Perspectives on Transactional Memory

Martín Abadi[1,2] and Tim Harris[1]

[1] Microsoft Research
[2] University of California, Santa Cruz

Abstract. We examine the role of transactional memory from two perspectives: that of a programming language with atomic actions and that of implementations of the language. We argue that it is difficult to formulate a clean, separate, and generally useful definition of transactional memory. In both programming-language semantics and implementations, the treatment of atomic actions benefits from being combined with that of other language features. In this respect (as in many others), transactional memory is analogous to garbage collection, which is often coupled with other parts of language runtime systems.

1 Introduction

The name "transactional memory" [21] suggests that a transactional memory (TM) is something similar to an ordinary memory, though perhaps with a slightly different interface and different properties. In particular, the interface would include means of initiating and committing transactions, as well as means of performing memory accesses. These memory accesses may be within transactions, and perhaps also outside transactions. The interface may provide other operations for aborting transactions, for delaying their execution, or for nesting them in various ways. As for the properties, we would expect certain guarantees that differentiate TM from ordinary memory. These properties should include, in particular, that all memory accesses within a successful transaction appear atomic.

Some interesting recent research aims to define TM more precisely, along these lines [9, 16–19, 29, 32]. TM may be modeled as a shared object that supports operations such as read, write, begin-transaction, and commit-transaction, with requirements on the behavior of these operations. Some of this research also examines particular implementations, and whether or not they satisfy those requirements.

Another recent line of research studies programming-language constructs that may be built over TM—typically `atomic` blocks [25] or other constructs for atomic actions [3, 6, 22]. A variety of semantics have been provided at different levels of abstraction. Some semantics model atomic actions that execute without the interleaving of operations of other threads. We call these "strong" semantics. Other semantics model low-level details of common implementations, such as conflict-detection mechanisms and roll-backs.

Despite encouraging progress, much work remains. In particular, the research to date does not fully explore the relation between the first style of definition (where TM is a shared object) and the second style of definition (modeling language constructs rather than TM *per se*).

In this paper, we argue that these gaps are not surprising, nor necessarily bad. Indeed, we find limiting the view that a TM is something similar to a memory, with a slightly different interface and properties. Although this view can sometimes be reasonable and useful, in many settings a clear delineation of TM as a separate, memory-like object is neither necessary nor desirable.

We consider TM from two perspectives: that of the programming language with atomic actions and that of the implementation of the atomicity guarantees.

- From the former perspective, we are primarily interested in the possibility of writing correct, efficient programs. The syntax and semantics of these programs may reflect transactional guarantees (for instance, by including `atomic` blocks), but they need not treat TM as a separate object. Indeed, the language may be designed to permit a range of implementations, rather than just those based on TM.
- From the latter perspective, we are interested in developing efficient implementations of programming languages. Implementations may fruitfully combine the TM with other aspects of a runtime system and with static program analysis, thus offering stronger guarantees at a lesser cost.

These two perspectives are closely related, and some of the same arguments appear from both perspectives.

Despite these reservations, we do recognize that, sometimes, a clear delineation of TM is possible and worthwhile. We explore how this approach applies in some simple language semantics, in Section 2. In Section 3, we consider the difficulties of extending this approach, both in the context of more sophisticated semantics and in actual implementations. We argue that it is best, and perhaps inevitable, to integrate the TM into the semantics and the implementations. In Section 4, we examine the question of the definition of TM through the lens of the analogy with garbage collection. We conclude in Section 5.

2 Transactional Memory in Semantics: A Simple Case

In our work, we have defined various semantics with atomicity properties [3, 5, 6]. Some of the semantics aim to capture a programmer's view of a language with high-level atomicity guarantees. Other semantics are low-level models that include aspects of implementations, for instance logs for undoing eager updates made by transactions and for detecting conflicts between concurrent transactions. Moore and Grossman [25] have defined some analogous semantics for different languages. Remarkably, although all these semantics specify the behavior of programs, none of them includes a separate definition of TM. Rather, the TM is closely tied to the rest of the semantics.

$$
\begin{array}{rl}
b \in \mathrm{BExp} & = \ldots \\
e \in \mathrm{NExp} & = \ldots \\
C, D \in \mathrm{Com} & = \texttt{skip} \\
& \mid\ x := e \qquad (x \in \mathrm{Vars}) \\
& \mid\ C; D \\
& \mid\ \texttt{if}\ b\ \texttt{then}\ C\ \texttt{else}\ D \\
& \mid\ \texttt{while}\ b\ \texttt{do}\ C \\
& \mid\ \texttt{async}\ C \\
& \mid\ \texttt{unprotected}\ C \\
& \mid\ \texttt{block}
\end{array}
$$

Fig. 1. Syntax.

