Unifying Memory and Database Transactions

Ricardo J. Dias   João M. Lourenço

CITI—Centre for Informatics and Information Technology, and
Departamento de Informática, Universidade Nova de Lisboa
Portugal
{rjfd,joao.lourenco}@di.fct.unl.pt

Abstract. Software Transactional Memory is a concurrency control technique gaining increasing popularity, as it provides high-level concurrency control constructs and eases the development of highly multi-threaded applications. But this easiness comes at the expense of restricting the operations that can be executed within a memory transaction, and operations such as terminal and file I/O are either not allowed or incur in serious performance penalties. Database I/O is another example of operations that usually are not allowed within a memory transaction. This paper proposes to combine memory and database transactions in a single unified model, benefiting from the ACID properties of the database transactions and from the speed of main memory data processing. The new unified model covers, without differentiating, both memory and database operations. Thus, the users are allowed to freely intertwine memory and database accesses within the same transaction, knowing that the memory and database contents will always remain consistent and that the transaction will atomically abort or commit the operations in both memory and database. This approach allows to increase the granularity of the in-memory atomic actions and hence, simplifies the reasoning about them.

1 Introduction

Software transactional memory (STM) is a promising concurrency control approach to multithreaded programming. More than a concurrency control mechanism, it is a new programming model that brings the concept of transactions into the programming languages, by way of new language constructs or as a simple API and supporting library. Transactions are widely known as a technique that ensure the four ACID properties [1]: Atomicity (A), Consistency (C), Isolation (I) and Durability (D).

Memory transactions, with roots in the database transactions, must only ensure two of the ACID properties: Atomicity and Isolation. The Consistency and Durability properties may be dropped, as memory transactions operate in volatile memory (RAM). Volatile memory does not have persistence properties and does not have a fixed logical structure, like a database system, over which one can make consistency assertions.

In the past few years, several STM frameworks have been developed. Most of these STM frameworks take the form of software libraries providing an API to support the transactional model to the application [2–5]. This library-based approach allows the rapid prototyping of algorithms and their performance evaluation. Some other STM frameworks extend existing programming languages with transactional constructs supported directly by the compiler [6–9]. Most of these frameworks focus in managed languages such as Java, C#, and Haskell, while some other target unmanaged languages like C and C++.

One drawback of using STM lies in the execution of partially- and non-transactional operations within a transaction [10]. Pure-transactional operations are undone automatically by the STM transactional framework when a transaction aborts, e.g., changing the contents of memory. Non-transactional operations simply cannot be undone, e.g., writing data to the terminal. Partially-transactional operations are either revertible or compensable, and can be undone at some expense, e.g., explicit memory management operations and I/O to disk files. Some STM frameworks opt to not allow the execution of partially- and non-transactional operations within memory transactions [7]. Some others force the memory transactions to execute non- and partially-transactional operations in mutual exclusion to all other transactions in the system [9, 11].

Another example of partially-transactional (revertible) operations are accesses to a transactional database from within a memory transaction. Particularly interesting is the case where the application decides to commit the database transaction within a memory transaction. If the memory transaction later needs to abort, it would be necessary to rollback the already committed database transaction. This problem has been briefly enunciated in the past [12], but to our best knowledge to date no other work has addressed this matter.

This paper proposes a solution to the above problem by widening the transactional model scope to cover, without differentiating, both memory and database operations. Thus, the users are allowed to freely intertwine memory and database accesses within the same transaction, knowing that the memory and database contents will always be consistent and that the transaction will either abort or commit the operations in both memory and database atomically.

The remaining of this paper is organized as follows. Section 2 defines the unified model and its properties. Section 3 describes the implementation of the unified model using a specific STM framework and database management system. Section 4 evaluates the performance of our implementation of the unified model and compares it with other alternative approaches and, finally, Sect. 5 closes with some concluding remarks.

2 The Unified Model

Working with two or more transactional models in the same application, each with its own set of properties and guarantees, may become unbearable. This paper proposes an unified transactional model, merging two currently popular transactional models: memory and database transactions.

Transactions in the unified model can enclose two main classes of operations, memory and database operations. In the first class, two types of operations are allowed, read and write of memory locations. In the second class, all transactional operations supported by database are allowed.

The unified model preserves the minimum common set of properties from both of the transactional models it unifies—Atomicity and Isolation—,allowing to define sets of operations as transactions, acting upon the memory and the database atomically and isolated from other concurrent transactions. The unified model also inherits the Consistency and Durability properties from the database management system, but these properties only apply to database operations.

