock on the current node.

3.3. Soft Lock

Soft locks are basically the implicit locks that propagate upwards in a class hierarchy because of an explicit lock on a node. In an object-oriented database any instance or class is dependent on its class or super-classes. Therefore to lock any resource (instance/ class/ class hierarchy) explicitly in shared or exclusive mode all the ancestors of that resource must be locked in shared mode so that no one can modify them. If any node is explicitly locked by a transaction, soft locks are implicitly applied on the ancestors to prevent granting of exclusive lock requests on the ancestors.

3.4. Lock Modes in SAML

Here are the lock modes being proposed for the self-adjusting multi-granularity locking protocol. Based on the granularity units used in this locking protocol, the lock modes have been categorized into three different groups: instance lock, class lock and class hierarchy lock.

Two locks of the same or different modes on the same object are considered compatible if they can be granted concurrently for different transactions. All the lock modes used in this protocol are either shared lock or exclusive lock or variations of these two modes.

3.4.1. Instance Locks

In reality, transactions deal with a set of instances of one or more classes. Instances are the smallest granularity unit of the SAML protocol. According to this protocol, an instance can be locked either in shared (IS) or exclusive (IX) mode.

3.4.1.1. IS (Shared Instance Lock)

A shared instance lock on an instance means that the instance can be read by one or more transactions but can be modified by none of them.

3.4.1.2. IX (Exclusive Instance Lock)

An exclusive lock on an instance means that the instance can be read and modified only by a transaction holding the lock. No other transaction can read or modify the instance.

3.4.2. Class Locks

Class locks are the next higher granule for locking after instances. A class can have several instances. So, a lock on a class means that one or more instances of that class are locked by one or more transactions. A class can be locked in shared (CS), exclusive (CX), soft shared (CST) or soft exclusive (CXT) mode. A CS or CX lock on a class means that there is an explicit lock on the class itself, i.e., one or more transactions are holding explicit lock(s) on the class. On the other hand, CST or CXT lock on a class means one or more instances of the class are locked in IS or IX mode respectively.

3.4.2.1. CS (Shared Class Lock)

A shared lock on a class C means that all the instances of the class are implicitly locked in IS mode. The instances of the class can be read by the transactions that are holding the CS lock on the class C. But none of them can modify any instance of C.

3.4.2.2. CX (Exclusive Class Lock)

An exclusive lock on a class C means that all the instances of the class are implicitly locked in IX mode. Only the transaction holding the CX lock on a class can read or modify its instances. No other transaction can have any access to any of the instances of that class.

3.4.2.3. CST (Soft Shared Class Lock)

A soft shared lock on a class means that one or more instances of the class is/are explicitly locked in IS mode by one or more transactions. The locked instances as well as the other instances of the current class can be locked explicitly by other transactions as long as these lock requests by other transactions do not conflict with the existing locks by the existing transactions.

3.4.2.4. CXT (Soft Exclusive Class Lock)

A soft exclusive lock on a class means that one or more instances of the class is/are explicitly locked in IS or IX modes, with at least one IX lock, by one or more transactions. The locked instances as well as the other instances of the current class can be locked explicitly by other transactions as long as these lock requests by other transactions do not conflict with the existing locks by the existing transactions.

3.4.3. Class Hierarchy Locks

Locking a class in hierarchy mode means all its instances along with all the descendents of the class and their instances are locked in the same mode. In a situation where a transaction is accessing instances of multiple classes, it is better to lock the super class in hierarchy mode. This can reduce locking overhead greatly. In SAML, four kinds of hierarchy lock modes are provided:

3.4.3.1. HS (Shared Hierarchy Lock)

A shared hierarchy lock on a class means the class, its subclasses and all the instances of those classes are implicitly locked in shared mode. This includes the entire tree structure rooted at the current class and recursively traversing down to the leaves of the tree. The whole hierarchy rooted at class C can be read by the transactions that are holding a HS lock on the class C. But none of them can modify any instance of any class of the hierarchy.

3.4.3.2. HX (Exclusive Hierarchy Lock)

An exclusive hierarchy lock on a class means the class, its subclasses and all the instances of those classes are implicitly locked in exclusive mode. This includes the entire tree structure rooted at the current class and recursively traversing down to the leaves of the tree. The whole hierarchy rooted at class C can be read and modified only by the transaction that is holding the HX lock on the class C. But no other transaction can access any instance of any class of the hierarchy.

3.4.3.3. HST (Soft Shared Hierarchy Lock)

A soft shared hierarchy lock on a class C means that one or more descendent classes of C or the class C itself is/are locked in CS or CST mode or HS (only for descendent classes) mode by one or more transactions. Other transactions can lock the entire hierarchy or any part of it only in shared mode (IS, CS or HS).

3.4.3.4. HXT (Soft Exclusive Hierarchy Lock)

A soft exclusive hierarchy lock on a class C means that one or more descendent classes of C or the class C itself is/are locked in CS, CX, CST, CXT mode or HS/HX mode (only for descendent classes), with at least one lock in exclusive mode, by one or more transactions. No transaction can lock the entire hierarchy but any part of it can be locked by other transactions in shared or exclusive mode (IS/IX, CS/CX or HS/HX) only if they do not conflict with any of the existing locks.

3.5. Why Soft Lock?

A soft lock on any level of the hierarchy indicates the presence of an explicit lock somewhere down the hierarchy. This prevents a transaction to be granted a conflicting lock on the hierarchy.

Let us assume that there is a shared (IS) lock on an instance of a class C. In this situation, if another transaction appears and tries to lock the super class of C in HX mode, it would be granted if the super class does not have any information about the presence of a lock in any part of its sub hierarchy. To prevent this kind of conflict, soft locks are used.

3.6. Precedence of lock modes

Figure 3-4 shows the precedence and lock coverage of the lock modes discussed for the SAML protocol. A lock of lower granularity is contained in a lock of higher granularity. For example, an IS lock in contained in a CS/CST lock whereas a CS/CST lock is contained in a HS/HST lock. Again, in the same granularity level, an exclusive lock has higher precedence over a shared lock and an explicit lock has higher precedence over a soft lock.

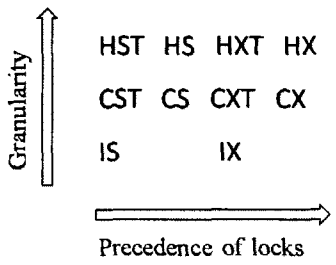

Figure 3-4. Precedence of lock modes

3.6.1. Lock Coverage

A lock L_1 is said to be covered by another lock L_2 if L_2 has higher granularity than L_1 or if L_2 has a higher precedence over L_1 where both L_1 and L_2 have the same granularity level.

3.7. Lock Compatibility Matrix

Based on the proposed lock modes, the 10×6 compatibility matrix is shown in Table 3-1. Compatibility among the lock modes has been determined from their semantics. In SAML, a transaction can only request explicit locks. The soft locks are implicitly applied by the SAML protocol because of the existence of some explicit locks. So, the new lock mode being requested can only be one of the six explicit locks, i.e. IS, IX, CS, CX, HS or HX.

Each cell in the lock compatibility matrix represents whether the requested new lock mode in the column is compatible with the existing lock mode in the row. A "Y" means compatible and hence the lock request is granted. A "N" means conflicting lock request which will not be granted according to SAML. A "D" means the de-escalation of the new lock. It denotes that there is an existing soft lock on the current object and the new lock requested is a conflicting one but it may be granted on a lower granularity unit. A "DD" means the de-escalation of both the existing and the requested lock. It denotes that there is an existing explicit lock on the current object and the new lock requested is a conflicting one. According to SAML, in this case, the existing lock must de-escalate first in order to accommodate any new lock on a smaller granularity level. So, the existing lock de-escalates first and after that the requested lock may be granted on a lower granularity unit.

Existing Lock Modes \ New Lock Modes	IS	IX	CS	CX	HS	HX
IS	Y	N				
IX	N	N				
CST			Y	D	Y	D
CS			Y	DD	Y	DD
CXT			D	D	D	D
CX			DD	DD	DD	DD
HST			Y	Y	Y	D
HS			Y	DD	Y	DD
HXT			Y	Y	D	D
HX			DD	DD	DD	DD

Table 3-1. Lock Compatibility Matrix

In Table 3-1, N = conflicting lock request that will not be granted, Y = lock request granted, D = de-escalation of the new lock, DD = de-escalation of both existing and requested lock.

3.8. Main Components of SAML

Unlike the other existing locking protocols, in SAML the lock table alone does not maintain all the locks in the system. Along with the lock table, a class inheritance graph of the database is used for this purpose. The main purpose of a class inheritance graph is to maintain the soft locks as they propagate recursively upwards in the class hierarchy till the root. In this section, the class inheritance graph, the lock table and the transaction list of SAML is discussed.

3.8.1. Class Inheritance Graph

The Class Inheritance Graph (CIG) is used only to maintain the soft locks as well as the existence of explicit locks on a class or class hierarchy. This graph enables navigation from a class to its subclasses or its super classes. Unlike AMGL [1], the lock table in SAML does not have to retain any information about the soft locks because of the presence of CIG. This significantly reduces the size of the lock table.

For example, if n transactions concurrently acquire explicit shared locks on an object at the l-th level of the class hierarchy, then according to AMGL, the lock table has to preserve the information of additional n × (l−1) soft locks along with the n explicit locks. On the contrary, SAML deals with all the soft locks using only the class inheritance graph which is static in size. So, the lock table in SAML does not have to maintain any information about the soft locks.

Figure 3-5 represents an example of a class inheritance graph along with the some locks on objects. Each node in CIG stands for a class and each node has two colors: class color and hierarchy color. The term "color" has been used for CIG to indicate the lock modes on the nodes. In the following figure the shaded half of a node represents hierarchy color and the other half represents class color.

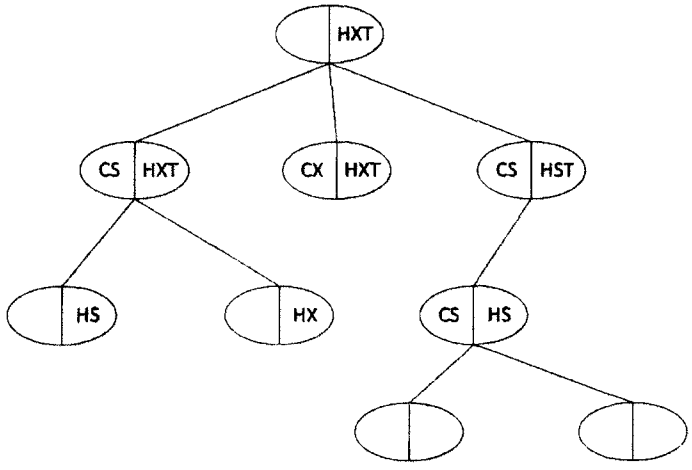

Figure 3-5. Class Inheritance Graph

3.8.2. Lock Table

A lock table is the list of explicit locks present in the system. Each entry of the lock table is a lock with the following information:

1) Lock ID (LId)
2) Transaction ID (TId)
3) Set of target resources (instances) {RT} ≡ {(r_i, m_i)}

Where, r_i is the target resource to be locked and m_i is the mode with which r_i is to be locked

4) Current resource locked (RC)
5) Mode of the lock (LM)

So, a lock L can be represented as follows:

L = {LId, TId, $\{(r_1, m_1), (r_2, m_2), \ldots, (r_i, m_i)\}$, RC, LM}

$\forall\, i$, (r_i, m_i) is covered [section 3.6] by (RC, LM)

3.8.3. Transaction List

The transaction list contains the list of the transactions that are currently present in the system. Each entry of the transaction list is a transaction with the following information:

1) Transaction ID (TId)
2) Set of locks held by the transaction $\{L\}$
3) Set of instances to be locked with mode $\{(r, m)\}$

So, a transaction T can be represented as follows:

T = {TId, $\{L_1, L_2, \ldots, L_i\}$, $\{(r_1, m_1), (r_2, m_2), \ldots, (r_k, m_k)\}$} where $i \leq k$

3.9. Rules for Requesting Locks

3.9.1. Locking an instance

3.9.1.1. Shared Lock (IS)

1) To lock an instance in shared mode, an IS lock must be obtained for the instance.
2) Before an IS lock can be obtained, we must
   a) search the lock table to check whether there is any conflicting lock
   b) the class must be locked in CST mode
3) Add a corresponding entry to lock table

3.9.1.2. Exclusive Lock (IX)

1) To lock an instance is shared mode, an IX lock must be obtained for the instance.
2) Before an IX lock can be obtained, we must
   a) search the lock table to check whether there is any conflicting lock
   b) the class must be locked in CXT mode.
3) Add a corresponding entry to lock table.

3.9.2. Locking a class

3.9.2.1. Shared lock (CS/CST)

1) To lock a class is shared mode, a CS/CST lock must be obtained for the instance.
2) Before a CS/CST lock can be obtained for class C, the super class, if there is one, must be locked in HST mode.
3) Before locking a class in CS mode,
   a) check the corresponding node's colour in the class inheritance graph to see whether there is any conflicting lock.
   b) colour the node in CS, if it is not already coloured or coloured in CST.
   c) add an entry in the lock table.
4) Before locking a class in CST mode,
   a) colour the node in CST, if the node is not coloured.

3.9.2.2. Exclusive mode (CX/CXT)

1) To lock a class in exclusive mode, a CX/CXT lock must be obtained for the class.
2) Before a CX/CXT lock can be obtained for a class C, the super class, if there is one, must be locked in HXT mode.
3) Before locking a class in CX mode
   a) check the corresponding node's colour in the class inheritance graph to see whether there is any conflicting lock.
   b) colour the node in CX, if the node is not coloured.
   c) add an entry in the lock table.
4) Before locking a class in CXT,
   a) colour the node in CXT, if it is not coloured or coloured in CST mode.

3.9.3. Locking a class hierarchy

3.9.3.1. Shared mode (HS/HST)

1) To lock a hierarchy in shared mode, a HS/HST lock must be obtained for the class hierarchy.
2) Before a class hierarchy can be locked in HS/HST mode, its super class, if there is one, has to be locked in HST mode.
3) To lock a class hierarchy in HS mode
   a) Check the corresponding node's color in the class inheritance graph to see if there is any conflicting lock.
   b) Colour the node in HS, if it is not coloured or is coloured in HST mode.
   c) Add an entry in the lock table.
4) Before locking a class hierarchy in HST mode,
   a) Colour the node in HST if the node is not coloured.

3.9.3.2. Exclusive mode (HX/HXT)

1) To lock a class hierarchy in exclusive mode, a HX/HXT lock must be obtained.
2) Before a HX/HXT lock can be obtained for a class hierarchy, the super class, if there is any, must be locked in HXT mode.
3) Before locking a class hierarchy in HX mode
   a) Check the corresponding node's colour in the class inheritance graph to see if there is any conflicting lock.
   b) Colour the node in HX mode, if the node is no coloured.
   c) Add an entry in the lock table.
4) Before locking a class hierarchy in HXT mode
   a) Colour the node if it is not coloured or is coloured in HST mode.

3.9.4. Lock Release

When a transaction commits/aborts, all the locks held by the transaction must be released. To release a lock:

1) Delete the lock from the lock table.

2) Search the remaining lock table to check if there is any other explicit lock on the object.
    a) If yes, then there is no change in colour.
    b) If there is no other lock,
        i) check the colour of all the children of current node
        ii) update the colour of the node accordingly in the *class Inheritance graph.*
        iii) re-colour the super class of current node in class inheritance graph

To re-colour a node in class inheritance graph,

1) Decide the new colour of the node from the colour of its sub-classes.
2) If the new colour is different from the old colour
    a) colour the node with new colour
    b) re-colour the super class of current node.
3) If the new colour is same as the colour
    a) return

3.10. Lock Granularity Graph

The database resources are grouped and structured in a manner much like a tree and it is called a Lock Granularity (LG) graph. Although it looks like a tree, it is a Directed Acyclic Graph (DAG) when the class hierarchy is considered. The resources are arranged according to the varying depth of their granularity. The source node, or the root, of the graph represents the entire database and the intermediate nodes in the graph correspond to consecutively finer granularity units. The nodes of the graph depict the successive refinements of granularity from coarse towards finer granules from the root to the leaf level of the tree.

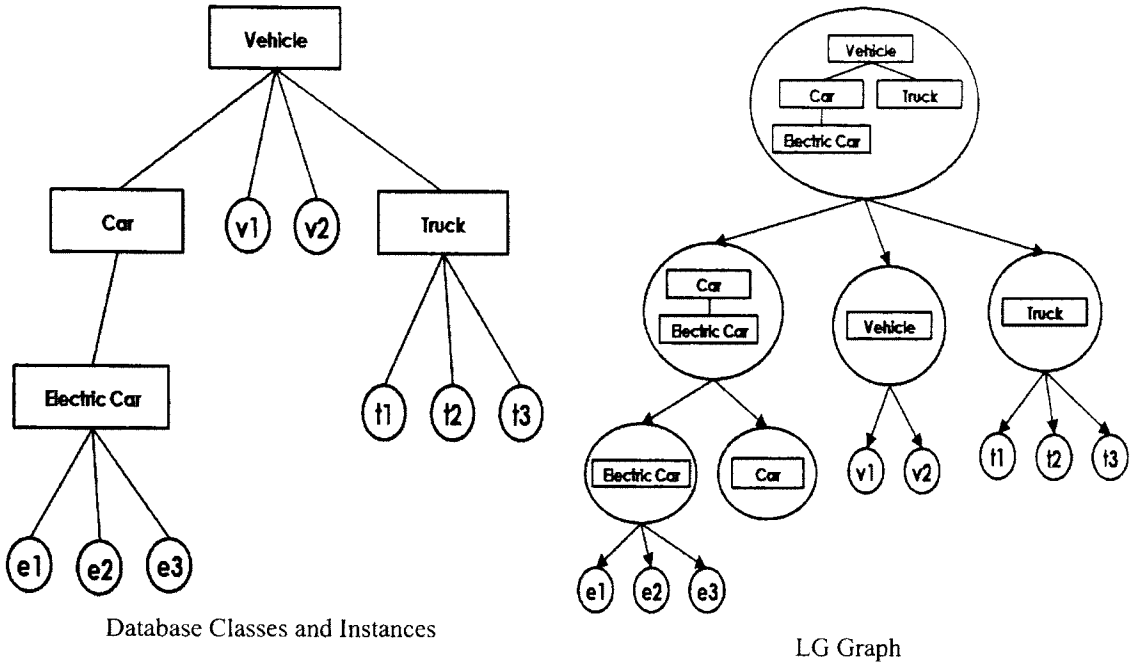

Figure 3-6. Lock Granularity Graph

3.10.1. Formal Definition

The lock granularity graph is a Directed Acyclic Graph (DAG) where each node of the graph is a granularity unit of the database.

DAG is a type of graph which is part tree and part graph. A DAG is a pair (V, E) where V is a set of nodes, and E is a set of edges between the nodes $E \subseteq \{(U, V) \mid U, V \in V\}$ with no path that starts and ends at the same node, that is, $E \subseteq \{(U, V) \mid U, V \in V \wedge U \neq V\}$ [19]. A node p is a parent of a child c and c is a child of parent p, if $<p,c> \in E$. Again, a path is any route along the edges of a graph. A path may follow a single edge between two nodes or multiple edges through multiple nodes. So, a path is any sequence of two or more nodes $<n_i \mid i = 1, ..., k>$, such that for each $1 \leq i \leq k$, $<n_i, n_{i+1}> \in E$. Node n is an ancestor of node c if n is on some path from a root to node c.

In the single root lock granularity graph the root node is used to represent the whole database or in other words, the class inheritance tree rooted at the root class. There are three different types of nodes in the graph corresponding to the different lockable granularity units:

- Instance Node: A single instance of class
- Class Node: A single class
- Class Hierarchy Node: A set of classes that forms a class hierarchy.

The set of edges of the Lock Granularity graph follows these rules:

1) <p,c> exists if p is a class and c is an instance node where c represents an instance of p.

2) <p,c> exists if a p is a class hierarchy node and c is a class hierarchy node, where c is an immediate subtree of p in the class hierarchy tree.

3) <p,c> exists if p is a class hierarchy node and c is a class node, where c is the class of p.

Each of the nodes of the LG graph can be locked as each node represents a lockable unit in the graph. As the child node is absolutely contained in a parent node, so, the child node represents a granularity unit which is contained within the granularity unit represented by the parent node.

3.11. Lock De-Escalation in SAML

Like all other granularity locking methods, SAML tries to minimize the number of locks implemented to access a database. The lock modes of SAML defined in section 3.1, if used for locking, can guarantee that the various updates by different transactions of the system will be reflected correctly as well as consistently. But, maintaining these locks could incur high overhead. The number of objects being locked, the kind of object being locked and the lock coverage of the lock being implemented can affect the level of overhead of lock maintenance.

For example, if most of the instances of a class need to be accessed then acquiring one lock for the entire class, instead of acquiring one lock for each instance, can greatly reduce the number of locks being maintained. A lock on a class means a lock on each instance of the class. On the other hand, if a few instances of a class are to be accessed, it is better to acquire individual locks for the instances so that other transactions get to access the rest of the instances concurrently. So, the optimal choice of locking

granularity will vary from one transaction to another. In addition to that, this optimal choice might vary for a single transaction at different times. Hence the optimal granularity of locking for each transaction can only be determined dynamically at run-time.

Most of the available locking protocols choose lock granularity for all transactions in a database statically which produces average result. If these protocols are choosing the fine granularity unit unless there is other guidance, then they would get the maximum degree of concurrency but the lock maintenance might result in a huge overhead. Again, if they are choosing a coarser granularity unit, then the lock maintenance issue can be resolved but as the granularity unit is not changing, even when there is a better granularity unit to lock later in the transaction, the desired level of concurrency cannot be achieved. As a result, over time, this results in a granularity unit that is not the best choice for being locked by the transactions of the system.

To resolve these issues, lock de-escalation has been used in SAML so that it can facilitate choosing the appropriate locking granularity according to the requirement of the transaction. This operation is performed dynamically during the transaction. The largest granularity unit is locked at the beginning of a transaction even if most of the objects being locked are not necessary at first. However, it keeps track of the objects being accessed and the modes they are being accessed in. This information is used while de-escalation is performed. Later, if there is any conflict from other transactions, lock de-escalation is performed. Each conflicting transaction then would lock the next higher granularity unit that is not in conflict. So when the locks of two transactions conflict, both transactions keep on reducing their granule size until the conflict is resolved. Therefore, the protocol adjusts itself to the transaction requirements by using multiple granularity units to provide maximum concurrency but keeping minimum locking overhead. Thus, the Self Adjusting Multi-Granularity Locking (SAML) protocol aims for a fair trade-off between locking overhead and concurrency.

Among the three levels of granularity, namely instance level, class level and hierarchy level, a hierarchy lock on a class C after de-escalation can generate one or more

locks which are either a class lock on C or hierarchy locks on the sub classes of C. A hierarchy lock can be represented as follows:

$L = \{ID, TID, \{(r, m)\}, RC, M\}$

Where, $M = \begin{cases} HS & if\ m_i = IS\ \forall\ i \\ HX & otherwise \end{cases}$

A hierarchy lock can be de-escalated like this:

$L \rightarrow \{\{L_H\}, L_C\}$ or $\{L_H\}$ or L_C

Here, L_H is hierarchy lock on a sub- hierarchy of RC

L_C is a class lock on RC.

L_H and L_C can be defines as follows:

$L_H = \{ID_H, TID, \{(r^H, m^H)\}, RC_H, M_H\}$

Where,

$$M_H = \begin{cases} HS & if\ m_i^H = IS\ \forall\ i \\ HX & otherwise \end{cases}$$

$\{(r^H, m^H)\} \subseteq \{(r, m)\}$ and

RC_H is a sub class of RC

Again,

$L_C = \{ID_C, TID, \{(r^C, m^C)\}, RC, M_C\}$

$$M_C = \begin{cases} CS & if\ m_i^C = IS\ \forall\ i \\ CX & otherwise \end{cases}$$

and $\{(r^C, m^C)\} \subseteq \{(r, m)\}$

A class lock on a class C after de-escalation can generate one or more instance locks on the instances of class C.

If $L = \{ID, TID, \{(r, m)\}, RC, M\}$ is a class lock, i.e. $M = CS/CX$, then after de-escalation, for each (r_i, m_i) in $\{(r, m)\}$, a new lock L_i is created such that,

$L_i = \{ID_i, TID, (r_i, m_i), r_i, m_i\}$

This section outlined the types of locks, the lock modes, the lock request rules as well as the lock de-escalation concept used for the design and development of SAML.

The following section provides a simulation overview of SAML. The database, the transaction model and the simulation model used for the purpose has been discussed in details.

CHAPTER IV

SIMULATION OVERVIEW

This chapter elaborates on the simulation of the concurrency control protocols. It discusses the database model, the transaction model, as well as the simulation model used for implementation of the protocols. The pseudo code for procedure SAML has been provided in this section too.

