# Subtleties of Transactional Memory Atomicity Semantics

Colin Blundell E Christopher Lewis Milo M. K. Martin
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, Pennsylvania, USA
{*blundell,lewis,milom*}@*cis.upenn.edu*

***Abstract*— Transactional memory has great potential for simplifying multithreaded programming by allowing programmers to specify regions of the program that must appear to execute atomically. Transactional memory implementations then optimistically execute these transactions concurrently to obtain high performance. This work shows that the same atomic guarantees that give transactions their power also have unexpected and potentially serious negative effects on programs that were written assuming narrower scopes of atomicity. We make four contributions: (1) we show that a direct translation of lock-based critical sections into transactions can introduce deadlock into otherwise correct programs, (2) we introduce the terms *strong atomicity* and *weak atomicity* to describe the interaction of transactional and non-transactional code, (3) we show that code that is correct under weak atomicity can deadlock under strong atomicity, and (4) we demonstrate that sequentially composing transactional code can also introduce deadlocks. These observations invalidate the intuition that transactions are strictly safer than lock-based critical sections, that strong atomicity is strictly safer than weak atomicity, and that transactions are always composable.**

## I. Introduction

In response to the performance and complexity challenges of locks, researchers have proposed hardware and software mechanisms for synchronization via *transactions* [2], [5]–[9], [11], [12]: segments of code that execute atomically[1] with respect to each other, *i.e.*, each transaction executes without interference from other transactions. Like lock-based critical sections, transactions are a mechanism for mutual exclusion, but transactions are simpler (specifying atomicity without naming a lock) and more efficiently implemented (optimistically executing concurrently, rolling back on dynamically detected inter-transaction conflicts). This combination of intuitive interface and efficient implementation has the potential to solve many lock-related problems.

Although transactions have great potential, this work describes subtle issues and common misconceptions about their semantics.[2] In particular, we investigate the implications of different *scopes* of atomicity. The scope of atomicity determines precisely what code must be atomic and with respect to what other code it must appear atomic. Naturally, narrowing the scope of atomicity can break a program, because a narrower scope introduces additional possible interleavings that may contain data races. Conversely, we show that in several circumstances, correct programs created assuming one scope of atomicity (*e.g.*, that of lock-based critical sections) can deadlock when run on a system supporting a *broader* scope of atomicity (*e.g.*, that of transactions). This result is counter-intuitive because broader atomic scope limits interleaving, which we would expect to eliminate—not introduce—bugs. We make four main contributions:

- We show that a direct translation of lock-based critical sections into transactions can introduce deadlock into an otherwise correct program.
- We define two models of atomicity scope for transactional systems: *strong atomicity* guarantees atomicity between transactions and non-transactional code, and *weak atomicity* guarantees atomicity among only transactions.
- We show that a program that is correct under the weak atomicity model may deadlock under the strong atomicity model. The intuitive view that a stronger atomicity model will correctly execute a superset of the code that is correct under a weaker atomicity model is fallacious.
- We demonstrate that sequential composition of transactions to form a single, larger transaction can also cause deadlock.

This work invalidates the intuitive notion that transactions are strictly safer than lock-based critical sections, that strong atomicity is strictly safer than weak atomicity, and that transactions are always composable. In all of these cases, broadening the scope of atomicity restricts the set of legal program interleavings, which would seem to only help the programmer by removing (potentially) buggy interleavings. However, programmers may create programs that intentionally or unintentionally exploit such interleavings, resulting in a program that *requires* concurrent execution of these regions to avoid deadlock.

## II. Critical Sections ≠ Transactions

Transactions are a promising replacement for lock-based critical sections, so we would like to extend the benefits of transactional systems to legacy lock-based programs. However, directly transforming lock-based critical sections into transactions (by replacing lock acquires and releases with transaction begin and end operations, respectively) is not always safe. This conversion broadens the scope of atomicity, thus changing the program's semantics: a critical section that was previously atomic only with respect to other critical sections guarded

[1] In the ACID properties of database transactions, it is isolation that guarantees non-interference, not atomicity; however, the programming languages and software verification communities have long used the term "atomic" to mean "in isolation", and the transactional memory community has largely adopted this usage.

[2] This manuscript is an enhanced version of our earlier workshop paper [3].

```
Boolean flagA = false, flagB = false;
Object o1, o2;
```

| $P_1$ | $P_2$ |
|---|---|
| ~~synchronize(o1) {~~ | ~~synchronize(o2) {~~ |
| atomic { | atomic { |
| ... | ... |
| while(!flagA) {} | flagA = true; |
| flagB = true; | while(!flagB) {} |
| ... ✪ ... | ... ✪ ... |
| } | } |

Fig. 1. A program with benign data races that executes correctly using locking but deadlocks when directly converted to transactions.

by the same lock is now atomic with respect to *all* other critical sections. This broadened scope of atomicity disallows (previously legal) interleavings, and a correct program could *require* one of these disallowed interleavings to make progress. As a result, this semantic change can cause some correct lock-based programs to deadlock.

