Transactional Semantics with Zombies*

Michael L. Scott
University of Rochester
scott@cs.rochester.edu

July 2014

Abstract

Different formal models of transactional memory are required at different levels of the system stack. This paper focuses on the run-time level, where the semantics of individual operations (start, read, write, try-commit) govern the interactions between the compiler and the TM system. For *sandboxing* TM systems, which allow a doomed transaction (a "zombie") to continue for some time beyond an inconsistent read, run-time–level semantics cannot be captured by opacity as currently defined: we need a formal model of zombie execution.

1 Introduction

Transactional Memory spans multiple levels of abstraction [1]. At the language level, programmers write applications using syntactically identified atomic blocks. At the run-time level, the compiler calls methods of a speculation API (start, read, write, try-commit) to implement atomic blocks. At the system level, an STM algorithm implements the speculation API and promises some sort of (at least probabilistic) liveness to the run-time level. At the hardware (instruction set and microarchitecture) level, special instructions may (typically without any guarantee of liveness) support efficient implementation of at least some atomic blocks.

Each of these levels calls for different formalism. We focus here on the language and (especially) run-time levels. At the former, programmers typically care only about atomicity: they want *transactional sequential consistency* [6]. At the latter, the compiler needs something more complex—semantics for the various speculative operations that guarantee strict serializability for whichever transactions (attempted executions of atomic blocks) are able to commit successfully. These semantics may be explicitly concurrent (as in the *opacity* of Guerraoui and Kapałka [12]), or they may specified sequentially and then extended to the concurrent case with the standard machinery of linearizability [22]. In the former case, at least, run-time–level semantics have traditionally required that reads be mutually consistent—simultaneously valid.

Unfortunately, the requirement for consistent reads appears to preclude important implementations—specifically, those that use *sandboxing* to tolerate "zombie" transactions. The point of consistency, after all, is to prevent the creation of zombies—transactions that have seen inconsistent state but have not yet aborted, and may therefore behave unpredictably. Sandboxing, by contrast, ensures that while a zombie may be unpredictable, it is nonetheless *harmless*—unable to influence externally visible behavior between its first inconsistent read and its eventual abort.

Sandboxing appears in several STM systems for managed languages [2, 15]. It can also be implemented via binary rewriting [19] or through careful instrumentation and signal interposition [7], even in C and C++. It can also appear in hardware: the HTM of the IBM Blue Gene/Q [25], for example, is implemented entirely outside the processor core. A transaction can execute several instructions beyond an inconsistent read, but the need for kernel intervention on both faults and commits (the only dangerous events that can occur on that time scale) ensures that the processor will "sync up" with the memory system before any harm can be done. Among other things, a sandboxing runtime raises the possibility of detecting and resolving conflicts "out of band"—in a separate thread or special hardware. Casper et al. [5] and Kestor et al. [17] report significant speedups by exploiting this possibility.

One could perhaps argue for the correctness of a sandboxed implementation via some sort of equivalence to opacity. One might, for example, argue that all operations of a transaction beginning with its first inconsistent read and continuing through its subsequent abort should somehow be seen as a *single* operation—the simple abort envisioned by opacity. Aside from notational messiness, however, this strategy has the disadvantage of moving all aspects of a zombie's behavior out of the domain of

*Invited presentation, Sixth Workshop on the Theory of Transactional Memory, Paris, France, July 2014.

This work was supported in part by NSF grants CCF-0963759, CCF-1116055, CNS-1116109, CNS-1319417, CCF-1337224, and CCF-1422649, and by support from the IBM Canada Centres for Advanced Study.

TM semantics and into the implementation. One of the key advantages of opacity is that by embracing aborted transactions it allows one to consider liveness properties that will obtain for any conforming implementation. This advantage disappears if zombies aren't part of the formalism.

2 Transactional Sequential Consistency

In previous papers [6, 24], my students and I have argued that transactional sequential consistency (TSC) constitutes the "right" semantics for language-level atomic blocks. Informally, TSC requires that a program's reads and writes appear to occur in some global total order, consistent with program order in each thread, and with the accesses of each dynamic instance of an atomic block contiguous in that order. In practice, we may require TSC only for programs that are *transactionally data-race–free*—that is, for programs for which no execution that happens to be TSC has a data race.