4.1. Database System Model

The database model used in the simulation of SAML is similar to the one used in AMGL proposed by C.T.K. Chang [1]. The database consists of a number of classes and instances. The classes form a class hierarchy. The exact nature of the database class hierarchy is controlled by two parameters: *numSubClassPerClass* and *numClassLevel*. The parameter *numSubClassPerClass* outlines the number of sub classes per class and *numClassLevel* stands for the depth of the class hierarchy tree or the number of class levels. The total number of classes in the database is determined by these two parameters. The total number of classes in the database will be $\sum_{i=0}^{n} C^i$ or $\frac{C^{n+1}-1}{C-1}$ where C is the number of sub classes per class and n is the number of levels in the class hierarchy. Each class has a number of instances which is defined by the parameter *numInstancesPerClass*.

Each class contains the following attributes: a class id, the class level in the hierarchy, a reference to its parent class, a list of its sub classes and a list of instances of that class. Each instance has two attributes: an instance id and a reference to its class.

Parameters	Description
numSubClassPerClass	Number of sub classes per class
numClassLevel	The depth of the class hierarchy
numInstancesPerClass	Number of instances in a class

Table 4-1. Summary of Database Parameters

4.2. Transaction Model

The transactions required for the simulation were generated offline by a transaction generator. During simulation, a transaction injector injects those transactions into the system following a Poisson distribution pattern. Each transaction has the following parameters: *transactionID*, *numInstances*, *writeRatio*, *duration*, *positionInDB* and *instanceList*. The workload of the transaction is controlled by the parameter *numInstances*. It is the total number of instances that a transaction works with. The ratio of number of instances written to the number of instances read is characterized as *writeRatio*. The parameter *duration* controls the lifetime of a transaction. The parameter *positionInDB* indicates the part of database hierarchy (namely: first half, second half and overall) from which the instances are chosen. The set of instances used by the transaction is stored in *instanceList*. For each transaction, the list of instances is generated from the database by a transaction generator based on the parameters *numInstances* and *positionInDB*. The pseudo code for transaction generator is given in section 4.2.1.

Parameters	Description
transactionID	Id of the transaction
numInstances	Total number of instances a transaction works with
Duration	The lifetime of a transaction
positionInDB	The part of database class hierarchy from which instances are chosen
writeRatio	Number of instances written to the number of instances read
instanceList	Set of instances used by the transaction

Table 4-2. Summary of Transaction Parameters

4.2.1. Pseudo Code for Transaction Generator

Input: database, positionInDB, numInstances

Output: Transaction

Procedure ***Generate_Transaction***:

1) *instanceRemaining* = *numInstances*

2) *minLevel* =0
3) *maxLevel* = depth of database class hierarchy
4) if *positionInDB* = First half
   a) *maxLevel* = *maxLevel*/2
5) if *positionInDB* = Second half
   a) *minLevel* = *maxLevel*/2
6) Create a new transaction *T*
7) while *instanceRemaining* > 0
   a) *maxInstanceFromAClass* = minimum of *instanceRemaining* and *numInstancePerClass* of the database
   b) *noOfInstance* = a random number between 1 and *maxInstanceFromAClass*
   c) Choose a class *C* randomly from the database having level between *minLevel* and *maxLevel*
   d) Choose *noOfInstance* instances of *C* randomly and add them to the *instanceList* of transaction *T*
   e) *instanceRemaining* = *instanceRemaining* – *noOfInstance*
8) return transaction *T*

4.3. Simulation Model

The simulation model is constructed by abstracting i.e., by performing the process of generalization by reducing the information content of an object-oriented database management system, in order to retain only information which is relevant for simulation purpose. It has been assumed that there is no schema change and no multiple inheritances in the database. This condition is a basic model in the experiments done. It simulates the execution of transactions in a database management system. In addition, the transaction generator and database generator have also been simulated. For each of the locking protocols, the concurrency control protocol along with lock manager and transaction manager have been simulated to control the database operations.

The simulation environment was built using Java. Java's multi-threading feature has been used to simulation concurrent processing of multiple transactions. Each component of the simulation environment, for example, the concurrency control

manager, the lock manager and the transaction manager runs concurrently in separate threads during the simulation process. In addition to that each active transaction runs in separate threads as well.

Transactions are generated offline by a transaction generator prior to the simulation. During the simulation, a transaction injector injects the transactions into the system using a Poisson distribution [1], with mean rate of 10 transactions per unit time, and puts them at the end of *ready queue*. The concurrency control protocol takes one transaction at a time from the front of the ready queue and requests a lock for the transaction. If the request is granted, then the transaction is put into the *active transaction list* and a new thread is created for the transaction which remains alive during the lifetime of the transaction. On failure of the lock request, it goes to the *waiting transaction list* and the conflicting transaction in the active transaction list is updated with this information. When a transaction ends, it is removed from the active transaction list. If there is any transaction waiting for the completion of that transaction, it is removed from the waiting transaction list and put at the front of the ready queue. Figure 4-1 is a pictorial representation of the simulation model discussed.

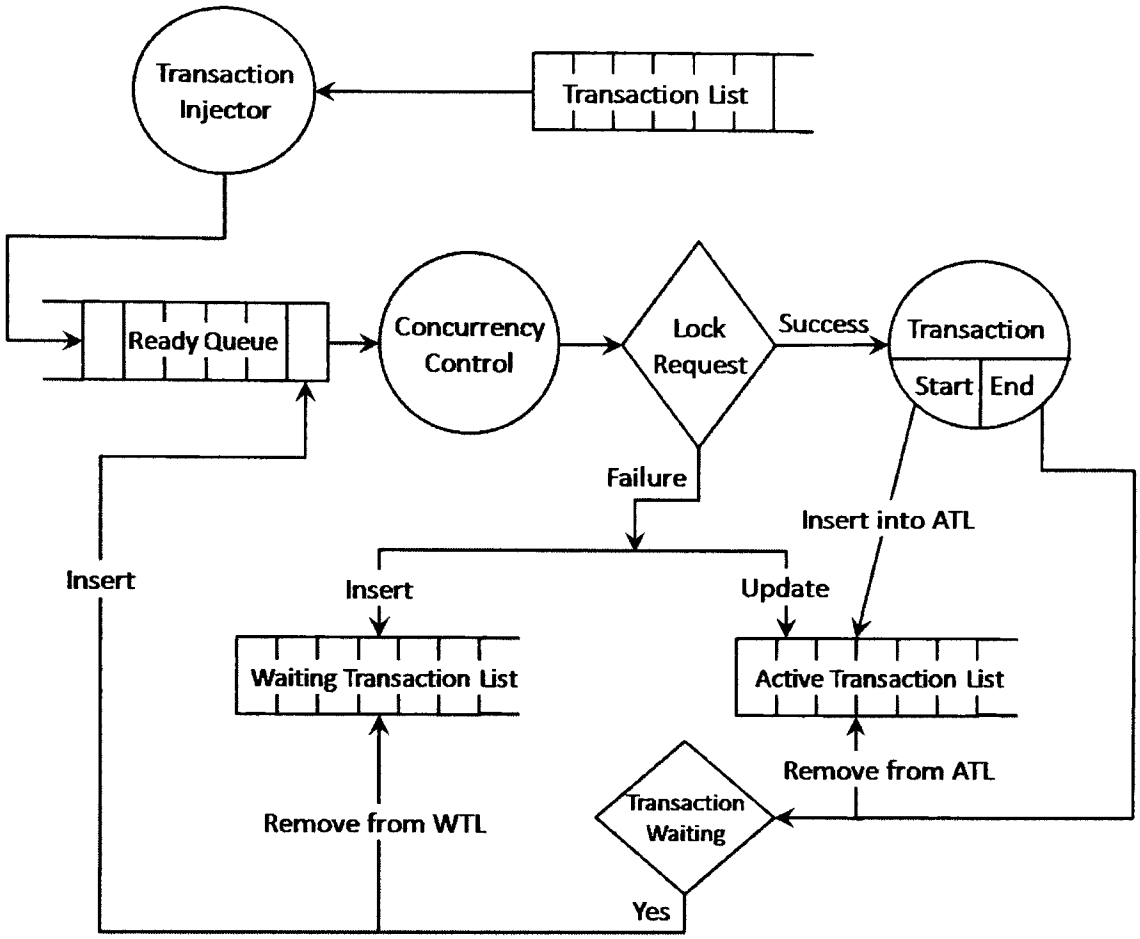

Figure 4-1. Simulation Model

4.4. Pseudo Code for SAML Protocol

The SAML protocol discussed in chapter CHAPTER III has been implemented using the simulation model described in section 4.3. The pseudo code for the implementation of SAML protocol is given below:

- Procedure ***SAML***:-

A) For each transaction *T* in the Ready Queue

1) Request_lock_for_transaction(*T*) and obtain the lock set *L*
2) If *L* is ***null*** then
   a) Put the transaction *T* in Wait Queue
   b) Find the existing conflicting transaction (*CT*)
   c) Add *T* to the waiting transaction list of *CT*

3) If L is ***not null*** then

a) For each lock l in L

i) Apply the lock l

ii) update the color of the corresponding node in the *class hierarchy graph*

4) Put T in the active transaction list

- Procedure ***Request_lock_for_transaction(T):-***

A) Create an empty lock set L for the transaction T

B) Create a new hierarchy lock l for transaction T with root as the hierarchy granularity unit

C) Insert l into L

D) $i = 0$

E) while i < size of L

1) currentLock = i th lock of L

2) request_Lock(currentLock)

a) if successful then

i) i ++;

b) if failure then

i) if current lock is instance lock then

find conflicting transactions from lock table

return ***null*** (lock request for T failed)

ii) if not then

- remove current lock l from L

de-escalate current lock and obtain de-escalation lock set

insert all the locks from de-escalation lock set to the end of L

F) Return ***L*** (lock request for T succeeded)

- Procedure ***Request_Lock (currentLock):-***

A) if current lock is a *class lock* or *hierarchy lock* then

1) Check the color of the corresponding mode in the *class hierarchy graph* to find lock compatibility

a) If current lock is compatible,

i) return ***success***

b) else

i) return ***failure***

B) If current lock is instance lock then

1) Search lock table to find if there is any conflicting lock

a) If no conflicting lock is found

i) return ***success***

b) else

i) return ***failure***

- When a transaction *T* finishes:

A) For each lock *l* of the transaction

1) Remove lock *l* from lock table

2) update the color of the corresponding node in the *class hierarchy graph*

3) Remove transaction *T* from the active transaction list

4) For each transaction *t* in the waiting list of *T*

a) Put *t* in Ready Queue

Remove *t* from waiting transaction

In the next section an experimental comparison of SAML with ILP and CLP has been done and an analytical comparison of SAML and AMGL has also been presented.

CHAPTER V

EXPERIMENTS, RESULTS AND ANALYSIS

This chapter gives a detailed description of the simulation experiments that were performed. Two different protocols were tested and compared experimentally against the SAML protocol. For each of these three protocols, several sets of experiments were done. The simulation parameters for each experiment as well the performance criteria have been discussed. At the end of each experiment, the results are presented, the data has been analyzed and the performance statistics have been determined. SAML was compared against another multi-granularity protocol AMGL. The AMGL protocol has not been implemented and hence was not compared experimentally with SAML but a mathematical comparison is discussed in this chapter.

SAML and AMGL have a completely different locking granularity compared to CLP and ILP. So at any given instance, it is difficult to predict how many instance, class and/or hierarchy locks are present at the system as it depends completely on transaction's workload, duration, access area, write ratio, the incoming rate of transactions in the system. So it is difficult to predict mathematically, beforehand how many locks and which locks these two protocols might need. As a result SAML was compared against ILP and CLP experimentally. Again, the key difference between SAML and AMGL is the way each protocol handles the soft locks. So an analytical comparison between them can properly indicate their variation. Hence, SAML was compared against AMGL analytically.

5.1. Locking Protocols

Three locking protocols have been used for experimental purposes. All three of the locking protocols use different granularity units for locking. While instance granularity locking uses instances as the lock granule and class granularity locking used class for the same, SAML has a self-adjusting lock granule which uses instance, class or class hierarchy as the lock granule depending on the requirement of the transaction. SAML has

been compared mathematically with another multi-granularity locking protocol AMGL which has been discussed in this section as well.

5.1.1. Instance Granularity Locking

This protocol uses instances as the lockable granule. So locks are set on instances that transactions want to access. If a read or write access is needed by a transaction on an instance of a class, an explicit shared or explicit exclusive lock has to be acquired and set on the instance. Again, if a read access on all instances of a class is needed, a lock on each of the instances has to be acquired and if a read access on all instances of a class hierarchy is needed, a lock for each of the instances of the parent class as well as all the instances of the subclasses has to obtained. In this locking, the maximum number of locks that can be set on a database is equal to the total number of instances of the classes of that database.

5.1.2. Class Granularity Locking

This protocol considers classes as granularity units. Here, locking a class implicitly locks all its instances too. To access an instance the class of that instance has to be locked explicitly in shared or exclusive mode. If a class hierarchy has to be accessed, an explicit shared or exclusive lock has to be set not only on the class but also each of the subclasses of that class and the subclasses of those subclasses. In this case, if a granularity unit for the lock remains fixed at class level and does not change during the lifetime of the transaction.

5.1.3. Adaptive Multi-granularity Locking (AMGL)

This is the only adaptive multi-granularity protocol available. In this locking protocol, instance, class and class hierarchy locking is supported. Locking a class or an instance is conducted in the same manner as in class granularity locking. Locking a class hierarchy is performed in the same manner as in multi-granularity locking [17]. They have used firm locks and soft locks. Soft locks are like intention locks [7] but this also keeps track of information about underlying objects covered by the lock Once a lock is obtained for any granularity unit, according to this protocol, the degree of granularity can be adjusted

during the lifetime of the transaction. The granularity is adjusted according to the transaction requirements by using lock escalation and de-escalation. This is done to try to reduce the lock overhead and increase the degree of concurrency amongst transactions.

5.1.4. Self Adjusting Multi-Granularity Locking (SAML)

Like existing multi-granularity locking protocols, the SAML protocol also supports instance, class and class hierarchy locking. Instances are locked in the same way as in the instance lock protocol. To lock a class, a class lock is applied like the class granularity locking protocol, which implicitly locks all the instances of the class. If a class hierarchy has to be locked, a hierarchy lock has to be set on the root class of the hierarchy. However, once a lock is obtained, the granularity of the lock can be adjusted during the transaction so that the concurrency among transactions can be increased. It is an optimistic locking, so it starts with the largest possible granularity lock possible, assuming there will be no conflict, and on the event of a conflict with another transaction, it de-escalates to the next possible granule for locking. The details of this protocol have been discussed in CHAPTER III.

5.2. Criteria for Performance Evaluation

In the simulation experiments, three criteria have been measured to evaluate the performance of each protocol.

- *Active Transactions (AT):* At a certain instance, the number of transactions in running state in the system.
- *Lock Count (LC):* It is the size of the lock table at a certain instance. It represents the number of explicit locks on various resources of the database.
- *Waiting Transactions (WT):* It is the total number of transactions that are waiting for the completion of other transactions because of the presence of some conflicting transactions in the active transaction list.

5.3. Values of Simulation Parameters

For experiment purposes, three kinds of databases have been used. The Type 1 database has a lot of instances but a lesser number of subclasses per class. The Type 2 database is

similar to the first type but the hierarchy depth of the database is double that of Type 1. The Type 3 database has the same depth as Type 1, but with more subclasses per class and lesser instances per class.

Database Parameters	Database Type		
	Type 1	Type 2	Type 3
numSubClassPerClass	3	3	10
numInstance perClass	50	50	15
numClassLevel	5	10	5

Table 5-1. Database Generation Parameters

The transactions used for the experiments can be classified into two categories depending on the number of instances they work with: a large load or a small load. Large load transactions work with 200 instances whereas small load transaction works with 20 instances. Each of these two types of transactions has been tested for two different durations. Short duration transactions exist for 2 units of time and long duration transactions stay in the system for 4 units of time. Again these transactions with different loads and different durations can have three types of access areas in the database. This signifies the area of database from where the instances, with which the transactions work with, have been chosen. These accesses can either be concentrated near to the root; near to leaf or the access area can be from the overall database.

5.4. Experiments

Depending on the types of the databases used, the experiments performed can be categorized into three major groups. For each type of databases, transactions of different workloads and various durations have been run on three different access areas of the database. T_1, T_2, T_3 are the three types of databases used. A_R, A_L, A_O defines the access areas, namely near root, near leaf and overall respectively. L_S and L_H stands for small load and heavy load and D_1, D_2, D_4 represents the different durations of transactions. D_1 uses one unit of time as the duration of transaction, D_2 and D_4 uses two and four units of time respectively. According to the notation provided here, an experiment denoted by $T_3A_OL_HD_4$ means that the experiment has been done on type-3 database with an overall

access area for transactions with heavy load and 4 units of duration. For simulation purposes, 1 unit time = 100 milliseconds has been used. Heavy load transactions consist of 200 instances whereas small load transactions use 20 instances. For each experiment, a total of 400 transactions have been run which were injected into the system using a Poisson distribution with mean rate of 10 transactions per unit of time.

The simulation results of each experiment have been presented in three charts in Appendix. The first chart of each experiment shows the number of locks held by different locking protocols at different instances of time during the lifetime of all the transactions. The second chart denotes the number of active transactions present in the database management system for different locking protocols at different instances of time during the lifetime of all the transactions. The third chart denotes the same for the number of waiting transactions that are waiting for some resources to be released. The x-axis of the charts represent the time in milliseconds and the y- axis denotes the values of lock count, active transactions and waiting transactions respectively.

5.4.1. Type-1 Database

Type 1 database has very few numbers of classes. Each class has three sub-class and 50 instances. The depth of the class hierarchy is 5. So the total number of classes in the database is only 121. Therefore the degree of conflict among the transaction is very high in this type of database.

	Lock Count (Avg)			Active Transaction (Avg)			Waiting Transaction (Avg)		
	SAML	CLP	ILP	SAML	CLP	ILP	SAML	CLP	ILP
$T_1A_OL_SD_2$	364	58	367	19	7	18	7	126	7
$T_1A_OL_SD_4$	656	61	664	33	7	33	27	160	26
$T_1A_OL_HD_2$	454	17	454	2	1	3	154	166	165
$T_1A_OL_HD_4$	460	18	455	2	1	3	169	180	186
$T_1A_IL_SD_2$	362	58	369	19	7	18	7	119	7
$T_1A_IL_SD_4$	648	62	648	33	7	33	30	155	30
$T_1A_IL_HD_2$	451	16	438	2	1	2	151	173	169
$T_1A_IL_HD_4$	453	18	457	2	1	3	166	178	186
$T_1A_RL_SD_2$	365	57	372	19	7	19	7	122	7
$T_1A_RL_SD_4$	670	61	670	34	7	34	28	156	28
$T_1A_RL_HD_2$	460	18	432	2	1	2	157	164	166
$T_1A_RL_HD_4$	449	18	457	2	1	3	160	174	185

Table 5-2. Experimental results of type-1 database

Experiments show that the performance of the SAML protocol is exactly similar to Instance Locking protocol in terms of lock count, active transactions and waiting transactions. The reason behind this is that, due to the high rate of conflicts among the transactions, SAML has to deescalate almost all the locks to instance level, which is the smallest granularity unit to accommodate more transactions to run concurrently. Whereas in the CLP protocol each transaction holds locks on a good portion of the database until it completes, which leads to the worst concurrency in the database. Although the locking overhead of CLP is very low compared to SAML or ILP, our simulations show that, in type 1 database, the total time required by the Class Lock protocol to complete all the 400 transactions is always almost three to five times than the time required by ILP or SAML protocol.

Although ILP and SAML performs similarly in terms of locking overhead, number of active and waiting transactions, the SAML protocol has additional overhead

for lock de-escalations and maintaining the class inheritance graph. Therefore, for Type1 databases the instance locking protocol will be the best choice for concurrency control and the class locking protocol will be the worst choice.

5.4.2. Type-2 Database

The type-2 database has similar characteristics like the type-1 database, but the difference is that the depth of the class hierarchy is twice that of a type-1 database. As a result it has much more number of classes than a type-1 database and hence less conflicts among the transactions.

	Lock Count (Avg)			Active Transaction (Avg)			Waiting Transaction (Avg)		
	SAML	CLP	ILP	SAML	CLP	ILP	SAML	CLP	ILP
$T_2A_OL_SD_2$	174	169	383	19	19	19	<1	<1	<1
$T_2A_OL_SD_4$	346	329	741	37	37	37	<1	1	<1
$T_2A_OL_HD_2$	707	493	3809	19	18	19	2	5	2
$T_2A_OL_HD_4$	1685	920	4374	37	34	37	6	19	6
$T_2A_LL_SD_2$	170	167	382	19	19	19	<1	<1	<1
$T_2A_LL_SD_4$	344	328	744	38	37	37	1	1	1
$T_2A_LL_HD_2$	725	502	3788	19	19	19	3	5	3
$T_2A_LL_HD_4$	1763	920	7330	37	34	37	7	20	7
$T_2A_RL_SD_2$	172	169	382	19	19	19	<1	<1	<1
$T_2A_RL_SD_4$	343	329	742	37	36	37	1	1	1
$T_2A_RL_HD_2$	706	496	3774	19	19	20	2	4	2
$T_2A_RL_HD_4$	1681	935	7331	37	35	37	6	18	6

Table 5-3. Experimental results of type-2 database

Table 5-3 presents the experimental results of the three locking protocols performed on various access areas of type-2 database for transactions with different workloads and durations. It has been observed that, for small workload the locking overhead of SAML is similar to CLP and much lower than ILP protocol whereas it has

same degree of concurrency like the ILP protocol. For large workload the number of locks held by SAML is a little bit more than CLP but again it is much lower than ILP Also SAML has same degree of concurrency like ILP as both of these protocols have almost same number of active and waiting transactions. The CLP protocol is worse than both SAML and ILP in terms of concurrency, as it has less active transactions and more waiting transactions in the system at any point of time and hence requires much more time to complete all the transactions, especially for transactions with heavy workload and long durations.

Though both SAML and ILP provide the same degree of concurrency, the locking overhead of SAML is significantly less than ILP in spite of the additional overhead of maintaining the class inheritance graph. Therefore, for this type of database, the SAML protocol will be a better choice for concurrency control for transactions with heavy workload, whereas with a small workload the CLP protocol will be the better choice.

5.4.3. Type-3 Database

Type-3 database has more number of classes and less number of instances per class. Each class has 10 subclasses and 15 instances and the depth of the class hierarchy is 5. Because of the presence of more classes and less instances per class, each lock in the class locking protocol blocks a very small portion of the database and hence the CLP protocol can provide better concurrency in this type of database than a type-1 or type-2 database.

Table 5-4 presents the experimental results of the three locking protocols performed on various access areas of type-3 databases for transactions with different workloads and durations. It is observed that for transactions with small workload all the three protocols provide a similar degree of concurrency as the conflicts among the transactions is very low. The ILP protocol has a high locking overhead, whereas the CLP and SAML protocol have a similar locking overhead which is much less compared to the ILP protocol. As the SAML protocol has additional overhead for maintaining the Class Inheritance Graph and lock de-escalations, for type-3 databases the CLP protocol will be the better choice for small load transactions.

	Lock Count (Avg)			Active Transaction (Avg)			Waiting Transaction (Avg)		
	SAML	CLP	ILP	SAML	CLP	ILP	SAML	CLP	ILP
$T_3A_OL_SD_2$	180	173	381	19	19	19	<1	1	<1
$T_3A_OL_SD_4$	358	329	740	37	36	37	1	4	1
$T_3A_OL_HD_2$	1900	657	3419	17	11	17	27	90	26
$T_3A_OL_HD_4$	2661	749	4794	21	13	24	53	142	82
$T_3A_LL_SD_2$	180	175	382	19	19	19	<1	1	<1
$T_3A_LL_SD_4$	361	342	747	37	40	37	1	4	1
$T_3A_LL_HD_2$	1862	667	3410	16	11	17	28	88	29
$T_3A_LL_HD_4$	2770	751	4725	21	13	24	59	138	85
$T_3A_RL_SD_2$	180	175	385	19	19	19	<1	1	<1
$T_3A_RL_SD_4$	364	341	744	37	37	37	1	5	1
$T_3A_RL_HD_2$	1801	356	3384	16	11	17	29	90	29
$T_3A_RL_HD_4$	2725	735	4677	21	12	24	53	138	84

Table 5-4. Experimental results of type-3 database

Transactions with a heavy workload have much more conflicts, especially for long duration transactions. So the CLP protocol provides less concurrency and takes much more time to finish all the 400 transactions compared to the ILP or the SAML protocol. The locking overhead of SAML in this case is higher than the CLP protocol, but it is much less than the ILP protocol. Although the degree of concurrency in the SAML protocol in this scenario is not exactly similar to the ILP protocol like the other cases, it is very close to ILP protocol. Therefore, considering concurrency and locking overhead, the SAML protocol will be the better choice for type-3 databases for transactions with heavy workload.

