Contents

4.1 Linearizable objects

Concurrent programming using mutual exclusion is considered to be difficult but developing software using Memory Transactions is not necessarily easier. Transactional Memory systems have complex concurrent semantics and are prone to isolation pathologies, such as cascading aborts, which make their run-time behaviour unpredictable. Weak isolation can be identified as the cause of these problems. To avoid these problems Memory Transactions should be strongly isolated and shared state should be encapsulated in linearizable objects. Linearizable objects have intuitive concurrent semantics and are free from isolation pathologies.

Transactional Memory systems weaken transactional isolation for several reasons. Firstly, to make programming easier by minimising the application changes required when implementing atomic sections. Secondly, to improve concurrent performance, by allowing transactions to share values. Thirdly, to allow transactions to be composed by nesting.

The main contribution of this section is the identification of weak transactional isolation as one of the reasons why concurrent programs are so difficult to write. This section focuses on implementing strongly isolated Memory Transactions.

4.1.1 Weak Isolation

Weakly isolated transactions appear to make programming more convenient by allowing active transactions to pass values to each other. However, weakly isolated transactions interact with each other in many different ways which makes their concurrent semantics very complex. Programming a system with complex semantics is much more difficult than programming a system with simple intuitive semantics. In a large program the complex semantics of weak isolation overwhelm the programming convenience of value passing.

Section 4.1.5 identifies weak isolation as the origin of the semantic complexity of transactional systems.

Weak isolation appears to improve the performance of a transactional system by allowing a value to be shared between transactions as soon as it is produced, but this introduces isolation pathologies which make the behaviour of the concurrent system unpredictable. As the number of participating processors increases so does the overhead of the mechanisms required to avoid pathologies. Eventually, this overhead exceeds the benefits of sharing values.

Section 4.1.6 describes how weak isolation causes the isolation pathologies.

A programming system should permit the programmer to compose a complex application from simple components and the act of composition should not add complexity. Transactional Memory systems support nesting so that transactions can be composed. Nested transactions have complex concurrent semantics because they are a form of weak isolation in which value sharing is restricted to the parent-child relationship. To compose complex transactional applications from simpler components it is not necessary to support nested transactions. Commercial database applications can be very complex, yet nested transactions are rarely used.

Section 4.1.7 describes the semantics of nested transactions.

A solution to the problem of weak isolation and its associated pathologies must address the reasons why isolation is typically weakened. Weak isolation permits state to be shared between processors efficiently, minimises the application changes required to support Memory Transactions and facilitates transactional composition.

4.1.2 Strong Isolation

The requirements that motivate the use of weak isolation should be satisfied by the interface to shared state. Memory Transactions should be strongly isolated and shared state should be encapsulated in linearizable objects.

Linearizability is a correctness condition that characterises the concurrent behaviour of an object. Informally, an object is said to be linearizable if all of its fields are private and the execution of each of its methods appear to take place atomically, at a single moment in time, between their invocation and response [Her08].

Section 4.1.3 describes the property of linearizability in detail.

A mutating method of an object can be seen as a transformation from a set of pre-conditions, that are true of the object before the method call, to a set of post-conditions, that are true afterwards. When these conditions are met the object is said to be consistent. A linearizable object can ensure the consistency of the data that it encapsulates.

A method of a linearizable object can be regarded as a Memory Transaction because it is atomic, isolated and can ensure consistency. The execution of a method of a linearizable object forms a strongly isolated Memory Transaction which is free from isolation pathologies and has intuitive concurrent semantics.

A linearizable object satisfies our requirement for a solution to the problems caused by weak isolation because it allows state to be shared between processors efficiently, minimises changes to the calling application and allows transactions to be composed.

In a concurrent system, shared state should be represented to applications exclusively as linearizable objects because they have intuitive concurrent semantics and predictable run-time behaviour.

4.1.3 Linearizability

Linearizability can be viewed as a special case of serialisability in which a transaction is restricted to a single method applied to a single object.

Linearizability is a non-blocking property of objects. An invocation of a method is never required to wait for another pending invocation to complete so the methods of linearizable objects are not prone to the progress pathology of dead-lock.

Linearizability is a local property. The methods of an object can enforce linearizability without reference to any other object or to any global state so it is not necessary to invoke the concept of a global transaction manager to enforce linearizability.