In this section we illustrate, through a simple example, the style of those semantics. We also consider and discuss a variant in which TM is presented more abstractly and separately.

More specifically, we consider a simple imperative language and an implementation with transaction roll-back. This language omits many language features (such as memory allocation) and implementation techniques (such as concurrent execution of transactions). It is a fragment of the AME calculus [3], and a small extension (with `unprotected` sections) of a language from our previous work [6]. Both the high-level semantics of the language (Section 2.2) and a first version with roll-back (Section 2.3) treat memory as part of the execution state, with no separate definition of what it means to be a correct TM. On the other hand, a reformulation of the version with roll-back (Section 2.4) separates the semantics of language constructs from the specification of TM.

2.1 A Simple Language

The language that we consider is an extension of a basic imperative language, with a finite set of variables Vars, whose values are natural numbers, and with assignments, sequencing, conditionals, and while loops (IMP [36]). Additionally, the language includes constructs for co-operative multi-threading:

- A construct for executing a command in an asynchronous thread. Informally, `async` C forks off the execution of C. This execution is asynchronous, and will not happen if the present thread keeps running without ever yielding control, or if the present thread blocks without first yielding control. The execution of C will be atomic until C yields control, blocks, or terminates.
- A construct for running code while allowing preemption at any point. Informally, `unprotected` C yields control, then executes C without guaranteeing C's atomicity, and finally yields control again.
- A construct for blocking. Informally, `block` halts the execution of the entire program.

We define the syntax of the language in Figure 1. We do not detail the usual constructs on numerical expressions, nor those for boolean conditions.

$$
\begin{array}{lll}
\langle \sigma, T, \mathcal{E}[x := e] \rangle & \longrightarrow \langle \sigma[x \mapsto n], T, \mathcal{E}[\texttt{skip}] \rangle & \text{if } \sigma(e) = n \\
\langle \sigma, T, \mathcal{E}[\texttt{skip}; C] \rangle & \longrightarrow \langle \sigma, T, \mathcal{E}[C] \rangle & \\
\langle \sigma, T, \mathcal{E}[\texttt{if } b \texttt{ then } C \texttt{ else } D] \rangle & \longrightarrow \langle \sigma, T, \mathcal{E}[C] \rangle & \text{if } \sigma(b) = \texttt{true} \\
\langle \sigma, T, \mathcal{E}[\texttt{if } b \texttt{ then } C \texttt{ else } D] \rangle & \longrightarrow \langle \sigma, T, \mathcal{E}[D] \rangle & \text{if } \sigma(b) = \texttt{false} \\
\langle \sigma, T, \mathcal{E}[\texttt{while } b \texttt{ do } C] \rangle & \longrightarrow \langle \sigma, T, \mathcal{E}[\texttt{if } b \texttt{ then } \ldots \texttt{ else } \ldots] \rangle & \\
\langle \sigma, T, \mathcal{E}[\texttt{async } C] \rangle & \longrightarrow \langle \sigma, T.C, \mathcal{E}[\texttt{skip}] \rangle & \\
\langle \sigma, T, \mathcal{E}[\texttt{unprotected } C] \rangle & \longrightarrow \langle \sigma, T.\mathcal{E}[\texttt{unprotected } C], \texttt{skip} \rangle & \\
\langle \sigma, T, \mathcal{E}[\texttt{unprotected skip}] \rangle & \longrightarrow \langle \sigma, T.\mathcal{E}[\texttt{skip}], \texttt{skip} \rangle & \\
\langle \sigma, T.C, \texttt{skip} \rangle & \longrightarrow \langle \sigma, T, C \rangle &
\end{array}
$$

Fig. 2. Transition rules of the abstract machine.

2.2 High-Level Strong Semantics

A first semantics for our language is given in terms of small-step transitions between states. A state $\langle \sigma, T, C \rangle$ consists of the following components:

- a store σ, which is a mapping of the finite set Vars of variables to the set of natural numbers;
- a finite multiset of commands T, which we call the thread pool;
- a distinguished active command C.