Figure 1 presents a short (admittedly contrived) program that has this property. When the code fragment in Figure 1 uses locks via `synchronize` blocks (rather than transactions via `atomic` blocks), the two code fragments synchronize on different objects (`o1` versus `o2`), so they are guarded by different locks. In this code, the programmer intends that neither $P_1$ nor $P_2$ can reach the lines marked by ✪ until the other can also reach this line (*i.e.*, effecting a barrier).[3] This program operates as intended because the code executed by $P_1$ and $P_2$ is protected by different locks, so their execution can be interleaved (assuming pre-emptive thread scheduling). Suppose we directly convert these critical sections to transactions (*i.e.*, replace each `synchronize` block with an `atomic` block). Now the transactions executed by $P_1$ and $P_2$ must execute atomically with respect to each other, meaning that one transaction must appear to execute before the other. This restriction allows either $P_1$ to observe $P_2$'s update of `flagA` or $P_2$ to observe $P_1$'s update of `flagB`, *but not both*. As a result, the program will deadlock because one or both of the transactions will be unable to make progress beyond the `while` loop.

The example shows that directly converting locks into transactions may result in deadlock in legal lock-based programs. This situation will persist even if the underlying implementation aborts and restarts these transactions. Schemes that dynamically transform locks to transactions (*e.g.*, transactional lock removal [10]) revert to acquiring the lock after a timeout, avoiding this problem. Although these systems improve the performance of lock-based code, they do not provide programmers with the benefits of a transactional interface.

Our intent is not to exhibit a real program on which the direct conversion is unsafe, but rather to show that it is possible for this conversion to cause deadlock. In practice, such a conversion may almost always be safe. Nonetheless,

[3]The unprotected references to `flagA` and `flagB` give rise to benign data races; protecting these references with a lock does not change the result.

```
  int A = 0, B = 0;
   P1          P2
atomic {
❶ ld A
          st A = 1
  ...
❷ ld A
}
```

```
  int A = 0, B = 0;
    P1          P2
atomic {
  st B = 1
  ...          ❸ ld B
  st B = 2
}
```

Fig. 2. A strongly atomic semantic must provide both *non-interference* and *containment* with respect to non-transactional code. Non-interference (left) prevents ❶ and ❷ from observing different values. Containment (right) prevents ❸ from observing the internal value 1.

any system that translates lock-based critical sections into transactions cannot assume that this translation is always safe. Determining when this direct translation can be safely applied is now an open research question.

## III. Strong versus Weak Atomicity

Transactions should be atomic with respect to each other, but their relationship to non-transactional code is less clear. This ambiguity would at first appear to be merely an implementation detail; however, legal programs may contain unprotected references to shared variables (*i.e.*, outside transactions) without creating malignant data races, so both transactional and non-transactional code can refer to the same data.

To account for these cases, we present two models for reasoning about scope of atomicity. We define *strong atomicity* to be a semantics in which transactions execute atomically with respect to both other transactions *and* non-transactional code. Strong atomicity has two components: it requires both *non-interference* and *containment* from non-transactional code (see Figure 2). In essence, strong atomicity implicitly treats each instruction appearing outside a transaction as its own singleton transaction. We define *weak atomicity* to be a semantics in which transactions are atomic only with respect to other transactions (*i.e.*, their execution may be interleaved with non-transactional code), therefore violating either non-interference or containment (or both).

An atomicity model for a transactional system is analogous to a memory consistency model for a traditional shared memory multiprocessor. A memory consistency model defines the observable orderings of memory operations between threads [1]. A strong memory consistency model, which limits the observable reordering of memory operations, is easiest to reason about for programmers, but it is difficult to implement efficiently. A relaxed memory consistency model, which allows for counter-intuitive reordering of memory operations, is more complex for programmers to reason about because it requires them to explicitly insert memory barriers to enforce ordering. However, weak ordering models are easier to implement efficiently. Similarly, strong atomicity provides a simple and intuitive view of transactional atomicity, which may be more difficult to implement efficiently (especially in software-based transactional memory systems). In contrast, weak atomicity provides a less intuitive model (as transactions may not appear atomic when interleaved with non-transactional code),

```
Boolean flagA = false, flagB = false;

        P1                          P2
...                          atomic {
❶ while(!flagA) {}             ...
❷ flagB = true;                flagA = true;
...                          ❸ while(!flagB) {}
                               ...
                             }
```

Fig. 3. A program that executes correctly under weak atomicity but deadlocks under strong atomicity.

but it may be easier to implement efficiently. Finally, just as a properly synchronized program will execute correctly under a weak memory consistency model, we anticipate that a properly synchronized transactional program will execute correctly under a weak atomicity model. Interestingly, as transactional systems are also shared memory systems, such systems must define *both* a transactional atomicity model and a memory consistency model, as well as any previously unconsidered interactions between the two [4].

