The Transactional Memory / Garbage Collection Analogy

Dan Grossman
University of Washington
djg@cs.washington.edu

Abstract

This essay presents remarkable similarities between transactional memory and garbage collection. The connections are fascinating in their own right, and they let us better understand one technology by thinking about the corresponding issues for the other.

Categories and Subject Descriptors D.1.3 [*Programming techniques*]: Concurrent Programming—Parallel programming; D.3.3 [*Programming languages*]: Language Constructs and Features—Concurrent programming structures; D.3.4 [*Programming languages*]: Processors—Memory management (garbage collection)

General Terms Languages

Keywords Transactional Memory, Garbage Collection

1. Introduction

Transactional memory is currently one of the hottest topics in computer-science research, having attracted the focus of researchers in programming languages, computer architecture, and parallel programming, as well as the attention of development groups at major software and hardware companies. The fundamental source of the excitement is the belief that by replacing locks and condition variables with transactions we can make it easier to write correct and efficient shared-memory parallel programs.

Having made the semantics and implementation of transactional memory a large piece of my research agenda [44, 46, 32, 19], I believe it is crucial to ask why we believe transactional memory is such a step forward. If the reasons are shallow or marginal, then transactional memory should probably just be a current fad, as some critics think it is. If we cannot identify crisp and precise reasons why transactions are an improvement over locks, then we are being neither good scientists nor good engineers.

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.

The purpose of this article is not to rehash excellent but previously published examples where software transactions provide an enormous benefit (though for background they are briefly discussed), nor is it to add some more examples to the litany. Rather, it is to present a more general perspective that I have developed over the last two years. This perspective is summarized in a one-sentence analogy:

> *Transactional memory (TM) is to*
> *shared-memory concurrency*
> *as*
> *garbage collection (GC) is to*
> *memory management.*

Having shared this sentence with many people, I have come to realize that this sort of pithy analogy has advantages and disadvantages. On the one hand, it sparks discussion and is easy to remember. When fully understood, such an analogy can inspire new research ideas and let one adapt terminology from one side of the analogy for use in the other. On the other hand, it is easy to misinterpret an analogy, apply it too broadly, or dismiss it as just a slogan. The key is to understand that an analogy like this one is not a complete argument; it is an introductory remark for a more complete discussion pointing out the remark's deeper meaning and limitations. This article is designed to provide a cogent starting point for that discussion. The primary goal is to use our understanding of garbage collection to better understand transactional memory (and possibly vice-versa).

The presentation of the TM/GC analogy that follows will demonstrate that the analogy is much deeper than, "here are two technologies that make programming easier." However, it will not conclude that TM will make concurrent programming as easy as sequential programming with GC. Rather, it will lead us to the balanced and obvious-once-you-say-it conclusion that transactions make it easy to define critical sections (which is a huge help in writing and maintaining shared-memory programs) but provide no help in identifying where a critical section should begin or end (which remains an enormous challenge).

I begin by providing a cursory review of memory management, garbage collection, concurrency, and transactional memory (Section 2). This non-analogical discussion simply introduces relevant definitions for the two sides and may

leave you wondering how they could possibly have much to do with each other. I then present the core of the analogy, uncovering many uncanny similarities even at a detailed level (Section 3). This discussion can then be balanced with the primary place the analogy does not hold, which is exactly the essence of what makes concurrent programming inherently more difficult no matter what synchronization mechanisms are provided (Section 4).

Having completed the crux of the argument, I then provide some additional context. First is a brief detour for an analogous type-theoretic treatment of manual memory management and locking, a prior focus of my research that provides some backstory for how the TM/GC analogy came to be (Section 5). Second are some conjectures one can make by pushing the analogy (perhaps) too far (Section 6). Finally, the conclusion describes the intended effects of publishing this article (Section 7).

2. Background

A full introduction to garbage collection and transactional memory is clearly beyond our scope (excellent overviews exist for GC [48, 33] and TM [35]), so this section will just introduce enough definitions to understand most of the claims that follow and provide some motivation for TM. Some readers may be able to skip much of this section.

For the sake of specificity, I will assume programs are written in a modern object-oriented language (e.g., Java) and interthread communication is via mutable shared-memory. Much of the discussion applies to other paradigms (e.g., mostly-functional programming) but less to communication via message-passing. (I will not wade into the merits of shared memory versus message passing. Assuming that shared memory is one model we will continue to use for the foreseeable future, it is worth improving.)

2.1 Garbage Collection

When a program creates an object, space for it is allocated in the heap. In a language with manual memory management, this space is explicitly reclaimed (freed) by the programmer. Accessing an object after reclaiming its space is a dangling-pointer dereference and behavior is typically undefined. Not reclaiming space in a timely manner is a space leak and hurts performance.

Garbage collection automates memory reclamation. The key idea is determining reachability: an object is reclaimed only after there is no sequence of references (i.e., a path) from a root (a global variable or (live) local variable) to the object. Reachability can be determined via tracing (an algorithm that starts with the roots and finds all reachable objects) or automatic reference-counting (an algorithm that maintains the number of references to each object, reclaiming an object when its count reaches zero). These algorithms are duals [5]. In practice, efficient garbage collectors use various high-level techniques (e.g., generational collection) and low-level tricks (e.g., pointer-reversal).

