ACM SIGACT News Distributed Computing Column 29

Idit Keidar
Dept. of Electrical Engineering, Technion
Haifa, 32000, Israel
idish@ee.technion.ac.il

Concurrent computing, once an esoteric pastime of Distributed Computing Theorists and High Performance Computing Extremists, is suddenly important in the computing world at large. This new interest in concurrency stems from a dramatic paradigm shift in computer architecture: No longer are hardware manufacturers making faster and faster (uni-)processors. Nowadays, chip companies are producing multi-processors with more and more cores. Only concurrent (multi-threaded) programs can effectively exploit the potential of such multi-core processors. And thus, as multi-processors become mainstream, so does concurrent computing.

This column features three contributions that reflect on the role of Distributed Computing research in the brave new world of multi-processors. What new challenges are raised by the new-found ubiquitous relevance of concurrency? What can we, the Distributed Computing community, do to address them?

The most notable challenge stemming from the shift to multi-core architectures is the difficulty of programming concurrent code. One concept that tackles this challenge is *transactional memory*, which allows multiple processes (or threads) to concurrently access memory objects with transaction-like consistency semantics. As this concept is already taken seriously by industry, and real-world implementations begin to materialize, one may ask what role the Distributed Computing community has in its continued development. This column provides three different takes on this question.

The first contribution, by Pascal Felber, Christof Fetzer, Rachid Guerraoui, and Tim Harris, questions whether transactions in memory are any different from the well-studied database transactions; and if there are differences, what do they entail? The authors point out some aspects in which conventional database wisdom does not provide adequate answers for transactional memory systems. The second contribution, by Hagit Attiya, is a call for more foundational research, or Theory, for transactional memory. Last but not least, Maurice Herlihy and Victor Luchangco provide their perspective on the role of Distributed Computing in the multi-core revolution. All three highlight some promising research directions.

Many thanks to Hagit, Pascal, Christof, Rachid, Tim, Maurice, and Victor for their contributions.

Call for contributions: Please send me suggestions for material to include in this column, including news, communications, and authors willing to write a guest column or to review an event related to distributed computing. In particular, I welcome tutorials, either exposing the community to new and interesting topics, or providing new insight on well-studied topics by organizing them in new ways. I also welcome open problems with a short description of context and motivation.

Transactions are back—but are they the same?

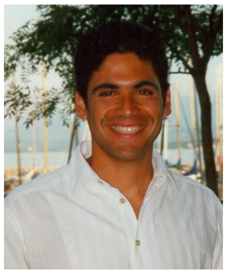
"Le Retour de Martin Guerre" (Sommersby)

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 transactions, 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.

Needed: Foundations for Transactional Memory

Hagit Attiya
Dept. of Computer Science, Technion
hagit@cs.technion.ac.il

"It's all very well in practice, but it will never work in theory"
(French management saying)

Transactional memory is a leading paradigm for designing concurrent applications for tomorrow's multi-core architectures. It follows and draws much inspiration from earlier research on concurrent data structures and concurrency control. Quite remarkably, it has succeeded in breaking out of the research community, and is being seriously considered by the industry—both as part of software solutions and as the basis for novel hardware designs.

But this success comes at a price, as every new research paper is now being judged by its immediate relevance and applicability to current technology, while ignoring the long-term development of foundations for this important area.

So, while there is a large body of practical work, the theoretical principles are still lacking: There are no agreed-upon concepts, and even the terminology is muddled; there is great confusion between specifications, policies and implementations; even when they exist, correctness properties do not distinguish between safety, liveness and performance.

This lack of foundations hinders communication and interaction, both within the community of researchers investigating concurrent data structures and in its interactions with other communities, most notably those investigating programming languages and verification.

Previous foundations developed for similar architectural advances include *concurrency control* theory [11] and consistency models for *distributed shared memory* [9]. This experience indicates what should be incorporated in a theory for transactional memory—and in a broader perspective, concurrent data structures.

The first ingredient are *specifications*. These should include clear interfaces, as well as definitions of safety and liveness properties. There are several possibilities for picking these properties, differing along several dimensions, including, most importantly, the following issues:

(1) Is the transaction stated as a list of atomic actions (like in classical database transactions) or more semantically as capturing a high-level operation (as in distributed shared memory)? This choice determines whether the data set of a transaction is explicitly stated (in the former case) or not; in turn, this has implications on detecting dependencies and conflicts among transactions.