We write $\sigma[x \mapsto n]$ for the store that agrees with σ except at x, which is mapped to n. We write $\sigma(b)$ for the boolean denoted by b in σ, and $\sigma(e)$ for the natural number denoted by e in σ. We write $T.C$ for the result of adding C to T. As usual, a context is an expression with a hole [ ], and an evaluation context is a context of a particular kind. Given a context $\mathcal{C}$ and a command C, we write $\mathcal{C}[C]$ for the result of placing C in the hole in $\mathcal{C}$. We use the evaluation contexts defined by the grammar:

$$\mathcal{E} = [\,] \mid \mathcal{E}; C \mid \texttt{unprotected } \mathcal{E}$$

Figure 2 gives rules that specify the transition relation (eliding straightforward details for `while` loops). According to these rules, when the active command is `skip`, a command from the pool becomes the active command. It is then evaluated as such until it produces `skip`, yields, or blocks. No other computation is interleaved with this evaluation. When the active command is not `skip`, each evaluation step produces a new state, determined by decomposing the active command into an evaluation context and a subexpression. Yielding happens when this subexpression is a command of the form `unprotected` C.

This semantics is a strong semantics in the sense that any `unprotected` sections in the thread pool will not run while an active command is running and does not yield.

2.3 A Lower-Level Semantics with Roll-Back

A slightly lower-level semantics allows roll-back at any point in a computation. (Roll-back may make the most sense when the active command is blocked, but

$$
\begin{array}{lll}
\langle S, \sigma, T, \mathcal{E}[x := e] \rangle & \longrightarrow \langle S, \sigma[x \mapsto n], T, \mathcal{E}[\texttt{skip}] \rangle & \text{if } \sigma(e) = n \\
\langle S, \sigma, T, \mathcal{E}[\texttt{skip}; C] \rangle & \longrightarrow \langle S, \sigma, T, \mathcal{E}[C] \rangle & \\
\langle S, \sigma, T, \mathcal{E}[\texttt{if } b \texttt{ then } C \texttt{ else } D] \rangle & \longrightarrow \langle S, \sigma, T, \mathcal{E}[C] \rangle & \text{if } \sigma(b) = \texttt{true} \\
\langle S, \sigma, T, \mathcal{E}[\texttt{if } b \texttt{ then } C \texttt{ else } D] \rangle & \longrightarrow \langle S, \sigma, T, \mathcal{E}[D] \rangle & \text{if } \sigma(b) = \texttt{false} \\
\langle S, \sigma, T, \mathcal{E}[\texttt{while } b \texttt{ do } C] \rangle & \longrightarrow \langle S, \sigma, T, \mathcal{E}[\texttt{if } b \texttt{ then } \ldots \texttt{ else } \ldots] \rangle & \\
\langle S, \sigma, T, \mathcal{E}[\texttt{async } C] \rangle & \longrightarrow \langle S, \sigma, T.C, \mathcal{E}[\texttt{skip}] \rangle & \\
\langle S, \sigma, T, \mathcal{E}[\texttt{unprotected } C] \rangle & \longrightarrow \langle S, \sigma, T.\mathcal{E}[\texttt{unprotected } C], \texttt{skip} \rangle & \\
\langle S, \sigma, T, \mathcal{E}[\texttt{unprotected skip}] \rangle & \longrightarrow \langle S, \sigma, T.\mathcal{E}[\texttt{skip}], \texttt{skip} \rangle & \\
\langle S, \sigma, T.C, \texttt{skip} \rangle & \longrightarrow \langle \langle \sigma, T.C \rangle, \sigma, T, C \rangle & \\
\langle \langle \sigma_0, T_0 \rangle, \sigma, T, C \rangle & \longrightarrow \langle \langle \sigma_0, T_0 \rangle, \sigma_0, T_0, \texttt{skip} \rangle &
\end{array}
$$

Fig. 3. Transition rules of the abstract machine, with roll-back.

it is convenient to allow roll-back at any point.) For this purpose, the semantics relies on extended states $\langle \langle \sigma_0, T_0 \rangle, \sigma, T, C \rangle$ with two additional components: an extra store σ_0 and an extra thread pool T_0. Basically, the current σ and T are saved as σ_0 and T_0 when a transaction starts, and restored upon roll-back. Figure 3 gives the rules of the semantics. Only the last two rules operate on the additional state components.

Our work and that of Moore and Grossman include more elaborate semantics with roll-back [3, 5, 25]. Those semantics model finer-grain logging; in them, roll-back is not a single atomic step. Some of the semantics are weak, in the sense that `unprotected` sections may execute while transactions are in progress, and even during the roll-back of transactions. We return to this complication in Section 3.2, where we also consider concurrency between transactions.