GC eliminates dangling-pointer dereferences. Space leaks are possible exactly when reachability is an imprecise approximation of object lifetime. To avoid imprecision, programmers need to avoid having dead objects (i.e., objects they are done with) remain reachable from roots. One helpful language feature is weak pointers — references that do not "count for reachability". If the garbage collector reclaims the target of a weak pointer, it updates the pointer to indicate it cannot be used (e.g., by setting it to `null`).

Conservative collection treats integers as possible pointers, so an object allocated at address a is not reclaimed (nor therefore, is anything reachable from it) if a live piece of memory holds the integer a. Accurate (i.e., nonconservative) collection uses tighter coupling between the compiler, which generates code for creating and accessing objects, and the run-time system, which includes the garbage-collector proper. The compiler can provide other static information to the collector, such as whether local variables are live at certain program points.

Real-time collection ensures the garbage collector never pauses the program for longer than a fixed threshold. The key complication is ensuring the collector can "keep up" (i.e., reclaim garbage as fast as it can be created) while meeting its deadlines, else space can become exhausted even though there is memory available for reclamation.

2.2 Transactional Memory

The assumed concurrency model allows programmers to create additional threads to execute code in parallel with all the other threads. Preemptive scheduling means a thread can be stopped at any point so other threads can use one of the available processors. Threads must communicate to coordinate the computation they are completing together. With shared memory, one thread can write to a field of an object and another thread can then read the value written. Shared memory and preemption (or true parallelism) are a difficult combination (e.g., a thread might be preempted between executing `data=x;` and executing `data_changed=true;`), so languages provide synchronization mechanisms by which programmers can prevent some thread interleavings.

For example, mutual-exclusion locks have acquire and release operations. If thread A invokes the acquire operation on a lock that thread B has acquired but not yet released, then thread A is blocked (does not run) until A releases the lock and B holds the lock. Incorrect locking protocols can lead to races (undesirable interleavings) or deadlocks (a cycle of threads that can never proceed because they are all waiting for a blocked thread to release a lock).

Transactional memory provides a synchronization mechanism that is easier-to-use but harder-to-implement than locks. At its simplest, it is just a new statement form `atomic{`s`}` that executes the statement s *as though* there is no interleaved computation from other threads. In principle,

s can include arbitrary code, but in practice systems typically limit some operations, such as I/O, foreign-function calls, or creating new threads. An explicit abort statement lets programmers indicate the body of the atomic block should be retried again later.[1] For example, a dequeue method for a synchronized queue might be:

```
// block until an object is available.
// getNextObject fails if the queue is empty.
Object dequeue() {
  atomic {
    if(isEmpty())
      abort;
    return getNextObject();
  }
}
```

TM implementations try to execute the atomic-block body s concurrently with other computation, implicitly aborting and retrying if a conflict is detected. This is important for performance (not stopping all other threads for each atomic block) and fairness: if s runs too long, other threads must be allowed to continue and the thread executing s should retry the transaction. In a different shared-memory state, s may complete quickly. Conflicts are usually defined as memory conflicts: s and another thread access the same memory and at least one access is a write. The essence of a TM implementation is two-fold: detecting conflicts and ensuring all of a transaction's updates to shared memory appear to happen "at once".

The distinction between weak- and strong-atomicity [8] refers to a system's behavior when a memory access not within the dynamic scope of an atomic block conflicts with a concurrent access (by another thread) within such a scope. Weak-atomicity systems can violate a transaction's isolation in this case, and can produce much stranger program behavior than is generally appreciated [46].

Prohibiting memory conflicts between parallel transactions is sometimes unnecessarily conservative. For example, if two transactions both use a unique-ID generator, they may both increment a counter but there is no logical conflict. Open nesting is a language construct supporting such non-conflict access. The statement `open{`s`}` executes s within a transaction, but (1) accesses in s are not considered for conflict detection and (2) accesses in s are not undone if the transaction aborts.

Obstruction-freedom is, roughly speaking, the property that any transaction can continue (i.e., advance its program counter) even if all other transactions are suspended. Some TM implementations have this property and some do not; its importance is fairly controversial [14].

Transactions are a classic concept in databases and distributed systems. Transactional support in hardware [30], programming languages [21], and libraries [45] had early advocates, with recent interest beginning with Harris and Fraser's work for Java [24]. Approaches to implementing TM in compilers [46, 27, 2, 25, 26, 44, 36, 32], libraries [29, 37, 28], hardware [23, 9, 38, 42, 4, 40, 41], and software/hardware hybrids [12, 34] have been published, and transactions are part of several next-generation languages [3, 10, 11].

[1] This abort is an abort-and-retry; some systems also have an abort-and-continue.

2.3 Motivations for Transactional Memory

In general, TM advocates believe it is better than locking because it has software-engineering benefits—avoiding locks' difficulties—and performance benefits—due to optimistic concurrency, transactions proceed in parallel unless there are dynamic memory conflicts. Several idioms where TM is superior have been given:[2]

- It is easier to evolve software to include new synchronized operations. For example, consider the simple bank-account class in Figure 1. If version 1 of the software did not anticipate the need for a `transfer` method, the self-locking approach makes sense. Given this, modifying the software to support `transfer` without potential races (see `transfer_wrong1`) or deadlock (see `transfer_wrong2`) requires wide-scale changes involving subtle lock-order protocols. This issue arises in Java's `StringBuffer` append method, which is presumably why this method is not guaranteed to be atomic [15].
- It is easier to mix fine-grained and coarse-grained operations. For example, most hashtable operations access only a small part of the table, but supporting parallel insert and lookup operations while still having a correctly synchronized "resize table" operation is difficult with locks and trivial with TM.
- It is easier to write code that is efficient when memory-conflicts are rare while remaining correct in case they occur. For example, allowing parallel access to both ends of a double-ended queue is difficult with locks because there can be contention, but only when the queue has fewer than two elements [39]. A solution using TM is trivial.
- With the addition of the "orelse" combinator [25], in which `atomic {` s_1 `} orelse {` s_2 `}` tries s_2 atomically if s_1 aborts (retrying the whole thing if s_2 also aborts), we can combine alternative atomic actions, such as trying to dequeue from one of two synchronized queues, blocking only if both are empty. This addition affords shared-memory concurrency some of the advantages of Concurrent ML's choice operator for message passing [43].