Aborted and incomplete transactions play no role in TSC-based semantics—they are confined entirely to the implementation. Their invisibility simplifies the programming model by avoiding the need to speak about speculation and rollback. In any valid history, transactions appear in their entirety or not at all. As discussed by Dalessandro et al. [6], TSC-based semantics can even capture condition synchronization (e.g., retry [14]): a history is valid only if each of its transactions occurs at a point in time when all of its preconditions hold. (Extensions are required for language-level abort and orElse [23].)

Unfortunately, the lack of aborted and incomplete transactions implies that TSC-based semantics are declarative rather than operational. Given a history, we can easily verify whether it is valid: are the transactions contiguous, does each read return the value of the most recent prior write to the same location, and does each individual step (inside or outside a transaction) adhere to the (sequential) semantics of the underlying language? Unfortunately, given an incomplete history, there are no small-step rules that will allow us to enumerate all the possible extensions.

The lack of an operational model for TSC makes it difficult to address issues of liveness. It also precludes discussion of the interface between the compiler and the run-time system, where aborted transactions are part of the API.

3 Sequential Semantics

At the run-time level, where speculation must be explicit, we can think of the TM system as a big concurrent object, with operations that include (among others) start, read, write, and try-commit. Linearizability—the traditional safety criterion for concurrent objects—considers program histories in which each of these operations is represented by separate call (request) and return (response) events. It then requires that each concurrent history satisfy two properties. First, the operations must appear to occur in some global total order that is consistent (a) with program order in each thread and (b) with any ordering that threads are able to observe by other means. (Requirement (b) implies that each individual operation must appear to occur, instantaneously, at some point between its call [request] and its return [response].) Second, linearizability requires that the total order on operations satisfy the *sequential semantics* of the object (i.e., the TM system) for which we are building a concurrent implementation.

But what are the sequential semantics of TM? At the first TRANSACT, in 2006, I proposed a framework (hereinafter referred to as *SSTMS*) in which to address this question [22]. The intent was to characterize the behavior of TM at the level of the run-time API, without regard to the underlying implementation.

SSTMS assumes that each thread subhistory is *well-formed*: it respects the sequential semantics of the source programming language and it comprises a series of *transactions*, each of which takes the form (start (read | write)* (try-commit | abort)). (This pattern is easily extended to allow additional reads and writes outside of any transaction.) The try-commit operation returns an indication of success or failure. The explicit abort operation admits cases in which the program or compiler has determined, from above, that the current transaction cannot—or should not—succeed.

Key features of the SSTMS framework include the following (keep in mind that these are all defined on *sequential* histories):

1. a memory model in which (a) each read must return the value provided by the most recent previous write to the same location in some already-completed successful transaction, and (b) each such value must still be valid at the time of the subsequent try-commit, if that try-commit is successful—that is, the value must still (or again) be the one that would be seen if the read occurred at commit time.[1]
2. the notion of *conflict functions*, which identify, in a given history, pairs of transactions that cannot both succeed.

[1] In the SSTMS paper, what I have here called the "memory model" was referred to somewhat inaccurately as "consistency." This particular memory model turns out to be sufficient but not necessary to ensure the more conventional meaning of consistency—namely, that all values seen by a transaction are simultaneously valid. Other memory models that would also suffice for consistency are posited later in this section.

3. optional *arbitration functions*, which specify which of two conflicting transactions must fail. These serve to capture the notion of *contention management* in TM implementations [10, 16, 21].

Semantic differences among TM systems in the literature can be captured, to a large extent, by their respective conflict functions. At one extreme, mutual exclusion corresponds to specifying that transactions conflict whenever they overlap (whenever each begins before the other ends). At the other extreme, *lazy invalidation* specifies that transactions S and T conflict if S reads a location that T subsequently writes, and T commits—successfully—before the linearization of S's try-commit. The SSTMS paper proved that lazy invalidation is the weakest conflict function (the one with the smallest number of conflicts) compatible with the (original) memory model.

The SSTMS paper also proved that a sequential TM history H that respects the memory model is always equivalent to some *serial* history, consisting of all and only the successful transactions of H, ordered in such a way that the operations of each individual transaction are contiguous, and each read sees the value provided by the most recent previous write to the same location. This property was dubbed the *fundamental theorem of TM*.

To avoid artificial constraints on the code within transactions, the original SSTMS memory model must be extended to allow a transaction to see its own writes. To accommodate the publication idiom [18], it must also be extended to allow a transaction to see writes in non-transactional code that precede any already-completed successful transaction. Both of these extensions have trivial impact on the proof of the fundamental theorem.