2.4 Separating the Transactional Memory

Figure 4 presents a reformulation of the semantics of Section 2.3. States are simplified so that they consist only of a thread pool and an active command. Memory is treated through labels on the transition relation. These labels indicate any memory operations, and also the start and roll-back of atomic computations. The labels for start and roll-back include a thread pool (which could probably be omitted if thread pools were tracked differently). The commit-point of atomic computations can remain implicit.

A separate definition can dictate which sequences μ of labels are legal in a computation $\langle T, \texttt{skip} \rangle \longrightarrow_\mu^* \langle T', \texttt{skip} \rangle$. This definition may be done axiomatically. One of the axioms may say, for instance, that for each label **back**T in μ there is a corresponding, preceding label **start**T, with the same T, and with no intervening other **start** or **back** label. Another axiom may constrain reads and writes, and imply, for example, that the sequence $[x \mapsto 1][x = 2]$ is not legal. Although such axiomatic definitions can be elegant, they are both subtle and error-prone. Alternatively, the definition may have an operational style. For this purpose we define a transition relation in Figure 5, as a relation on triples of the

$$
\begin{array}{lcl}
\langle T, \mathcal{E}[x := e]\rangle & \longrightarrow_{[e=n][x\mapsto n]} & \langle T, \mathcal{E}[\texttt{skip}]\rangle \\
\langle T, \mathcal{E}[\texttt{skip}; C]\rangle & \longrightarrow & \langle T, \mathcal{E}[C]\rangle \\
\langle T, \mathcal{E}[\texttt{if } b \texttt{ then } C \texttt{ else } D]\rangle & \longrightarrow_{[b=\texttt{true}]} & \langle T, \mathcal{E}[C]\rangle \\
\langle T, \mathcal{E}[\texttt{if } b \texttt{ then } C \texttt{ else } D]\rangle & \longrightarrow_{[b=\texttt{false}]} & \langle T, \mathcal{E}[D]\rangle \\
\langle T, \mathcal{E}[\texttt{while } b \texttt{ do } C]\rangle & \longrightarrow & \langle T, \mathcal{E}[\texttt{if } b \texttt{ then } \ldots \texttt{ else } \ldots]\rangle \\
\langle T, \mathcal{E}[\texttt{async } C]\rangle & \longrightarrow & \langle T.C, \mathcal{E}[\texttt{skip}]\rangle \\
\langle T, \mathcal{E}[\texttt{unprotected } C]\rangle & \longrightarrow & \langle T.\mathcal{E}[\texttt{unprotected } C], \texttt{skip}\rangle \\
\langle T, \mathcal{E}[\texttt{unprotected skip}]\rangle & \longrightarrow & \langle T.\mathcal{E}[\texttt{skip}], \texttt{skip}\rangle \\
\langle T.C, \texttt{skip}\rangle & \longrightarrow_{\mathbf{start}_{T.C}} & \langle T, C\rangle \\
\langle T, C\rangle & \longrightarrow_{\mathbf{back}_{T'}} & \langle T', \texttt{skip}\rangle
\end{array}
$$

Fig. 4. Transition rules of the abstract machine, with roll-back, reformulated.

$$
\begin{array}{lcll}
\langle \sigma_0, T, \sigma\rangle & \longrightarrow_{[e=n]} & \langle \sigma_0, T, \sigma\rangle & \text{if } \sigma(e) = n \\
\langle \sigma_0, T, \sigma\rangle & \longrightarrow_{[b=v]} & \langle \sigma_0, T, \sigma\rangle & \text{if } \sigma(b) = v \\
\langle \sigma_0, T, \sigma\rangle & \longrightarrow_{[x\mapsto n]} & \langle \sigma_0, T, \sigma[x \mapsto n]\rangle & \\
\langle \sigma_0, T, \sigma\rangle & \longrightarrow_{\mathbf{start}_{T'}} & \langle \sigma, T', \sigma\rangle & \\
\langle \sigma_0, T, \sigma\rangle & \longrightarrow_{\mathbf{back}_{T}} & \langle \sigma_0, T, \sigma_0\rangle &
\end{array}
$$

Fig. 5. Operational definition of legal sequences of memory operations.

form $\langle \sigma_0, T, \sigma\rangle$. Given an initial triple S, we say that the sequence of memory operations μ is legal if there is another triple S' such that $S \longrightarrow_{\mu}^{*} S'$.

Figures 4 and 5 amount to a decomposition of Figure 3, separating the definition of TM from the language semantics. Having a clear delineation of TM can be helpful for factoring semantics. Further, one may study how to implement memory systems that satisfy the definition in Figure 5—for instance, with various forms of logging.