## IV. Code Assuming Weak Atomicity Can Break under Strong Atomicity

It would appear reasonable to assume that any program that executes correctly under weak atomicity will also execute correctly under strong atomicity. However, this assumption is not true; some programs that are correct under weak atomicity will deadlock under strong atomicity. A program executing under weak atomicity can interleave non-transactional code arbitrarily with transactional code, and such interleavings may be necessary for the program to make progress. If the system actually provides strong atomicity, these interleavings are not allowed and the program may deadlock as a result.

For example, consider the two concurrently executing threads in Figure 3. The programmer intends that the two threads proceed in a coordinated way through the use of the shared variables `flagA` and `flagB`, effecting a barrier. Under weak atomicity, the program will execute correctly: the two threads' reads and writes can interleave arbitrarily, and the threads proceed as the programmer intended. However, consider what occurs if the program is executing under strong atomicity. The loop labeled ❶ in $P_1$ will terminate only after the transaction in $P_2$ propagates its update of `flagA` when the transaction commits; however, the transaction in $P_2$ can commit only after the update to `flagB` (labeled ❷) executes (because of the loop labeled ❸). The resulting circular dependency causes this program to deadlock under strong atomicity, despite correctly executing under weak atomicity.

The above example illustrates the need for transactional memory systems to specify whether they are strongly atomic or only weakly atomic and then implement that semantics precisely. If a programmer believes that a transactional system is strongly atomic, but it is only weakly atomic, the programmer may write a buggy program due to race conditions between a transaction and non-transactional code. Conversely, if a program is written with the assumption that a transactional system

```
int A = 0, B = 0;

        P1                          P2
atomic { ❹                   atomic (A==1) { ❷
  atomic { ❶                   B = 1;
    A = 1;                   }
  }
  atomic (B==1) { ❸
    ...
  }
}
```

Fig. 4. A program that executes correctly when the boxed `atomic` block is not present but deadlocks when the `atomic` block is introduced to compose the two transactions it contains (❶ and ❸).

is weakly atomic, but it in fact implements strong atomicity, the program may deadlock because it relies on transactions being non-atomic with respect to non-transactional code. As such, neither atomicity model alone can serve as a safe "least common denominator" target model for programmers. However, it may be possible to specify such a target model via a semantics in which there is no guaranteed behavior for the case of simultaneous data accesses by a transaction and non-transactional code.

## V. Transactions are Not Always Composable

One touted advantage of transactions is that they are composable, whereas lock-based synchronization is not. For example, atomically moving an element from one set to another (*i.e.*, the element will be observed by other processors as being in exactly one set) requires either exposing the internal locking of the set data structure or adding a new method for just such an operation. With transactions, a programmer can accomplish this behavior by sequentially composing the delete and insert invocations by wrapping them in a single transaction. This composability property has been identified as an important advantage of transactions [7].

However, not all such sequential compositions preserve program correctness. Just as broadening the scope of atomicity led to problems when converting locks to transactions and executing code for weak atomicity on a strongly atomic system, broadening the scope of atomicity by sequentially composing code within a transaction can introduce deadlocks.

Figure 4 shows a program that will deadlock when two of its transactions are composed into a single transaction. This example uses the conditional transaction notation in which `atomic (`*`condition`*`)` {*`code`*} will wait until *condition* is true before executing the transaction [6]. If we (for now) ignore the boxed `atomic` block (labeled ❹), the code from $P_2$ can be interleaved in between $P_1$'s two transactions. Because of the conditions on the transactions, the transactions must execute in the order: ❶, ❷, ❸. If we now introduce the boxed `atomic` block, $P_1$'s two transactions are now sequentially composed into a single larger transaction. This code now deadlocks, because $P_1$'s transaction can commit only once $P_2$ commits the assignment of `B`. Conversely, $P_2$'s transaction

cannot commit until $P_1$ commits the assignment of `A`. Although the cross-coupling in this example is straightforward, the two inner transactions could occur deeply nested within different code modules (or in even in library code), making identification of such cross-coupling more difficult.