[2] None of these examples are new. To the best of my knowledge, they were originally presented in various forms by Flanagan, Harris, Herlihy, and Peyton Jones, respectively.

```
class Account {
 float balance;
 synchronized void deposit(float amt) {
    balance += amt;
 }
 synchronized void withdraw(float amt) {
    if(balance < amt)
      throw new OutOfMoneyError();
    balance -= amt;
 }
 void transfer_wrong1(Acct other, float amt) {
    other.withdraw(amt);
    // race condition: wrong sum of balances
    this.deposit(amt);
 }
 synchronized void transfer_wrong2(Acct other, float amt) {
    // can deadlock with parallel reverse-transfer
    other.withdraw(amt);
    this.deposit(amt);
 }
}
```

```
class Account {
 float balance;
 void deposit(float amt) {
   atomic { balance += amt; }
 }
 void withdraw(float amt) {
   atomic {
     if(balance < amt)
       throw new OutOfMoneyError();
     balance -= amt;
    }
  }
 void transfer(Acct other,
               float amt) {
   atomic {
     other.withdraw(amt);
     this.deposit(amt);
   }
 }
```

Figure 1. Example showing the difficulty of extending software written with locks. If the `transfer` method must not allow other threads to see an incorrect sum-of-balances-in-all-accounts, then the first attempt on the left is wrong and the second attempt can deadlock. In the TM code on the right, adding `transfer` is straightforward.

In addition, advocates sometimes claim TM does not suffer from races or deadlocks or that, "it is obviously easier." The reasons for these claims and the limitations of them will become apparent in the subsequent two sections.

3. The Core Analogy

Without further ado, I now present the similarities between transactional memory and garbage collection, from the problems they solve, to the way they solve them, to how poor programming practice can nullify their advantages. The points in this section are all technical in nature; any analogies between the social processes behind the technologies of GC and TM are relegated to Section 6.

To appreciate fully the analogy I recommend reading each section twice. First read the descriptions of GC and TM all at once to make sure they are accurate and relevant. Then read the descriptions by interleaving sentences (or even phrases) to appreciate that the structure is identical with the difference being primarily the substitution of a few nouns.

The Problems

Memory management is difficult because one must use memory reclamation to balance correctness, i.e., avoiding dangling-pointer dereferences, and performance, i.e., avoiding the loss of space or even the ability to continue due to space exhaustion. In programs that manually manage memory, the programmer uses subtle whole-program protocols to avoid errors. One of the simpler approaches manually associates each data object with a reference count and requires a nonzero count when accessing the data (freeing the memory when the count is zero). To avoid lost space, it is sufficient to disallow cycles in the object graph, but in practice this requirement is too burdensome. Sharing reference counts among objects (cf. region-based memory management [16]) reduces the number of reference counts but may increase space consumption.

Unfortunately, memory-management protocols are nonmodular: Callers and callees must know what data the other may access to avoid deallocating needed data or making unneeded data unreachable. A small change — for example, a new function that needs data previously deemed no longer necessary — may require wide-scale changes or introduce bugs. In essence, memory management involves nonlocal properties: Correctness requires knowing what data subsequent computation will access. One must reason about how data is used across time to determine when to deallocate an object. If a program change affects when an object is used for the last time, the program's memory management may become wrong or inefficient.

Concurrent programming is difficult because one must use synchronization to balance correctness, i.e., avoiding race conditions, and performance, i.e., avoiding the loss of parallelism or even the ability to continue due to deadlock. In programs that manually manage mutual-exclusion locks, the

programmer uses subtle whole-program protocols to avoid errors. One of the simpler approaches associates each data object with a lock and holds the lock when accessing the data. To avoid deadlock, it is sufficient to enforce a partial-order on the order a thread acquires locks, but in practice this requirement is too burdensome. Sharing locks among objects reduces the number of locks but may reduce parallelism.

Unfortunately, concurrency protocols are non-modular: Callers and callees must know what data the other may access to avoid releasing locks still needed or acquiring locks that could make threads deadlocked. A small change — for example, a new function that must update two thread-shared objects atomically with respect to other threads — may require wide-scale changes or introduce bugs. In essence, concurrent programming involves nonlocal properties: Correctness requires knowing what data concurrently executing computation will access. One must reason about how data is used across threads to determine when to acquire a lock. If a program change affects when an object is used concurrently, the program's synchronization protocol may become wrong or inefficient.