5.5. Experiment Summary

Table 5-5 lists the best and the worst locking protocol among SAML, ILP and CLP protocols on different types of databases for transactions of different workloads, considering both locking overhead and the degree of concurrency. The experiments in section 5.4 show that the performance of the locking protocols does not change

significantly in different access areas of the database. So the access area criterion has been excluded from the summary in Table 5-5. Also if the duration of the transactions increases, the relative performance of the locking protocols remains the same. Hence the duration of a transaction has also been excluded from the summary table. Only the workload of the transaction significantly affects the concurrency and locking overhead of the locking protocols. For type-1 databases the ILP protocol has the best performance as it has a very high degree of conflicts among the transactions. For type-2 and type-3 databases the SAML protocol has the best performance for heavy workload whereas the CLP protocol performs well for a small workload. In all cases the performance of the SAML protocol is never the worst.

Database type	Workload	Best	Worst
Type-1	Small	ILP	CLP
	Heavy	ILP	CLP
Type-2	Small	CLP	ILP
	Heavy	SAML	ILP
Type-3	Small	CLP	ILP
	Heavy	SAML	CLP

Table 5-5. Best and Worst Protocol of Experiments

5.6. SAML vs. AMGL

Both SAML and AMGL are adaptive multi-granularity locking protocols for concurrency control in object oriented databases. The SAML protocol can be considered as a simplified version of AMGL protocol. AMGL has treated composite objects differently than other database objects and it assumes the presence of schema modifications but SAML does not handle composite objects differently than regular instance objects and assumes that there is no schema change in the database.

The SAML protocol is similar to AMGL in terms of the number of lock de-escalations and it provides the same degree of concurrency. Both of these protocols require exactly the same number of lock de-escalations to lock any object in the database. But to lock any class or instance explicitly in any mode, the AMGL protocol must apply corresponding soft locks explicitly on all the ancestors of that class or instance and it

keeps all the soft locks in the lock table along with the explicit firm locks. This makes the AMGL protocol even worse than the ILP protocol in terms of locking overhead in some cases [5]. SAML, on the other hand, takes a different approach to deal with soft locks. To maintain the soft locks it uses a Class Inheritance Graph which is static in size. The color of each node of the graph denotes the soft lock present on that object in the database at that point. The use of CIG significantly reduces the number of locks and hence the locking overhead compared to AMGL.

To lock any class or instance at level l, the AMGL protocol needs (l–1) explicit soft locks along with the explicit firm lock. Whereas to lock any class or instance, SAML needs to apply a new color on the corresponding node in CIG (only for class or hierarchy locks) and update the color of its immediate ancestor node. It does not require more than one entry in the lock table for a lock. This significantly reduces the locking overhead in SAML. For example, if n different transactions simultaneously lock an object o at level l of the class hierarchy, then according to the AMGL protocol there will be $n\times(l–1)$ soft locks and n firm locks in the lock table. Hence the total number of locks for AMGL protocol in this scenario will be $(n\times (l–1) + n) = n\times l$, which is directly proportional to the level of the class in the hierarchy. Whereas the number of locks required by SAML protocol is only n, this is independent of the level of the object in the hierarchy. So, the space complexity will be the main comparison parameter here as the larger the lock table size, the more it will need to search the lock table and time complexity will vary accordingly. The space complexity associated with the AMGL protocol would be O(nxl) and for SAML it would be O(n). This proves that the SAML protocol will have less locking overhead than AMGL in all cases.

CHAPTER VI

CONCLUSION AND RECOMMENDED FUTURE WORK

In the preceding chapters, to improve the concurrency in Object Oriented Databases, a self-adjusting multi-granularity locking protocol (SAML) has been presented which facilitates choosing appropriate locking granularity according to the requirement of the transactions and encompasses less overhead in various situations compared to some of the existing protocols. This chapter gives a summary of the protocol, highlights the contribution and recommends some directions where the future research efforts could be made.

The most attractive feature of SAML is that it maximizes the parallelism of transactions while keeping the locking overhead as low as possible. This protocol uses three levels of granularity for locking, namely instance lock, class lock and class hierarchy lock. SAML uses lock de-escalation which enables the lockable granules to be adjusted automatically during the lifetime of a transaction. This optimistic locking protocol uses conservative two-phase locking to avoid deadlocks. This protocol uses mainly three types of locks: explicit locks, implicit locks and soft locks. SAML consists of three components. First is the ten lock modes on different granules; second, a 10x6 compatibility matrix which is used to decide whether two lock modes are compatible to be set on the same object and the third is a complete protocol to guide how lock requests are issued. The basic idea is to lock the largest granularity unit is at the beginning of a transaction even if most of the objects being locked are not necessary at first. Later, if there is any conflict from other transactions, lock de-escalation is performed. Each conflicting transaction then would lock the next higher granularity unit that is not in conflict. So when locks of two transactions conflict, both locks keep on reducing their granule size until the conflict is resolved. Therefore, the protocol adjusts itself to the transaction requirements by using multiple granularity units to provide maximum concurrency but keeping minimum locking overhead.

SAML protocol was compared against instance, class and class hierarchy locking experimentally for three different types of databases and transactions having

various access areas, different workload and duration. Out of all the situations it was found that SAML performs the best, in terms of better concurrency and less overhead, when the transactions are long-duration transactions with heavy workload. SAML was compared analytically with another adaptive multi-granularity protocol called AMGL. Both of these protocols provide same degree of concurrency. Both of these protocols require exactly the same number of lock de-escalations to lock any object in the database. But to lock any class or instance explicitly in any mode, the AMGL protocol must apply corresponding soft locks explicitly on all the ancestors of that class or instance and it keeps all the soft locks in the lock table along with the explicit firm locks. This makes AMGL protocol even worse than ILP protocol in terms of locking overhead in some cases and SAML uses a class inheritance graph which significantly reduces the number of locks and hence the locking overhead compared to AMGL.

One possible improvement of SAML in terms of reducing locking overhead can be achieved if we can figure out beforehand whether a lock de-escalation is going to result in resolving the existing lock conflict or not. If it does, then only the de-escalation would be allowed to take place. This scenario is most critical for a lock de-escalation of a class lock to instance locks.

We have only considered a single inheritance scenario in our protocol but SAML can easily be extended to incorporate multiple inheritances. In SAML, while locking an object explicitly, soft locks are applied on all the ancestors of that object and the class inheritance graph has been used to navigate from that object to all its ancestors. So, if multiple-inheritance is present in a database, this graph would be able to find all the ancestors through multiple parents and apply a soft lock on them.

We have not considered composite objects separately from primitive objects in SAML. If there is a composite object in the system, then each component object of the composite object has to be locked separately as individual primitive objects. Recognizing composite objects as being different could be investigated for the existing SAML protocol.

We have assumed that there will be no schema modification of the database. But in reality, that might not be the case always. So, including the effects of schema modification in SAML protocol could be an interesting area to explore.