The atomicity and isolation properties apply to any operation valid in the unified model. This includes both memory and database transactional operations. A transaction will not see intermediate memory nor database states of other transactions, and all the effects caused in both memory and database will either persist upon the transaction commit or be rolled back in case of an abort.

The database transactional model allows multiple applications to access the database concurrently and the ACID properties always hold. The transactional memory model, however, only applies to the memory shared between multiple control flows, typically multiple threads within a single application. The unified model will hold the most restrictive properties, i.e., will hold the AI properties for a single multithreaded application. This assumption can be somewhat relaxed when using a distributed transactional memory framework [13]. In this case the application breaks the physical node barrier, but still has to be a single distributed application whose multiple components are cooperating using the distributed STM framework. In other words, the unified model assumes exclusive access to the database, i.e., that no other application is accessing the database at the same time.

3 Implementation

The unified model was implemented resorting to a library based STM framework, the Consistent Transactional Layer [14] (CTL), a STM implementation for the C programming language and derived from the TL2 [2] library. CTL extends the TL2 framework with a large set of new features and optimizations [15]. CTL also implements a full featured handler system that allow the users to define reverting operations. These operations are executed in key moments of a memory transaction and allow to revert the effects of partially-transactional operations executed within a memory transaction.

3.1 CTL Handler System

The CTL handler system [10] extends the life-cycle of memory transaction by executing user-defined functions in specific key moments. Users can define five types of handlers which are executed by the CTL run-time at four different key moments of the transaction life-cycle: *prepare-commit* and *pre-commit* handlers are executed before the transaction commits; *pos-commit* handlers are executed after the transaction commits; *pre-abort* handlers are executed before a transaction aborts; and *pos-abort* handlers are executed after the transaction aborts.

Both prepare-commit and pre-commit handlers are executed after validating the memory transactions and, thus, may assume that the memory transaction will commit. The former may still force the transaction to abort while the latter cannot do so and must definitely assume that the transaction will commit. The adequate combination of prepare-commit and pre-commit handlers allows to execute a *two-phase-commit protocol* [16] between several transactional subsystems in which one of them is the memory transaction. The two-phase-commit protocol includes two main phases: the preparation phase, where all transactions must first agree whether they will commit or abort; and the commit/abort phase, where all the transactions must perform the decision previously agreed. If the decision was to commit, the transaction manager will request all the transactions to commit, otherwise it will request all the transactions to abort.

3.2 Unified Model Implementation

The unified model was implemented using two distinct transactional systems: one for memory transactions and another for database transactions. Dealing with two autonomous transactional systems as a single one requires that the commit/abort phase of both transactional systems to be atomic: both must either commit or abort. The *two-phase-commit* protocol (2PC) allows to commit N transactions, from N different transactional systems, atomically. This algorithm was implemented by having the STM framework playing two roles, one as the 2PC controller, and another as a partner in the 2PC protocol together with the database system.

The database is accessed by way of an ODBC interface for the C programming language. This approach allows to use any ODBC compliant database management system and initial experiments were made with two different ODBC compliant databases, DB2 and PostgresSQL. This dual database tests were essential to functionally validate the approach, but experiments demonstrated that PostgresSQL outperformed DB2 to the point that it became irrelevant to run performance tests with DB2. Thus, this paper only reports on performance tests with PostgresSQL.

Without support for the database prepare phase, the commit of the database transaction will be attempted as a prepare-commit handler. When the prepare-commit handlers are executed, the memory transaction has already been validated, thus it is known that the memory transaction will not abort due to concurrency conflicts. As prepare-commit handlers can still abort the transaction, this approach is safe as long as the database commit is the last prepare-handler to be executed, and this rule can be enforced by the TM framework. If the database commit is not successful, hence the database transaction has aborted, then the last of the prepare-commit handlers will also fail and the memory transaction will be rolled back. In the presence of a concurrency conflict, either in memory or in the database, the memory transaction will abort, the pre-abort handler will be executed and the database transaction will also be aborted.