(2) Are the intermediate states of the transaction observable and hence they should be consistent (as in *view serializability*) or should only the final state be checked for consistency?

(3) Should the real-time order among transactions be respected (as in *linearizability* [8] and *strict serializability* [10]) or not?

(4) What liveness properties should be demanded, namely, when are transactions guaranteed to terminate?

Obviously, these aspects are not always orthogonal, and they interact with performance specifications and issues of transactions' abort. There is initial study of these concerns [7], but further elucidation is needed and the consequences should be understood.

My belief is that the difficult choice between possibilities will not yield a single agreed-upon specification, but several alternatives corresponding to interesting combinations of the above dimensions. (As is the state of affairs in specifications of *group communication* middleware [4].) Even a clear choice between a small number of specifications will make feasible the derivation of verification techniques. This will let us address one of the most important challenges of multi-core programming, namely, validating that applications are indeed correct.

Comparison between various specifications, so as to choose the most appropriate for a certain set-up, will be assisted by *complexity measures*. Such measures should allow evaluating the worst case, as was done for distributed shared memories [3], perhaps even the average case. Because transactional memory is optimized for the *common case*, it is also important to figure out what is the best case and evaluate its cost.

The biggest question seems to be *what* to measure exactly? Since most transactional memory designs allow individual transactions to starve, they must be evaluated according to the overall performance. One suggestion is to measure their *Makespan* (the total time to complete a set of transactions) compared to their execution by a clairvoyant scheduler [6, 2], but this is an overly pessimistic measure that does not distinguish between designs that behave differently in practice. A recent proposal suggests taking into account the data conflicts when evaluating performance, allowing interferences only among closely conflicting transactions [1].

Once there are accepted complexity measures, they should be used to appraise proposed implementations. What would be even more beneficial would be the derivation of impossibility results, since these have the foremost implications on what can be achieved. This includes lower bounds on the costs of obtaining various safety and liveness properties, and trade-offs among them.

Developing such complexity measures will allow evaluating designs beyond existing technology. Currently, the performance of proposed designs is mostly measured through benchmarks. These benchmarks are often micro-scale and therefore, non-predictive for large-scale deployment [5]; moreover, it is not obvious what kind of workloads should be used even in full-scale benchmarks.

Finally, good theory will help us learn from the past, namely, by using results of previous research on similar topics like distributed shared memory and database concurrency control. It also allows us to leave the lessons for the future, because while concurrent access to shared data will always be an innate factor of computing, technology and computer architecture will eventually be transformed in way that will make transactional memory obsolete and necessitate new solutions. When this time comes, good principles will allow exploiting ideas from current research to address future needs.

References

[1] Hagit Attiya. Concurrency and the Principle of Data Locality. *IEEE Distributed Systems Online*, 8(9), September 2007.

[2] Hagit Attiya, Leah Epstein, Hadas Shachnai and Tami Tamir. Transactional Contention Management as a Non-Clairvoyant Scheduling Problem In *Proceedings of the 25th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 308–315, 2006.

[3] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, May 1994.

[4] Gregory V. Chockler, Idit Keidar and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, December 2001.

[5] Aleksandar Dragojevic, Rachid Guerraoui and Michal Kapalka. Dividing Transactional Memories by Zero. To appear in *3rd ACM SIGPLAN Workshop on Transactional Computing*, 2008.

[6] Rachid Guerraoui, Maurice Herlihy and Bastian Pochon. Toward a Theory of Transactional Contention Management. In *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 258–264, 2005.

[7] Rachid Guerraoui and Michal Kapalka. On the Correctness of Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008, to appear.

[8] Maurice Herlihy and Jeanette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, June 1990.

[9] Bill Nitzberg and Virginia Lo. Distributed shared memory: a survey of issues and algorithms. *Computer*, 24(8)52–60, August 1991.

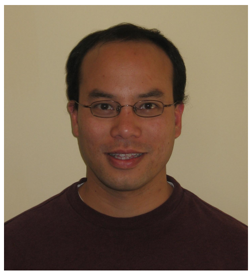
[10] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4)631–653, October 1979.

[11] Gerhard Weikum and Gottfried Vossen, *Transactional Information Systems*, Morgan Kaufmann, 2001.