APPENDIX A

A.1. Graphs of Experimental Results

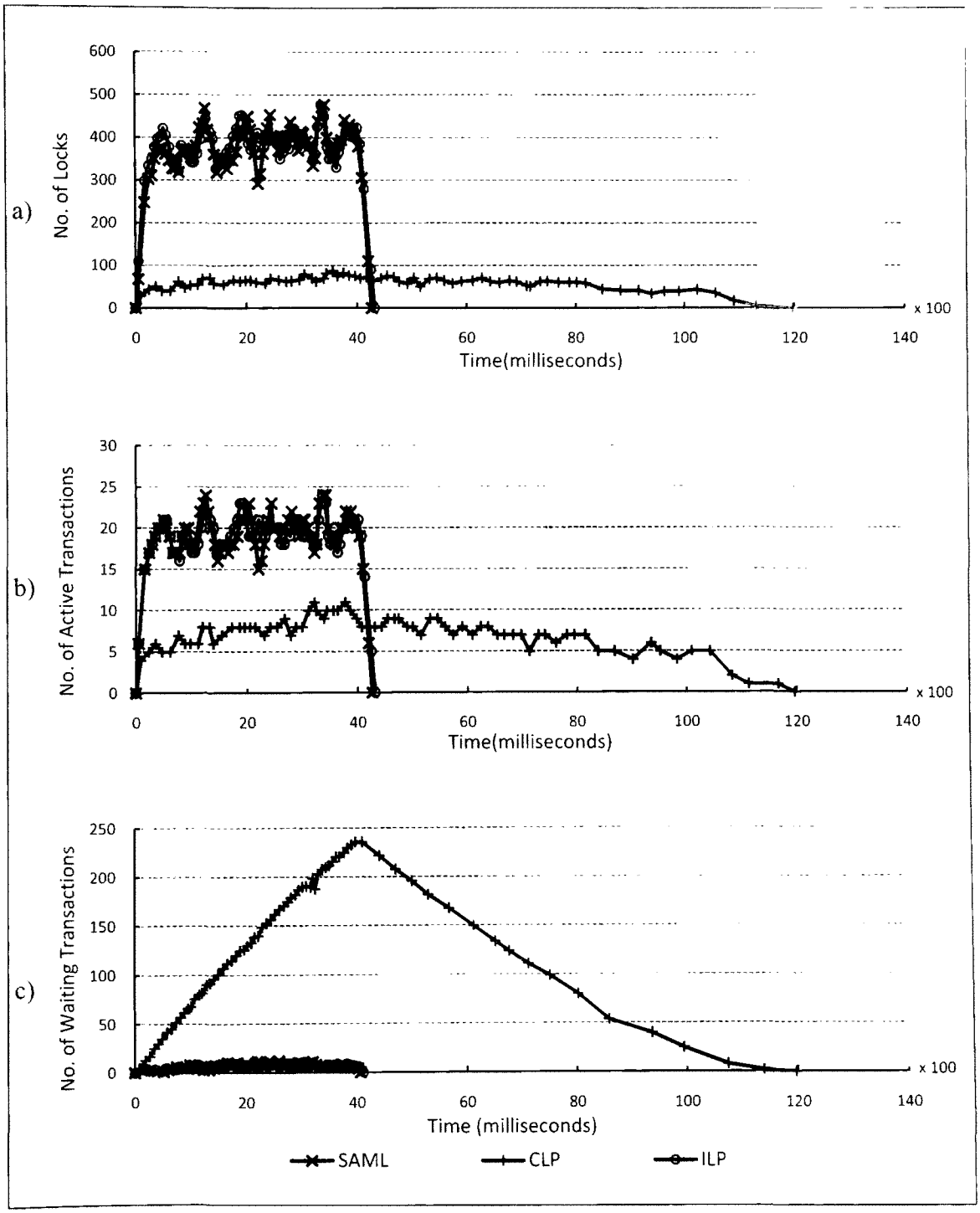

Figure 6-1. Result of experiment $T_1A_0L_SD_2$

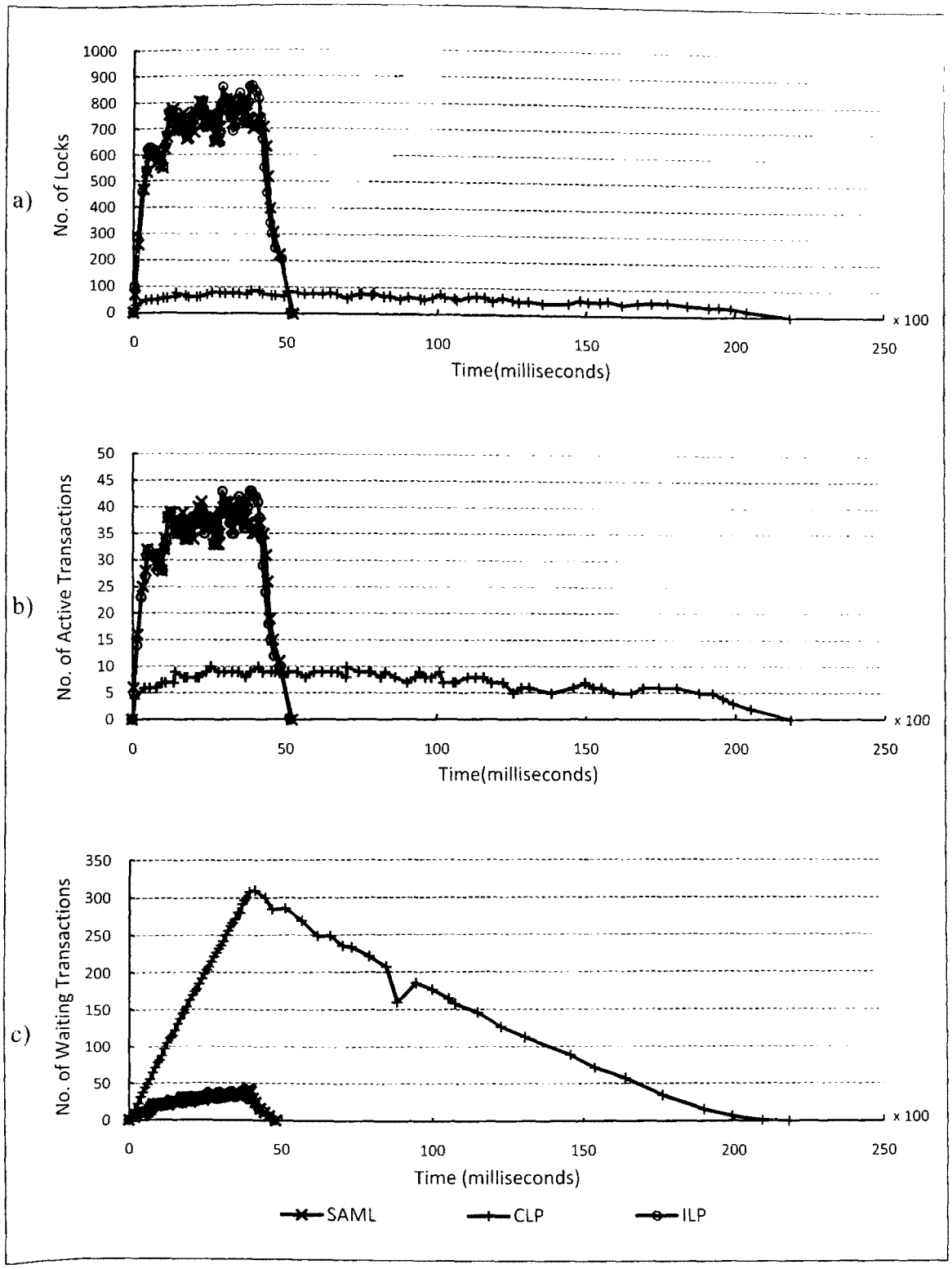

Figure 6-2. Result of experiment $T_1A_OL_SD_4$

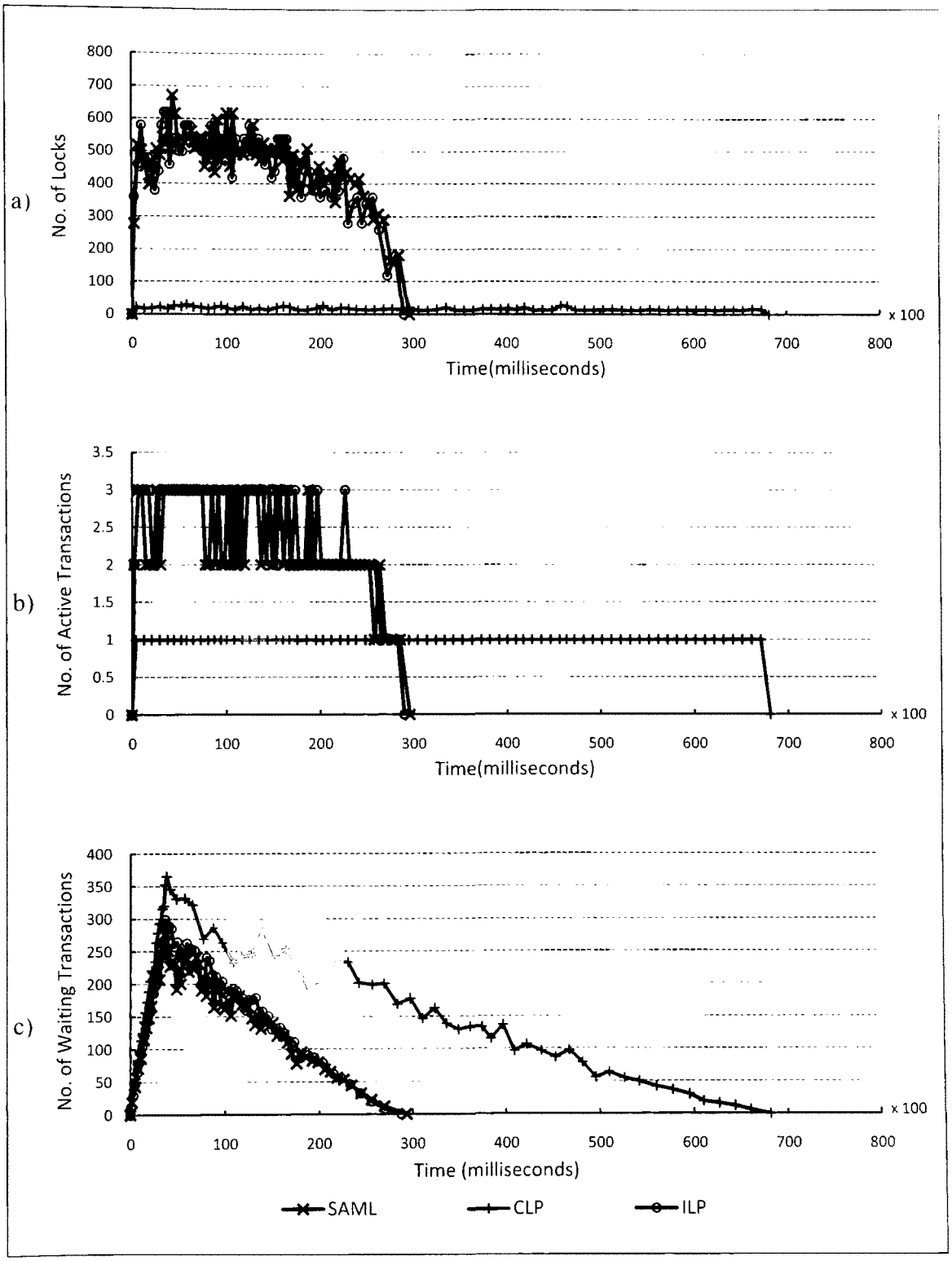

Figure 6-3. Result of experiment $T_1A_OL_HD_2$

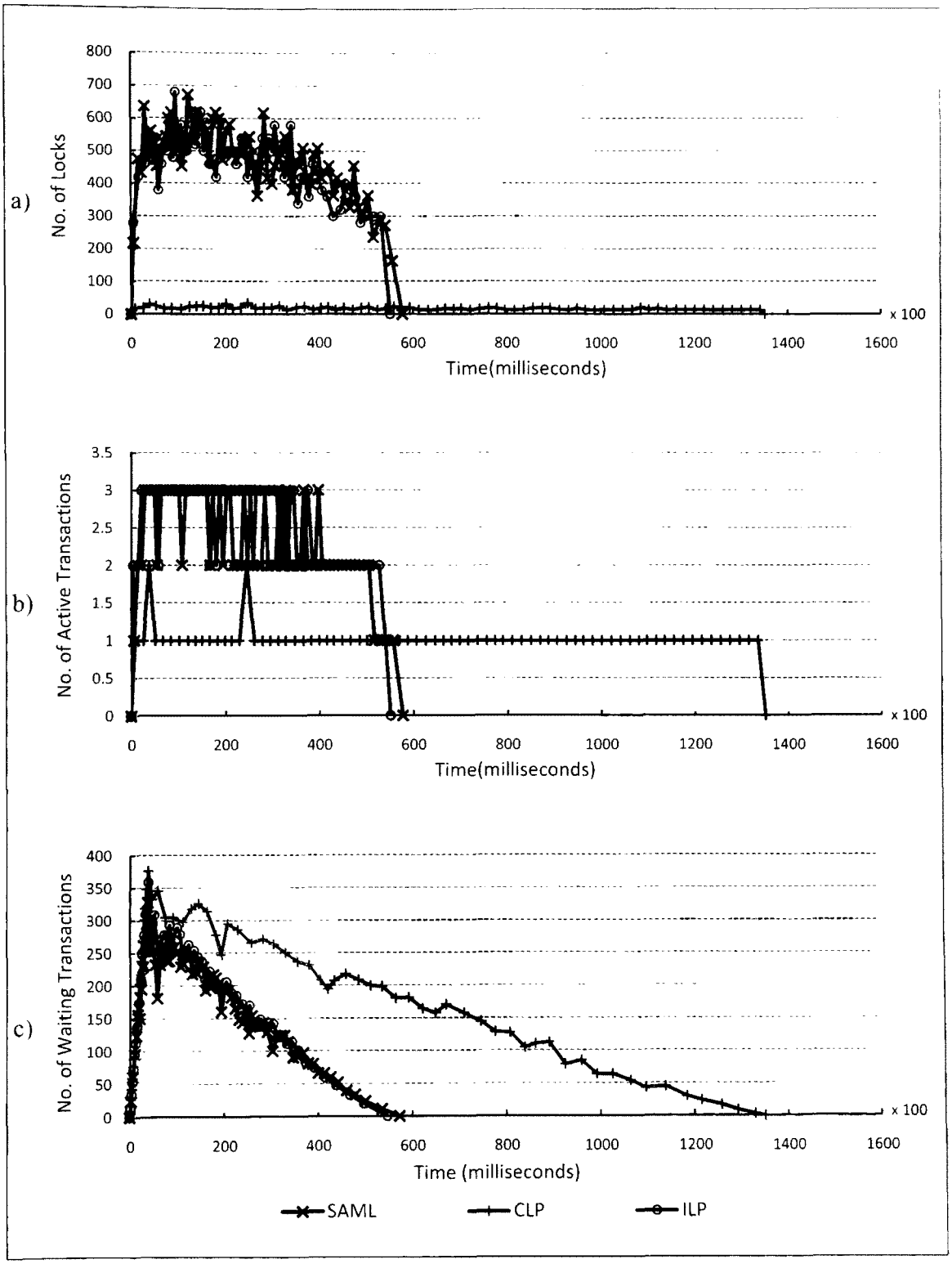

Figure 6-4. Result of experiment $T_1A_OL_HD_4$

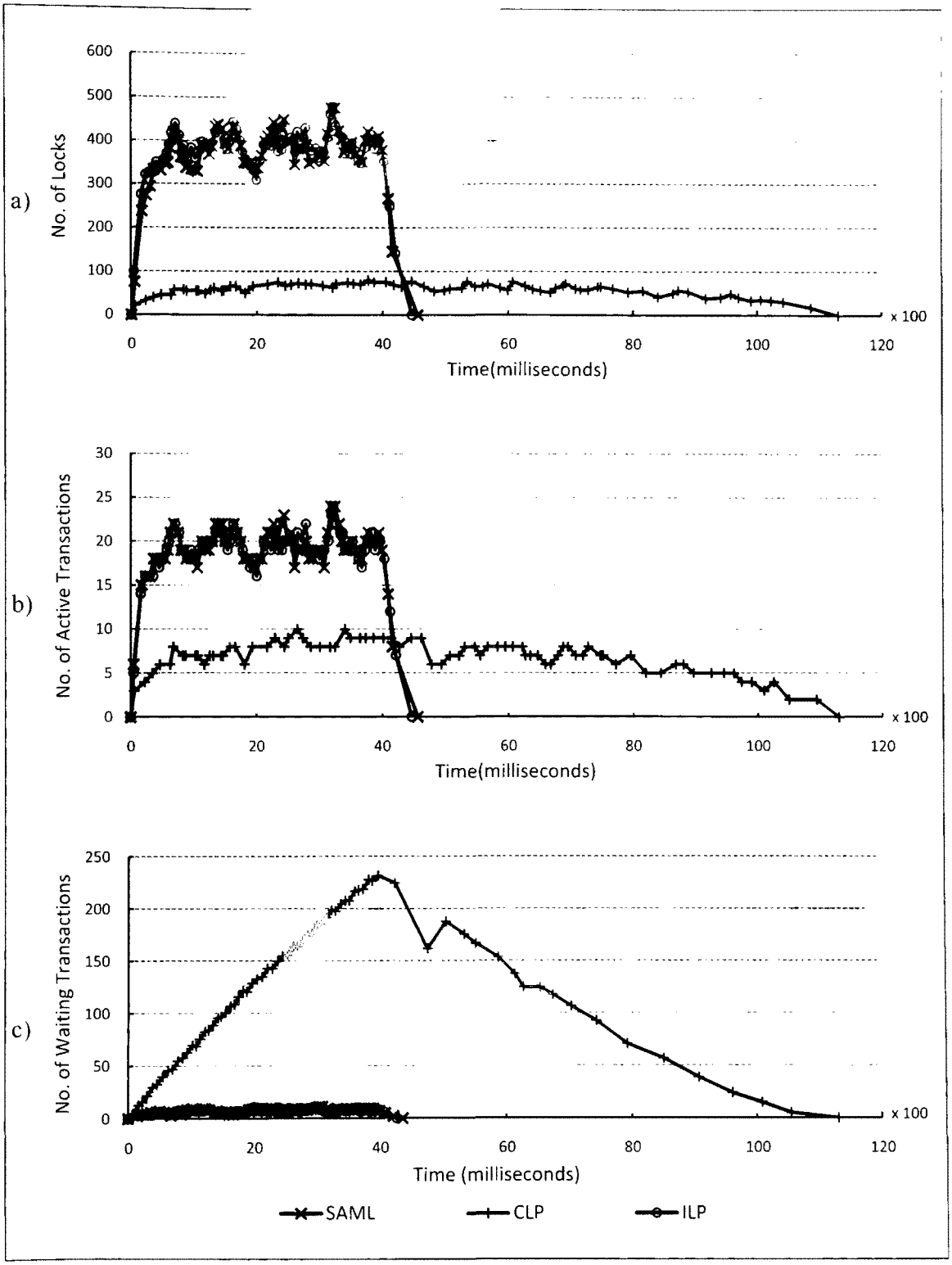

Figure 6-5. Result of experiment $T_1A_LL_SD_2$

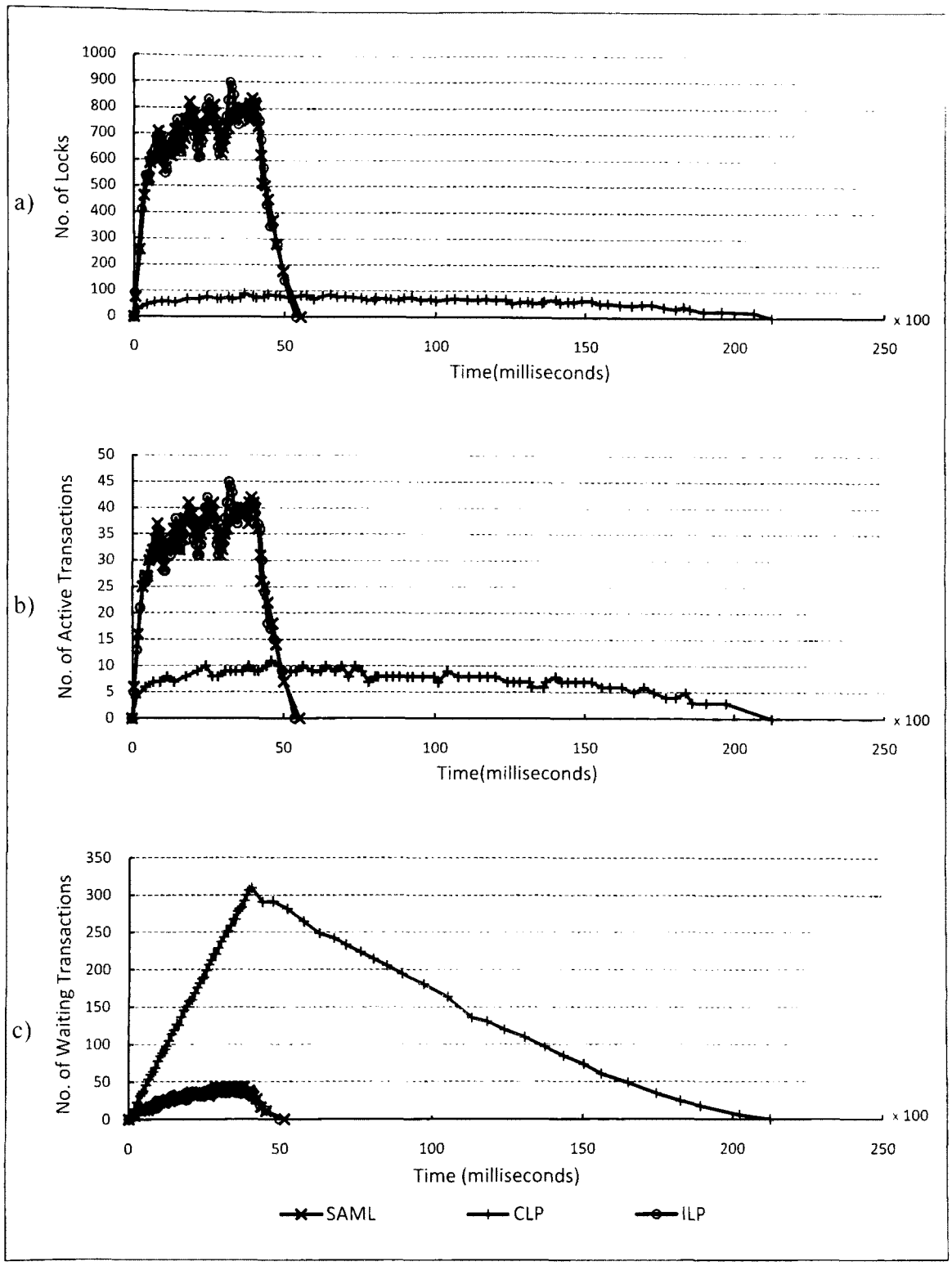

Figure 6-6. Result of experiment $T_1A_LL_SD_4$

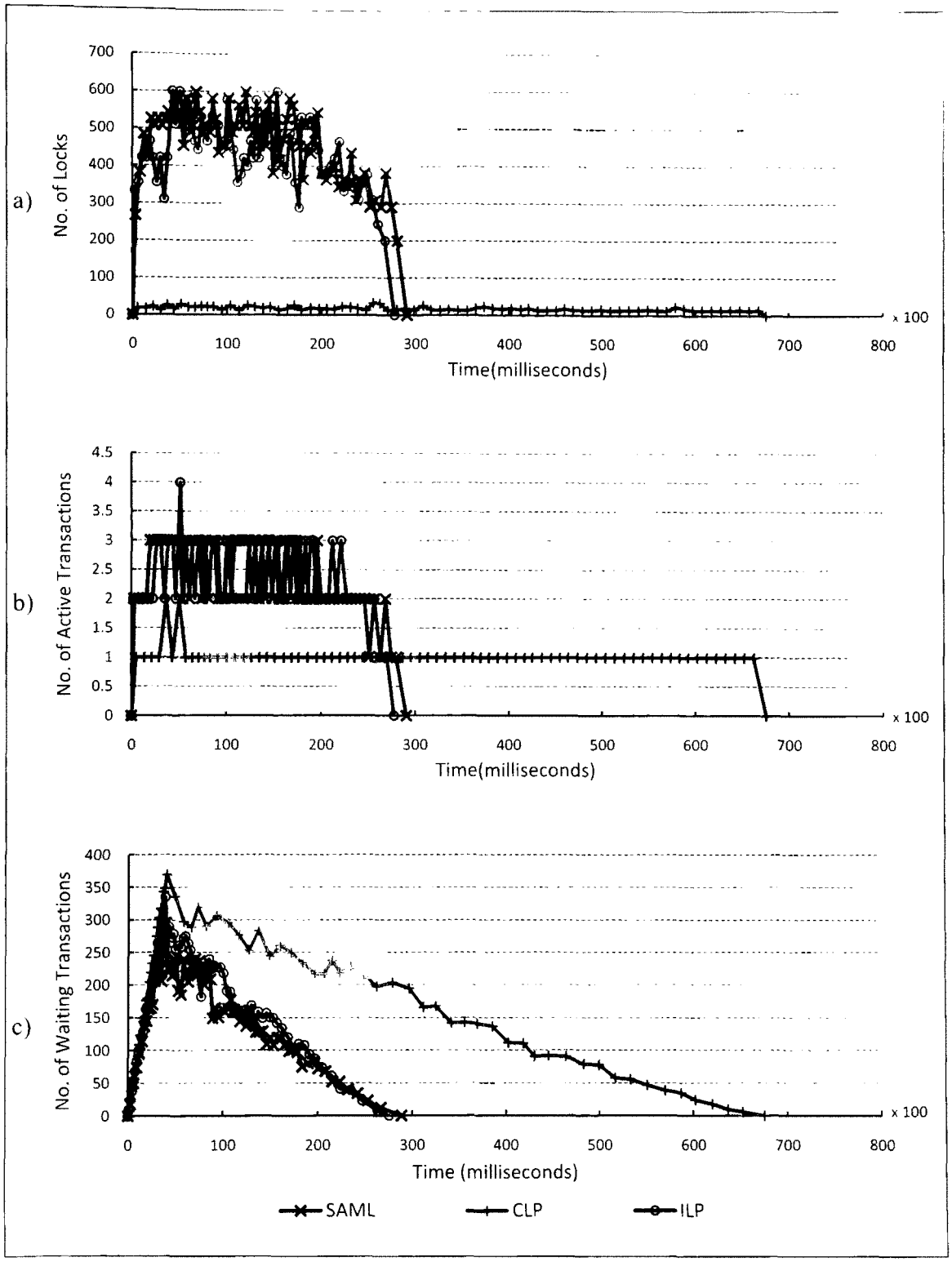

Figure 6-7. Result of experiment $T_1A_LL_HD_2$

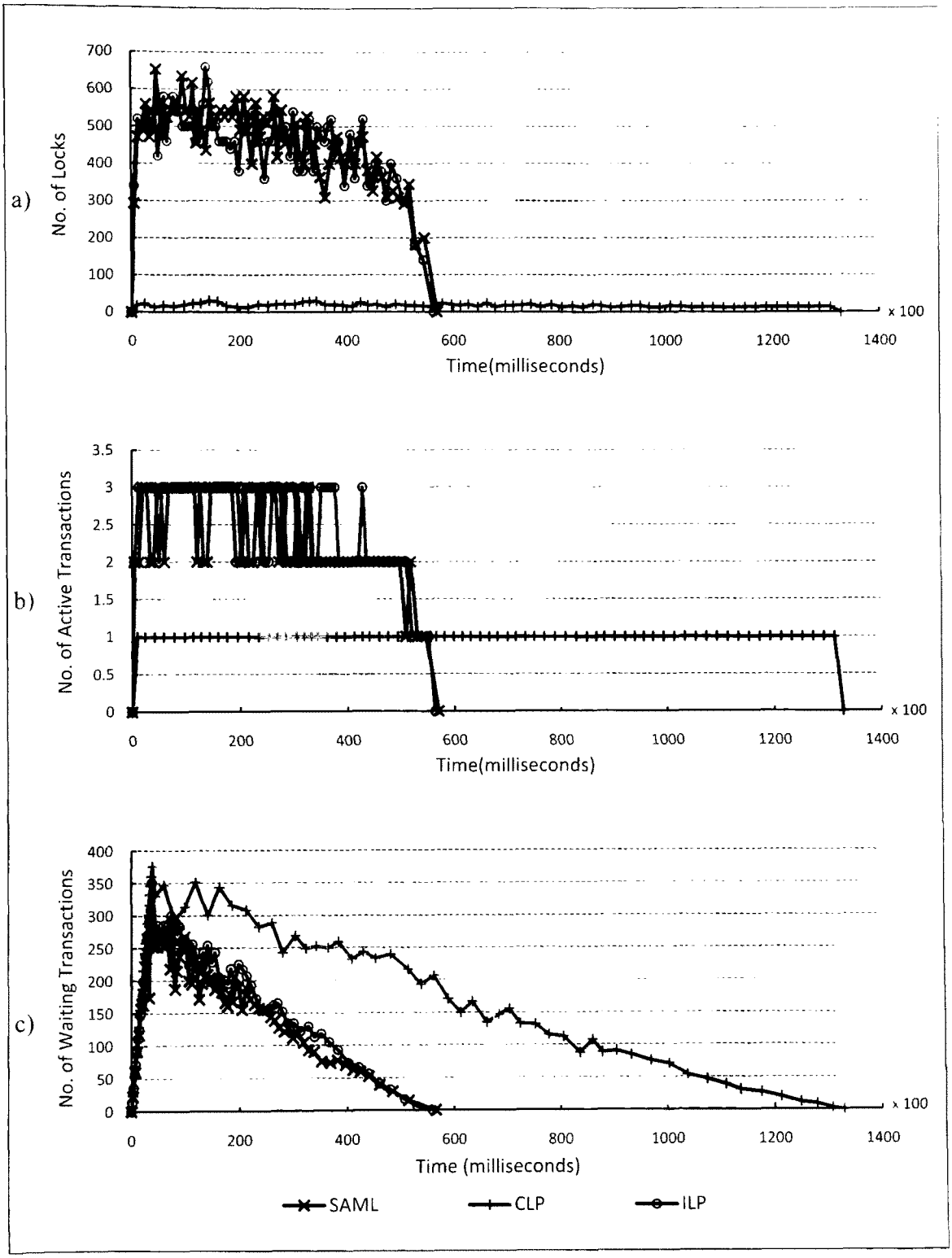

Figure 6-8. Result of experiment $T_1A_LL_HD_4$

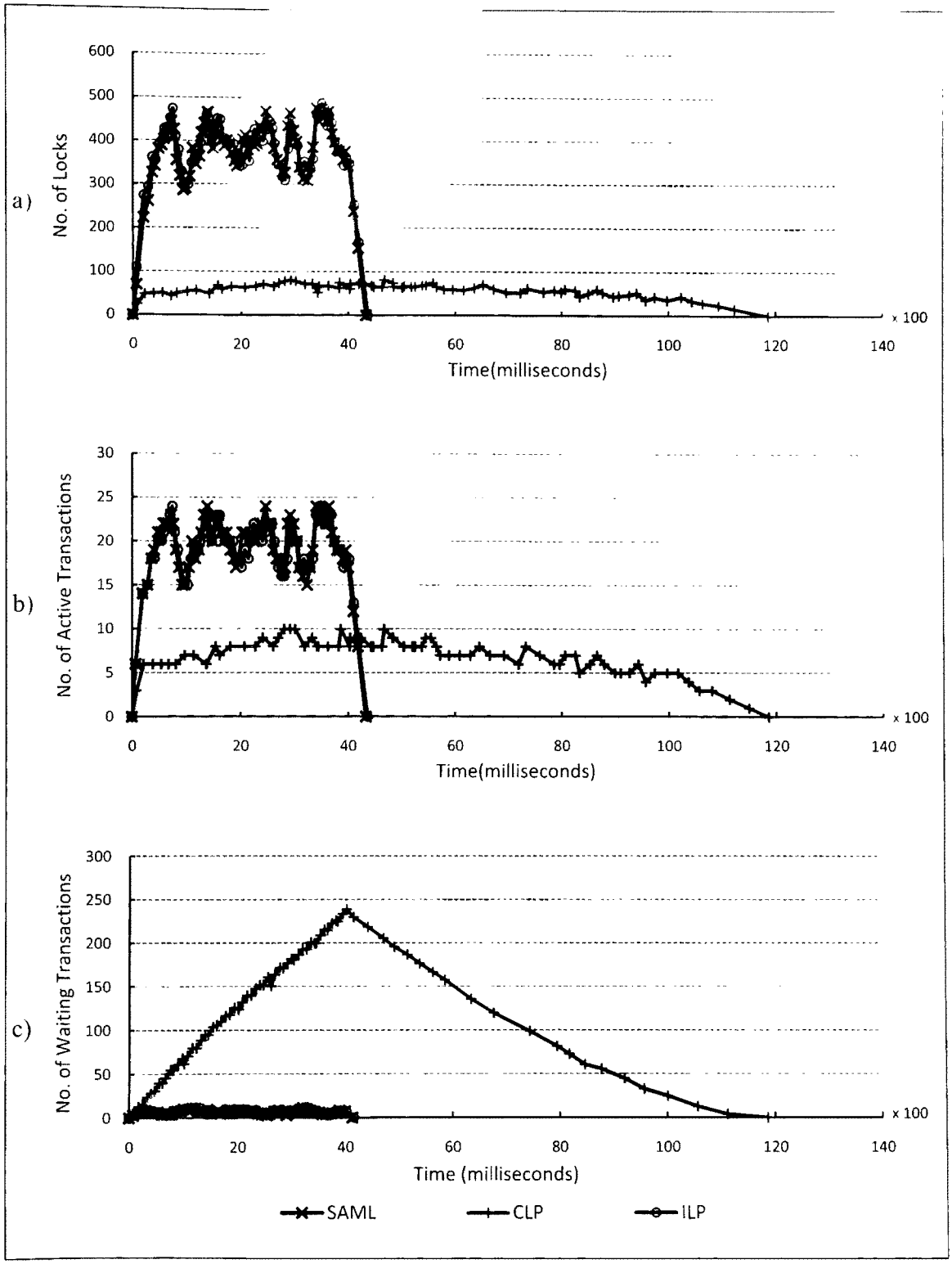

Figure 6-9. Result of experiment $T_1A_RL_SD_2$

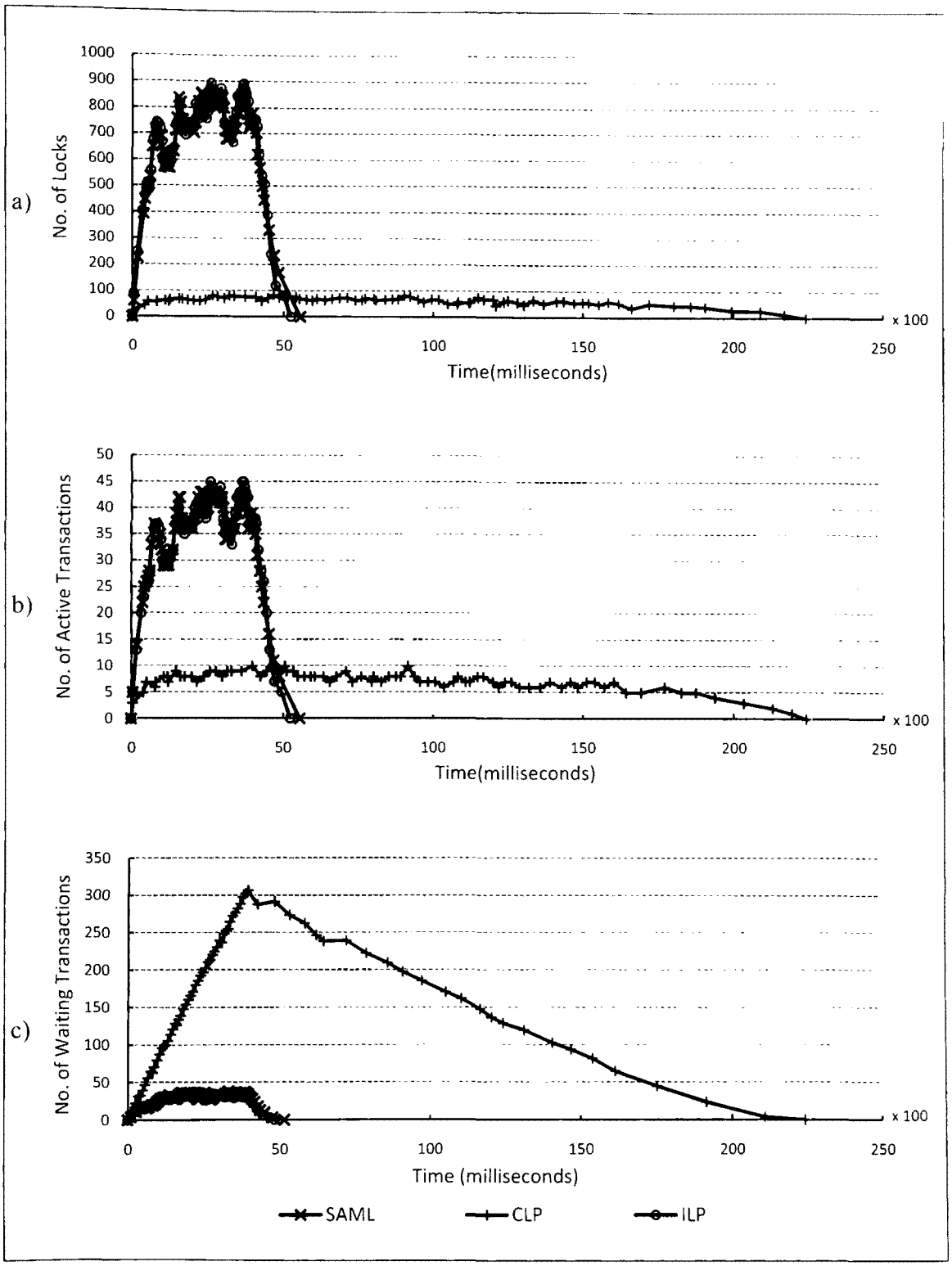

Figure 6-10. Result of experiment $T_1A_RL_SD_4$

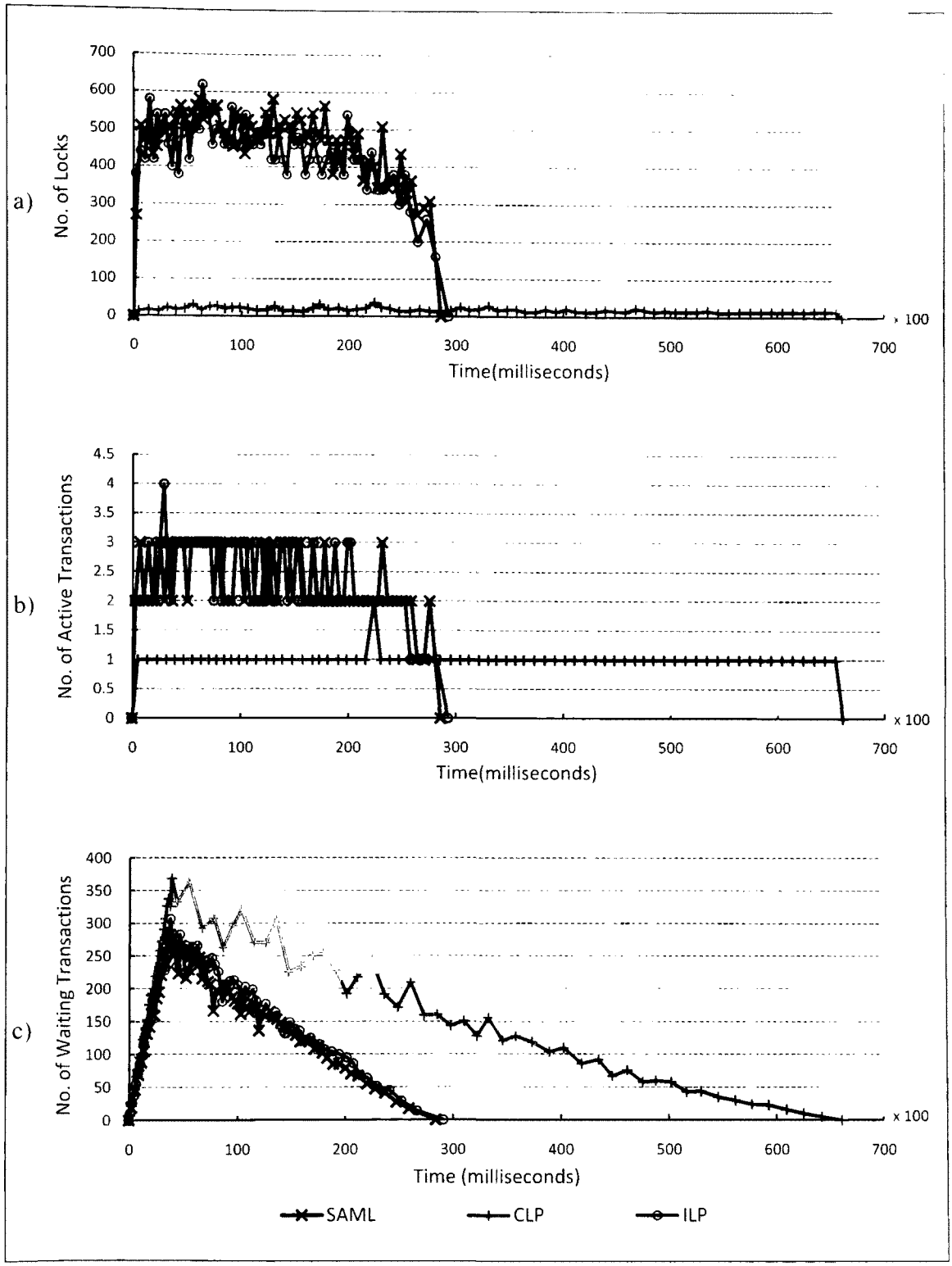

Figure 6-11. Result of experiment $T_1A_RL_HD_2$

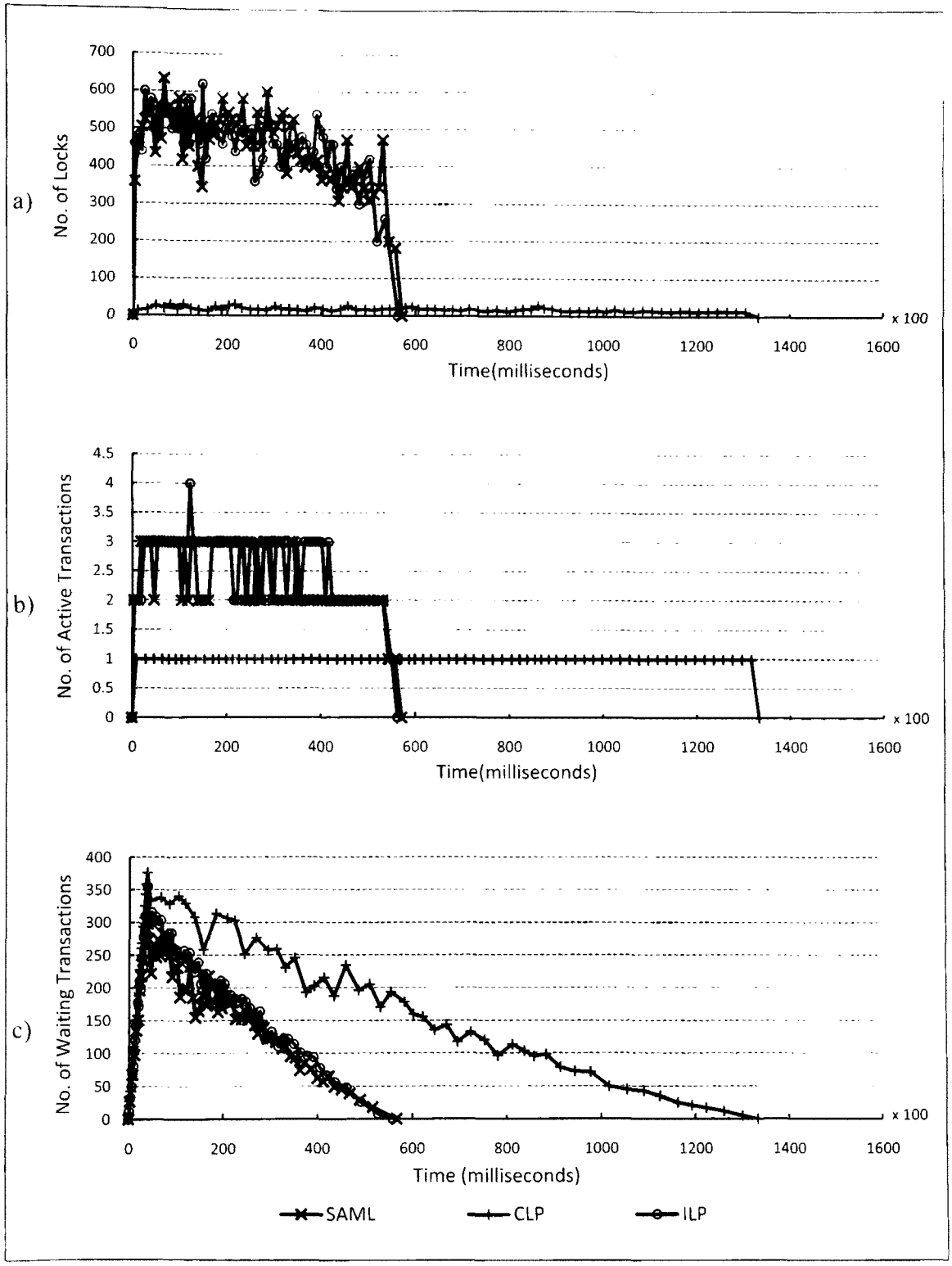

Figure 6-12. Result of experiment $T_1A_RL_HD_4$

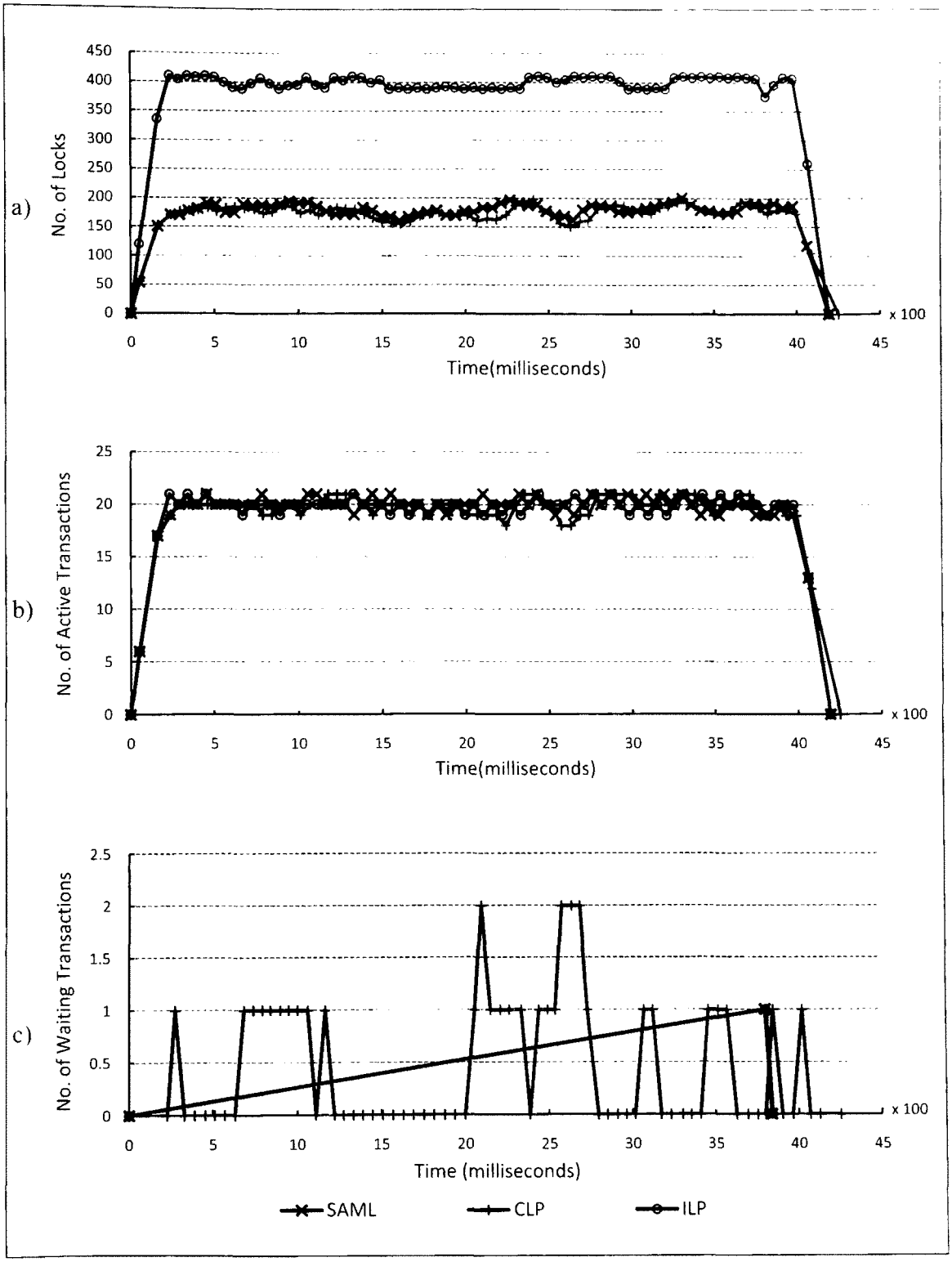

Figure 6-13. Result of experiment $T_2A_OL_SD_2$

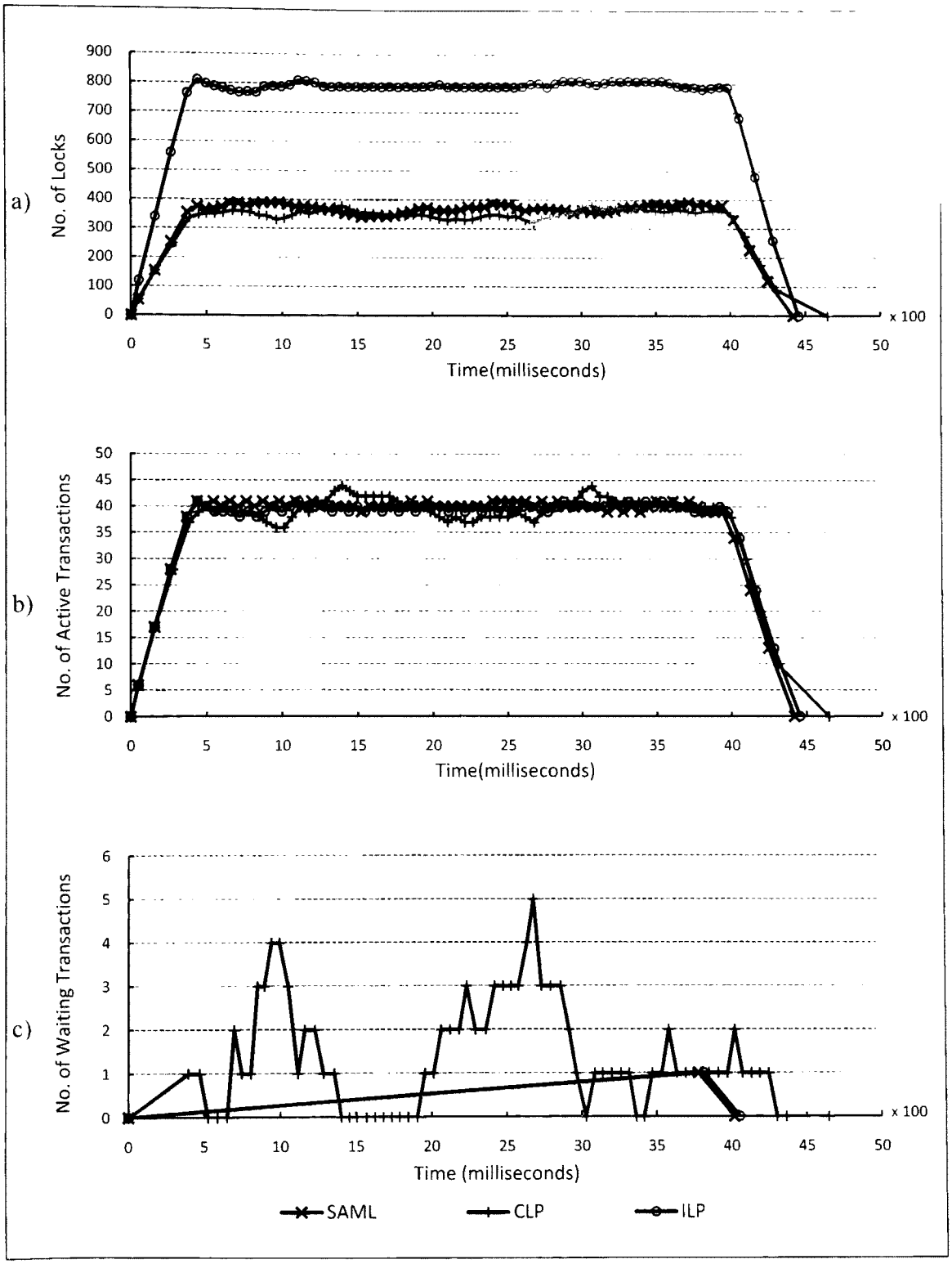

Figure 6-14. Result of experiment $T_5A_{10}L_5D_4$

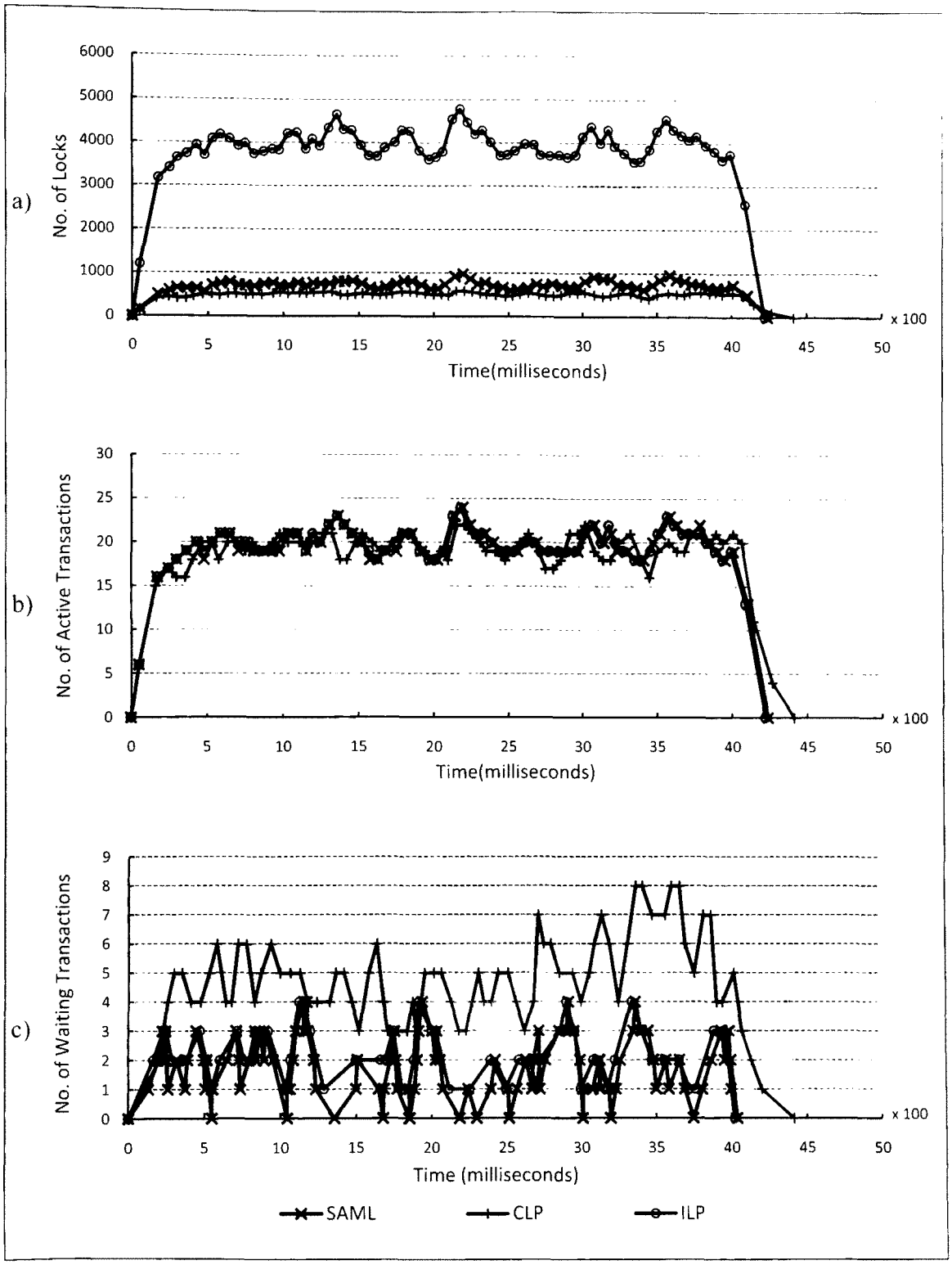

Figure 6-15. Result of experiment $T_2A_OL_HD_2$

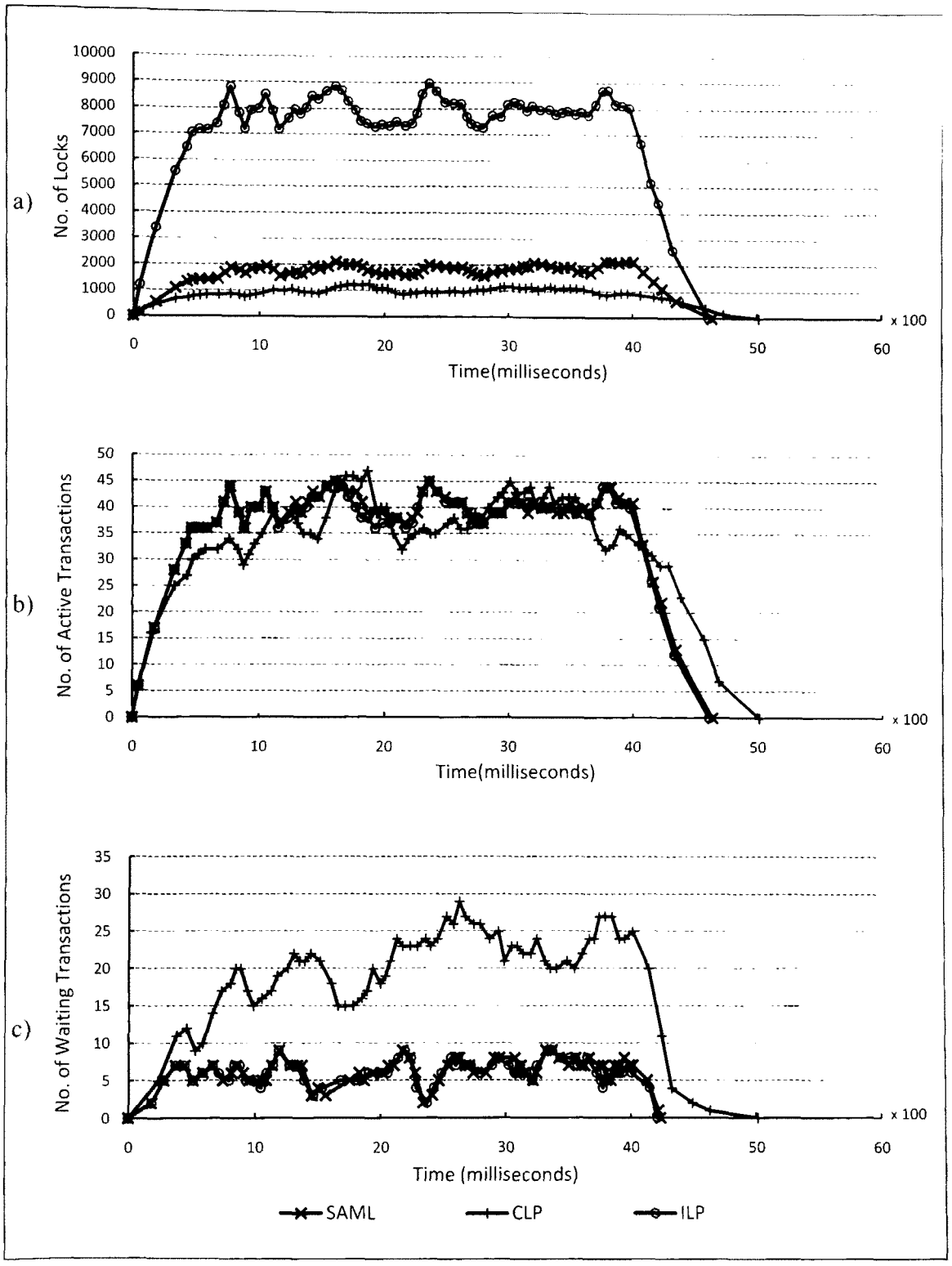

Figure 6-16. Result of experiment $T_2A_OL_HD_4$

