Transactions are back—but are they the same?

"Le Retour de Martin Guerre" (Sommersby)

Pascal Felber
University of Neuchâtel
Switzerland
pascal.felber@unine.ch

Christof Fetzer
TU Dresden
Germany
christof.fetzer@tu-dresden.de

Rachid Guerraoui
EPFL
Switzerland
rachid.guerraoui@epfl.ch

Tim Harris
Microsoft Research
UK
tharris@microsoft.com




Abstract

Transactions are back in the spotlight! They are emerging in concurrent programming languages under the name of transactional memory (TM). Their new role? Concurrency control on new multi-core processors. From afar they look the same as good ol' database transactions. *But are they really?*

In this position paper, we reflect about the distinguishing features of these *memory transactions* with respect to their *database* cousins.

Disclaimer: By its very nature, this position paper does not try to avoid subjectivity.

1 Introduction

Over the last few decades, much of the gain in software performance can be attributed to increases in CPU clock frequencies. However, the last few years have seen processor frequency leveling out and the focus shifting to multi-core CPUs, i.e., chips that integrate multiple processors, as a way to provide increasing computing power. To get a continued speedup on these processors, applications need to be able to harness the parallelism of the underlying hardware. This is commonly achieved using multi-threading.

Yet writing correct and scalable multi-threaded programs is far from trivial. While it is well known that shared resources must be protected from concurrent accesses to avoid data corruption, guarding individual resources is often not sufficient. Sets of semantically related actions may need to execute in mutual exclusion to avoid semantic inconsistencies. Currently, most multi-threaded applications use lock-based synchronization, which is not always adequate: coarse-grained locking limits concurrency and scales poorly, while fine-grained locking is inherently complex and error-prone, leading to problems such as deadlocks and priority inversions.

Concurrency control has been studied for decades in the field of database systems, where different operations can access tables simultaneously without observing interference. *Transactions* are a powerful mechanism to manage such concurrent accesses to a database. Transactions guarantee the four so-called ACID properties: *atomicity*, i.e., transactions execute completely or not at all; *consistency*, i.e., transactions are a correct transformation of the state; *isolation*, i.e., even though transactions execute concurrently, it appears for each transaction T that other transactions execute either before T or after T, but not both; and *durability*, i.e., modifications performed by completed transactions survive failures. This behavior is implemented by controlling access to shared data and undoing the actions of a transaction that did not complete successfully (roll-back).

The synchronization problems encountered by multi-threaded applications are somewhat reminiscent of those encountered in a database. Shared objects must be accessed in isolation by multiple threads, while consistency and atomicity must be maintained for sets of semantically-related actions.

The concept of transactions has recently been proposed as a mechanism to manage concurrent accesses to shared (in-memory) data in multi-threaded applications. *Transactional memory* [32] provides programmers with constructs to delimit transactional operations and implicitly takes care of the correctness of concurrent accesses to shared data. Such *memory transactions* have constituted an active field of research over the last few years, e.g., [15, 18, 19, 31, 24, 5, 11, 10, 28, 30]. Transactions provide the programmer with a high-level construct—simple to use, familiar, efficient, and safe—to delimit the statements of its application that need to execute in isolation.

Clearly, memory transactions share many commonalities with database transactions, from terminology and syntactical similarities to the properties they provide. In this position paper, we argue that memory transactions differ from database transactions in several important areas. They offer new research opportunities and carry promising perspectives for the development of future applications on multi-core computers. We discuss these specificities along three dimensions:

- The *language* dimension relates to the way transactions are supported by programming languages and libraries for software developers;

- The *semantics* dimension focuses on the consistency and progress guarantees provided by transactions;
- The *implementation* dimension discusses various design and realization aspects that differentiate both approaches.

The organization of the paper follows these three dimensions.

2 Programming Languages

The first topic we turn to is the manner in which transactions are exposed to programmers.

How to demarcate transactions?

One needs to define when a transaction starts, when it commits or aborts, and whether to re-execute it upon abort. With database transactions, SQL and its extensions form the dominant way of issuing queries and updates to a database. Transactions are explicitly constructed from individual SQL statements or by a series of such statements. Typically, individual statements are executed as separate transactions, and series of statements are grouped into transactions by explicit operations to start a transaction and to attempt to commit it.