Distributed Computing and the Multicore Revolution

Maurice Herlihy
Computer Science Department, Brown University
herlihy@cs.brown.edu

Victor Luchangco
Sun Microsystems Laboratories
victor.luchangco@sun.com



1 Introduction

Changes in technology can have far-reaching effects on theory. For example, while Turing's work on computability predated the first electronic computers, complexity theory flowered only after computers became a reality. After all, an algorithm's complexity may not matter much in a mathematics journal, but matters quite a bit in a FORTRAN program. We argue that something similar is going on with parallel and concurrent computation: after decades of being respected but not taken seriously, research on multiprocessor algorithms and data structures is going mainstream.

To make a long story short, in the near future, nearly all computers, ranging from supercomputers to smoke detectors, will be shared-memory multiprocessors. This change will affect the distributed computing community in two ways. First, the experience of confronting real multiprocessors, like early theorists' experience confronting FORTRAN, will force us to address problems obscured by many of today's elegant but naive computational models. Second, perhaps for the first time ever, multiprocessor research matters to people outside the multiprocessor research community. Let us expand on this latter point.

For a long time, advances in multiprocessors were overshadowed by the ever-increasing speed of uniprocessors. By the time any particular multiprocessor system was actually working, one could often buy a cheaper off-the-shelf uniprocessor with better computational power. At best, multiprocessors found a niche as exotic, high-end products.

Those days, however, appear to be over. It is harder and harder to increase processor clock speed (the chips overheat), but easier and easier to cram more processor cores onto a chip (thanks to Moore's Law). As a result, uniprocessors are giving way to dual-cores, dual-cores to quad-cores, and so on. Concurrent architectures have arrived because chip manufacturers had no choice: they can no longer provide increased value by increasing clock speed, so they are forced to increase concurrency.

There is one problem. Except for "embarrassingly parallel" applications, no one really knows how to exploit lots of cheap threads; not application programmers, not operating system designers, and perhaps not even the hardware architects. There are many ideas, and many vigorous debates, but it is too early to tell what works.

This multicore challenge has implications for all branches of computer science. Our ability to attract good students, to attract funding, to feel we are addressing important questions, all depend on the sense that computer science is a vibrant and growing field. Ultimately, this growth, even for theory, is driven by Moore's Law: we can think about doing things next year that would have been impractical last year. Today, we regularly upgrade our laptops and software not because they break, but because the new versions provide more value. If we fail to meet the multicore challenge, then laptops will become like washing machines: we buy a new one only when the old one breaks. We do not want Computer Science to become Washing Machine Science.

Here is where the distributed computing community can play a role. With sudden relevance comes sudden responsibility: many of these challenges involve concurrent algorithms, data structures, formal models, complexity analysis and lower bounds, all areas where the community has substantial experience, and much to contribute.

This article presents the authors' opinions about some promising research questions. The choice of areas is, of course, biased by the authors' experiences and interests, and should be evaluated accordingly.

2 The Current State of Concurrent Programming

In today's programming practice, programmers typically rely on combinations of locks and conditions, such as monitors, to prevent concurrent access by different threads to the same shared data. While this approach allows programmers to treat sections of code as "atomic", and thus simplifies reasoning about interactions, it suffers from a number of severe shortcomings.

First, programmers must decide between *coarse-grained* locking, in which a large data structure is protected by a single lock, and *fine-grained* locking, in which a lock is associated with each component of the data structure. Coarse-grained locking is simple, but permits little or no concurrency, thereby preventing the program from exploiting multiple processing cores. By contrast, fine-grained locking is substantially more complicated because of the need to ensure that threads acquire all necessary locks (and only those, for good performance), and because of the need to avoid deadlock when acquiring multiple locks. The decision is further complicated by the fact that the best engineering solution may be platform-dependent, varying with different machine sizes, workloads, and so on, making it difficult to write code that is both scalable and portable.

Second, conventional locking provides poor support for code composition and reuse. For example, consider a lock-based hash table that provides atomic `insert` and `delete` methods. Ideally, it should be easy to move an element atomically from one table to another, but this kind of composition simply does not work. If the table methods synchronize internally, then there is no way to acquire and hold both locks simultaneously. If the tables export their locks, then modularity and safety are compromised.

Finally, such basic issues as the mapping from locks to data, that is, which locks protect which data, and the order in which locks must be acquired and released, are all based on convention, and violations are notoriously difficult to detect and debug. For these and other reasons, today's software practices make lock-based concurrent programs too difficult to develop, debug, understand, and maintain.