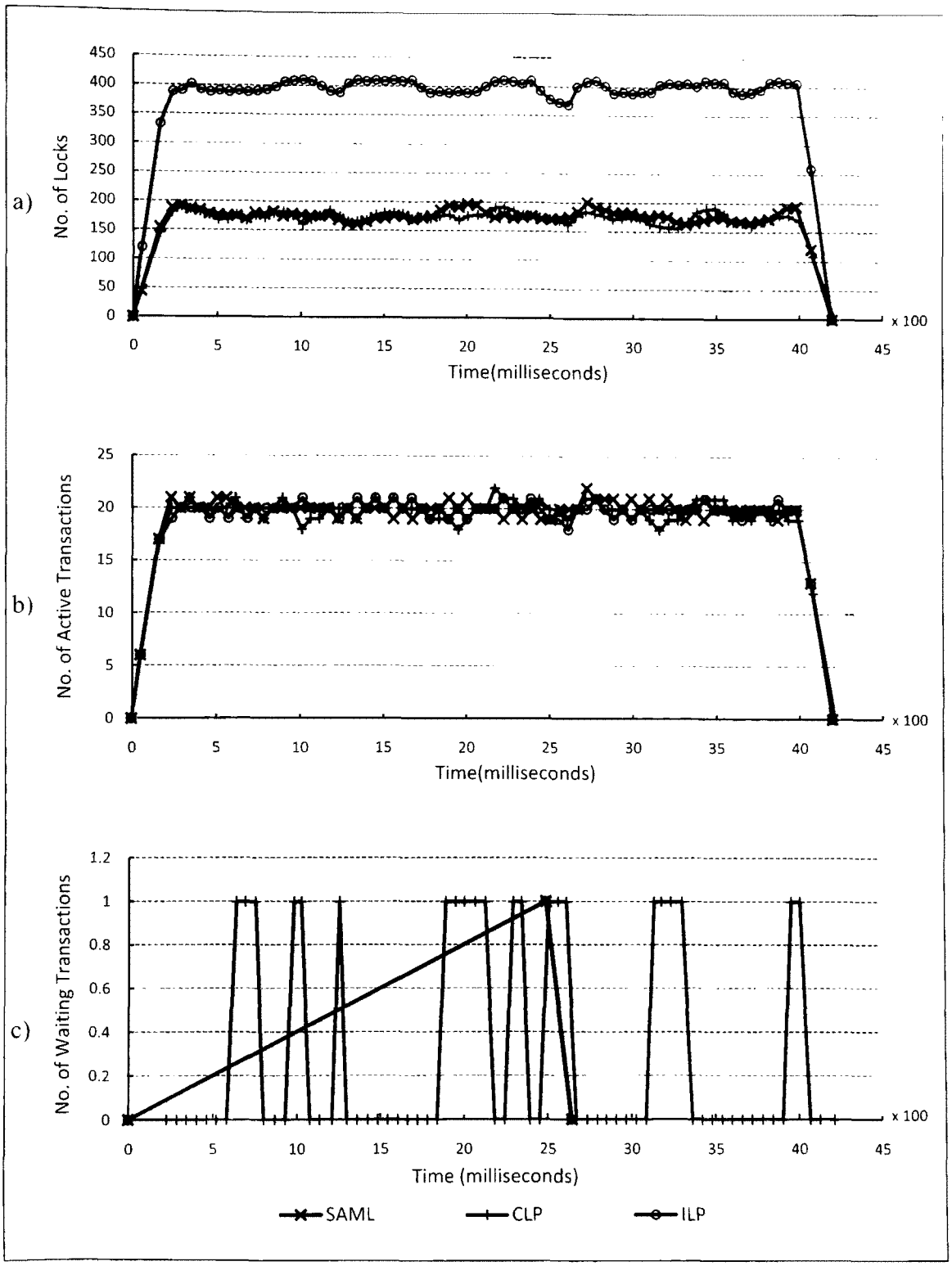

Figure 6-17. Result of experiment $T_2A_1L_SD_2$

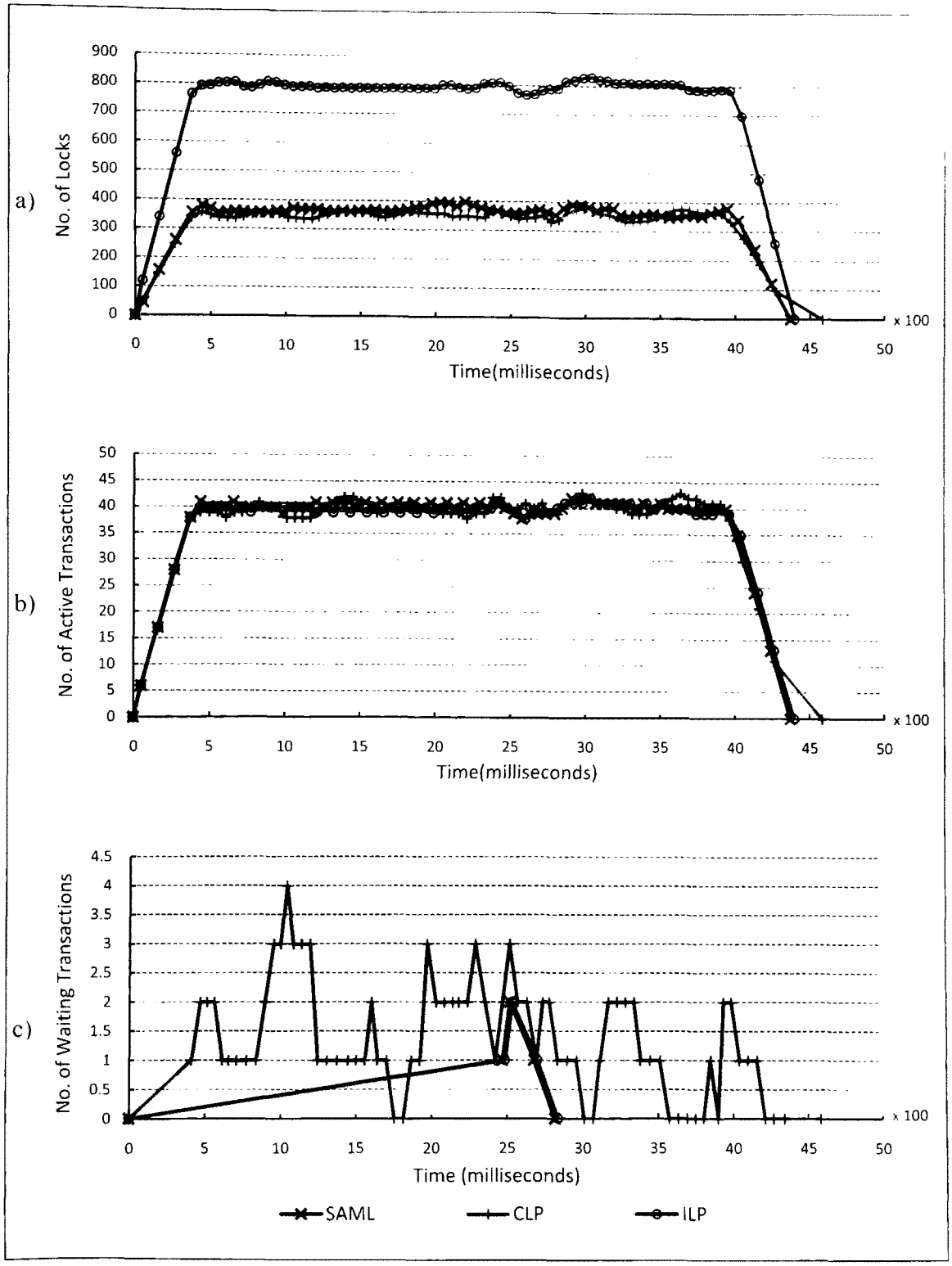

Figure 6-18. Result of experiment $T_2A_LL_SD_4$

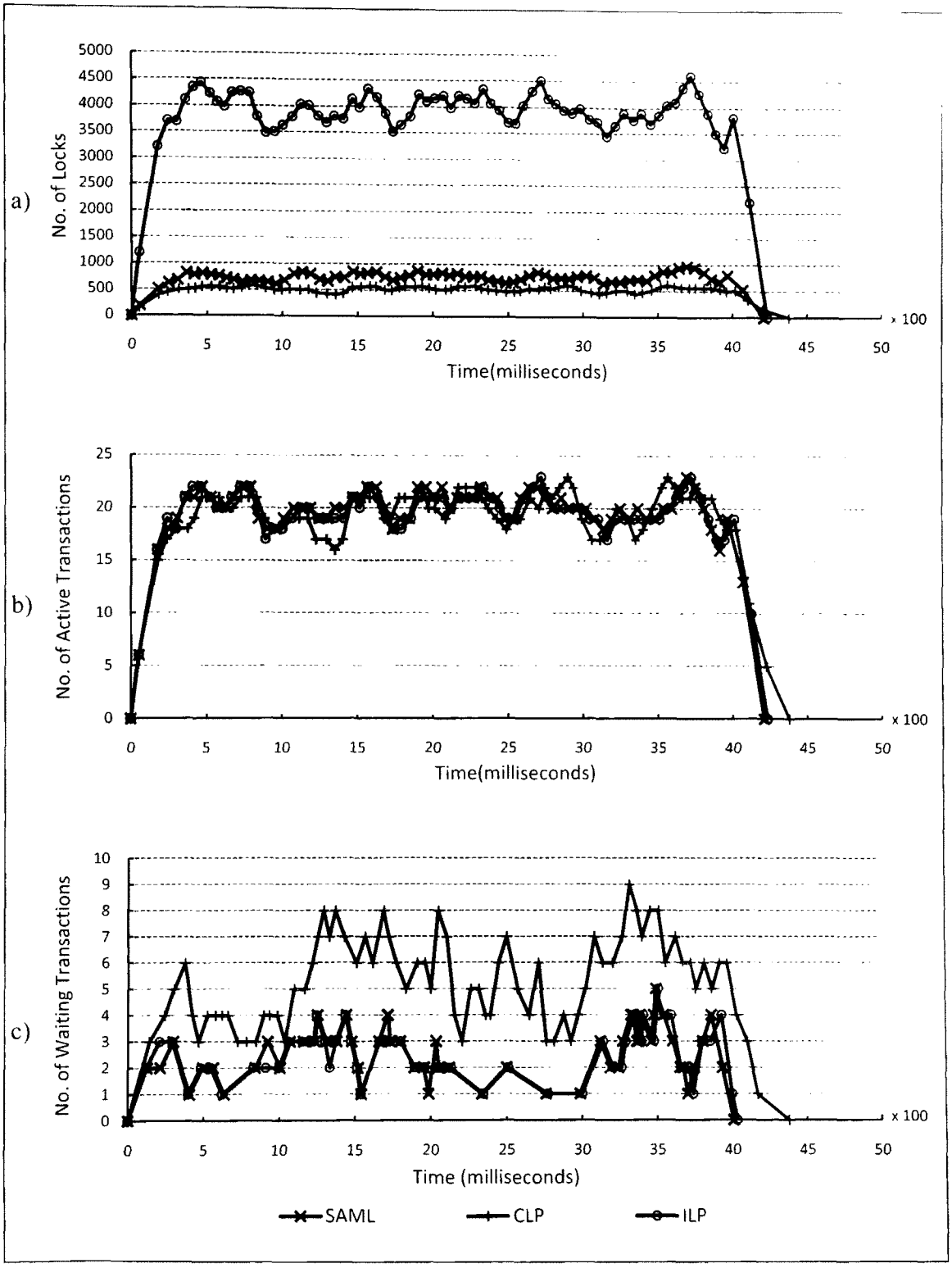

Figure 6-19. Result of experiment $T_2A_LL_HD_2$

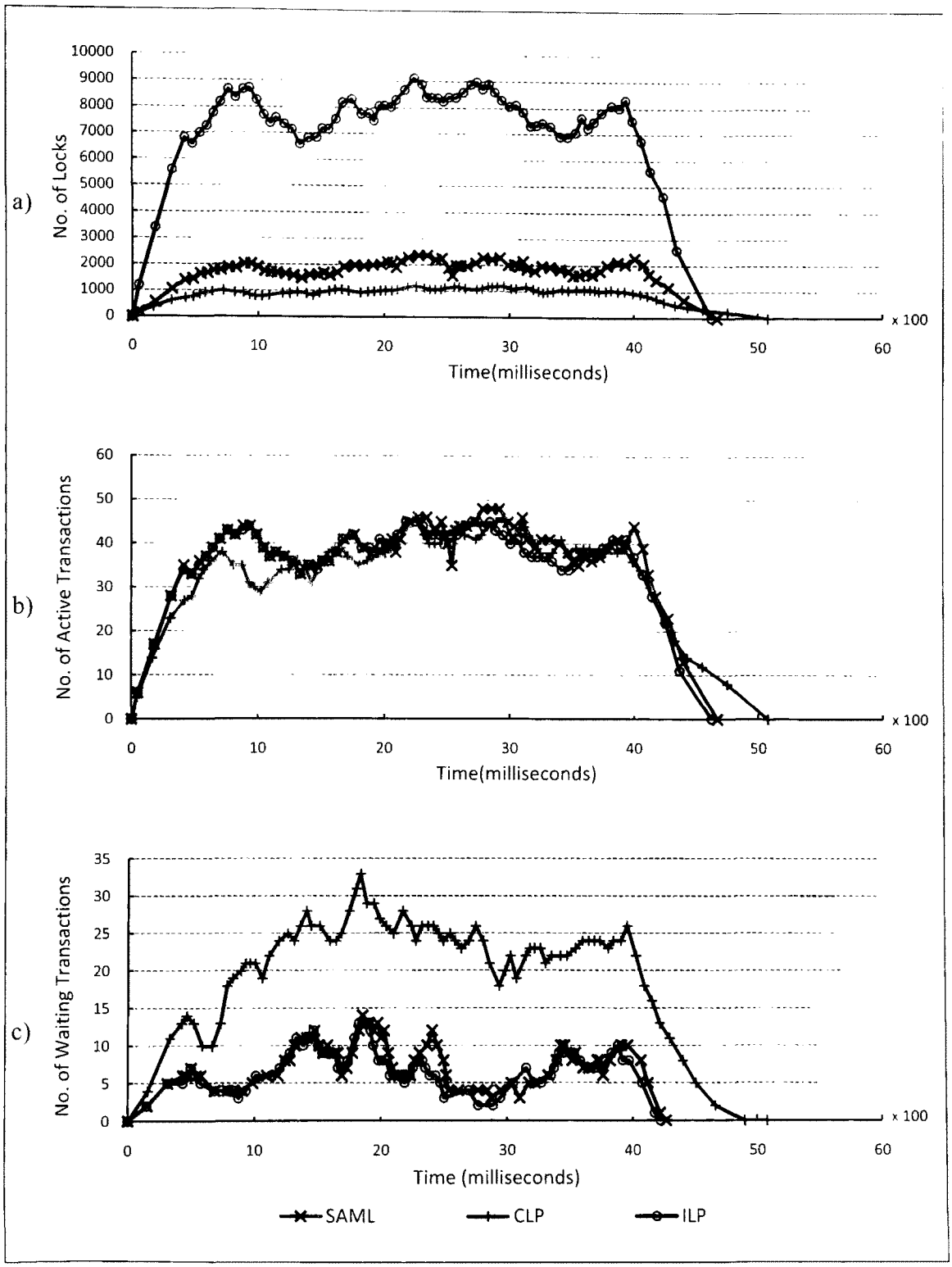

Figure 6-20. Result of experiment $T_2A_LL_HD_4$

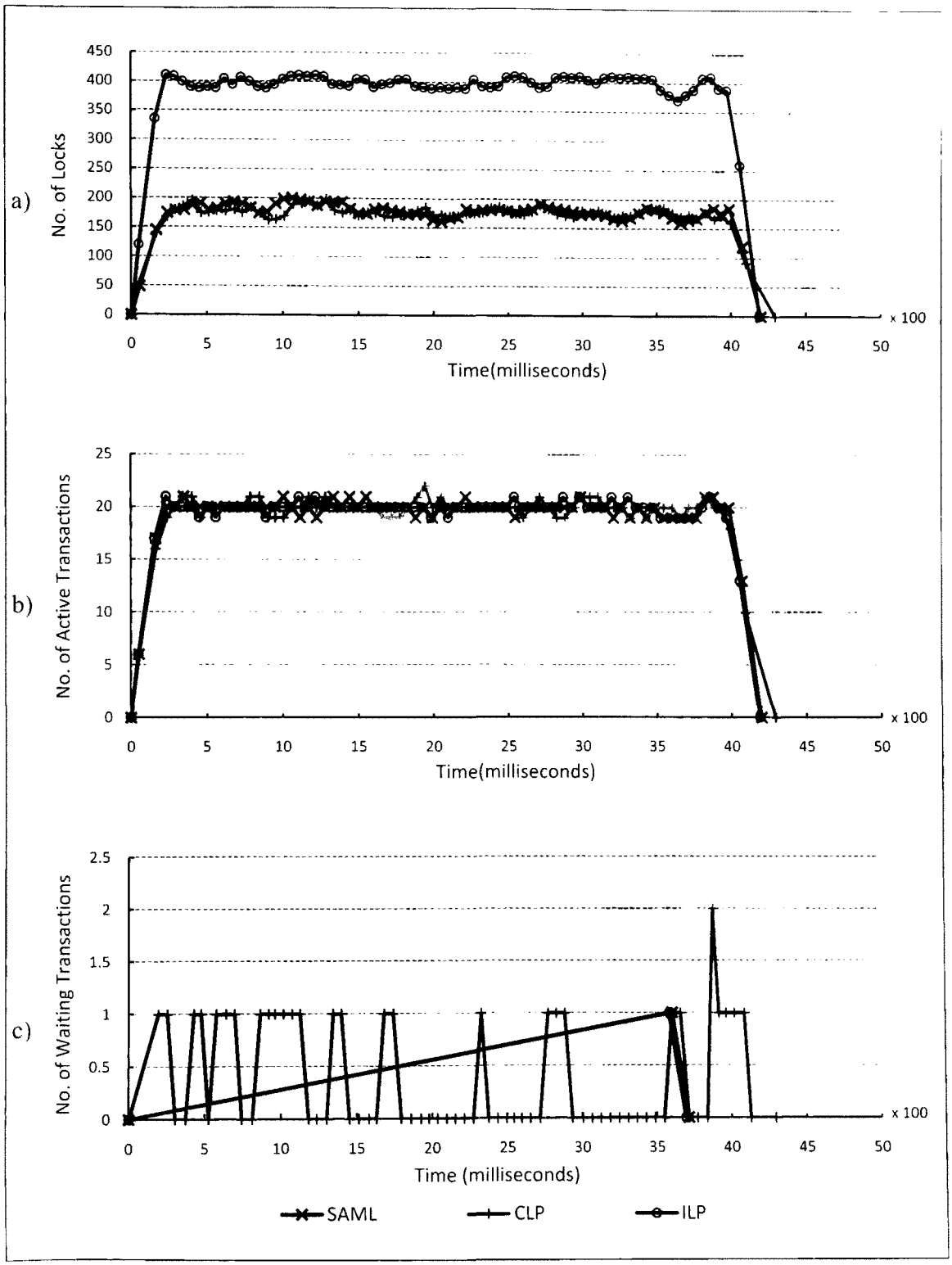

Figure 6-21. Result of experiment $T_2A_RL_SD_2$

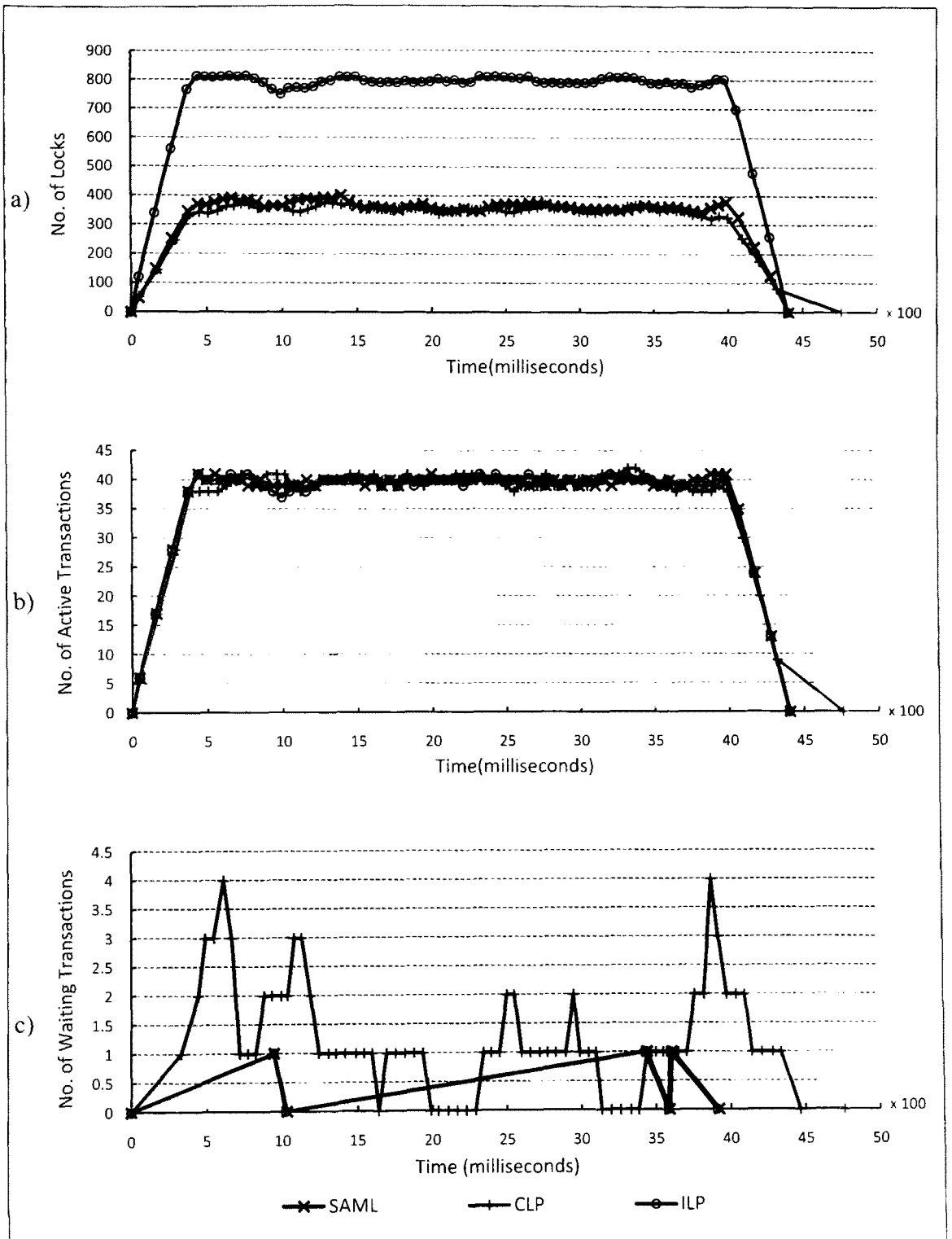

Figure 6-22. Result of experiment $T_2A_RL_SD_4$

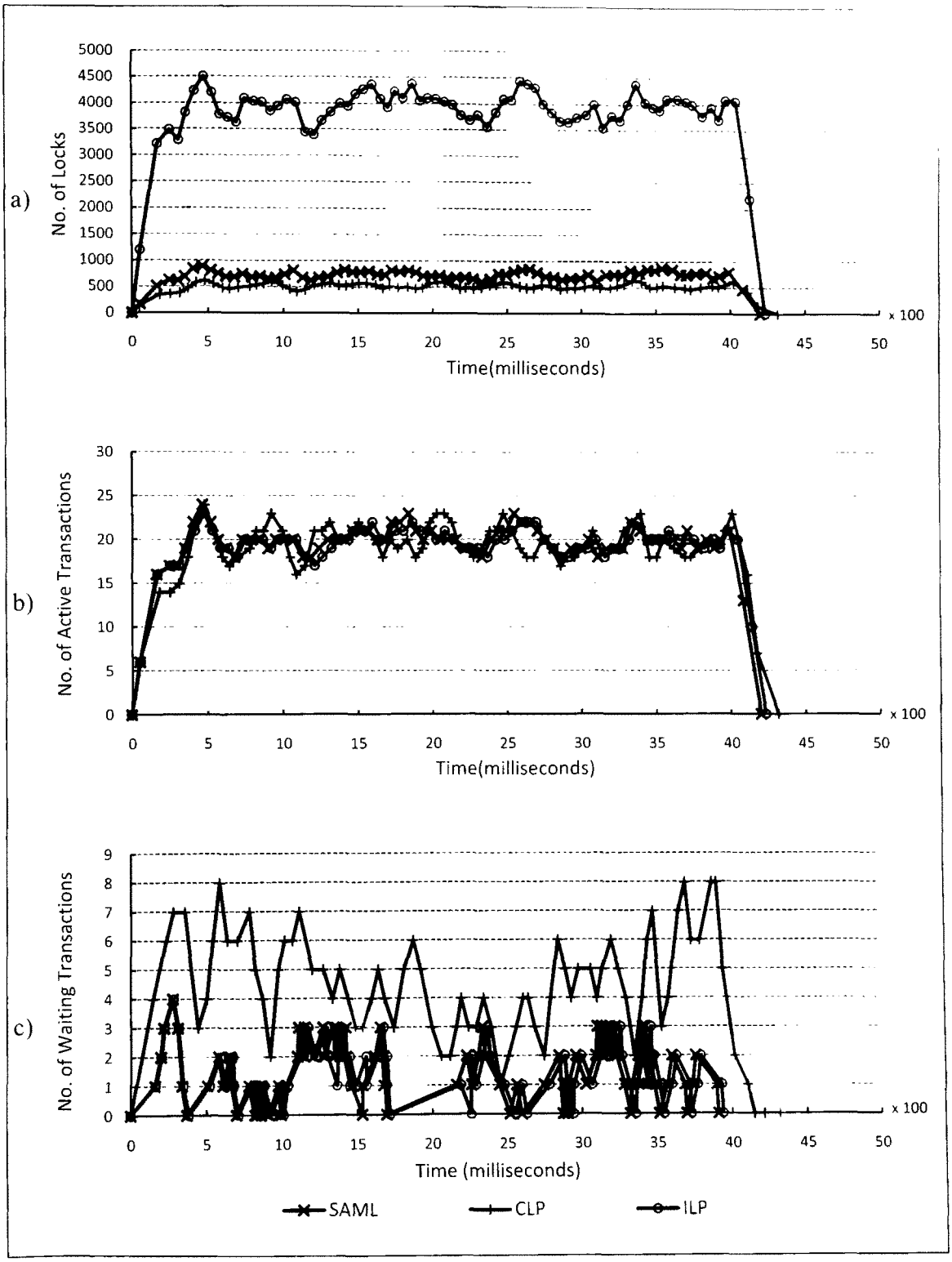

Figure 6-23. Result of experiment $T_2A_RL_HD_2$

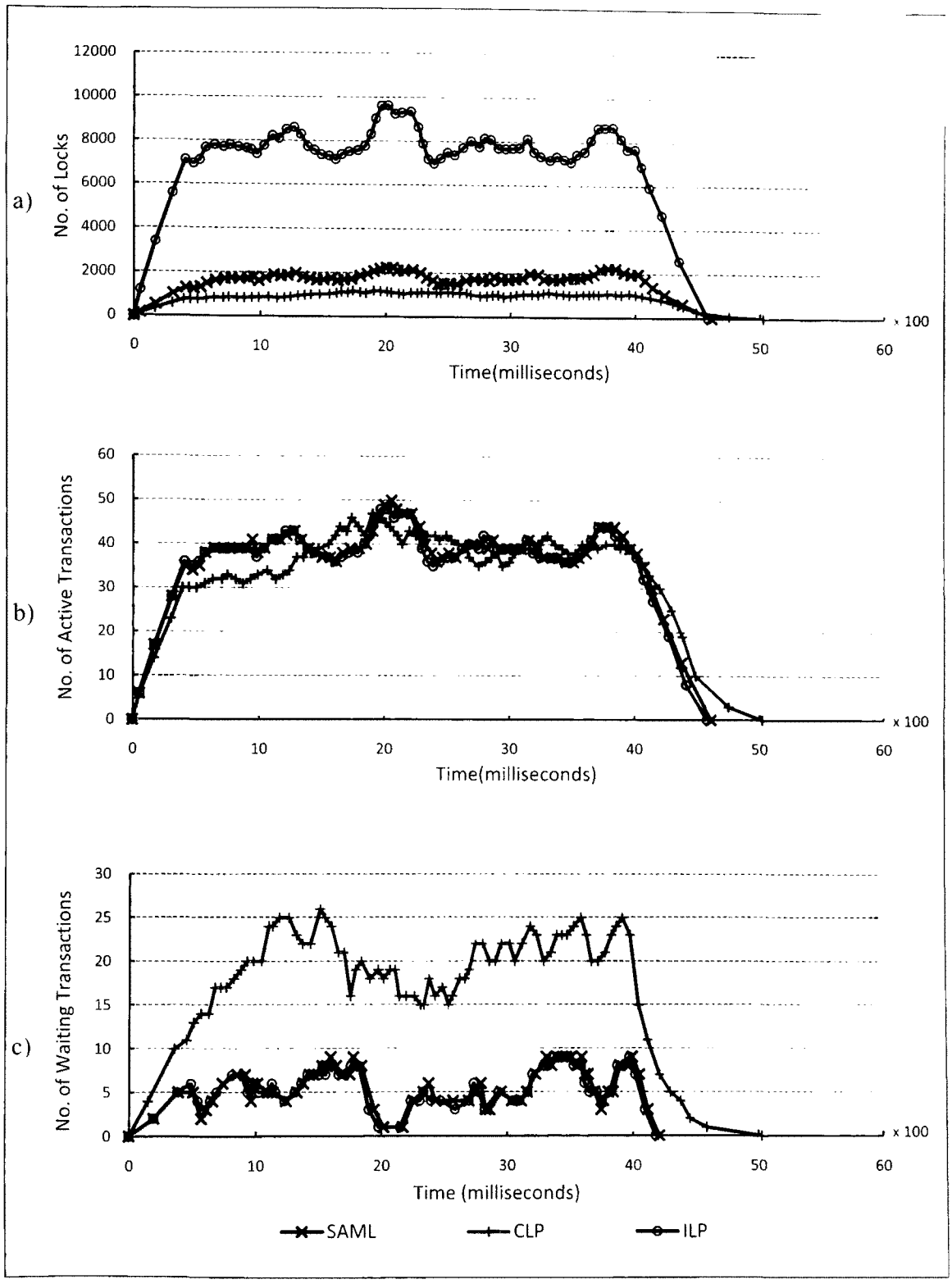

Figure 6-24. Result of experiment $T_2A_RL_HD_4$

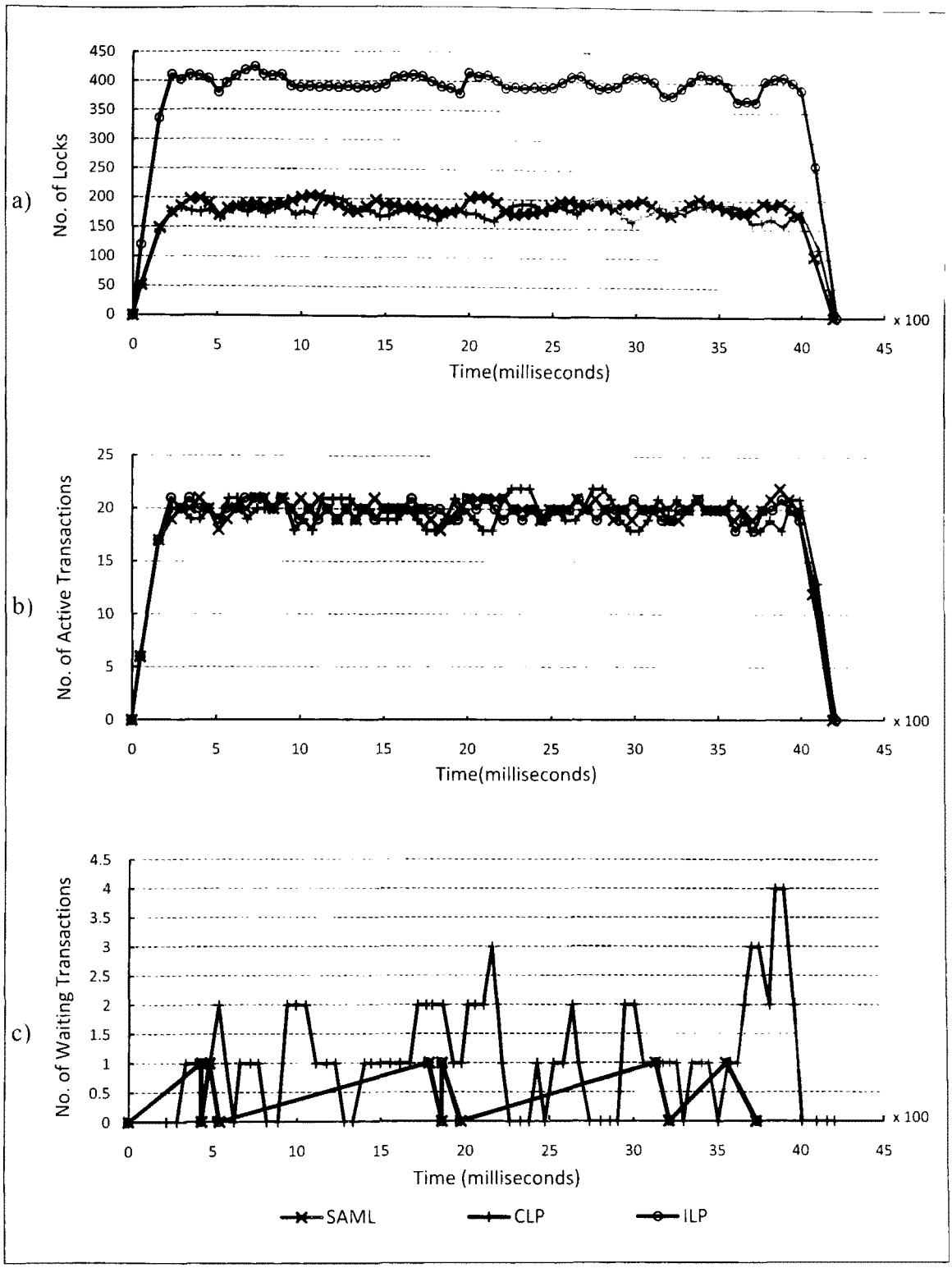

Figure 6-25. Result of Experiment $T_3A_0L_5D_2$

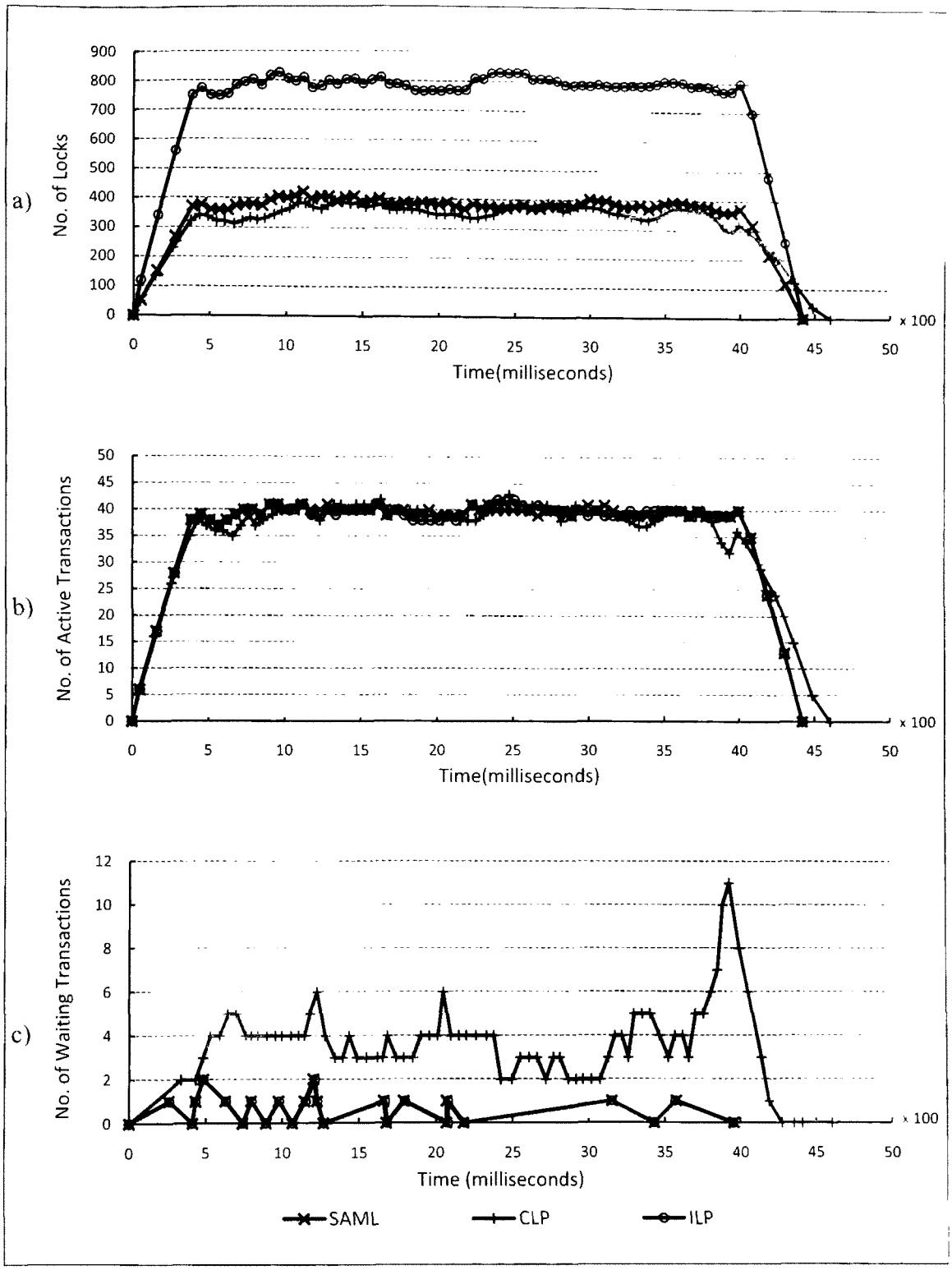

Figure 6-26. Result of Experiment $T_3A_0L_5D_4$

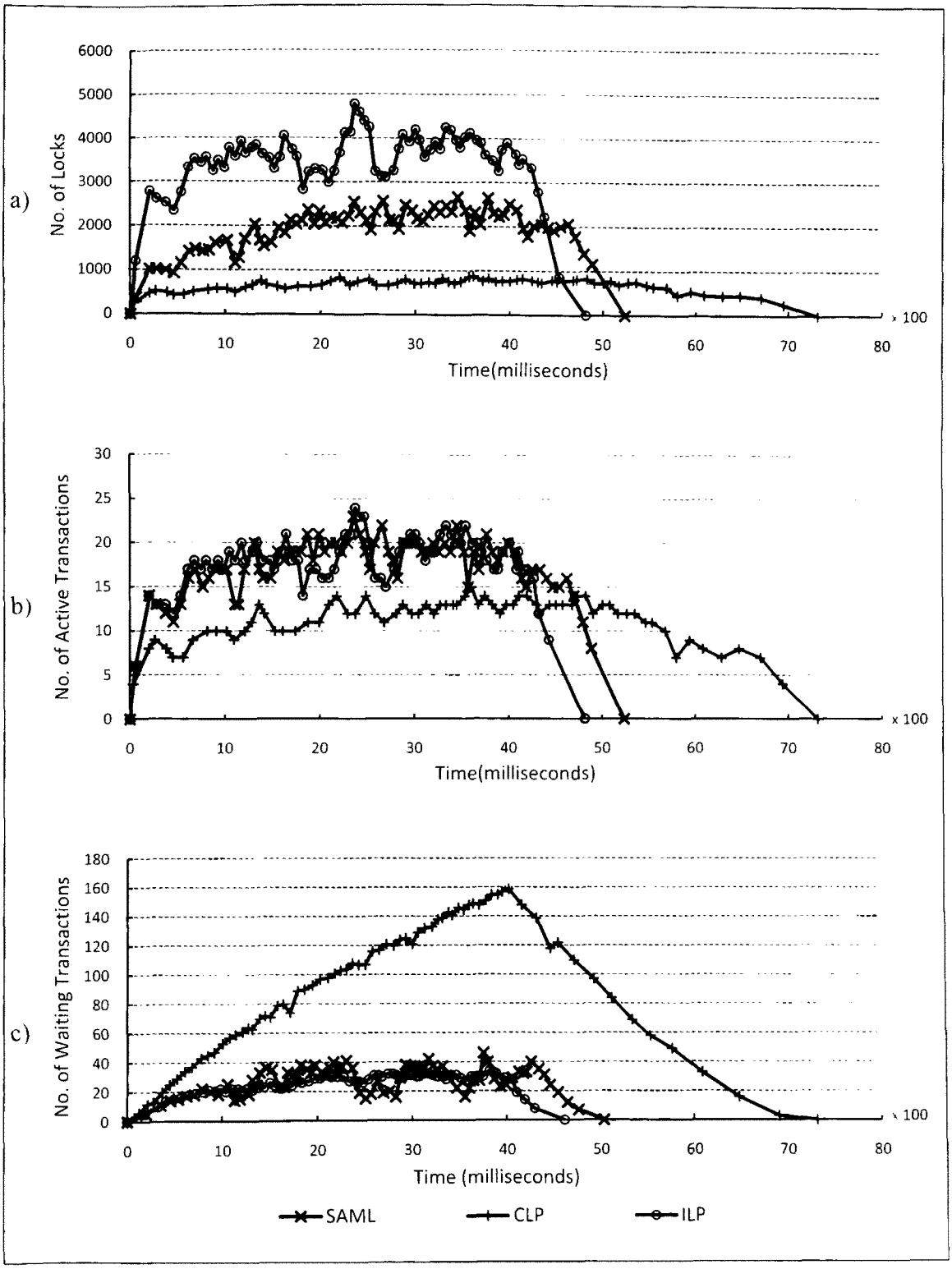

Figure 6-27. Result of experiment $T_3A_0L_HD_2$

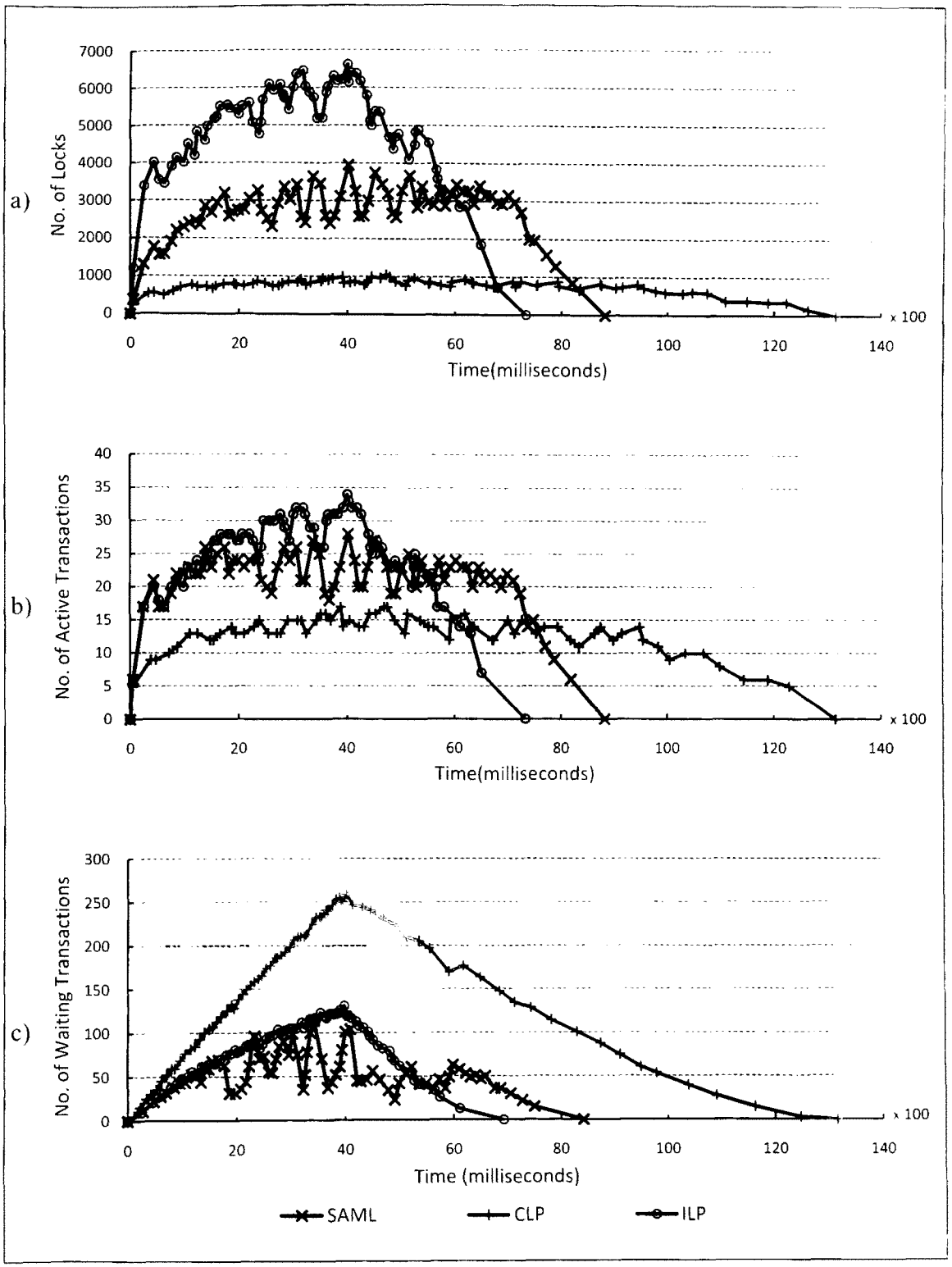

Figure 6-28. Result of experiment $T_3A_0L_HD_4$

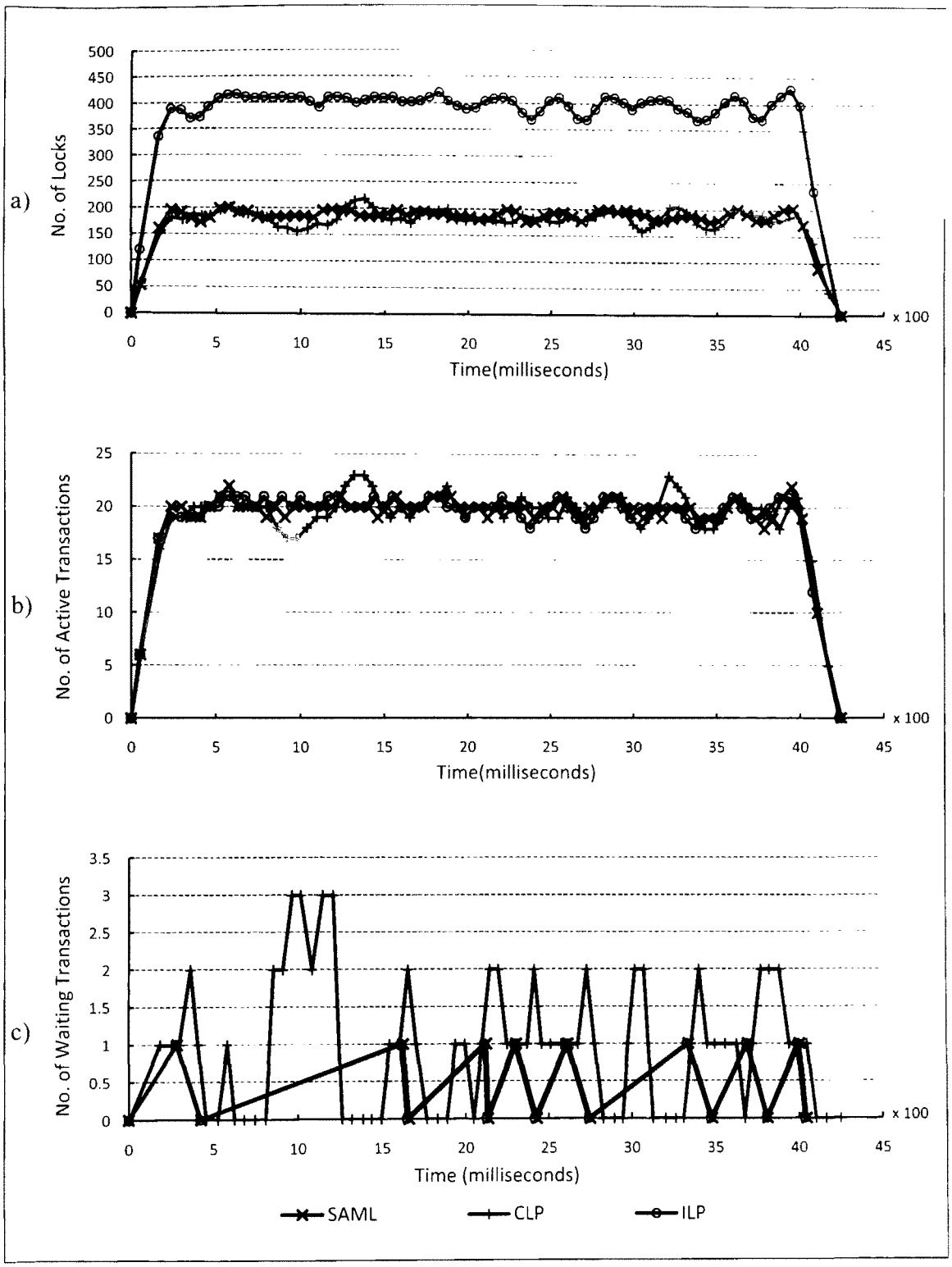

Figure 6-29. Result of experiment $T_3A_1L_3D_2$

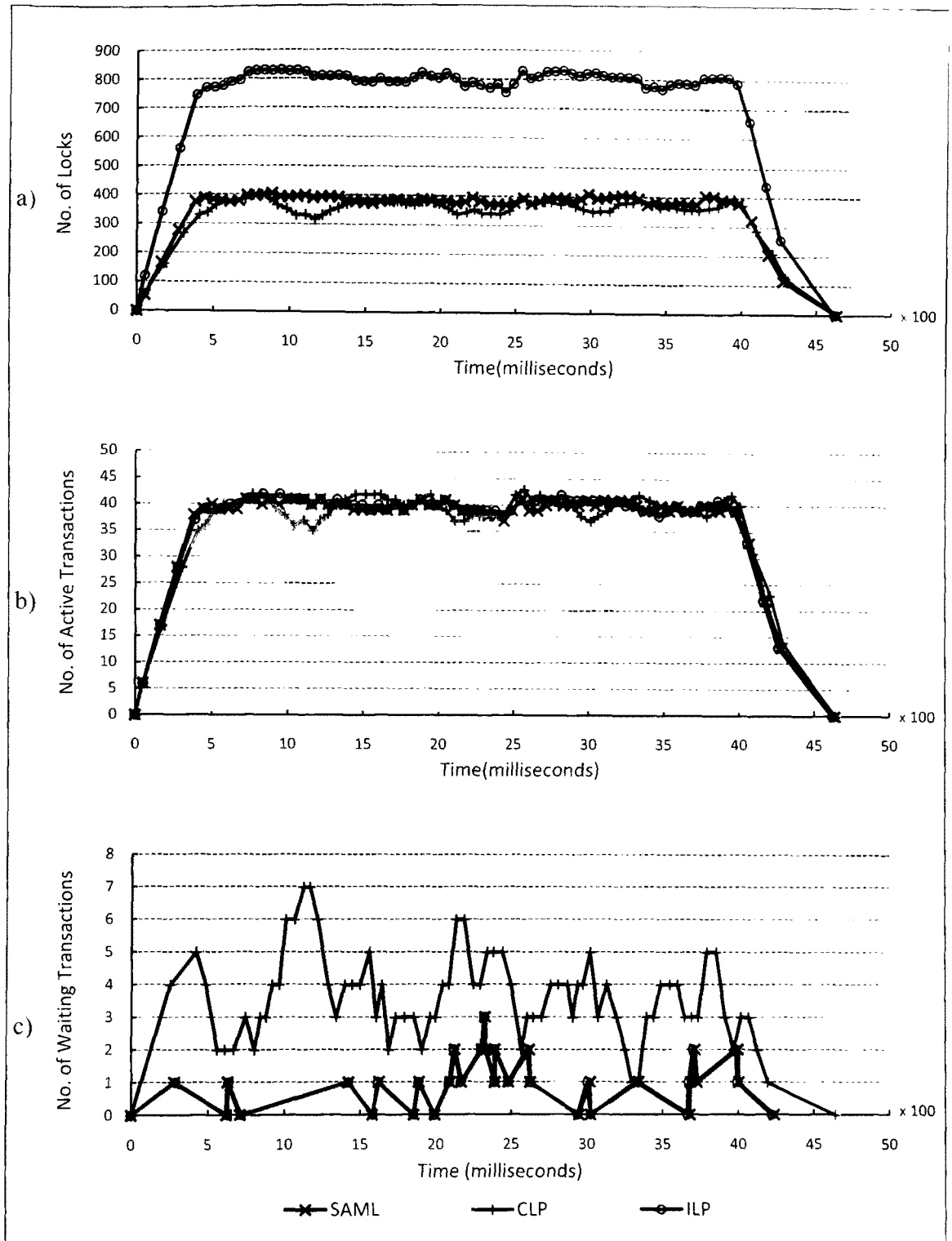

Figure 6-30. Result of experiment $T_3A_1L_5D_4$

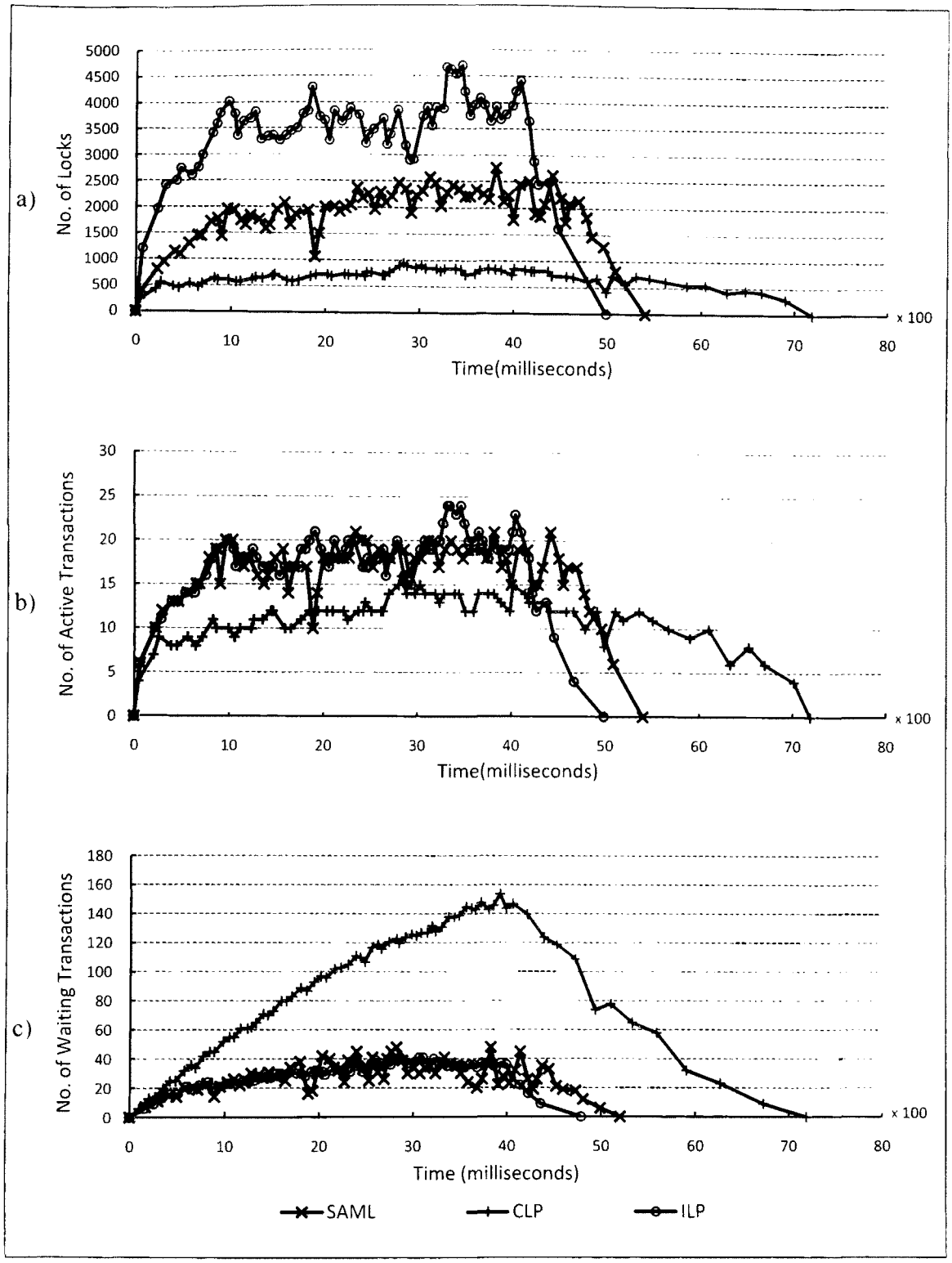

Figure 6-31. Result of experiment $T_3A_1L_HD_2$

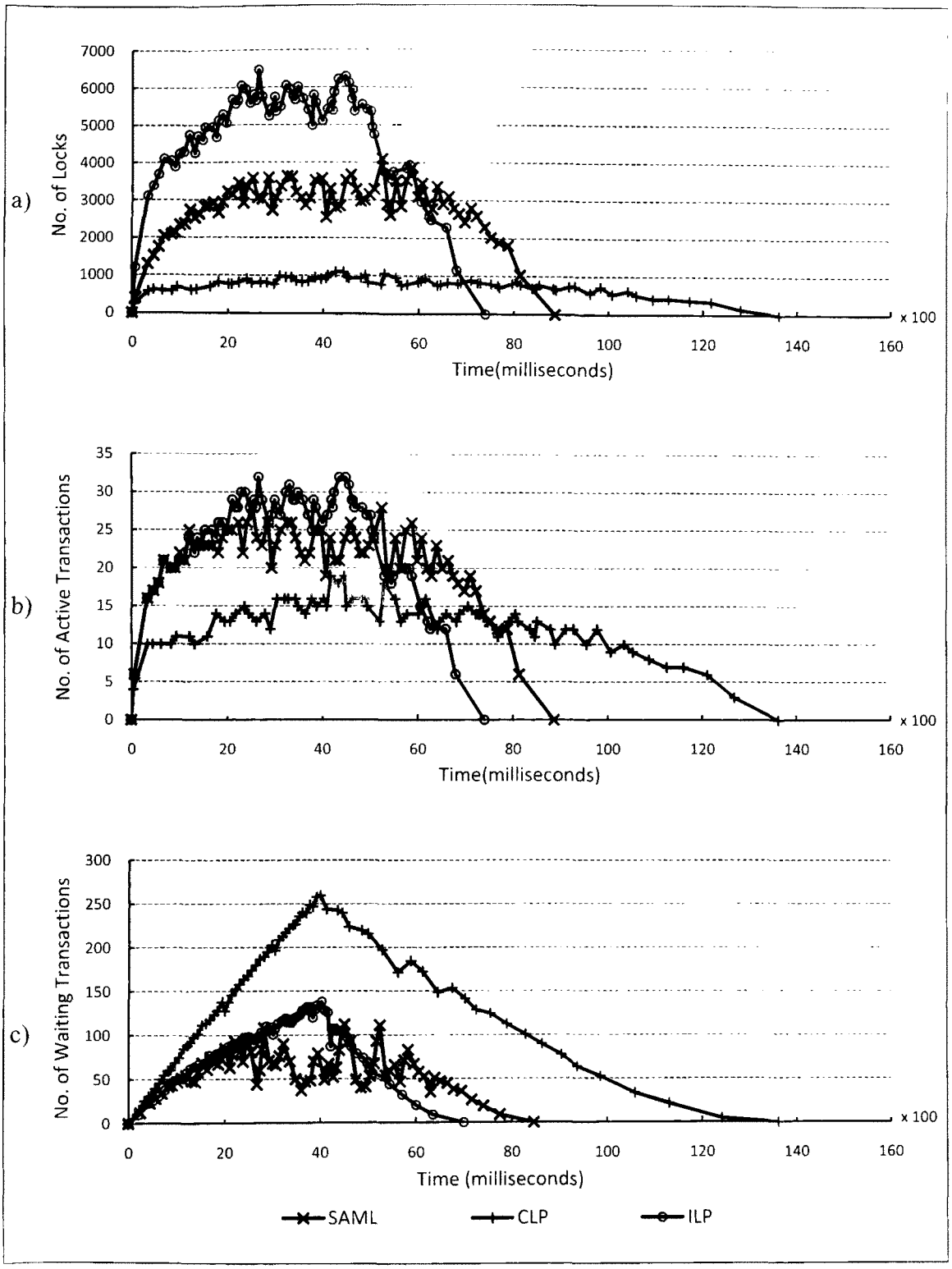

Figure 6-32. Result of experiment $T_3A_LL_HD_4$

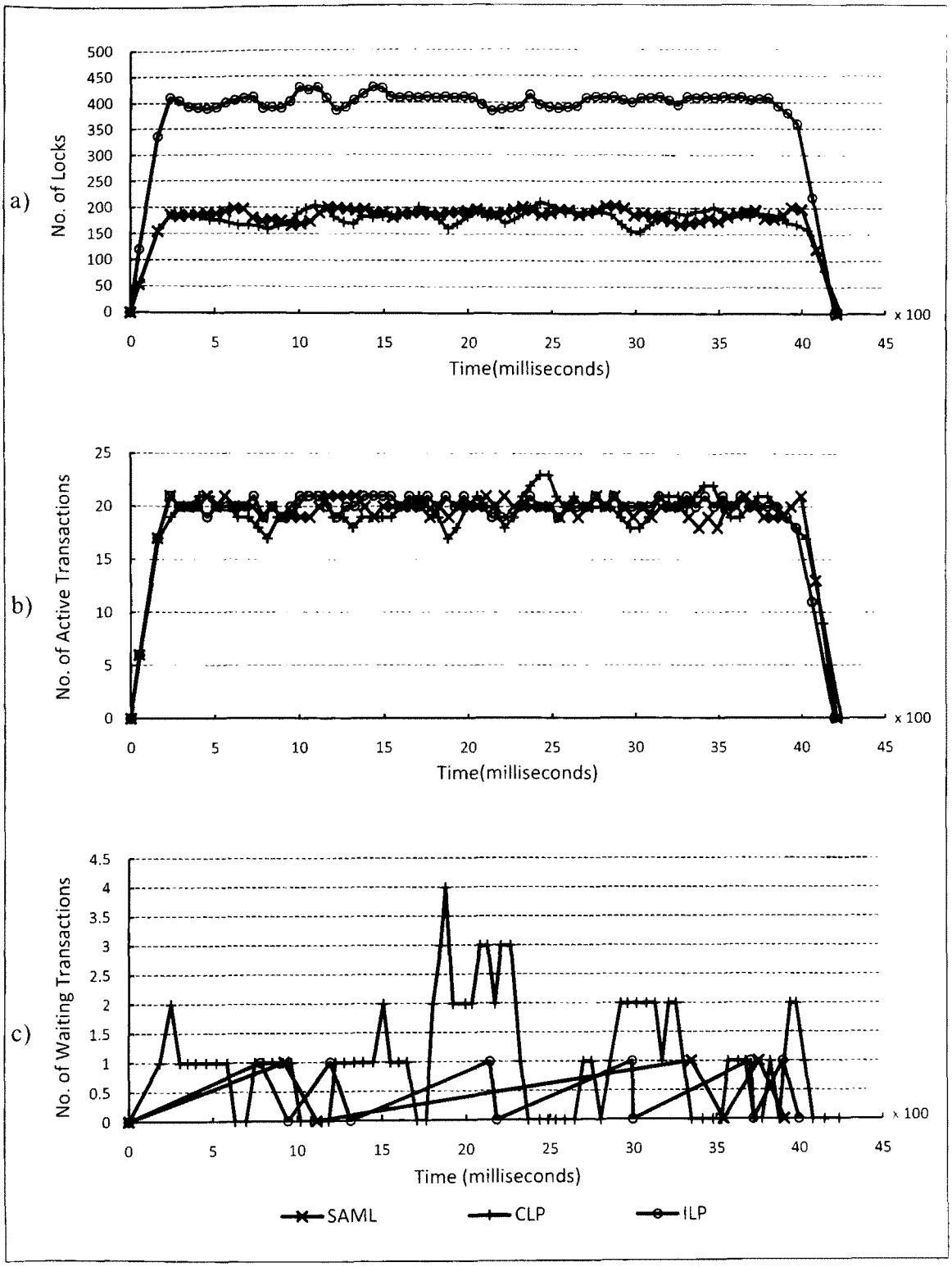

Figure 6-33. Result of experiment $T_3A_RL_SD_2$