In addition to the effect on the programmer that such non-composability may create, this issue also affects what optimizations are allowed within the transactional memory system. For example, static or dynamic techniques to coalesce two or more smaller transactions into a larger transaction to reduce per-transaction overheads may encounter problems. Similarly, combining transactional and non-transactional code into a larger transaction can also introduce deadlock. For example, even if the `A = 1` assignment in Figure 4 was not within a transaction, the code in Figure 4 would still deadlock. This result has ramifications for both static compiler optimizations (*e.g.*, code motion optimizations that increase the scope of a transaction) and dynamic optimizations (*e.g.*, naively expanding the scope of a transaction to simplify a hardware implementation). To avoid these problems, static schemes may require additional analysis to determine when such a transformation is legal. Dynamic schemes may be able to avoid this problem by detecting a lack of forward progress and falling back to a more exact enforcement scheme (analogous to how transactional lock removal [10] avoids such problems when dynamically transforming locks into transactions).

## VI. Conclusions and Open Questions

The main contribution of this paper is the counter-intuitive observation that programs that execute correctly under one scope of atomicity can break when executing under broader scopes. In particular, broader scopes of atomicity restrict legal interleavings that may be necessary for program correctness. Further work on transactions should consider this observation when proposing any transparent strengthening of atomicity. We have illustrated this situation by showing three ways in which this phenomenon can occur.

First, transactions do not strictly subsume lock-guarded critical sections in the sense that any program that works correctly with locks will work correctly when directly converted to transactions. The stronger guarantees that transactions provide result in different requirements for correct execution: locks enforce atomicity only among segments of code that are guarded by the same lock, whereas transactions enforce atomicity among all concurrent transactions. Hence, a program that depends on non-atomicity between critical sections guarded by different locks may break when converted to transactions.

Second, introducing atomicity between non-transactional and transactional code (strong atomicity) can break a program that correctly executes when non-transactional code can interleave with transactions (weak atomicity). Therefore, a system should specify its atomicity model as part of its transactional semantics. We encourage designers of transactional memory systems to use these terms to explicitly state the semantics of their proposals.

Third, broadening the granularity of atomicity by sequentially composing a transaction with other code (transactional or non-transactions) can also introduce deadlock.

This paper raises several questions. We have given contrived program examples that give rise to the problems we describe, but how often (if ever) do these types of codes arise in practice? Is it possible to build tools that determine—either statically or dynamically—when it is safe to broaden the scope of atomicity (both in the context of converting lock-based critical sections into transactions and composing transactions)? Does expanding the scope of atomicity always preserve partial correctness, *i.e.*, the translated program has the property that it will give a correct answer if it gives any answer? What are the relative benefits and drawbacks of strong atomicity and weak atomicity? Is a single transactional semantics appropriate for all applications and implementations? If not, how many different semantics are necessary? We hope that this work inspires researchers to investigate these and other questions of transactional semantics.

## Acknowledgments

The authors thank Mark Hill, Christos Kozyrakis, Ravi Rajwar, Steve Zdancewic, and Craig Zilles for their helpful comments on drafts of this paper. This work is funded in part by gifts from Intel Corporation and NSF awards 0311199, 0347290, and 0541292.

## References

[1] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, no. 12, pp. 66–76, Dec. 1996.

[2] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded transactional memory," in *Proceedings of the 11th Symposium on High-Performance Computer Architecture*, Feb. 2005.

[3] C. Blundell, E. C. Lewis, and M. M. K. Martin, "Deconstructing transactional semantics: The subtleties of atomicity," in *Proceedings of Workshop on Duplicating, Deconstructing, and Debunking*, June 2005.

[4] D. Grossman, J. Manson, and W. Pugh, "What do high-level memory models mean for transactions?" in *Proceedings of the 2006 Workshop on Memory System Performance and Correctness (MSPC)*, 2006.

[5] L. Hammond *et al.*, "Transactional memory coherence and consistency," in *Proceedings of the 31th Annual International Symposium on Computer Architecture*, June 2004, pp. 102–113.

[6] T. Harris and K. Fraser, "Language support for lightweight transactions," in *Proceedings of the 18th SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA)*, Oct. 2003, pp. 388–402.

[7] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, "Composable memory transactions," in *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, June 2005, pp. 48–60.

[8] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[9] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "LogTM: Log-based transactional memory," in *Proceedings of the 12th Symposium on High-Performance Computer Architecture*, Feb. 2006.

[10] R. Rajwar and J. R. Goodman, "Transactional lock-free execution of lock-based programs," in *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002, pp. 5–17.

[11] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing transactional memory," in *Proceedings of the 32th Annual International Symposium on Computer Architecture*, June 2005.

[12] N. Shavit and D. Touitou, "Software transactional memory," in *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, Aug. 1995, pp. 204–213.