More interesting extensions—alternative models, really—would accommodate global timestamp-based STM systems, which allow certain read-only transactions to succeed even when some of their reads are no longer valid at commit time. In TL2 [8], for example, a read-only transaction T can succeed so long as all writes by other threads to locations read by T occur before T starts or after it has performed all its reads. In TinySTM [9], T can succeed so long as there exists a time t, between T's start and its try-commit, such that all of T's reads are still valid at t, and all writes by other threads to locations read by T occur before t or after T has performed all its reads.

More ambitious memory models could accommodate multi-version systems like JVSTM [4], LSA [20], and their successors, which allow a read-only transaction to serialize at any point between its start and its try-commit, by reading—if necessary—historic (overwritten) values. In principle, one could even imagine models under which a writer transaction could read historic values and still succeed, provided it introduced no circularity in serialization order.

Significantly, both global timestamp and multi-version-memory–based memory models would *change the sequential semantics* of TM: they would induce a different set of valid sequential histories. At the same time, they would still suffice to prove the fundamental theorem of TM. A natural question thus arises: what characteristics in a memory model and conflict function (and thus in the behavior of API calls) are *necessary* to prove the fundamental theorem?

4 Opacity

In a 2007 technical report [11], subsequently published at PPoPP'08, Guerraoui and Kapałka proposed a framework for run-time–level semantics that sidesteps the question of necessary characteristics of API calls by making serializability and consistency fundamental rather than derived. Specifically, in any *opaque* TM system [12, Sec. 5]:

1. all operations performed by every committed transaction appear as if they happened at some single, indivisible point during the transaction lifetime,
2. no operation performed by any aborted transaction is ever visible to other transactions (including live ones), and
3. every transaction always observes a consistent state of the system.

Note that these properties do not specify the behavior of individual API calls—in particular, the values returned by reads. Rather, they constrain the *overall* behavior of the TM system.

Over the past few years, opacity—property (3) in particular—has been widely accepted as the default safety criterion for TM systems. It differs from SSTMS in several significant ways. In particular, property (2) in SSTMS is a matter of sequential semantics; property (3) in SSTMS is true (and again part of the sequential semantics) only for successful (committed) transactions;[2] and property (1) is implied by properties (2) and (3), as extended to concurrent histories via linearizability. Putting it very informally, opacity property (1) says that a TM system is correct if, when considered in the SSTMS framework, the memory model and contention function of the run-time–level API suffice to prove the fundamental theorem of TM.

To avoid the creation of zombie transactions, opacity requires that a transaction abort before returning an inconsistent value in response to a read request. An abort

[2] As noted in footnote 1, the SSTMS paper actually stated the stronger requirement that every transaction observe the state of the system at the time of its try-commit. This requirement should be relaxed to admit implementations (e.g., those based on multiversioning) in which certain transactions may "commit in the past."

event may thus be generated either by a program ("from above") or by the run-time system ("from below").

By contrast, SSTMS maintains safety in the presence of zombie transactions by assuming that the compiler will invoke an explicit validate operation "whenever the use of inconsistent data might lead to unacceptable behavior" [22, Sec. 6]. The validate operation is defined to return false in the event of inconsistency, whereupon correctly compiled code will promptly call abort. Subsequent work [7] clarifies that the compiler must validate (1) before propagating an exception out of a transaction, (2) whenever using the result of a potentially inconsistent read might lead to externally visible erroneous behavior (e.g., an uninstrumented write to a dynamically determined location, or a branch to a dynamically determined address), and (3) within a bounded amount of time on every transactional code path (to avoid the possibility of an erroneous infinite loop). Precise specification of these conditions can be considered part of the contract for the run-time–level API, and may vary from one language, compiler, and TM system to another.

5 Tolerating Zombies

Both SSTMS and opacity can be used to prove a variety of liveness properties. The SSTMS paper did this only in the context of sequential histories, with each call to a TM operation considered as a single "step." In this context, it proved (among other things) that a TM system based on lazy invalidation is livelock free, but still admits starvation [22, Thms. 5 & 6]. In a similar vein, but in the richer context of concurrent histories, Bushkov, Guerraoui, and Kapałka showed that "no TM implementation can ensure *local progress* ... [i.e., that] every process that keeps executing a transaction (say keeps retrying it in case it aborts) eventually commits it" [3, Sec. 1.1].