To ease the work of registering all the necessary handlers to allow the execution of the 2PC protocol with the transactional memory framework acting as the controller, a new call, TxDBStart, was introduced into the CTL transactional memory API to replace the call previously used to start a new memory transaction. Figure 1 shows the definition of this new front-end.

```
void TxDBStart (Thread *Self, HDBC dbc, int roflag) {
    TxStart (Self, roflag);
    _ctl_register_prepare_handler (Self, do_commit, (void *)dbc);
    _ctl_register_pre_abort_handler (Self, do_abort, (void *)dbc);
}
```

Fig. 1. TxStart front-end for using database transactions.

The first and third parameters of TxDBStart are the same as for the CTL TxStart function. The second parameter is the database connection handler for the ODBC interface. Thus, obtaining the ODBC connection handler to the database and replacing all the calls to TxStart to the new TxDBStart is all that is needed for an application to switch from the pure transactional memory model into the new unified model.

TxDBStart starts by calling TxStart and initiating a new memory transaction, then it registers the do_commit function as a *prepare-commit handler* and the do_abort function as a *pre-abort handler*. Thus, when the memory transaction terminates either by committing or aborting, the appropriate handler will be executed and the database transaction will also terminate. The definitions of do_commit and do_abort are similar, as both only call the ODBC SQLEndTran function with the appropriate flag indicating whether to commit or abort.

Figure 2 shows the implementations of the do_commit function. The database transaction is implicitly started by the ODBC upon the first operation over the database and will end upon the commit or abort of the transaction.

```
int do_commit (Thread *Self, void *args) {
    SQLRETURN ret;
    ret = SQLEndTran (SQL_HANDLE_DBC, (HDBC)args, SQL_COMMIT);
    return (SQL_SUCCEEDED(ret) != 0);
}
```

Fig. 2. do_commit function handler definition.

The database connection handler is passed to the function TxDBStart, identifying which database should be used in the current transaction. It is possible to use different databases in different transactions, but it is not possible to use more than one database per transaction in this implementation of the unified model. This limitation is due to the fact that the ODBC does not support the prepare service in the database. Transactions that are known to not access the database can still use the original `TxStart` instead of the newly defined `TxDBStart`.

Guaranteeing that both memory and database transactions follow the 2PC protocol and commit and abort atomically, is not sufficient to guarantee the isolation property for the unified model. Some stronger requirements concerning the order in which concurrent transactions are scheduled must be fulfilled.

3.3 Implementation requirements

Most transactional database systems allow to relax the isolation level, by admitting the execution of non-serializable transaction schedules [17], in order to increase the throughput of transaction processing. To guarantee the provision of Isolation to the unified model, both transactional models must run under the same isolation level which must be the strongest (most restrictive) from both. Since the common isolation level for transactional memory is full serialization, the database transactional system used must also run in full serialization isolation level. Another requirement for the correctness of the unified model is that both transactional systems generate equal serialization schedules. Serialization schedules are created dynamically and independently by both memory and database transactional systems, and the generated schedules are affected by the isolation level and by the concurrency control policies.

Although the model as described in Sect. 2 is generic enough to unify memory and database transactional models, the transactional memory framework and the database management system must be carefully chosen and parametrized to satisfy the two requirements described above.

3.4 Prototype Evaluation

We implemented a prototype using the PostgreSQL database management system [18]. Although PostgresSQL supports a Serializable Isolation Level, its performance is much worse than Snapshot Isolation Level (SIL). However, using this DBMS in SIL with no other modifications would not satisfy the requirements to support Isolation in the unified model. The problem with SIL is that the transactional system does not detect conflicts between two transactions where one is reading data from a record and the other is writing data into the same record. Once this type of conflicts are detected, there is no need to impose stronger restrictions on the isolation level.

One possible approach to force the detection of the read-write conflicts was to force read operations to be treated as write operations by the DBMS [19]. A simple way to achieve such a goal (with limitations) would be to force, for each `SELECT` statement of a data item D_i, to update the value of the data item to itself, i.e., force the operation $V(D_i) = V(D_i)$, followed by the desired `SELECT` statement. This solution would transform all read operations into write operations and would therefore impose a strong performance overhead. The PostgreSQL database [18] implements two variants of the `SELECT` SQL statement to address such problem, the statements `SELECT FOR UPDATE` and `SELECT FOR SHARE`. The difference between the two statements is that `SELECT FOR UPDATE` acquires an exclusive lock of the rows being accessed while `SELECT FOR SHARE` acquires a shared lock, thus permitting other `SELECT FOR SHARE` statements to execute concurrently over the same rows. If any row is locked by an exclusive or shared lock, any write attempt to the locked row will block.