The distributed computing community has addressed this issue for more than fifteen years by developing nonblocking algorithms [25] for stacks, queues, deques and other data structures (e.g., [5, 22, 38, 36, 45]). These algorithms are subtle, and the discovery of a new one is sufficiently difficult that it merits publication at a top theory conference. Although wonderful for theory researchers, this is hardly a task for typical everyday programmers. If concurrent programming is to be commonplace, practitioners must not require years of study to gain proficiency and everyday problems must not require research to solve.

3 Transactional Memory

Recently the *transactional memory* programming paradigm [29] has gained momentum as an alternative to locks in concurrent programming. With transactional memory, rather than using locks to give the illusion of atomicity by preventing concurrent access to shared data, programmers designate regions of code as transactions, and the system guarantees that such code appears to execute atomically. A transaction that cannot complete is aborted—its effects are discarded—and may be retried. Transactions have been used to build large, complex and reliable database systems for over thirty years [16]; with transactional memory, we hope to translate that success to shared-memory multiprocessor systems. The underlying system may use locks or nonblocking algorithms to implement transactions, but the complexity is hidden from the application programmer. Proposals exist for implementing transactional memory in hardware (e.g., [4, 21, 29, 35, 39, 40]), in software (e.g., [23, 24, 28, 30, 33, 37, 46]), and in schemes that mix hardware and software (e.g., [13, 31, 43, 47, 51]). This area is growing at a fast pace; a list of citations can be found on the transactional memory web page at `http://www.cs.wisc.edu/trans-memory/biblio/list.html`.

There are lots of opportunities for the distributed computing researchers to contribute to the state of the art of transactional memory. One possibility, of course, is to design new transactional memory algorithms, which may improve the time or space complexity, or perhaps the progress guarantees, of previous proposals. Or they might allow greater concurrency among transactions, make different assumptions about the underlying hardware support, or just be simpler algorithms.

Many transactional memory implementations "optimistically" execute transactions in parallel, with conflicts resolved by a contention manager [28], which decides whether each transaction continues, waits or is aborted. The contention management policy of a transactional memory implementation can have a profound effect on its performance, and even its progress guarantees. Some evaluation and analysis of contention management policies has been done [18, 19, 53], but we do not yet have a theory of contention management to aid practitioners in choosing policies appropriate for their applications.

When evaluating transactional memory designs, it is important to be sensitive to the real-world concerns of practitioners, who care not only about asymptotic behavior but also about the constant factors. Indeed, defining relevant complexity measures and progress guarantees—ones that actually interest practitioners—is an open challenge.

We need more experience developing, using and evaluating transactional-memory-based algorithms. Thus far, most programs using transactional memory have replaced locked critical sections with transactions (e.g., [20, 42]). However, the promise of transactional memory is to make concurrent programming simpler, so we conjecture that programs designed from the start to use transactional memory will look quite different from lock-based programs. For example, a transactional memory that allows a transaction to explicitly abort itself may encourage a speculative style of programming. We also expect that the costs in transactional-memory-based programs will be quite different.

All proposed transactional memory implementations so far prevent read/write conflicts among committing transactions. However, it is well known that this is too restrictive for certain kinds of data structures, and that more sophisticated approaches can permit more concurrency. Much work has been done on using commutativity and related properties, and many algorithms exist, particularly from the database community. To exploit these algorithms, we may need to extend the transactional memory interface to allow programmers to specify conflict relations.

Although transactional memory is a promising approach for concurrent programming, it is not a panacea, and in any case, transactional programs will need to interact with legacy code, which may use locks or other means to control concurrency. We will also aid the adoption of transactional memory if we can easily convert legacy programs to transactional ones. Transactional boosting is an example of such a technique [26]; we need to develop more of them.

One major challenge for the adoption of transactional memory is that it has no universally accepted specification. Consider, for example, the following questions:

- Can memory be accessed outside a transaction, and if so, what are the guarantees for such memory access, and for transactional accesses that may be concurrent with nontransactional accesses? (This is often presented as a choice between "strong" or "weak" atomicity [34], though there is really a range of possible guarantees.)
- What effect does relaxed memory consistency have on transactional memory?
- Is there some useful notion of relaxed transactional guarantees?
- How should transactional memory interact with locks, I/O and system calls?
- How should exceptions and asynchronous interrupts within a transaction be handled?