Memory transactions are typically used as an implementation of *atomic blocks* (although several alternatives have been proposed during the last decades [23, 35, 25]) demarcated by the programmer. Such blocks of code identify critical regions that should appear to execute atomically and in isolation from other threads.

Several approaches have been considered to make the code inside the block transactional: language-level constructs coupled with a custom compiler or virtual machine (e.g., [15]), source code or bytecode instrumentation (e.g., [7]), code weaving using AOP[1] (e.g., [29]), or high-level APIs and runtime support. In certain extreme cases, memory transactions are hidden from the programmer and automatically generated. This can be provided for instance at the level of individual methods, by executing the body of the method in the context of a new transaction. This idea originates in Argus [22] where nested invocations correspond to nested transactions. Obviously, not all methods need to be transactional: one can use various mechanisms for specifying which methods are transactional, such as annotations as supported by Java and several other programming languages, and one can fork independent transactions through asynchronous invocations [9].[2]

Determining the right balance between expressiveness and efficiency is generally tricky and many research challenges, possibly involving compilation techniques, remain open.

[1] Aspectizing transactions is appealing but raises many non-trivial issues [21].

[2] Declarative transaction demarcation around methods can be compared to container-managed transaction demarcation in EJB [6].

Do transactional objects need to be segregated?

When accessing a database through SQL from an ordinary programming language, such as C# or Java, there is typically a strict segregation between data under the control of the DB and that under the control of the programming language; the types involved are different (tables versus objects) as are the operations for accessing them (select/update versus read/write).

Historically, this data segregation can be seen as a consequence of the separation between an application and the database that it accesses—the data is held remotely in the database, and not locally in the application. Typically, the programmer must optimize the application logic to reduce the number of interactions between the database and the application, and keep them as short and infrequent as possible.

This segregation is often not present in programming languages with atomic blocks or in STM libraries. For instance, extensions to C# [17], Java [2] and new "transactional" languages [4] do not make any distinction between ordinary objects and transactionally-accessed objects, nor between ordinary memory accesses and transactional memory accesses.

It is also unrealistic to take an existing application and transparently map its in-memory data structures to a database without prohibitive overhead.[3] In contrast, this is exactly what memory transactions are meant for: The goal is typically to take general-purpose sequential code and make it multi-thread-safe by having sections of it execute atomically and in isolation. Memory transactions constitute in this sense a lightweight approach to guaranteeing consistency without sacrificing scalability in concurrent applications.

Not segregating transactional state raises an important question that does not hold with database transactions: what happens if the *same* data is accessed in both modes? For instance, consider the following pair of operations that could be performed by two threads:

```
// Initially x == 0
// Thread 1                 // Thread 2
atomic {
  x = 42;                   temp = x;
}
```

What are the possible results for `temp` after executing this code? Indeed, is this a correctly synchronized program at all? The analogous question with database transactions would perhaps be "what guarantees are provided if a table is accessed directly through the file system at the same time as from a database transaction?" This question is typically not relevant for databases: management operations on database files (such as backup) are usually performed with the database offline, or under the exclusive control of the DBMS.

More subtle problems occur when the same data changes of access mode over time: for example if it is initialized directly by one thread, subsequently accessed transactionally by several threads, and finally cleaned-up by the original thread. For instance, consider the following code where `x_shared` is used to indicate whether or not `x` is to be accessed transactionally:

[3] There exist transparent approaches to persistence (e.g., hibernate [20]) and transaction management (e.g., entity beans in EJB [6]), but they are only effective when the data is properly encapsulated in specific types of objects and interaction follow well-defined patterns.

```
// Initially x == 0, x_shared == true
// Thread 1                 // Thread 2
atomic {                    atomic {
  if (x_shared) {             x_shared = false;
    x = 42;                 }
  }                         x ++;
}
```

One may reason informally that either `x==43` or `x==1` depending on the serialization order of the two transactions. However, many implementations of `atomic` blocks will produce other answers—for example `x==42` [1].

There are several points of view here. One is that this kind of sharing between transacted and non-transacted accesses should be prevented through the type system [16]. A second is that a language should support "strong atomicity" [3] in which each non-transacted access behaves like a serializable transaction [33]. An intermediate point is that a class of "correctly synchronized" idioms should be supported [1].