With this solution, the transaction schedules generated by the DBMS and by the transactional memory framework were identical, thus satisfying the requirements for the Isolation property to hold.

4 Evaluation of the Unified Model

The evaluation of the prototype of the unified model was twofold: functional and performance. Functional evaluation aimed at verifying the correctness of the system behavior, even under very stressing conditions, and its fitness to application development. Performance evaluation aimed at comparing absolute performance and scalability of our approach with coarse and finer grain locks.

The testing application stores and retrieves scientific articles from a database. The articles can be indexed by author name, by keywords, or both. The article repository has an interface with four services: insert an article, remove an article, find an article by author, and find an article by keyword. Depending on the version of the testing application, each service will access multiple shared data structures in main memory (when applicable), multiple tables in the database, or both. The benefits of this approach depend on the application being able to store part of the relevant information in main memory, thus avoiding to access the database for read-only operations.

The application is divided in two components: a server and a client. The server will manage the repository, allowing concurrent calls to the repository interface. The client is a single threaded component issuing a sequence of service requests to the server. This application will use the unified model to maintain an in-memory replica of the database indexation structures. As the unified model guarantees the consistency between both data repositories, if the application is closed all information still persists in database and when the application is restarted all indexation information is reloaded into main memory.

4.1 Database model

The database scheme is very simple. It has three entities: articles, authors, and keywords. An article is represented by an internal *id*, a title, and the file path to the article document. An author is represented by its name (we assume that each author name is unique). A keyword is represented by itself (and must also be unique among the keywords). Each article must have at least one author and one keyword, but may have more. Figure 3 illustrates the entity-relation model of the database scheme just described.

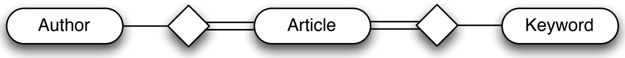

Fig. 3. Application database entity-relation model.

The relation between authors and articles is supported by an association table where each row makes the link between an author and an article. The same applies to the relation between keywords and articles.

4.2 Memory Data Structures

The indexing structures in memory are represented by a single linked list and a hash table. The structure is simple: one hash table for indexing authors and another for indexing keywords. Each element of the hash table is a list of article identifiers that are associated with that author or keyword.

The single linked list was implemented using CTL to protect it from concurrent accesses. This list implements three operations: insert an element, remove an element, and lookup for an element. Each of these operations uses the handler system, described in Sec 3.1, to manage the memory allocation and deallocation of list nodes inside memory transactions.

The hash table was also implemented using CTL to protect it from concurrent accesses. It also implements three operations: insert an element, remove an element, and lookup for an element. Similar to the linked list, each hash table operation also resorts to the handler system to manage memory allocation and deallocation inside memory transactions.

4.3 Description of Repository Operations

Inserting an article into the repository requires several steps to be executed in sequence. First the article is inserted into the database. If the database insertion succeeds (meaning that the article was not yet in the database), then the associations between the article and each of its authors are also inserted into the database. References to this article are also inserted, one for each author, in the hash table that represents the association in main memory. A similar process is executed for the keywords. Removing an article also requires several steps. First, the article is removed from the memory hash tables that maps authors to articles. Then, a similar operation is executed to remove that same association from the database. A similar process is executed for the keywords. Finding an article in the repository by author or by keyword involves only indexing the respective hash table and, for every article in the list, retrieve the info from the database.

The operations of insert and remove from the database repository, only use `INSERT` and `DELETE` SQL statements and do not use any `SELECT` statement. In this case, where we only preform write operations in database, no read-write conflict will ever occur, and therefore we can use the PostgreSQL DBMS resorting to the standard SQL SELECT statement.

4.4 Functional Evaluation

To validate the correction of the unified model algorithm, we developed an alternative version of the same testing application that would periodically validate the coherence between the data present in memory against its equivalent in the database. The assertion test was: *every data that is in memory must be also in database*. An assertion failure would mean that a coherency problem was found, either as a bug in the algorithm or in its implementation. We made several long runs of the application. In each run, the application was periodically paused to verify the consistency assertion, and later resumed. Although this test only provides statistical confidence on the correction of our algorithm, we must say that in the many accumulated hours of execution, the assertion was never broken.