3 Difficulties in Separating Transactional Memory

In this section we discuss the difficulties of having a separate TM in the context of more sophisticated semantics and in actual implementations. We have explored several such semantics and implementations, particularly focusing on avoiding conflicts between transactional and non-transactional memory accesses [2–5]. Recent research by others [8, 28] develops implementations with similar goals and themes, though with different techniques. Our observations in this section are drawn primarily from our experience on the implementation of `atomic` blocks and the AME constructs.

In Section 3.1, we consider systems with memory allocation, for which the difficulties appear somewhat interesting but mild. In Section 3.2, we consider concurrency, and the important but delicate distinction between strong semantics (of the kind presented in Section 2) and the property of strong atomicity [7] that may be ensured by a TM. In Section 3.3, we identify areas where aspects

of the implementation of atomic actions can either be provided by a TM with strong guarantees or be layered over a TM with weaker guarantees.

We conclude that, in such settings, it is beneficial and perhaps inevitable to integrate TM with other parts of semantics and implementations (for instance, with static analysis, garbage collection, scheduling, and virtual-memory management).

3.1 Memory Allocation

In Section 2, as in some works in the literature, the operations on memory do not include allocation. However, allocation must be taken into account in the context of TM.

In particular, some semantic definitions say that roll-backs do not undo allocations [3, 20, 25]. This choice simplifies some of the theory, and it is also important in practice: it helps ensure that—no matter what else happens—a dangling pointer will not be dereferenced after a roll-back. Thus, this choice represents a sort of defense in depth.

Adding allocation to the TM interface does not seem particularly challenging. However, we may wonder whether allocation is the tip of an iceberg. Class loading, initialization, finalization, exceptions, and perhaps other operations may also have interesting interactions with transactions. A definition of TM that considers them all may well become unwieldy.

Implementations vary a great deal in how allocation is treated inside transactions. Some consider the memory-management work to be part of transactions—for example, the memory manager may be implemented using transactional reads and writes to its free-lists. In other cases, the memory manager is integrated with the transactional machinery—for example, maintaining its own logs of tentative allocations and de-allocations that can be made permanent when a transaction commits and undone when a transaction aborts.

3.2 Concurrency, and Strong Atomicity vs. Strong Semantics

Much of the appeal of TM would not exist if it were not for the possibility that transactions execute in parallel with one another and also possibly with non-transactional code. We can extend the semantics of Section 2 to account for such concurrency. The semantics do get heavier and harder, whether TM is built into the semantics or is treated as a separate module. In the latter case, the operations on transactions may include transaction identifiers, and the reads and writes may be tied to particular transactions via those identifiers.

Such semantics may reflect the choice of particular implementation techniques (for instance, the use of eager updates or lazy updates). Nevertheless, the definitions should guarantee that a class of "correctly synchronized" programs run (or appear to run) with strong semantics. Researchers have explored various definitions of correct synchronization, for instance with static separation between

Thread 1	Thread 2

`atomic {`
`  ready = true;`
`  data = 1;`
`}`
	`tmp1 = ready;`
	`if (tmp1 == true) {`
	`  tmp2 = data;`
	`}`

Fig. 6. Initially `ready` is `false`. Under strong semantics, if `tmp1` is `true` then `tmp2` must be `1`.

transactional and non-transactional data [3, 20, 25], dynamic separation with operations for moving data between those two modes [1, 2, 5], and dynamic notions of data races between transactional and non-transactional code [3, 10, 29].

If TM is a separate module, we should also make explicit its own guarantees in the presence of concurrency. With this style of definition, if we wish to obtain strong semantics for all programs in a language, the TM should probably ensure strong atomicity, which basically means that transactions appear atomic not only with respect to one another but also with respect to non-transactional memory accesses. Unfortunately, the exact definition of strong atomicity is open to debate, and the debate might only be settled in the context of a particular language semantics. Stronger notions of strong atomicity might be needed to provide strong semantics when the language implementation is aggressive, and weaker notions of strong atomicity might suffice in other cases. The relation between strong semantics (for a programming language) and strong atomicity (for a TM) is particularly subtle.

Strong Semantics Without Strong Atomicity. Some languages have strong semantics but do not rely upon a TM with strong atomicity. For instance, in STM-Haskell, a type system guarantees that the same locations are not concurrently accessed transactionally and non-transactionally [20]. In other systems, scheduling prevents transactional and non-transactional operations being attempted concurrently [26]. These cases suggest that, at the very least, the TM should be closely tied to program analysis and scheduling.