None of these approaches is satisfactory and new ideas need to be proposed and determined in the context of languages-level transactions.

Is there a life after the death of a transaction?

Transaction completion introduces some subtle problems. The general pattern employed for ensuring isolation with TM systems is to transparently abort and re-execute the memory transaction if it cannot commit because of a conflict. If the code executed in the context of a transaction throws an exception, e.g., because an overdraft occurs when transferring money between two accounts, what should be the proper behavior?

- Should we abort the transaction and re-execute it? The same exception will probably be thrown again.
- Should we abort the transaction and propagate the exception? The state of the application may be inconsistent because an exception is thrown but its cause has been rolled back.
- Should we commit the partial changes and propagate the transaction? This corresponds to the behavior expected by the programmer for his code executing in isolation but it conflicts with the atomicity property expected from a transaction.

The third choice might seem sensible from an application programmer perspective, as it does not modify the semantics of exception handling mechanisms. However, one could also leverage transactions to implement failure atomicity by automatically undoing the changes performed before an exception is thrown [8]. Indeed, transactions were initially introduced in the database domain to simplify error handling.

In the domain of programming languages one rarely wants to have simple all or nothing semantics. Instead, for certain operations some graceful degradation is the preferred behavior. For example, a "disk full" error could be tolerated by writing log messages to the console instead,

without requiring to abort the whole computation. An option to supporting such behavior is to allow the programmer to specify alternative actions that are executed if an action fails. The use of transactions for error handling is a promising approach that needs further investigation.

3 Semantics

How isolated should a transaction be?

A transaction, be it a memory or a database one, should be *isolated* from other transactions. This intuition is the key to the selling argument that transactions reduce the difficult problem of preserving the consistency of a concurrent program into the simpler one of preserving the consistency of a set of sequential programs. But what does thus intuition mean exactly?

For database transactions, the intuition was captured through the theory of *serializability* [27]—one of the most commonly required properties of database transactions—generalized to arbitrary objects (or *strict serializability* when the real-time order of transactions is accounted for). In short, this says that a history H of possibly concurrent transactions should look like a sequential history H' of the transactions that have been committed in the original history. Clearly, this does not say what happens to live or aborted transactions in the original history H. In particular, nothing prevents a transaction from observing an inconsistent state (as long that the transaction is aborted): one that cannot be produced by a sequence of committed transactions. A transaction that observes an inconsistent state can cause various problems, even if it is later aborted. Whilst this is not really a problem in a database context where transactions run in a fully controlled environment, things are different when transactions run within an application and cannot be surrounded by control structures. A transaction that works on an inconsistent state might lead the program to throw unexpected exceptions, enter infinite loops, or access invalid memory addresses.

The problem of preventing a live transaction from observing an inconsistent state might look similar to that of preventing cascading aborts. This issue was addressed in the database world through the *recoverability* [14] theory. This theory puts restrictions on the state observed by *every* transaction, including live ones. Intuitively, recoverability says that no transaction should read an update from another live transaction. It may seem at first that recoverability, when combined with serializability, matches the requirements of memory transactions. This is not the case however: a transaction might read two updates produced by two committed transactions, one overwriting the other. While respecting recoverability, such a scenario would be inconsistent [12]. A new precise formulation is needed, and writing it down carefully is not trivial.

Is the world really all made of transactions?

Memory transactions might be composed with legacy concurrent code initially written without transactional support. In this case, and ideally, one would expect that every non-transactional operation be automatically transformed into a transaction that cannot abort. This is a clear departure from traditional databases where all concurrent code is supposed to be encapsulated within trans-

actions, each possibly committing or aborting. Precisely capturing the idea that every operation be encapsulated inside an immortal transaction is however not trivial.

A more pragmatic approach consists in requiring consistency only for transactional code [34] and exempting non-transactional code from any such requirement. While weird, this approach might lead to significant performance gains with respect to automatically transforming non-transactional operations into transactions. In a sense, this is like assuming that the program executes as if certain threads (transactions) were executed under a single global lock but some of the threads do not need to acquire the lock. Threads that access objects outside transactions have no guarantee of consistency. Minimizing the impact of this freedom on the semantics of transactions is not obvious.