Although preliminary investigation and discussion of these issues has been done [7, 11, 17, 32, 52, 54], there is no consensus on the answers to these questions. Indeed, different answers are probably appropriate for different settings. For example, it may be too expensive to guarantee strong atomicity with a purely software implementation of transactional memory (assuming transactional and nontransactional access is permitted), whereas a hardware implementation may be perfectly reasonable.

Whatever semantics we desire for transactional memory, we need to specify it precisely so that we can verify both implementations of the transactional memory and applications built on top of it. Again, there is some nascent work (e.g., [32, 41, 44]), but we still have a long way to go. And to our knowledge, only trivial transactional memory implementations have been verified thus far [8, 9], not any ones that have actually been proposed.

Finally, we would like to see lower bounds and other impossibility results. For example, is there an inherent cost to guaranteeing strong atomicity? For such results, we need a precise specification of the underlying computation model, one that captures the capabilities and limitations of real shared-memory multiprocessors (e.g., with atomic synchronization primitives such as compare-and-set, rather than just read/write shared memory).

Many of these issues are not specific to transactional memory. Rather, they are relevant to any modern shared-memory multiprocessor. We discuss some of these issues in this more general context below.

4 System Model

Good theory rests on the strength of its underlying model, and we are in dire need of good models for explaining the behavior of concurrent programs, to reason about both its semantics and its performance. We could use at least two such models.

First, we urgently need an abstract model for a shared-memory multiprocessor that properly accounts for performance. In the 80s, the PRAM model [15] became a standard model for parallel computation, and the research community developed many elegant parallel algorithms for this model. Unfortunately, PRAM assumed that processors were synchronous, and that memory could be accessed only by read and write operations. Also, it did not model the effects of contention nor the performance implications of multilevel caching, assuming instead a flat memory with uniform-cost access. More realistic models (e.g., [12]) were proposed to account for the costs of interprocess communication, but these models still assumed synchronous processors with only read and write access to memory.

Modern computer architectures are asynchronous, and their performance is heavily influenced by the effects of contention and hierarchical memory. Furthermore, to reduce synchronization costs, these architectures provide relaxed memory consistency guarantees, requiring programmers to insert explicit "memory fences" to prevent memory operations from being reordered. Memory consistency models were an active area of research in the 90s (see [1, 32] for more on these models). These architectures also provide atomic primitives such as compare-and-set (CAS) and load-linked/store-conditional (LL/SC) instructions, which are the only realistic means of synchronization on these systems. Upcoming architectures will provide hardware support for transactional memory [48, 49]. We must reconsider models for parallel computation in light of current technology, and derive foundational models and algorithms that match current practice and realistically model performance.

We also need a model for concurrent programming at a higher level, above the operating system. Because a processor may to wait a long time for a memory access, modern systems typically allow multiprogramming (i.e., multiple threads executing on a single processor). Indeed, to use processors effectively, it is often necessary to provide many more threads than processors, so that the scheduler has "slack" [50]. In addition, many concurrent programming languages allow threads to be created dynamically. We need a model that supports this functionality.

5 Progress, Performance Metrics and Lower Bounds

Obstruction-freedom is a weak nonblocking progress condition designed to give as much freedom to implementors as possible while still ensuring that no thread can be blocked by the failure of any other thread [27]. Anecdotal evidence suggests that obstruction-free algorithms are simpler, more efficient and easier to reason about than their lock-free or wait-free counterparts. However, there are no formal separation results to back up this and similar claims in a system that provides universal synchronization primitives such as CAS. (In an asynchronous system with only read/write memory, obstruction-free consensus is solvable while lock-free and wait-free consensus are not.)

Most analyses of concurrent algorithms focus on worst-case performance rather than the performance in the "normal case". In practice, however, the worst case hardly ever happens (and if it does, you have other problems as well). Thus, worst-case performance can be a very poor predictor of the observed performance of an algorithm. Average-case performance may be no better, for example, if the algorithm can "get stuck" in the worst case. One possibility is to characterize the "normal case" in the problem and system description. Another is to adopt complexity measures such as *contention-free step complexity* [3], or *obstruction-free step complexity* [14].

Algorithms might then be characterized by both their worst-case and their normal-case performance. For algorithms that are not wait-free, the strongest progress condition may replace the worst-case performance. Ideally, we might even have a measure for "graceful degradation", which rewards an algorithm for "doing the best it can". Whether these or any other complexity measures are good indicators of actual observed performance is unknown and worth studying. We could also use a formal characterization of "scalable" algorithms.