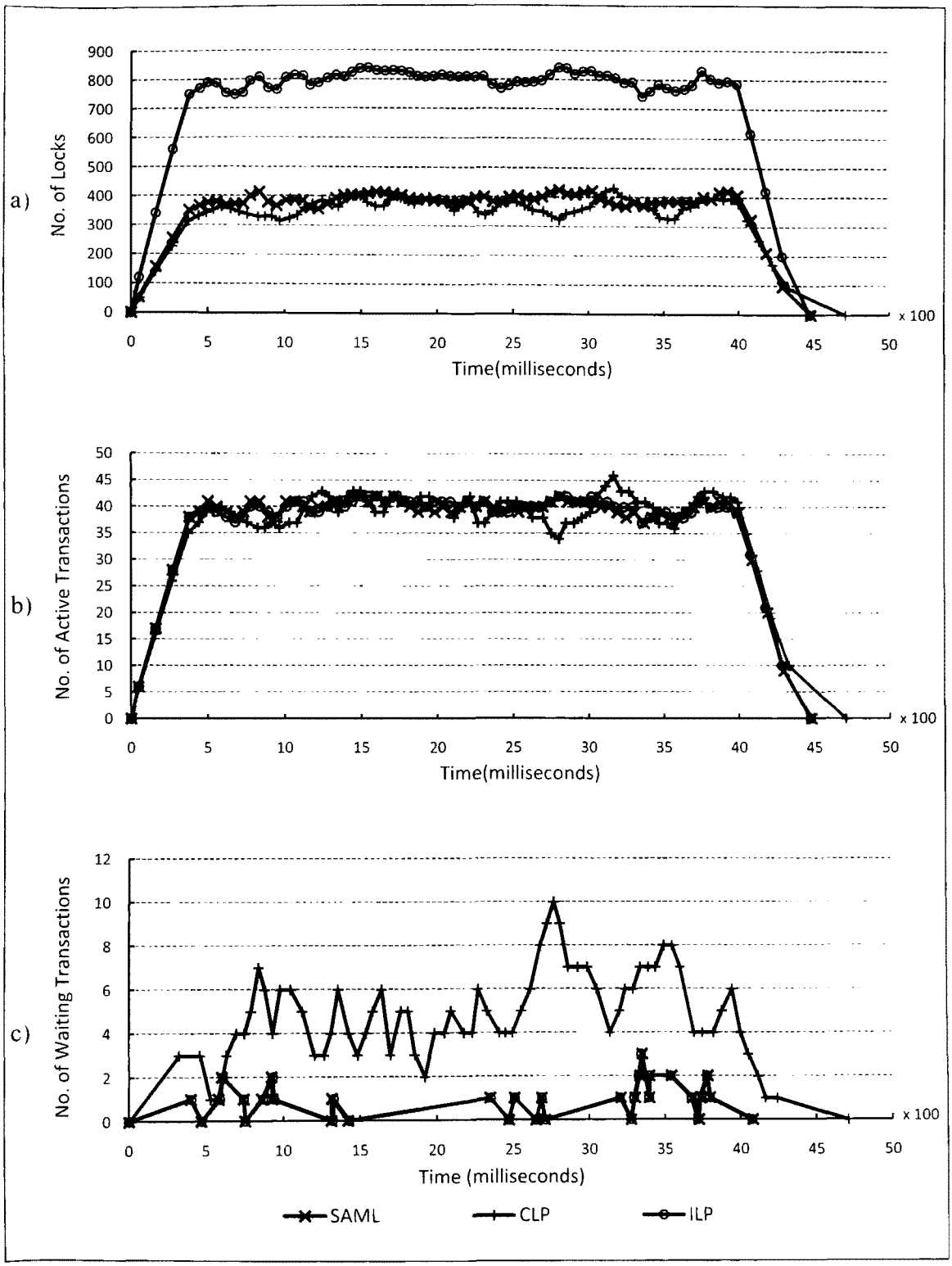

Figure 6-34. Result of experiment $T_3A_RL_SD_4$

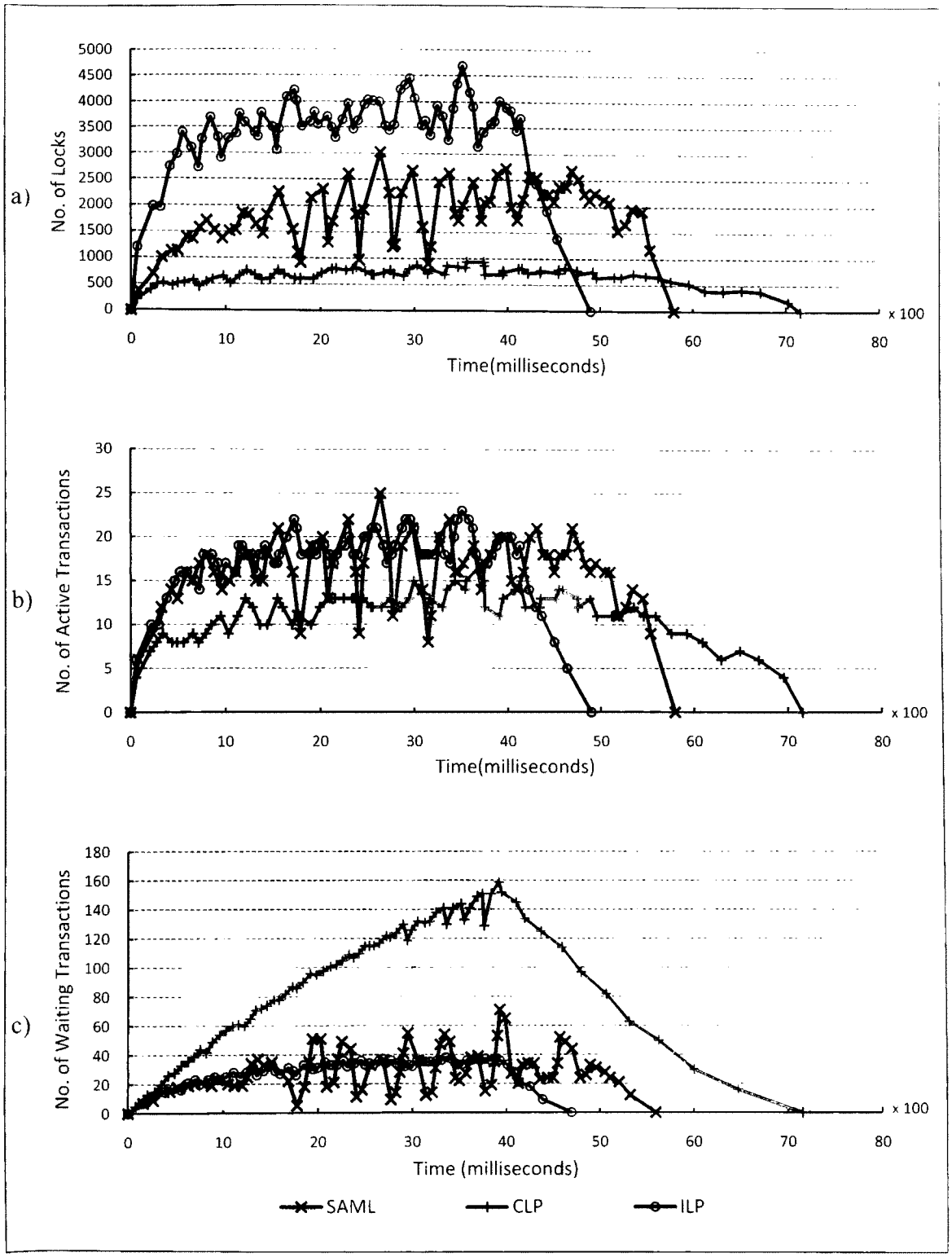

Figure 6-35. Result of experiment $T_3A_RL_HD_2$

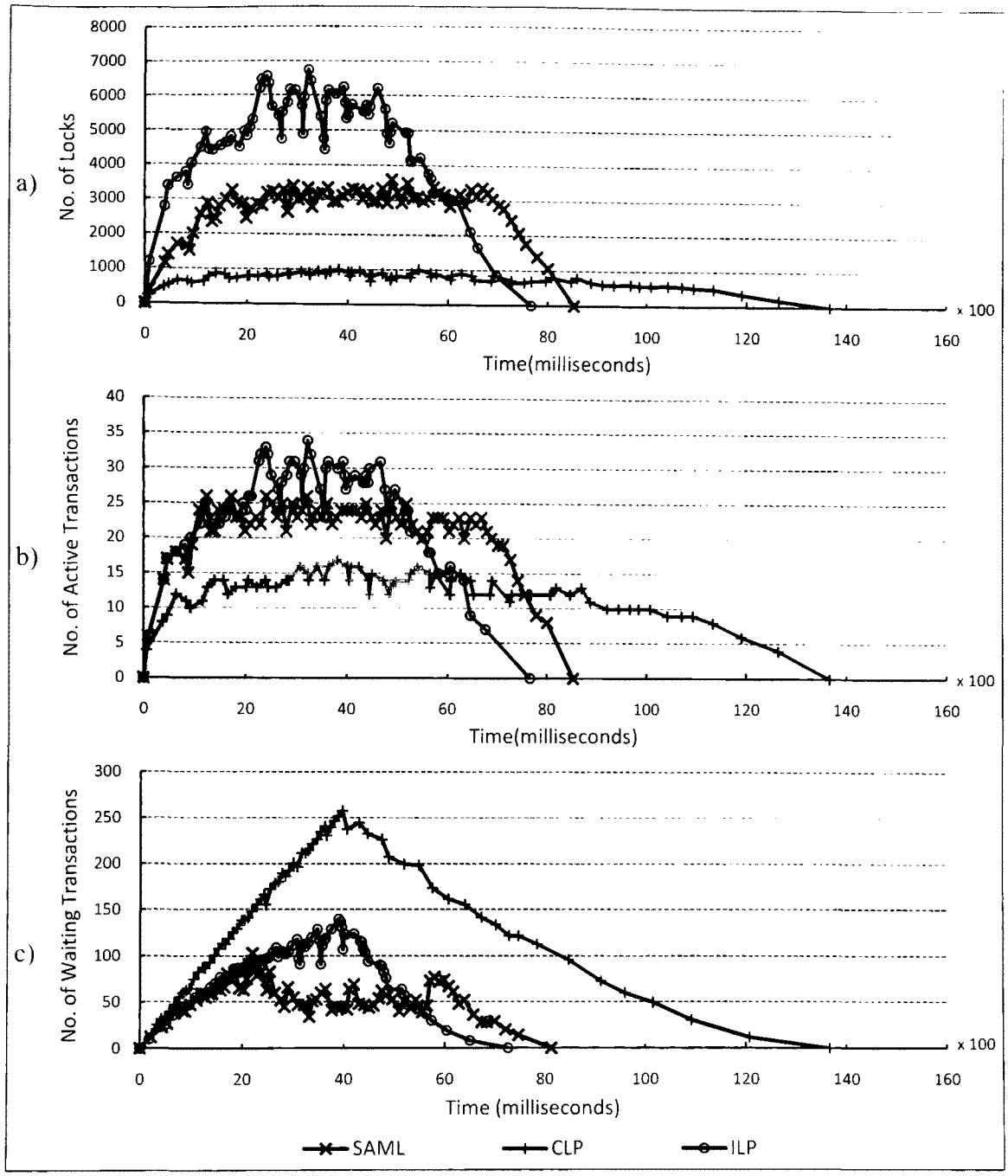

Figure 6-36. Result of experiment $T_3A_RL_HD_4$

REFERENCES

[1] C. T. K. Chang, "Adaptive multi-granularity locking protocol in object-oriented databases," PhD Thesis, University of Illinois, 2002.

[2] R. Grehan. (2005-2008) ODBMS.ORG :: Object Database (ODBMS). [Online]. http://www.odbms.org/introduction.html

[3] W. Kim, *Introduction to Object-Oriented Databases*. Cambridge, London: The MIT Press, 1990.

[4] M. E. Atkinson, "The Object-Oriented Database Manifesto," in *First International Conference on Deductive and Object-Oriented Databases*, Koyto, 1989, pp. 223-240.

[5] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*, Fifth Edition ed., A. R. Apt, Ed. New York, America: McGraw-Hill Companies, Inc., 2006.

[6] R. Elmasri and S. Navathe, *Fundamentals of Database Systems*. Redwood city, California: Benajamin/Cummings, 1989.

[7] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger, *Granularity of locks and degrees of consistency in a shared data base*. San Francisco, California, USA: Morgan Kaufmann Publishers Inc., 1998.

[8] R. G. G. Cattell, *Object Data Management: Object-Oriented and Extended* , 1st ed.. Boston, USA: Addison-Wesley Longman Publishing Co., Inc., 1994.