The Solutions

GC takes the subtle whole-program protocols sufficient to avoid dangling-pointer dereferences and space leaks and moves them into the language implementation. As such, they can be implemented "once and for all" by experts focused only on their correct and efficient implementation. Programmers specify only what data points to what, relying on the implementation to be correct (no dangling pointers) and efficient (reclaiming unreachable memory in a timely manner). Note the garbage collector does maintain subtle whole-program invariants, often with the support of the compiler and/or hardware. As examples, header words may identify which fields hold pointers and a generational collector may assume there are no unknown pointers from "mature" objects to "young" objects.

The whole-program protocols necessary for GC are most easily implemented in some combination of the compiler (particularly for read and/or write barriers) and the runtime system (including hardware) because we can localize the implementation of the protocols. Put another way, the difficulty of implementation does not increase with the size of the source program.

In theory, garbage collection can improve performance by increasing spatial locality (due to object-relocation), but in practice we pay a moderate performance cost for software-engineering benefits.

TM takes the subtle whole-program protocols sufficient to avoid races and deadlock and moves them into the language implementation. As such, they can be implemented "once and for all" by experts focused only on their correct and efficient implementation. Programmers specify only what must be performed atomically (as viewed from other threads), relying on the implementation to be correct (no atomicity violations) and efficient (reasonably parallel, particularly when transactions do not contend for data). Note the transactional-memory implementation does maintain subtle whole-program invariants, often with the support of the compiler and/or hardware. As examples, header words may hold version numbers and systems optimizing for thread-local data may assume there are no pointers from thread-shared objects to thread-local objects.

The whole-program protocols necessary for TM are most easily implemented in some combination of the compiler (particularly for read and/or write barriers) and the runtime system (including hardware) because we can localize the implementation of the protocols. Put another way, the difficulty of implementation does not increase with the size of the source program.

In theory, transactional memory can improve performance by increasing parallelism (due to optimistic concurrency), but in practice we may pay a moderate performance cost for software-engineering benefits.

Incomplete Solutions

GC is probably not a natural match for all parts of all applications. Examples may include applications where trading space for time is a bad performance decision or where heap-allocated data lifetime follows an idiom not closely approximated by reachability. Language features such as weak pointers allow reachable memory to be reclaimed, but using such features correctly is best left to experts or easily recognized situations such as a software cache. Recognizing that GC may not always be appropriate, languages can complement it with support for other idioms, such as regions [6, 22, 20].

In the extreme, programmers can code manual memory management on top of garbage collection, destroying the advantages of garbage collection. Figure 2 shows one basic approach in Java.[3] More efficient implementations (e.g., using a free list) are straightforward extensions. A programmer can then treat `mallocT` as the way to get fresh `T` objects, but an object passed to `freeT` may be returned by `mallocT`, reintroducing the difficulties of dangling pointers.[4] In practice, we can expect less extreme idioms that still introduce application-level buffers for frequently used objects.

TM is probably not a natural match for all parts of all applications. Examples may include applications where optimistically attempting parallel transactions is a bad perfor-

[3] The code supports only objects of a single type, but separate per-type allocators or fancier tricks such as reflection can avoid this.

[4] The fact that the program cannot "seg fault" is little consolation; in fact, if my C program has dangling-pointer dereferences I hope it *does* "seg fault" rather than silently use aliases to just-allocated memory.

```
class AllocT {
  T[]       buffer    = new T[100];
  boolean[] available = new boolean[100];
  AllocT() {
     for(int i=0; i<1000; ++i) buffer[i]    = new T();
     for(int i=0; i<1000; ++i) available[i] = true;
  }
  T mallocT() {
     for(int i=0; i < 1000; ++i) {
       if(!available[i]) continue;
       available[i] = false;
       return buffer[i];
     }
     throw new OutOfMemoryError(); //could resize buffer
  }
  void freeT(T t) {
     for(int i=0; i < 1000; ++i)
       if(buffer[i]==t) available[i] = true;
  }
}
```

```
class Lock {
  boolean held = false;
  void acquire() {
    while(true)
      atomic {
        if(!held) {
          held=true;
          return;
        }
      }
  }
  void release() {
    atomic { held = false; }
  }
}
```

Figure 2. Left: Java code building manual memory management on top of garbage collection. Right: Java code with `atomic` statements building locks on top of transactional memory.

mance decision or where correct synchronization follows an idiom not closely approximated by data conflicts. Language features such as open-nested transactions allow transactions with data conflicts to succeed in parallel, but using such features correctly is best left to experts or easily recognized situations such as unique-identifier generation. Recognizing that TM may not always be appropriate, most prototypes continue to support other idioms, such as locks and condition variables.

In the extreme, programmers can code locks on top of transactional memory, destroying the advantages of transactional memory. Figure 2 shows one basic approach in Java. More powerful libraries (e.g., supporting reentrancy) are straightforward extensions. A programmer can then treat the `Lock` methods as synchronization primitives and reintroduce the difficulties of locking. In practice, we can expect less extreme idioms that still reintroduce application-level races and deadlocks.

Two Basic Approaches

Despite a wide variety of garbage-collection algorithms in terms of details, there are two fundamental approaches. First, tracing collection uses tracing code to find all live data and garbage-collect the rest. The code running the application simply treats all data as live, delaying the check for unreachable data. Second, automatic reference-counting checks the number of references to an object while the application runs, allowing unreachable data to be found immediately. However, a cycle of garbage can lead to data being kept when it should not be, so other techniques (e.g., "trial deletion") can complement the core reference-counting mechanism. In practice, compilers employ deferred reference counting to avoid the overhead of constantly manipulating reference counts while delaying the check for garbage.