Note that unlike for sequential computation, asymptotics are not good enough: The performance of concurrent algorithms for many problems do not differ in their order of growth, but only in the constant factors. However, these constant factors are important, especially for systems with fewer than 100 or so processors.

With appropriate complexity measures, it would interesting to prove lower bounds and impossibility results for concurrent algorithms. For example, in a system with only trivial and overwriting operations, Attiya et al. prove a lower bound on the space and obstruction-free step complexity of a large class of objects [6]. However, this result does not apply to systems that support universal synchronization primitives such as CAS, as all modern shared-memory multiprocessors do. Can it be generalized?

6 Other Research Directions

There are many other directions that may appeal to distributed computing researchers. For example, Aguilera et al. generalize the notion of aborting a transaction to general objects [2] by defining *abortable* and *query-abortable* counterparts of objects. An abortable object is useful in concurrent programming because it allows contention management to be separated from the implementation of the object, just as it can be for transactions. Perhaps transactions could in turn be defined as implementations of abortable counterparts of some simple object.

As mentioned before, contention management is an area with huge possibilities for improvement. Indeed, contention management is just a special case of scheduling. Transactions make it particularly easy because the scheduler can always abort a low-priority transaction when it conflicts with a high-priority one, so priority inversion can be avoided, unlike with lock-based programs. Lock-free and wait-free programs, on the other hand, typically include elaborate mechanisms for helping threads that get "stuck". These mechanisms act as kind of internal scheduler, and often interact badly with policies of the global system scheduler. Obstruction-free algorithms avoid helping, and thus leave all the scheduling decisions to the global system scheduler, making it easier to design, implement and evaluate various scheduling policies.

And again, formal verification of concurrent algorithms is still in its infancy, and we desperately need new and better techniques and tools for this task, not only for transactional memory implementations, but for all concurrent algorithms. We have verified some nontrivial algorithms (e.g., [10]), but these are still relatively small and took a lot of effort.

7 Conclusions

In summary, we are at the dawn of a new age in concurrent computing. For the first time, concurrency and parallelism are entering the mainstream. We, the distributed computing community, have been studying this area for decades, and are uniquely qualified to contribute. To do so effectively, however, we must attack problems relevant to practitioners (whether they know it or not). Our models must capture the technology as it has actually developed, not the way we expected it to develop back in the 80s. Our community is in a unique position to build a theory of concurrent systems that captures the essential properties of modern machines, to serve as a foundation for the practice of concurrent programming. Here, we have outlined our opinions on the challenges and opportunities we see. We hope it stirs your interest and your desire to contribute to this endeavor.

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] M. Aguilera, S. Frolund, V. Hadzilacos, S. Horn, and S. Toueg. Abortable and query-abortable objects and their efficient implementation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 2007.

[3] R. Alur and G. Taubenfeld. Contention-free step complexity of shared memory algorithms. *Information and Computation*, 126(1):62–73, Apr. 1996.

[4] S. Ananian, K. Asanovic, B. Kuszmaul, C. Leiserson, and S. Lie. Unbounded transactional memory. In *Proc. 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Feb. 2005.

[5] N. Arora, R. D. Blumofe, and C. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2):115–144, 2001.

[6] H. Attiya, R. Guerraoui, D. Hendler, and P. Kouznetsov. Synchronizing without locks is inherently expensive. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 300–307, 2006.

[7] L. Baugh and C. Zilles. An analysis of i/o and syscalls in critical sections and their implications for transactional memory. In *Workshop on Transactional Computing (TRANSACT)*, 2007.

[8] A. Cohen, J. O'Leary, A. Pnueli, M. Tuttle, and L. Zuck. Verifying correctness of transactional memories. In *Proceedings of Formal Methods in Computer Aided Design*, 2007.

[9] A. Cohen, A. Pnueli, and L. Zuck. Verification of transactional memories that support non-transactional memory accesses. In *Workshop on Transactional Computing (TRANSACT)*, 2008.

[10] R. Colvin, L. Groves, V. Luchangco, and M. Moir. Formal verification of a lazy concurrent list-based set. In *Proceedings of Computer Aided Verification*, 2006.

[11] L. Crowl, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Integrating transactional memory into c++. In *Workshop on Transactional Computing (TRANSACT)*, 2007.

[12] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, pages 1–12, 1993.