A drastic approach consists in partitioning the objects, at any point in time, into those that are shared, accessed through transactions, and those that are private to a thread. (Notice that an object might be private at some point in time and shared at some other point in time.) This *privatization* problem does simply not hold for database transactions.

To boost the performance of transactions, the design decisions that need to be made are not the same when dealing with databases or memory.

How durable is a transaction?

It is sometimes argued that memory transactions do not need to be durable; i.e., memory transactions are simply ACI. Such a statement has to be taken with care. It is indeed expected that the effects of committed memory transactions be durable and accessible to other transactions. The difference is that the effects of a memory transaction do not need to survive the crash of the process hosting the transaction whereas those of a database transaction need to. In a sense, it is a different level of durability.

This difference has an impact. The cost of storing information on disk is several orders of magnitude higher than storing it in shared memory. Optimization criteria for accesses to disk and memory are not the same. One typically wants to serialize the accesses to the disk in order to improve performance and use specific index structures like B-trees. The mapping of the data tables to the discs also affects performance. These differences are the same as those between traditional disk-based databases and main memory databases [26]. As we will discuss below however, there are other significant differences between memory transactions and transactions in main memory databases.

Should transactions be sequential or parallel?

Main memory databases differ significantly from memory transactions with respect to how concurrency is viewed. It is often argued that transactions in main memory databases should perform best if executed sequentially [26] because this saves the overhead related to concurrency control as well as CPU cache flushes. In short, the goal is clearly to optimize for single processors with high performance computers. Cache flushes in this case are equivalent to thousands of instructions and this indeed calls for sequential executions.

In contrast, memory transactions are optimized for multi-core architectures. The objective is to keep the processors busy and schedule as many transactions as possible: throughput is the goal. This difference is reflected to a large extend with the benchmarks that are considered. In *Bench7*, a classical benchmark for databases, the ultimate goal is to speed up every individual transaction. There is no actual support to measure concurrency and throughput: it is a pure sequential program. Adapting such a benchmark to memory transactions require involved concurrency constructs [13].

When looking at throughput, what really matters is how many transactions do commit. This is intimately related to the progress property that a TM system should feature. In databases, it might be acceptable to abort certain transactions. This would simply lead, in many cases, to skipping certain updates that will be overwritten anyway. When transactions are sophisticated programs, some minimal progress property should be ensured. What can we require in this case? It is easy to see that no implementation can ensure that every transaction commits. But can we ensure that every transaction eventually commits? Even this is very hard to achieve without hampering performance [12]. Some work has been done around contention management and how to boost the liveness of memory transactions [11], but this research is still at its infancy and much work remains to be done.

What is the state of a transaction?

As transactions may abort and roll back, a TM implementation must provide support for checkpointing shared data accessed by transactions. Unlike DB tables, which have a well-defined format, a TM must deal with any type of shared data allowed by the programming language.

Depending on the type of TM implementation (word-based or object-based) and the programming language, state management can become quite intricate. With some languages, one must distinguish between primitive types and objects, as well as deal with complex graphs while checkpointing whole objects. Updates may be performed directly to shared data, keeping an "undo log" to handle aborts, or updates may be performed on private data and written to shared memory upon commit. Such design decisions obviously affect the performance of the TM implementation, but also the complexity of state management in various programming languages. Note that the problem of state management is not restricted to TM systems: similar problems are encountered when dealing with object replication, persistence, or migration.

Arguably, the most challenging issue with memory transactions is intercepting read and write accesses to transactional data before redirecting them to the relevant instance (an old version, a thread-local copy, or the actual data). In object-oriented languages, when dealing with properly encapsulated objects, it suffices to intercept method calls. Yet, it is not always obvious to distinguish a read from a write method: this distinction is important for reducing contention because read accesses create less conflicts than write accesses.

When dealing with other types of data accesses, we need to handle every set and get operation on transactional variables, which is far from trivial. This can be achieved with acceptable effort using code analysis and compiler support.

4 Concluding Remarks

Although similar in spirit, memory transactions have some specificities that induce different challenges than those extensively addressed in the database world. There is space for new research and it is not all about re-inventing the wheel.

References

[1] Martin Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL 2008*, January 2008.

