International Journal of Scientific Research in Computer Science, Engineering and Information Technology
© 2019 IJSRCSEIT | Volume 5 | Issue 2 | ISSN : 2456-3307
DOI : https://doi.org/10.32628/CSEIT1951117

The Transactional Memory

V. M. Dhivya Shri[1], Mrs. K. Reshma[2]
[1]B.Sc (Computer Technology), Department of Information and Computer Technology, Sri Krishna Adithya College of Arts and Science, Coimbatore, Tamil Nadu, India
[2]Assistant Professor, Department of Information & Computer Technology, Sri Krishna Adithya College of Arts and Science, Coimbatore, Tamil Nadu, India

ABSTRACT

Transactional memory (TM) promises to simplify concurrent programming. Language-based constructs allow programmers to denote atomic regions declaratively. Its implementations operate by tracking loads and stores to memory and by detecting concurrent conflicts. TM allows programmers to write simpler programs that are composable and deadlock-freeThis essay presents remarkable similarities between transactional Memory and garbage collection. The connections are fascinating in their own right, and they let us better stand one technology by thinking about the corresponding issues for the other.

Keywords : Transactional Memory, Garbage Collection

I. 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 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 improvement over locks, then we are being neither good scientists nor good engineers. 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 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 understanding of 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 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 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 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. Second are some conjectures one can make by pushing the analogy too far finally, the conclusion describes the intended effects of publishing this article.

II. BACKGROUND

A full introduction to garbage collection and transactional memory is clearly beyond our scope exist for GC and TM , 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 and interthread communication is via mutable shared-memory. Much of the discussion applies to other paradigms but less to communication via message-passing.

Transactional Memory

The assumed concurrency model allows programmers to create additional threads to execute code in parallel with all the other threads.Pre-emptive 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 pre-emption are a difficult combination so languages provide synchronization mechanisms by which programmers can prevent some thread inter leavings. 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 until A releases the lock and B holds the lock. Incorrect locking protocols can lead to races or deadlocks. 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 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. 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 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 behaviour than is generally appreciated 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 even if all other transactions are suspended. Some TM implementations have this property and some do not; its importance is fairly controversial Transactions are a classic concept in databases and distributed systems. Transactional support in hardware programming languages and libraries had early advocates, with recent interest beginning with Harris and Fraser's work for Java Approaches to implementing TM in compilers, libraries, hardware and software/hardware hybrids have been published, and transactions are part of several next-generation languages.

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:

• It is easier to evolve software to include new synchronized operations. For example, consider the simple bankaccount 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 .

• 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 memoryconflicts 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 { s1 } orelse { s2 } tries s2 atomically if s1 aborts (retrying the whole thing if s2 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.

III. 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 .

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. 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. 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. 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. In the extreme, programmers can code manual memory management on top of garbage collection, destroying the advantages of garbage collection. 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. 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

```
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; }
}
}
```

IV. 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 analysing, and continuing to make progress with any transaction while another thread could be suspended after having accessed any of the objects the transaction is accessing.

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.

V. 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 and lock-based data-race prevention 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 analogy.

In region-based memory management, we can have these 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 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.

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 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.

VI. 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.

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), 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; We still need better tools and methodologies to help programmers determine where transactions should begin and end. 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.

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 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 contention management, which some may feel are essential aspects of TM. Does considering these issues fundamentally change what we should conclude.

VII. 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]. 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.

[3]. 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.

[4]. 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 1.0_, Mar. 2007.http://research.sun.com/projects/plrg/Publications/fortress1.0beta.pdf.

[5]. 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.

[6]. 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.

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

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

[9]. P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An objectoriented approach to non-uniform cluster computing.

[10]. 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.

Cite this article as :

V. M. Dhivya Shri, Mrs. K. Reshma, "The Transactional Memory", International Journal of Scientific Research in Computer Science, Engineering and Information Technology (IJSRCSEIT), ISSN : 2456-3307, Volume 5 Issue 2, pp. 13-20, March-April 2019. Available at doi : https://doi.org/10.32628/CSEIT1951117
Journal URL : http://ijsrcseit.com/CSEIT1951117