Strong Atomicity Without Strong Semantics. Strong atomicity does not suffice for strong semantics, in particular because of program transformations. Figure 6 provides an example due to Grossman *et al.* [15]. After this code runs, under strong semantics, if `tmp1` is `true` then `tmp2` must be 1. However, a conventional optimizing compiler might perform Thread 2's read from `data` before the thread's read from `ready`. (For instance, an earlier read from `data` may still be available in a register.)

Arguments in favor of strong atomicity (in particular from a hardware perspective) often seem to overlook such examples. Unless these examples are regarded as racy, program transformations currently in use should be considered incorrect. As these examples illustrate, strong atomicity does not remove the need for a notion of "correct synchronization" in programs. For racy programs

that do not satisfy the correctness criterion, very few guarantees can be given in the presence of transformations by optimizing compilers and weak processor memory models.

3.3 Implementation Options at Multiple Layers

In practical implementations of language constructs, we encounter implementation options at multiple layers, and it would seem premature to fix a specific TM interface that would mandate one option or another. We consider several examples.

Implementing Strong Atomicity. In one of our implementations [4], we rely on off-the-shelf memory protection hardware for detecting possible conflicts between transactional accesses and normal (non-transactional) accesses. We organize the virtual address space of a process so that its heap is mapped twice. One mapping is used in transactions, while the other mapping is used in normal execution. This organization lets us selectively prevent normal access to pages while they remain accessible transactionally. We use this mechanism to detect possible conflicts between transactional accesses and normal accesses at the granularity of pages; we use an existing TM to detect conflicts between transactions at the granularity of objects.

This design provides a foundation that is sound but slow on conventional hardware. We introduce a number of optimizations. We allow the language runtime system to operate without triggering access violations. We use static analysis to identify non-transactional operations that are guaranteed not to conflict with transactions (these operations can bypass the page-level checks) and to identify transactions that are guaranteed not to conflict with normal accesses (these transactions need not revoke normal-access permissions on the pages involved).

Both the design and the optimizations raise a number of questions on the notion of TM. If a TM is a separate entity, should it know about virtual addresses, physical addresses, or both? How should it relate to memory protection? How can its guarantees be adjusted in the presence of program analysis?

Tolerating Inconsistent Views of Memory. With some TM implementations, a transaction can continue running as a "zombie" [12] after experiencing a conflict. Zombie transactions can have errant behavior that is not permitted by strong semantics. Consider the example in Figure 7. This program is correctly synchronized under all of the criteria that we have studied, so a correct implementation must not loop endlessly. However, if the TM does not guarantee that Thread 1 will see a consistent view of memory, it is possible for `temp1==0` and `temp2==1`, and consequently for Thread 1 to loop.

This flaw can be corrected by modifying the TM to give stronger guarantees (for instance, opacity [19]). Alternatively, the language runtime system can sandbox the effects of zombie transactions by adding periodic validation work to loops, and containing any side-effects from zombie transactions (for example,

Thread 1

```
atomic {
  temp1 = x1;
  temp2 = x2;
  if (temp1 != temp2) {
    while (1) { }
} }
```

Thread 2

```
atomic {
  x1 ++;
  x2 ++;
}
```

Fig. 7. Initially `x1==x2==0`. Under strong semantics, Thread 1 must not loop.

Thread 1

```
atomic {
  x = 100;
  x_initialized = true;
}
```

Thread 2

```
while (true) {
  atomic {
    if (x_initialized) break;
} }
Console.Out.Writeline(x);
```

Fig. 8. A publication idiom. Under strong semantics, if Thread 2 sees `x_initialized` true, then it must print `100`.

not raising a `NullReferenceException` in response to an access violation from a zombie transaction).

Granular Safety. Some TM implementations present data granularity problems. For instance, rolling back a write to a single byte might also roll back the contents of other bytes in the same memory word.

There are several viable techniques for avoiding this problem. First, the TM implementation may be strengthened to maintain precise access granularity (say, at the cost of extra book-keeping by tracking reads and writes at a byte level). Second, if correct synchronization is defined by static separation, then transactional and non-transactional data can be allocated on separate machine words. Third, various dynamic mechanisms can be used for isolating transactional and non-transactional data.

Only the first of these options places granular-safety requirements on the TM implementation. The other two options require that other parts of the language implementation be aware of the particular granularity at which the TM operates.

Ordering Across Threads. In privatization and publication idioms [3, 12, 13, 31, 34], a piece of data is accessed transactionally depending on the value of another piece of data, as for example in Figure 8. These idioms are frequently considered to be correctly synchronized. However, naïve implementations over TM may not execute them correctly. For the example in Figure 8, an implementation that uses lazy updates may allow Thread 2's non-transactional read of `x` to occur before Thread 1 has finished writing back its transactional update.