4.5 Performance Evaluation

The performance of the unified model was evaluated by measuring the transactional throughput (completed transactions per time unit). A total of four versions of the application have been developed: one using the unified model as described above; another using coarse grain locks, where each repository operation was protected with a global lock; a third using finer grain locks, where each hash table bucket has a separate lock; and finally a version where the indexing structure was not replicated in main memory, i.e., the lookup operation required querying the database with `SELECT` statements.

Since each repository operation deals with more than one hash table, the implementation for finer-grain locks was very complex and error prone, as any small mistake in the management of the locks, including the order in which they were acquired and released, would cause a deadlock.

Each of the application variants were tested in four different environments: read dominant context and a write dominant contexts; and low and high contention contexts. The tests are characterized by three type of operations: `insert`, `remove`, and `lookup`. The insert and remove operations are *read-write* operations, while the lookup by author or by keyword operations are *read-only*. Each operation has a predefined probability defined as an application parameter. The maximum number of different articles to be inserted in the repository will control the contention level and will be also defined as an application parameter.

The tests were performed in a Sun Fire X4600 M2 x64 server, with eight dual-core AMD Opteron Model 8220 processors @ 2.8 GHz, 1024 KBytes of cache in each processor and a total of 32 GByte of RAM. The database management system used was PostgreSQL 8.3. Figures 4 and 5 show the results for the different testing configurations. In these tests, each article always had two authors and two keywords associated.

The first remark is that in read-dominant contexts (Fig. 4) the database version performance is poor comparing to the other versions. This was an expected result because the database has to access the secondary memory for every

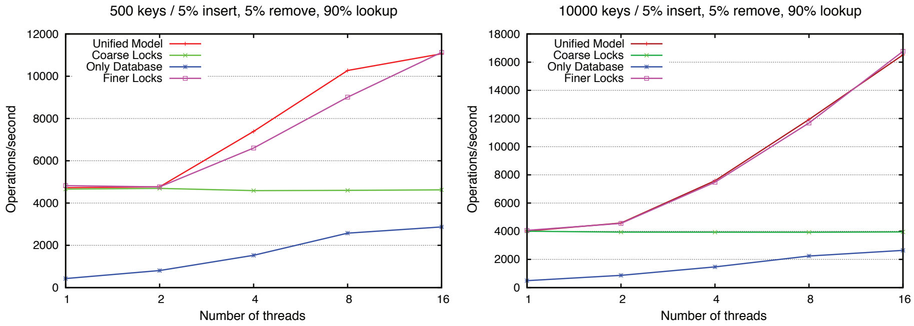

Fig. 4. Article repository used in a read dominant context, with high contention (left) and low contention (right)

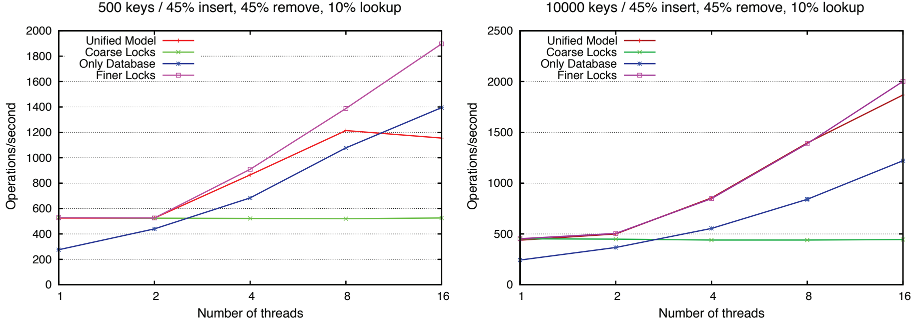

Fig. 5. Article repository used in a write dominant context, with high contention (left) and low contention (right)

read, while the other versions only have to access the main memory which is a few orders of magnitude faster. Another important aspect that can be identified from the graphics is that the difference between finer grain locks and our unified model is solely affected by the contention level. Our unified model scales as well as finer grain locks for low level contention environments, and development of the application version using the unified model implementation is much simpler than the equivalent version with finer grain locks. The salability of the unified model decreases when the contention level increases. This is more notorious when the number of parallel threads (each running a transaction) reach the maximum numbers of processors available. We believe this is due to the CPU time waisted by transactions that abort by contention conflicts, while the finer grain locks version blocks the thread when facing contention, never wasting CPU time.