Proofs of these properties are possible precisely because SSTMS and opacity expose run-time–level details—namely, the individual operations of the TM API—which TSC deliberately obscures. A key claim of this position paper is that opacity similarly—but unnecessarily—obscures an important run-time–level detail of many TM systems—namely, the indefinite continued execution of zombie transactions. If we want to understand the behavior—including liveness—of TM at the run-time level, we need to capture this zombie execution.

Perhaps the most obvious approach would allow any read to be inconsistent, and allow try-commit to succeed only in the absence of inconsistency. Unfortunately, this approach is insufficient to ensure even *weak progressiveness*—the requirement that "a transaction that encounters no conflict must be able to commit" [13]. To enable proofs of liveness, it seems reasonable to require, in a manner reminiscent of SSTMS conflict functions, that a read return an inconsistent value only if its transaction has an "excuse" for a subsequent try-commit to fail. Results in the SSTMS paper imply that lazy invalidation conflict (in a non–multi-version TM system) would be the weakest excuse sufficient to guarantee the consistency of successful transactions—thereby providing the strongest guarantees of liveness. Stronger conflict functions would lead to weaker guarantees of liveness. At the extreme of overlap conflict, liveness would still be ensured by (at least) an implementation that executes transactions one at a time.

The obvious objection to this "excused inconsistency" is that it allows a zombie to perform operations not envisioned by the program source. It is easy, in fact, to construct a scenario in which the first zombie in an execution performs (within its own transaction, at least), *any given undesirable operation*, even when that operation could never occur in an execution that never experiences inconsistency. To preserve the safety and liveness of the overall execution, we need semantic rules that capture the essence of sandboxing. We could use these to modify the definition of opacity, or we could embed them in a framework reminiscent of SSTMS. I prefer the latter option, because it allows us to reason locally about the behavior of individual operations of the TM run-time API. Serializability and consistency—both of which are global properties—then *emerge from* this local behavior.

Building on SSTMS, we might require that a correct implementation of TM embody a memory model, a conflict function, and (optionally) an arbitration function such that

1. we can prove the fundamental theorem of TM. Following the pattern of the SSTMS paper, it would suffice (a) for each read in a successful transaction to return the value provided by the most recent previous write to the same location in the current transaction, some already-completed successful transaction, or nontransactional code that precedes some already-completed successful transaction; (b) for a try-commit to be successful only if all reads of locations not previously written by the transaction would return the same values if repeated at commit time (ignoring local writes); and (c) for the conflict function to be at least as strong as lazy invalidation.
2. we can prove at least minimal liveness—specifically, a try-commit fails only in the presence of some other conflicting transaction.
3. a read r in an unsuccessful transaction T is inconsistent with previous reads of T only if there exists some other transaction S whose prefix prior to r conflicts with T (thereby giving T an excuse to abort).
4. zombie execution is bounded: given a partial execution history H with an uncompleted transaction T, if for any k there exists an extension of H in which

T performs at least k program steps, then there exists a (possibly different) extension of H in which T performs at least k program steps, none of which is an inconsistent read.

5. if the language semantics envision exceptions, these never escape an unsuccessful transaction.

6 Future Work

The proposal of the previous section requires formalization. It also invites exploration of the memory models and conflict functions sufficient to prove both safety (the fundamental theorem) and liveness. As noted in footnotes 1 and 2, the SSTMS convention (observe the most recently committed writes, enforce at least lazy invalidation conflict, and require reads to remain valid at commit time) is sufficient but not necessary.

For hardware transactions in particular, but also perhaps for software TM, it may be desirable to permit the occasional spurious (unexcused) failure. Conversely, it may desirable in real-world systems to place a fixed bound on the duration of zombie execution, though this need not necessarily be reflected in the semantics. Real implementations will all embody conflict functions, some of which will be stronger than lazy invalidation; for these, liveness proofs may require a crisp characterization in the formal semantics. For all implementations of interest, we will want proofs of both safety and liveness.

Perhaps the most interesting questions center around the values that may be seen by a zombie transaction, and the actions it may consequently take. At one extreme, we might require that every read—even in a zombie—return the most recent committed value (which may of course be inconsistent with the values seen by previous reads). At the other extreme, we might allow "out of thin air" reads. It is not yet clear what impact—if any—this choice will have on the compiler. It is at least conceivable that tighter requirements on inconsistent reads might reduce the number of validate calls required for correct sandboxing.

References

[1] M. Abadi and T. Harris. Perspectives on Transactional Memory. In *Proc. of the 20th Intl. Conf. on Concurrency Theory*, Bologna, Italy, Sept. 2009.