[13] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[14] F. E. Fich, V. Luchangco, M. Moir, and N. Shavit. Obstruction-free step complexity: Lock-free DCAS as an example. In *Proceedings of the 19th International Conference on Distributed Computing (DISC)*, pages 493–494, 2005.

[15] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[16] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, 1993.

[17] D. Grossman, J. Manson, and W. Pugh. What do high-level memory models mean for transactions? In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, pages 62–69, 2006.

[18] R. Guerraoui, M. Herlihy, M. Kapalka, and B. Pochon. Robust contention management in software transactional memory. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, 2005.

[19] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In P. Fraigniaud, editor, *Proceedings of the International Conference on Distributed Computing (DISC)*, volume 3724 of *Lecture Notes in Computer Science*. Springer, Sept. 2005.

[20] R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: A benchmark for software transactional memory. Technical report, EPFL, 2006.

[21] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proc. 31st Annual International Symposium on Computer Architecture*, June 2004.

[22] T. Harris. A pragmatic implementation of non-blocking linked lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC)*, pages 300–314, 2001.

[23] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 388–402. ACM Press, 2003.

[24] T. Harris, S. Marlowe, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2005.

[25] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[26] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th Annual ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008.

[27] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems*, pages 522–529, May 2003.

[28] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, pages 92–101. ACM Press, 2003.

[29] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300. ACM Press, 1993.

[30] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 151–160. ACM Press, 1994.

[31] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *Workshop on Transactional Computing (TRANSACT)*, 2007.

[32] V. Luchangco. *Memory Consistency Models for High Performance Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, 2001.

[33] V. Marathe, W. Scherer, and M. Scott. Adaptive software transactional memory. Technical Report TR 868, Computer Science Department, University of Rochester, May 2005.

[34] M. Martin, C. Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2):17, Feb. 2006.

[35] A. McDonald, J. Chung, B. Carlstrom, C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory, 2006.

[36] M. M. Michael and M. L. Scott. Nonblocking and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distribued Computing*, 51(1):1–26, 1998.

[37] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 219–228. ACM Press, 1997.

[38] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using elimination to implement scalable and lock-free fifo queues. In *Proceedings of the 17th Annual ACM Symposium on Parallel Algorithms and Architectures*, July 2005.

[39] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proc. 12th Annual International Symposium on High Performance Computer Architecture*, 2006.

[40] K. E. Moore, M. D. Hill, and D. A. Wood. Thread-level transactional memory. Technical Report CS-TR-2005-1524, Dept. of Computer Sciences, University of Wisconsin, Mar. 2005.

[41] K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *Proceedings of the 35th Annual ACM Symposium on Principles of Programming Languages*, pages 51–62, 2008.

[42] H. E. Ramadan, C. J. Rossbach, and E. Witchel. The Linux kernel: A challenging workload for transactional memory. In *Workshop on Transactional Memory Workloads*, June 2006.

[43] B. Saha, A.-R. Adl-Tabatabai, R. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 187–197, 2006.

[44] M. L. Scott. Sequential specification of transactional memory semantics. In *TRANSACT:First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[45] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM*, 53(3):379–405, 2006.

[46] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213. ACM Press, 1995.

[47] F. Tabba, C. Wang, J. R. Goodman, and M. Moir. NZTM: Nonblocking, zero-indirection transactional memory. In *Workshop on Transactional Computing (TRANSACT)*, 2007.

[48] M. Tremblay. Transactional memory for a modern microprocessor. In *26th Annual ACM Symposium on Principles on Distributed Computing*, 2007.

[49] M. Tremblay and S. Chaudhry. A third-generation 65nm 16-core 32-thread plus 32-scout-thread cmt sparc® processor. In *Proceedings of the 2008 IEEE International Solid-State Circuits Conference*, 2008.

[50] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[51] E. Vallejo, T. Harris, A. Cristal, O. Unsal, and M. Valero. Hybrid transactional memory to accelerate safe lock-based transactions. In *Workshop on Transactional Computing (TRANSACT)*, 2008.

[52] H. Volos, N. Goyal, and M. Swift. Pathological interaction of locks with transactional memory. In *Workshop on Transactional Computing (TRANSACT)*, 2008.

[53] I. William N. Scherer and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing*, pages 240–248, New York, NY, USA, 2005. ACM Press.

[54] C. Zilles and L. Baugh. Extending hardware transactional memory to support non-busy waiting and non-transactional actions. In *TRANSACT:First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.