[2] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proceedings of PLDI*, Jun 2006.

[3] Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *Proc. 2005 Workshop on Duplicating, Deconstructing and Debunking*, 2005.

[4] Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. Atomos transactional programming language. In *Proceedings of PLDI*, Jun 2006.

[5] C. Cole and M.P. Herlihy. Snapshots and software transactional memory. *Science of Computer Programming*, 2005.

[6] Enterprise JavaBeans. http://java.sun.com/products/ejb/.

[7] P. Felber, C. Fetzer, U. Müller, T. Riegel, M. Süsskraut, and H. Sturzrehm. Transactifying applications using an open compiler framework. In *TRANSACT'07*, August 2007.

[8] C. Fetzer, P. Felber, and K. Högstedt. Automatic detection and masking of non-atomic exception handling. *IEEE Transactions on Software Engineering*, 30(8):547–560, 2004.

[9] Rachid Guerraoui, Riccardo Capobianchi, Agne Lanusse, and Pierre Roux. Nesting actions through asynchronous message passing. In *Proceedings of ECOOP*, June 1992.

[10] Rachid Guerraoui, Maurice Herlihy, Michal Kapalka, and Bastian Pochon. Robust contention management in software transactional memory. In *Proceedings of SCOOL*, 2005.

[11] Rachid Guerraoui, Maurice Herlihy, and Sebastian Pochon. Toward a theory of transactional contention managers. In *Proceedings of PODC*, Jul 2005.

[12] Rachid Guerraoui and Michał Kapałka. On the atomicity of transactional memory. In *ACM Proceedings of PPoPP*, 2007.

[13] Rachid Guerraoui, Michał Kapałka, and Jan Vitek. STMBench7: A benchmark for software transactional memory. In *Proceedings of the Second European Systems Conference EuroSys 2007*, 2007.

[14] Vassos Hadzilacos. A theory of reliability in database systems. *Journal of the ACM*, 35(1):121–145, 1988.

[15] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of OOPSLA*, pages 388–402, Oct 2003.

[16] Tim Harris, Maurice Herlihy, Simon Marlow, and Simon Peyton-Jones. Composable memory transactions. In *Proceedings of PPOPP*, Jun 2005.

[17] Timothy Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *Proceedings of PLDI*, Jun 2006.

[18] M. Herlihy, V. Luchangco, M. Moir, and W.N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of PODC*, pages 92–101, Jul 2003.

[19] Maurice Herlihy. The transactional manifesto: software engineering and non-blocking synchronization. In *Proceedings of PLDI*, 2005.

[20] hibernate. http://www.hibernate.org.

[21] Jörg Kienzle and Rachid Guerraoui. Aop: Does it make sense? the case of concurrency and failures. In *Proceedings of ECOOP*, pages 37–61, 2002.

[22] Barbara Liskov. Distributed programming in argus. *Commun. ACM*, 31(3):300–312, 1988.

[23] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *Proceedings of an ACM conference on Language design for reliable software*, pages 128–137, 1977.

[24] Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Adaptive software transactional memory. In *Proceedings of DISC*, pages 354–368, 2005.

[25] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 346–358, New York, NY, USA, 2006. ACM Press.

[26] H. Garcia Molina and K. Salem. Automatic detection and masking of non-atomic exception handling. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[27] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[28] Torvald Riegel, Pascal Felber, and Christof Fetzer. A Lazy Snapshot Algorithm with Eager Validation. In *20th International Symposium on Distributed Computing (DISC)*, September 2006.

[29] Torvald Riegel, Christof Fetzer, and Pascal Felber. Snapshot isolation for software transactional memory. In *TRANSACT'06*, Jun 2006.

[30] Torvald Riegel, Christof Fetzer, and Pascal Felber. Time-based Transactional Memory with Scalable Time Bases. In *19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, June 2007.

[31] W.N. Scherer III and M.L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of PODC*, pages 240–248, Jul 2005.

[32] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of PODC*, Aug 1995.

[33] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in STM. In *PLDI'07: Proc. 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 78–88, 2007.

[34] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 338–339, 2007.

[35] Andy Wellings and Alan Burns. Implementing atomic actions in ada 95. *IEEE Trans. Softw. Eng.*, 23(2):107–123, 1997.