The Linearizability property of an object may be preserved when objects are composed. A system composed of objects is linearizable if and only if every object in the system is linearizable.

The property of linearizability does not permit method calls whose execution does not overlap to be re-ordered so it enforces a sequential order of events affecting an object and preserves the real time order of method calls.

The property of linearizability can be contrasted with that of sequential consistency which, when applied to objects, requires that method calls issued by different processors appear to take place in some global sequential order. Sequential consistency is a property of the method calls of objects in a concurrent system that many programmers expect [Lam97].

Sequential consistency is not a local property so a global view of state is required to ensure sequential consistency. It is a blocking property so an invocation of a method is required to wait for another pending invocation to complete. It is not a composable property so a system composed of multiple sequentially consistent objects is not necessarily sequentially consistent. Sequential consistency permits method calls whose execution does not overlap to be re-ordered so it does not preserve the real time order of method calls.

Linearizability is a stronger condition than sequential consistency. Every linearizable history is sequentially consistent but not vice versa.

4.1.4 Previous work

Herlihy introduced linearizability as a correctness condition [HW90]. Herlihy also provides an accessible introduction to linearizability [Her08]. Linearizability has not previously been considered as a correctness condition for Immutable Data Structures.

4.1.5 The semantics of weak isolation

Isolation levels are a way of describing the behaviour of weakly isolated transactions in terms of the access that a transaction has to the uncommitted state of another transaction. In Database systems the classification of isolation levels is formalised as the ANSI/ISO Isolation Levels [ISO92]. This formalism describes weak isolation by characterising a read access that would not be permitted in a strongly isolated transactional system.

A dirty read is an access to the uncommitted state of another transaction. The transaction from which the variable was read might never commit. A transactional system that permits dirty reads is regarded as having a transaction isolation level of *read uncommitted*. It is difficult to write a concurrent program for a system that permits dirty reads as there can be no happens-before relationship between transactions.

A non-repeatable read is an access to a shared variable that can be modified by another transaction. A variable can appear to have a different value when read for a second time within a single transaction. A transactional system that permits non-repeatable reads is regarded as having an isolation level of *read committed*. It is difficult to write a concurrent program in a system that permits non-repeatable reads as the value of variables can appear to change for reasons outside the immediate logic of the program.

A phantom read is an inconsistent access to shared state. A transactional system that permits phantom reads is regarded as having an isolation level of *repeatable read*. This isolation level is referred to as repeatable because a read access to a single variable will always return the same value within a transaction. However, the reading of multiple variables within a transaction may not, necessarily, present a consistent view of shared state. It is difficult to write a concurrent program in a system that permits phantom reads as the value of the variables accessed by a transaction do not necessarily represent a consistent state.

The ANSI/ISO Isolation Levels formalism has been criticised as being vague, incomplete, inconsistent and not corresponding to the levels implemented by commercial systems [BBG+95]. These criticisms support our assertion that weak isolation does not have intuitive concurrent semantics. If the ANSI committee could not come up with a logical way of classifying the semantics of weak isolation then there is little chance that ordinary programmers will be able to reason about them.

Transactional Memory systems compromise the strict isolation of transactions to make a program easier to write. However, weakly isolated concurrent systems have complex semantics that can make a concurrent program more difficult to write.

4.1.6 Isolation pathologies

Isolation pathologies arise when scheduling is applied to enforce reasonable behaviour on weakly isolated transactions.

A transaction schedule in which a transaction may commit before a transaction that wrote a variable that it has read is called non-recoverable. The transaction schedule is non-recoverable because if the transaction it read from aborts then it too should abort, because the value it read should never have been written. However, once the transaction has already committed it is not possible to abort. A transaction schedule in which a transaction can commit only after all the transactions it has read from have committed is called recoverable. Non-recoverability is an isolation pathology of transactional systems that leads to inconsistent results.

A transaction schedule in which a transaction is permitted to read uncommitted values can suffer from the pathology of cascading aborts. A cascading abort occurs when a transaction reads a value, written by another transaction, that has not yet committed. If the transaction from which the value was read is aborted then the reading transaction must also abort. A transaction schedule in which a transaction can only read committed values avoids the pathology of cascading aborts. Cascading aborts are an isolation pathology that causes unpredictable run-time behaviour.