For write-dominant contexts (Fig. 5), the database version scales almost as well as the other ones. This is due to the fact that now almost all the operations (90%) are either inserts or removes, and in both cases it is mandatory to always access the database. Please note that the vertical scales in the read- and write-dominant contexts are different, and that here is no performance improvement in the database-only version for the write-dominant context. In this case, all the remaining versions that use transactional memory are performing much worse, as the vast majority of the operations require the execution of database updates.

5 Concluding Remarks

This work has proposed a new approach to unify the memory and database transactional models as a single one. The proposed model still differentiates between memory and database operations, but allow them to be freely intertwined. The unified model guarantees that the Atomicity and Isolation properties hold for both memory and database operations. Additionally, it guarantees that memory and database contents are consistent if changed within the same transaction. Although the unified model was implemented using a specific transactional memory framework and a specific DBMS, we presented the implementation requirements for achieving the same result with other transactional systems.

The unified model is a high level approach to concurrency management covering both memory and database operations. It is considerably faster that database-only solutions, and much simpler to use than those based in finer grain lock. In the future we will extend the unified model to support transaction nesting, will evaluate it with standard benchmarks such as TPC-C and TPC-W, and compare it with hybrid in-memory/on-disk databases.

Acknowledgements

Our thanks to the reviewers of this paper for their valuable comments. This work was partially supported by Sun Microsystems and Sun Microsystems Portugal under the "Sun Worldwide Marketing Loaner Agreement #11497", by the Centro de Informática e Tecnologias da Informação (CITI) and by the Fundação para a Ciência e Tecnologia (FCT/MCTES) in the Byzantium research project PTDC/EIA/74325/2006 and research grant SFRH/BD/41765/2007.

References

1. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1992)
2. Dice, D., Shalev, O., Shavit, N.: Transactional locking ii. In: Distributed Computing. Volume 4167., Springer Berlin / Heidelberg (2006) 194–208
3. Cachopo, J., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. Sci. Comput. Program. **63** (2006) 172–185
4. Herlihy, M., Luchangco, V., Moir, M., William N. Scherer, I.: Software transactional memory for dynamic-sized data structures. In: PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing, New York, NY, USA, ACM (2003) 92–101
5. Luke, D., Marathe, V.J., Spear, M.F., Scott, M.L.: Capabilities and limitations of library-based software transactional memory in c++. In: Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing, Portland, OR (2007)

6. Harris, T., Fraser, K.: Language support for lightweight transactions. In: OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications, New York, NY, USA, ACM (2003) 388–402
7. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, New York, NY, USA, ACM (2005) 48–60
8. Felber, P., Fetzer, C., Müller, U., Riegel, T., Süßkraut, M., Sturzrehm, H.: Transactifying applications using an open compiler framework. In: Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing. (2007)
9. Ni, Y., Welc, A., Adl-Tabatabai, A.R., Bach, M., Berkowits, S., Cownie, J., Geva, R., Kozhukow, S., Narayanaswamy, R., Olivier, J., Preis, S., Saha, B., Tal, A., Tian, X.: Design and implementation of transactional constructs for c/c++. SIGPLAN Not. **43** (2008) 195–212
10. Dias, R., Lourenço, J., Cunha, G.: Developing libraries using software transactional memory. ComSIS **5** (2008) 104–117
11. Blundell, C., Lewis, E.C., Martin, M.M.K.: Unrestricted transactional memory: Supporting i/o and system calls within transactions. Technical Report CIS-06-09, Department of Computer and Information Science, University of Pennsylvania (2006)
12. Harris, T.: Exceptions and side-effects in atomic blocks. Sci. Comput. Program. **58** (2005) 325–343
13. Kotselidis, C., Ansari, M., Jarvis, K., Luján, M., Kirkham, C., Watson, I.: Distm: A software transactional memory framework for clusters. In: ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing, Washington, DC, USA, IEEE Computer Society (2008) 51–58
14. Cunha, G.: Consistent state software transactional memory. Master's thesis, Universidade Nova de Lisboa (2007)
15. Lourenço, J., Cunha, G.: Testing patterns for software transactional memory engines. In: PADTAD '07: Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging, New York, NY, USA, ACM (2007) 36–42
16. Gray, J.: Notes on data base operating systems. Operating Systems (1978) 393–481
17. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ansi sql isolation levels. In: SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data, New York, NY, USA, ACM (1995) 1–10
18. : (Postgresql database management system) http://www.postgresql.com.
19. Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., Shasha, D.: Making snapshot isolation serializable. ACM Trans. Database Syst. **30** (2005) 492–528