Despite a wide variety of transactional-memory algorithms in terms of details, there are two fundamental approaches. First, update-on-commit TM uses private copies of accessed data and relies on a validate/commit protocol to detect conflicts and reflect changes back to shared memory. The code running the application simply assumes there will be no conflicts, delaying the check for inconsistent data. Second, update-in-place TM checks the consistency of data while the application runs, allowing inconsistent data to be detected immediately and the transaction aborted. However, a cycle of transactions can cause themselves to all abort when it is unnecessary,[5] so other techniques (e.g., priority-based schemes) can complement the core update-in-place mechanism. In practice, compilers employ optimistic reads with update-in-place to avoid the overhead of checking consistency on every field read while delaying the check for transaction-consistency.

I/O

GC is difficult to reconcile with external effects such as input and output *of pointers* because it is not always possible to establish what pointer values may be encoded by

[5] This point is subtle: Suppose transaction A seeks to write to object o which has already been written to by transaction B. An update-in-place system might abort A, but if B is going to abort later (perhaps because it attempts to write something already written to by A), we could let A proceed.

output. Because such "hidden pointers" do not matter unless the external world uses them to generate input that "rematerializes" the pointer, the essence of the problem is input-after-output. In a distributed setting, GC becomes more difficult because it requires consensus on when data is no longer needed. In practice, serializing objects is often sufficient.

TM is difficult to reconcile with external effects such as input and output because it is not always meaning-preserving to delay output until a transaction commits. Because such output does not matter unless the external world uses it to generate input needed for the transaction, the essence of the problem is input-after-output. In a distributed setting, TM becomes more difficult because it requires consensus on when a transaction is completed. It is not yet clear what suffices in practice.

False Sharing

For reasons of performance and simplicity, garbage collectors typically reclaim only entire objects, rather than reclaiming parts of objects that contain dead fields. That is, memory management is done with object-level granularity. As a result, extra space can be consumed, but space-conscious programmers aware of object-level granularity can restructure data to circumvent this approximation because object size is under programmer control.

However, with conservative garbage collection, programmers can no longer fully control how much memory is reachable. Because the memory address at which an object is allocated is uncontrollable, a collision with a live integer value could lead to space consumption.

For reasons of performance and simplicity, some implementations of transactional-memory detect memory conflicts between entire objects, rather than permitting parallel access to distinct parts of objects. That is, conflict management is done with object-level granularity. As a result, extra contention can occur, but parallelism-conscious programmers aware of object-level granularity can restructure data to circumvent this approximation because object size is under programmer control.

However, with granularity coarser than objects (e.g., at cache lines), programmers can no longer fully control how many false conflicts occur. Because the memory address at which an object is allocated is uncontrollable, adjacent placement of independent objects could lead to lost parallelism.

Progress Guarantees

Most garbage collectors do not make real-time guarantees. Providing such worst-case guarantees can incur substantial extra cost in the expected case, so real-time collection is typically eschewed unless an application needs it. The key complication is continuing to make progress with collection while the program could be performing arbitrary operations on the reachable objects the collector is analyzing.

Some implementations of transactional memory do not make obstruction-freedom guarantees. Providing such worst-case guarantees can incur substantial extra cost in the expected case, so obstruction-freedom should perhaps be eschewed unless an application needs it. The key complication is continuing to make progress with any transaction while another thread could be suspended after having accessed any of the objects the transaction is accessing.

Static-Analysis Improvements

Compile-time information can improve the performance of GC. The most common approach is liveness analysis for determining that the contents of a local variable is not used after a certain program point. This information allows the collector to treat fewer local variables as roots. Other analyses can also prove useful. For example, in a generational setting, static analysis can remove write barriers for objects that are definitely in the nursery [49].

Compile-time information can improve the performance of TM. The most common approach is escape analysis for determining that the contents of a local variable is not reachable from multiple threads before a certain program point. This information allows the transactional-memory implementation to treat fewer memory accesses as potential memory conflicts. Other analyses can also prove useful. For example, in a strong-atomicity setting, static analysis can remove write barriers for objects that are definitely never accessed in a transaction [46].

Conclusion

Figure 3 summarizes some of the connections discussed. In general, the point is that GC and TM aim to automate what is otherwise an error-prone and detailed task of balancing correctness and performance. As general solutions, GC and TM rely on approximations that seem to work well for many but not all applications, and they cannot perform well with code specifically aimed at circumventing their advantages. Efficient and elegant approaches to implementing and improving GC/TM can involve any combination of code generation, static analysis, run-time systems, and hardware.

4. The Essence of Concurrency

The previous section described many ways that transactional memory is like garbage collection. The most exciting aspect is that it makes developing, maintaining, and evolving concurrent programs easier, by moving the low-level synchronization protocols into the programming-language implementation. In so doing, we let the application develop-

memory management	concurrency
dangling pointers	races
space exhaustion	deadlock
regions	locks
garbage collection	transactional memory
reachability	memory conflicts
nursery data	thread-local data
weak pointers	open nesting
I/O of pointers	I/O in transactions
tracing	update-on-commit
automatic reference counting	update-in-place
deferred reference counting	optimistic reads
conservative collection	false memory-conflicts
real-time collection	obstruction-freedom
liveness analysis	escape analysis

Figure 3. Summary of some differences in nouns between the two sides of the analogy.

ers focus on the higher-level concern of determining where critical sections should begin and end, i.e., determining which shared-memory states should be accessible to multiple threads.