[9] W. Lam, Y. Wang, and Y. Feng, "Concurrency Control in Object-Oriented Databases," University of Waterloo , Technical report, 1991.

[10] H. Garcia-Molina, "Using semantic knowledge for transaction processing in a distributed database," *ACM Transactions on Database Systems (TODS)*, vol. 8, no. 2, 983.

[11] P. Muth, T. C. Rakow, G. Weikum, P. Brossler, and C. Hasse, "Semantic concurrency control in object-oriented database systems," in *Ninth International Conference on Data Engineering*, 1993, pp. 233-242.

[12] R. F. Resende, D. Agrwal, and A. E. Abbadi, "Semantic locking in object-oriented database systems," *ACM SIGPLAN Notices*, vol. 29, no. 10, pp. 388–402, 1994.

[13] Y. Murakami, M. Nishikaku, T. Okada, and M. Sakaguchi, "Real-time scheduling for semantic concurrency control of object-oriented database systems," in *7th International Workshop on Database and Expert Systems Applications*, 1996, pp. 214-22.

[14] K. Kwon and S. Moon, "Semantic multigranularity locking and its performance in object-oriented database systems," *Journal of Systems Architecture*, vol. 44, no. 12, pp. 917-935, 1998..

[15] W. Jun and L. Gruenwald, "An effective class hierarchy concurrency control technique in object-oriented database systems," *Information and Software Technology*, vol. 60, pp. 45-53, 1998.

[16] S. K. Madriaa, M. A. Tubaishatb, and B. Bhargavaa, "Multi-level transaction model for semantic concurrency control in linear hash structures," *Information and Software Technology Journal*, vol. 42, no. 7, pp. 445–464, 2000.