Such idioms require that if non-transactional code executes after an atomic action, then the non-transactional code must see all the side effects of preceding atomic actions. This guarantee is provided by TMs with strong atomicity. Alternatively, it can be layered over a weaker TM by adding synchronization barriers when transactions start and commit [24], or it can be provided by page-based separation of transactional and non-transactional data [4]. The performance tradeoffs between these approaches are complicated, and there is no clear approach to favor in defining a clean common TM interface.

4 The Garbage-Collection Analogy

Grossman has drawn a compelling analogy between TM and garbage collection, comparing the programming problems that they aim to solve, and the features and limitations of the techniques [14]. In this section we consider what this analogy says about the problem of defining TM.

Garbage collection can be regarded as a tool for implementing "garbage-collected memory". A "garbage-collected memory" is simply an "infinite memory" over which one does not need to worry about freeing space. It is both common and helpful to describe language semantics over such a memory.

On the other hand, there is no canonical definition of garbage collection. Although language implementations may have internal interfaces that are defined with various degrees of precision and generality, there seems to be no separate, clean, portable garbage-collection interface.

Typically, garbage collection—much like TM—is coupled with a compiler and other parts of a language implementation. Some collectors exploit virtual-memory page-protection hardware (e.g., [23, 35]) and static analysis (e.g., [11, 33]). Much like TM, also, garbage collection can interact with program transformations. For instance, the trick of exchanging the contents of two reference-typed fields by using three XOR operations is correct only if the garbage collector will not see the intermediate non-reference values. Furthermore, program transformations can affect when finalizers (which run when an object becomes unreachable) will be eligible to run.

On this basis, garbage-collection machinery and TM machinery are indeed analogous. In the world of TM, however, there is no easy counterpart to garbage-collected memory, that is, to infinite memory. In our opinion the best candidate is not TM, but rather the concept of atomic action. Both infinite memory and atomic actions can be used in specifying language semantics, and both have a wide range of concrete implementations.

5 Conclusion

In this paper we examine transactional memory from the perspectives of language semantics and implementations. We believe that, from both perspectives, it is often impractical to define a separate, clean, portable internal TM interface.

Nevertheless, it may be productive to study the definition of TM interfaces, and to examine whether or not particular TM implementation techniques are compatible with them.

Furthermore, in some cases, programmers may use a TM interface directly, rather than via `atomic` blocks (for instance, for manipulating data structures). However, that TM interface need not coincide with one used internally by an implementation of `atomic` blocks, nor with a hardware TM interface, since hardware might provide lower-level primitives [27, 30]. Finally, a compiler framework may usefully include a common interface for multiple TM implementations—but this interface is likely to be specific to a particular framework, and much broader than that of TM as a shared object.

In summary, the question of what is transactional memory seems to remain open, and may well deserve further investigation. However, in our opinion, it is at least as worthwhile to study languages and implementation techniques based on transactional ideas, even without a separate definition of transactional memory.

Acknowledgements We are grateful to Dan Grossman, Rachid Guerraoui, Leslie Lamport, and Greg Morrisett for discussions on the subject of this paper.

References

1. Martín Abadi, Andrew Birrell, Tim Harris, Johnson Hsieh, and Michael Isard. Dynamic separation for transactional memory. Technical Report MSR-TR-2008-43, Microsoft Research, March 2008.
2. Martín Abadi, Andrew Birrell, Tim Harris, Johnson Hsieh, and Michael Isard. Implementation and use of transactional memory with dynamic separation. In *CC '09: Proc. 18th International Conference on Compiler Construction*, volume 5501 of *Lecture Notes in Computer Science*, pages 63–77, March 2009.
3. Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL '08: Proc. 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 63–74, January 2008.
4. Martín Abadi, Tim Harris, and Mojtaba Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *PPoPP '09: Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 185–196, February 2009.
5. Martín Abadi, Tim Harris, and Katherine F. Moore. A model of dynamic separation for transactional memory. In *CONCUR '08: Proc. 19th International Conference on Concurrency Theory*, pages 6–20, August 2008.
6. Martín Abadi and Gordon D. Plotkin. A model of cooperative threads. In *POPL '09: Proc. 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 29–40, January 2009.
7. Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *WDDD '05: Proc. 4th Workshop on Duplicating, Deconstructing and Debunking*, pages 48–55, June 2005.
8. Nathan G. Bronson, Christos Kozyrakis, and Kunle Olukotun. Feedback-directed barrier optimization in a strongly isolated STM. In *POPL '09: Proc. 36th ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 213–225, January 2009.