However, the claim that TM makes concurrent programming easier must *not* be misconstrued as a claim that it makes concurrent programming easy. Delimiting critical sections is a fundamentally difficult application-specific challenge with — by definition — no analogue in sequential programming. Critical sections that are too small lead to application-level races because other threads may see inconsistent state (e.g., the sum of two bank-account balances may be too large). Critical sections that are too large may impede application-level progress because they deny access to an intermediate state another thread needs.

An example adapted from Blundell et al. [8] in Figure 4 illustrates this point. In the example, Thread 2's critical section can complete successfully only after Thread 1's first critical section, and Thread 1's second critical section can complete successfully only after Thread 2's critical section. Therefore, program behavior is altered (from terminating to nonterminating) if Thread 1's critical sections are combined by placing them in a larger atomic block.

This example, however, does not contradict the claim that TM enables composable critical sections, such as the account-transfer example in Section 2. Unlike with programs built out of locks, we *can* create larger critical sections out of smaller ones without introducing deadlock or changing the locking protocol for code already written. Whether doing so is *appropriate* for the application falls under the essential difficulty of shared-memory concurrent programming, i.e., delimiting critical sections.

I have heard advocates of TM claim that atomic blocks are more "declarative" than locks, but I have not seen a

Initially, x = y = 0

```
Thread 1                 | Thread 2
//atomic {               |  atomic {
    atomic {             |    if(x==0)
      x = 1;             |      abort;
    }                    |    y = 1;
    atomic {             |  }
      if(y==0)           |
        abort;           |
    }                    |
//}                      |
```

Figure 4. An example showing that enlarging critical sections can break application correctness. Uncommenting the atomic statement in Thread 1 leads to application-level deadlock.

precise justification of this claim. Here it is: The essence of shared-memory concurrent programming is deciding where critical sections should begin and end. With atomic blocks, programmers do precisely that rather than encode critical sections via other synchronization mechanisms. That is, they declare where interleaved computation from other threads is and is not allowed.

Returning to our analogy, there *is* a connection to be made. Concurrency adds expressiveness (the ability to have multiple threads of control) and performance potential (via parallelism) just as memory reclamation adds expressiveness (the ability to compute with a conceptually unbounded amount of memory rather than statically allocating all objects) and performance potential (via a memory footprint matching dynamic behavior). On the memory management side, the essential difficulty introduced is to avoid using too much memory, something much easier to control, albeit conservatively, when you allocate all your memory at the beginning of program execution.

There is, however, a large qualitative difference in the two essential difficulties.[6] Avoiding space leaks in a garbage-collected program just seems much easier than avoiding incorrect interleavings in a transactional-memory program. *That* is why TM, despite being a great leap forward, will not make concurrent programming easy.

5. A Brief Digression for Types

It turns out GC and TM are not the only solutions to memory management and concurrency that enjoy remarkable similarities. The type systems underlying statically checked languages for region-based memory management [47] and lock-based data-race prevention [1] are essentially identically structured type-and-effect systems. In adapting work on both to the Cyclone programming language, I was able to exploit this similarity to provide a simpler and more regular

[6] There is also a theoretical difference: Unbounded memory is necessary for Turing-completeness; concurrency is not.

type system [17, 20, 18]. I will not repeat that development here, but by very briefly sketching the similarity, my intent is to:

- Provide further evidence that memory management and concurrency are problems with very similar structure, not just, "two different problems in software development."
- Suggest (though I cannot prove it, even to myself) that my background in type systems provided the intellectual grounding that allowed me to stumble across the GC/TM analogy.

In region-based memory management, we can have these primitives:[7]

`new_region()`	create a new region
`free_region(rgn)`	deallocate `rgn` and all its objects
`use_region(rgn){s}`	allow access to objects in `rgn` in statement `s`
`new (rgn) C()`	put a new object in region `rgn`

A type-and-effect system can ensure a region's objects are created or accessed only in the dynamic scope of an appropriate `use_region` and a region is freed only outside such a statement. As such, the only dynamic checks are for "has this region already been deallocated" and occur on entry to `use_region` or on `free_region`. The key to type soundness (no dangling-pointer dereferences) is using fresh type variables to ensure every region has a type distinct from every other region. The key to expressiveness is parametric polymorphism so that methods can be parameterized over the regions in which the data they access resides. A computation's effect is the set of regions that may need to be live while the computation is performed.

In lock-based data-race prevention, we can have these primitives:[8]

`new_lock()`	create a new lock
`synchronized(lk){s}`	allow access to objects guarded by `lk` in statement `s`
`new (lk) C()`	create new object guarded by `lk`

A type-and-effect system can ensure the objects guarded by a lock are accessed only in the dynamic scope of an appropriate `synchronized`. As such, the only dynamic checks are for "is this lock available" and occur on entry to `synchronized`. The key to type soundness (no data races) is using fresh type variables to ensure every lock has a type distinct from every other lock. The key to expressiveness is parametric polymorphism so that methods can be parameterized over the locks guarding the data they access. A computation's effect is the set of locks that may need to be held while the computation is performed.

[7] This description is similar to the primitives in many systems, but not exactly like any of them.

[8] See previous note.

6. Unsubstantiated Conjectures

So far, I have been careful to focus on the essence of the technologies being compared, emphasizing that GC and TM share a remarkably similar structure without jumping to the incorrect conclusion that TM will make concurrent programming as easy as sequential programming. While you might quibble with some specific aspect of the analogy as presented, the sheer number of correspondences suggests there is something fundamental between approaches to memory management and concurrency.