[17] S. Taniguchi, Budiarto, and S. Nishio, "On Locking Protocols in Object-Oriented Database Systems," *IEICE transactions on information and systems*, vol. E78-D, no. 11, pp. 1449-1457, 1995.

[18] S. Y. Lee and R. L. Liou, "A Multi-granularity Locking Model for Concurrency Control in Object- Oriented Database Systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 8, no. 1, pp. 144-156, 1996.

[19] P. E. Black. (2004, Apr.) directed acyclic graph. [Online]. http://www.nist.gov/dads/HTML/directAcycGraph.html

[20] O. D.H. and S. Ram, " Towards a Comprehensive Concurrency Control Mechanism for Object- Oriented Databases," *Journal of Database Management*, vol. 6, no. 4, pp. 24-34, 1995.

[21] N. S. Barghouti and G. E. Kaiser, "Concurrency Control in Advanced Database Applications," *Computing Surveys*, pp. 269-317, 1991.

VITA AUCTORIS

NAME	:	Deepa Saha
PLACE OF BIRTH	:	Dhaka, Bangladesh
YEAR OF BIRTH	:	1984
EDUCATION	:	Bachelor of Engineering in Computer Science and Technology, 2005. Bengal Engineering and Science University, Shibpur, WB, India. Master of Science, 2008. School of Computer Science, University of Windsor, Windsor, ON, Canada.
AWARDS	:	International Graduate Student Scholarship, 2004-2006. University of Windsor, Windsor, ON, Canada. Bangladesh-Sweden Travel Scholarship, 2004. Indian Council for Cultural Relations Scholarship, 2001-2005.
PUBLICATION	:	J. Lu, Y. Yu, D. Roy and D. Saha, "Web service composition: a reality check", in *Eighth International Conference on Web Information Systems Engineering*, Nancy, 2007, pp. 523-532.