[2] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and Runtime Support for Efficient Software Transactional Memory. In *Proc. of the SIGPLAN 2006 Conf. on Programming Language Design and Implementation*, Ottawa, ON, Canada, June 2006.

[3] V. Bushkov, R. Guerraoui, and M. Kapałka. On the Liveness of Transactional Memory. In *Proc. of the 31st ACM Symp. on Principles of Distributed Computing*, Madeira, Portugal, July 2012.

[4] J. Cachopo and A. Rito-Silva. Versioned Boxes as the Basis for Memory Transactions. *Science of Computer Programming*, 63(2):172-185, Dec. 2006.

[5] J. Casper, T. Oguntebi, S. Hong, N. G. Bronson, C. Kozyrakis, and K. Olukotun. Hardware Acceleration of Transactional Memory on Commodity Systems. In *Proc. of the 16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, Mar. 2011.

[6] L. Dalessandro, M. L. Scott, and M. F. Spear. Transactions as the Foundation of a Memory Consistency Model. In *Proc. of the 24th Intl. Symp. on Distributed Computing*, Cambridge, MA, Sept. 2010. Earlier but expanded version available as TR 959, Dept. of Computer Science, Univ. of Rochester, July 2010.

[7] L. Dalessandro and M. L. Scott. Sandboxing Transactional Memory. In *Proc. of the 21st Intl. Conf. on Parallel Architectures and Compilation Techniques*, Minneapolis, MN, Sept. 2012.

[8] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proc. of the 20th Intl. Symp. on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[9] P. Felber, T. Riegel, and C. Fetzer. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *Proc. of the 13th ACM Symp. on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[10] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a Theory of Transactional Contention Managers. In *Proc. of the 24th ACM Symp. on Principles of Distributed Computing*, Las Vegas, NV, Aug. 2005.

[11] R. Guerraoui and M. Kapałka. Opacity: A Correctness Condition for Transactional Memory. Technical Report LPD-REPORT-2007-004, EPFL, May 2007. `lpd.epfl.ch/kapalka/files/opacity-techreport07.pdf`.

[12] R. Guerraoui and M. Kapałka. On the Correctness of Transactional Memory. In *Proc. of the 13th ACM Symp. on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[13] R. Guerraoui and M. Kapałka. The Semantics of Progress in Lock-Based Transactional Memory. In *Proc. of the 36th ACM Symp. on Principles of Programming Languages*, Savannah, GA, Jan. 2009.

[14] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable Memory Transactions. In *Proc. of the 10th ACM Symp. on Principles and Practice of Parallel Programming*, Chicago, IL, June 2005.

[15] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing Memory Transactions. In *Proc. of the SIGPLAN 2006 Conf. on Programming Language Design and Implementation*, Ottawa, ON, Canada, June 2006.

[16] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proc. of the 22nd ACM Symp. on Principles of Distributed Computing*, Boston, MA, July 2003.

[17] G. Kestor, R. Gioiosa, T. Harris, O. S. Unsal, A. Cristal, I. Hur, and M. Valero. STM^2: A Parallel STM for High Performance Simultaneous Multithreading Systems. In *Proc. of the 2011 Intl. Conf. on Parallel Architectures and Compilation Techniques*, Galveston Island, TX, Oct. 2011.

[18] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical Weak-Atomicity Semantics for Java STM. In *Proc. of the 20th ACM Symp. on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[19] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In *Proc. of the 16th Intl. Conf. on Parallel Architectures and Compilation Techniques*, Brasov, Romania, Sept. 2007.

[20] T. Riegel, P. Felber, and C. Fetzer. A Lazy Snapshot Algorithm with Eager Validation. In *Proc. of the 20th Intl. Symp. on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[21] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proc. of the 24th ACM Symp. on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[22] M. L. Scott. Sequential Specification of Transactional Memory Semantics. In *1st ACM SIGPLAN Wkshp. on Transactional Computing*, Ottawa, ON, Canada, June 2006.

[23] M. L. Scott and L. Dalessandro. On the Orthogonality of Speculation and Atomicity. In *2nd Wkshp. on the Theory of Transactional Memory*, Cambridge, MA, Sept. 2010.

[24] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. Ordering-Based Semantics for Software Transactional Memory. In *Proc. of the 12th Intl. Conf. on Principles of Distributed Systems*, Luxor, Egypt, Dec. 2008.

[25] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *Proc. of the 21st Intl. Conf. on Parallel Architectures and Compilation Techniques*, Minneapolis, MN, Sept. 2012.