Leaving the realm of thoughtful technological inquiry to stretch the analogy further proves too irresistible. After all, GC and TM have both been developed in the real world, with all the engineering and social pressures that entails. Provided we do not take the conclusions too seriously, we can try to draw parallels at this "metalevel" as well. I will make three claims about the development and success of GC and then extrapolate to what it would mean for TM to follow a similar arc.

Claim #1: GC did not need hardware support to succeed.

Hardware support for GC has certainly been considered and built, and at least some believed GC would never be fast enough to be accepted without it. But special-purpose hardware has trouble supporting algorithmic advances in a timely fashion; GC hardware just never kept up. Business priorities may have been another, possibly primary, reason GC hardware never predominated. Nonetheless, architectural features (from the instruction set to the cache write-back policy) still can have significant effects on the performance of languages with GC [13, 7, 31].

TM can also benefit from hardware support, particularly the ability to detect memory conflicts with technology at the level of cache-coherence protocols. However, if the history of GC is a guide, hardware does not need any specific notion of a transaction for TM to succeed. Moreover, other architectural features may have a bigger effect on performance than we currently realize.

Of course, hardware support for transactions, particularly schemes for mixing software and hardware transactions [12, 34], are currently a more active area in the architecture community than hardware GC ever was. So, turning the analogy around, perhaps the time is ripe for a resurgence of research into GC hardware.

Claim #2: GC took decades longer to reach mainstream adoption than its initial developers expected.

The lag time between the basic research on GC and its widespread popularity is a standard example of computer-science research taking a nearly unbearable amount of time

to "pay off," but still being well worth it in the end. While it is possible the excitement around TM will help expedite the process of widespread adoption — and in general technology adoption is accelerating — I think we should be prepared for the TM lag time to be longer than anyone expects. This in no way reduces the importance of TM research.

Claim #3: Mandatory GC is usually sufficient despite its approximations.

As already described, GC essentially relies on the approximation that reachable objects may be live, and this approximation can make an arbitrary amount of memory live arbitrarily longer. For programmers to avoid suffering from this, unsafe languages can provide a "back-door" for explicit memory deallocation and safe languages can provide features like weak pointers. In practice, these features are sometimes necessary, but plenty of practical systems have been built that rely exclusively on reachability for determining liveness. Moreover, the exact definition of "what is reachable" — which in theory is necessary for reasoning about program performance — is typically left unspecified and compiler optimizations are allowed to subtly change reachability information.

I have argued the TM analogue of the reachability approximation is the memory-conflict approximation — assuming that two transactions accessing the same memory (where at least one access is a write) cannot proceed in parallel. The "back-door" for letting programmers avoid this approximation is open-nesting. The question then is whether open-nesting is so important that it must be addressed as a primary obstacle to developing transactional-memory implementations. The limitations of not having open-nesting and the situations where it is the best solution may be few, just as many programmers in garbage-collected languages never bother with weak pointers. Moreover, the exact definition of "what is a memory conflict" as well as related issues of how conflicts are arbitrated (e.g., notions of fairness) may not prove important for most programs.

7. Conclusion

A good analogy can provoke thought, provide perspective, guide research, and promote an idea. An analogy need not be valid science (i.e., a proof) nor a complete and total correspondence. Rather, it can serve to describe concisely (if imperfectly) one idea in terms of another better-known idea. Humans often learn and understand via analogies, so I believe a thought-provoking analogy can in and of itself serve as a contribution toward our common research goals.

My intent has been to present how one can view transactional memory from the perspective of garbage collection, which though perhaps surprising at first springs from fundamental similarities between memory management and synchronization. In so doing, I have made a case for transactional memory that I personally find quite compelling, which is why I continue to do research on the topic. To restate it succinctly, by moving mutual-exclusion protocols into the language implementation (any combination of compiler, run-time system, and hardware), we make it easier to write and maintain shared-memory concurrent programs in a more modular fashion. This argument is not the only one that has been put forth in favor of transactional memory; it is simply my personal opinion that it is the most important.

Equally important, this argument does not oversell transactional memory: We still need better tools and methodologies to help programmers determine where transactions should begin and end.[9] Delimiting transactions is the essential difficulty of concurrent programming, and making transactions a language primitive does not change this.

For me, the most important conclusion arising from the analogy is that GC and TM rely on simple and usually-good-enough approximations (namely, reachability and memory-conflicts) that are subject to false-sharing problems. This fact can inform how we teach programmers to use TM (and GC) effectively and can guide research into reducing the approximations. I can credit it with inspiring the not-accessed-in-transaction analysis I developed recently [46] and I am hopeful it and other points in the analogy can inspire more ideas.

Indeed, the primary intended effect of this presentation is to incite such thoughts in others, whether readers agree or—more interestingly—disagree with the analogy. In particular:

- If you believe the GC/TM analogy is useful, can you use it to advance our understanding of TM or GC? For example, is there a TM analogue of generational collection? This question is crucial if one ascribes to the interpretation of history in which GC was less practical prior to generational collection. More abstractly, is there a unified theory of TM as beautiful is Bacon et al's unified theory of GC [5] in which tracing and automatic reference-counting are algorithmic duals?
- If you believe the GC/TM analogy is flawed or deemphasizes some crucial aspect of TM, can you identify why? I have essentially ignored issues of fairness and contention management, which some may feel are essential aspects of TM. Does considering these issues fundamentally change what we should conclude?
- Are my conjectures about TM's future based on GC's past—that hardware support is unnecessary, that TM will take longer to reach the mainstream than we expect, and that open-nesting is not always necessary—likely? Are they preventable? Should we try to prevent them? If you are in a position to direct TM's future, can you seek guidance from the history of GC?