9. Ariel Cohen, John W. O'Leary, Amir Pnueli, Mark R. Tuttle, and Lenore D. Zuck. Verifying correctness of transactional memories. In *FMCAD '07: Proc. 7th International Conference on Formal Methods in Computer-Aided Design*, pages 37–44, November 2007.
10. Luke Dalessandro and Michael L. Scott. Strong isolation is a weak idea. In *TRANSACT '09: 4th ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, February 2009.
11. David Detlefs and V. Kirshna Nandivada. Compile-time concurrent marking write barrier removal. Technical Report SMLI-TR-2004-142, Sun Microsystems, December 2004.
12. Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC '06: Proc. 20th International Symposium on Distributed Computing*, pages 194–208, September 2006.
13. Dave Dice and Nir Shavit. What really makes transactions faster? In *TRANSACT '06, 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.
14. Dan Grossman. The transactional memory / garbage collection analogy. In *OOPSLA '07: Proc. 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, pages 695–706, October 2007.
15. Dan Grossman, Jeremy Manson, and William Pugh. What do high-level memory models mean for transactions? In *MSPC '06: Proc. 2006 Workshop on Memory System Performance and Correctness*, pages 62–69, October 2006.
16. Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. Completeness and nondeterminism in model checking transactional memories. In *CONCUR '08: Proc. 19th International Conference on Concurrency Theory*, pages 21–35, August 2008.
17. Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. Permissiveness in transactional memories. In *DISC '08: Proc. 22nd International Symposium on Distributed Computing*, pages 305–319, September 2008.
18. Rachid Guerraoui, Tom Henzinger, and Vasu Singh. Model checking transactional memories. In *PLDI '08: Proc. 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 372–382, June 2008.
19. Rachid Guerraoui and Michał Kapałka. On the correctness of transactional memory. In *PPoPP '08: Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 175–184, February 2008.
20. Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proc. 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60, June 2005.
21. Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA '93: Proc. 20th International Symposium on Computer Architecture*, pages 289–301, May 1993.
22. Suresh Jagannathan, Jan Vitek, Adam Welc, and Antony Hosking. A transactional object calculus. *Sci. Comput. Program.*, 57(2):164–186, 2005.
23. Haim Kermany and Erez Petrank. The compressor: concurrent, incremental, and parallel compaction. In *PLDI '06: Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 354–363, June 2006.
24. Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard Hudson, Bratin Saha, and Adam Welc. Practical weak-atomicity semantics for Java STM. In *SPAA '08: Proc. 20th Symposium on Parallelism in Algorithms and Architectures*, pages 314–325, June 2008.

25. Katherine F. Moore and Dan Grossman. High-level small-step operational semantics for transactions. In *POPL '08: Proc. 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 51–62, January 2008.
26. Cosmin E. Oancea, Alan Mycroft, and Tim Harris. A lightweight in-place implementation for software thread-level speculation. In *SPAA '09: Proc. 21st ACM Symposium on Parallelism in Algorithms and Architectures*, August 2009.
27. Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural support for software transactional memory. In *MICRO '06: Proc. 39th IEEE/ACM International Symposium on Microarchitecture*, pages 185–196, June 2006.
28. Florian T. Schneider, Vijay Menon, Tatiana Shpeisman, and Ali-Reza Adl-Tabatabai. Dynamic optimization for efficient strong atomicity. In *OOPSLA '08: Proc. 23rd ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 181–194, October 2008.
29. Michael L. Scott. Sequential specification of transactional memory semantics. In *TRANSACT '06: 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.
30. Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. Flexible decoupled transactional memory support. In *ISCA '08: Proc. 35th International Symposium on Computer Architecture*, pages 139–150, June 2008.
31. Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. Technical Report 915, CS Dept, U. Rochester, February 2007.
32. Serdar Tasiran. A compositional method for verifying software transactional memory implementations. Technical Report MSR-TR-2008-56, Microsoft Research, April 2008.
33. Martin T. Vechev and David F. Bacon. Write barrier elision for concurrent garbage collectors. In *ISMM '04: Proc. 4th International Symposium on Memory Management*, pages 13–24, October 2004.
34. Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *CGO '07, International Symposium on Code Generation and Optimization*, pages 34–48, March 2007.
35. Michal Wegiel and Chandra Krintz. The mapping collector: virtual memory support for generational, parallel, and concurrent compaction. In *ASPLOS '08: Proc. 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 91–102, March 2008.
36. Glynn Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, 1993.