[9] This conclusion is not to discount classic and current work in this area, but such a discussion is beyond the present scope.

Acknowledgments

Conversations with Emery Berger, Jim Larus, Jan Vitek, members of the WASP group at the University of Washington, and many others led to some of the ideas presented here.

The author's research on transactional memory has been generously supported by Intel Corporation, Microsoft Corporation, and the University of Washington Royalty Research Fund.

References

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems*, 28(2), 2006.

[2] A.-R. Adl-Tabatabai, B. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *ACM Conference on Programming Language Design and Implementation*, 2006.

[3] E. Allen, D. Chase, J. Hallet, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Tobin-Hochstadt. The Fortress language specification, version 1.0β, Mar. 2007. http://research.sun.com/projects/plrg/Publications/fortress1.0beta.pdf.

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

[5] D. F. Bacon, P. Cheng, and V. T. Rajan. A unified theory of garbage collection. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[6] G. Bellella, editor. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[7] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *SIGMETRICS - Proceedings of the International Conference on Measurements and Modeling of Computer Systems*, 2004.

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

[9] B. D. Carlstrom, J. Chung, A. McDonald, H. Chafi, C. Kozyrakis, and K. Olukotun. The Atomos transactional programming language. In *ACM Conference on Programming Language Design and Implementation*, 2006.

[10] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2005.

[11] Cray Inc. Chapel specification 0.4. http://chapel.cs.washington.edu/specification.pdf.

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

[13] A. Diwan, D. Tarditi, and J. E. B. Moss. Memory system performance of programs with intensive heap allocation. *ACM Transactions on Computer Systems*, 13(3), 1995.

[14] R. Ennals. Software transactional memory should not be lock free. Technical Report IRC-TR-06-052, Intel Research Cambridge, 2006. http://berkeley.intel-research.net/rennals/pubs/052RobEnnals.pdf.

[15] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *ACM Conference on Programming Language Design and Implementation*, 2003.

[16] D. Gay and A. Aiken. Language support for regions. In *ACM Conference on Programming Language Design and Implementation*, 2001.

[17] D. Grossman. *Safe Programming at the C Level of Abstraction*. PhD thesis, Cornell University, 2003.

[18] D. Grossman. Type-safe multithreading in Cyclone. In *ACM Workshop on Types in Language Design and Implementation*, 2003.

[19] D. Grossman, J. Manson, and W. Pugh. What do high-level memory models mean for transactions? In *ACM SIGPLAN Workshop on Memory Systems Performance & Correctness*, 2006.

[20] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *ACM Conference on Programming Language Design and Implementation*, 2002.

[21] N. Haines, D. Kindred, J. G. Morrisett, S. M. Nettles, and J. M. Wing. Composing first-class transactions. *ACM Transactions on Programming Languages and Systems*, 16(6), 1994.

[22] N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. In *ACM Conference on Programming Language Design and Implementation*, 2002.

[23] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (TCC). In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[24] T. Harris and K. Fraser. Language support for lightweight transactions. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2003.

[25] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *ACM Symposium on Principles and Practice of Parallel Programming*, 2005.

[26] T. Harris, S. Marlow, and S. Peyton Jones. Haskell on a shared-memory multiprocessor. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, 2005.

[27] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *ACM Conference on Programming Language Design and Implementation*, 2006.

[28] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2006.

[29] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *ACM Symposium on Principles of Distributed Computing*, 2003.

[30] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *International Symposium on Computer Architecture*, 1993.

[31] M. Hertz and E. D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2005.

[32] B. Hindman and D. Grossman. Atomicity via source-to-source translation. In *ACM SIGPLAN Workshop on Memory Systems Performance & Correctness*, 2006.

[33] R. E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[34] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *ACM Symposium on Principles and Practice of Parallel Programming*, 2006.

[35] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2006.

[36] J. Manson, J. Baker, A. Cunei, S. Jagannathan, M. Prochazka, B. Xin, and J. Vitek. Preemptible atomic regions for real-time Java. In *26th IEEE Real-Time Systems Symposium*, 2005.

[37] V. J. Marathe, W. N. Scherer, and M. L. Scott. Adaptive software transactional memory. In *International Symposium on Distributed Computing*, 2005.

[38] A. McDonald, J. Chung, B. D. Carlstrom, C. Cao Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *International Symposium on Computer Architecture*, 2006.

[39] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *ACM Symposium on Principles of Distributed Computing*, 1996.

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

[41] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in LogTM. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[42] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *32nd International Symposium on Computer Architecture*, 2005.

[43] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[44] M. F. Ringenburg and D. Grossman. AtomCaml: First-class atomicity via rollback. In *10th ACM International Conference on Functional Programming*, 2005.

[45] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, Special Issue(10), 1997.

[46] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. Hudson, K. Moore, and B. Saha. Enforcing isolation and ordering in STM. In *ACM Conference on Programming Language Design and Implementation*, 2007.

[47] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2), 1997.

[48] P. R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, 1994.

[49] K. Zee and M. Rinard. Write barrier removal by static analysis. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2002.