A transaction schedule in which all transactions appear to execute in isolation is said to be serialisable. The execution is called serialisable because it is equivalent to an execution in which all transactions execute one after the other. A serial transaction schedule in which the order of conflicting operations matches the order in which the transactions commit is said to be strict. Strictly serialisable schedules are recoverable and not prone to the pathology of cascading aborts.

Transactional Memory systems compromise the strict isolation of transactions to obtain concurrent speed-up. However, weak isolation leaves Transactional Memory systems prone to isolation pathologies that make their run-time performance unpredictable.

4.1.7 Nested Transactions

Nesting permits the composition of complex programs from simpler components. Transactional nesting is a form of weak isolation in which values may be shared between transactions if there is a parent-child relationship between them. Transactional nesting has complex semantics and guaranteeing the correctness of execution has high overheads.

A nested transaction is a transaction whose execution is properly contained within the dynamic extent of another transaction. However, transactional nesting is generally taken to mean the nesting of atomic sections so that an outer section shares speculative state with an atomic section contained within it.

Mutual exclusion is not a composable property and this is often cited as an argument to motivate the use of Transactional Memory [HMPJH05]. It is argued that in order for Memory Transactions to be composable a Transactional Memory system should support nesting.

To support nested transactions isolation must be weakened to permit a parent-child relationship between transactions. A parent transaction passes information to its child both explicitly, in the form of shared values, and implicitly, because the parent must exist in order for the child to be created.

Closed nesting has the simplest semantics but its implementation is complex. A parent transaction may start a child transaction but the child must commit before its parent can commit. The speculative state of the child is incorporated into the speculative state of its parent when it commits. If a child transaction aborts it can be restarted, without forcing the parent to abort. Closed nested transactions facilitate the composition of a complex transaction from simpler components and reduce wasted work. Maintaining the parent-child relationship between closed transactions has a high overhead because if the parent transaction is aborted then its children must also be aborted. However, the child transaction may have already committed so to ensure that the transaction schedule of a closed nested transaction is recoverable, all of the state produced by the child transaction must be contained within the parent.

Open nested transactions have complex semantics but the implementation can be simpler than that of closed nesting. When an open nested transaction commits, its changes become visible to all other transactions in the system. Concurrently executing transactions observe changes to shared state immediately [NMAT$^+$07]. It is not necessary to maintain multiple versions of shared state so implementation is simplified. Open nested transactions are composable, although great care must be taken to avoid pathologies because exposing changes of shared state leads to the phenomenon of non-repeatable reads and the isolation pathology of cascading aborts.

There is a precise definition of the semantics of both open and closed nested transactions [MH06]. However, other forms of nesting, of which there are many, do not have precise definitions.

Flattened nesting has complex semantics but simple implementation. Flattening is similar to closed nesting except that if a child transaction aborts the parent transaction must also abort. Flattened transactions are effectively nested subroutines, all that is required to implement them is a stack of pointers indicating the calling point in the parent, so implementation is straightforward. Flattened nested transactions are not composable so their utility is questionable [HLR10].

Many database systems support some form of nested transactions. However, the use of nested transactions in the database programming environment is not widespread [GR93]. Nested database transactions can reduce the overhead of transactional execution. Nesting facilitates the check pointing of transactions to reduce the amount of work wasted when a transaction aborts [HK08]. Nesting also permits short running transactions to abort without affecting their long running parents. However, the overhead of maintaining the parent-child relationship between transactions is significant. In the database environment the overheads of transaction management, relative to the work done by a transaction accessing disk, are very low. Even so, support for nested database transactions has a significant performance overhead [GR93].

There is wide disagreement on the semantics of transactional nesting and on the desirability of different forms of nesting [AFS08] [HLR10]. However, the debate about nesting is really a debate about weakening transactional isolation. The complexity of the issues surrounding transactional nesting obfuscates the undesirability of weak isolation. Nested transactions, like other forms of weak isolation, have complex semantics and their run-time execution is prone to isolation pathologies.

Transactional Memory systems permit composition through nesting which makes a program easier to write. However, nesting is a form of weak isolation with complex semantics that makes a concurrent program more difficult to write.

Bibliography

[AFS08] Kunal Agrawal, Jeremy T. Fineman, and Jim Sukha. Nested parallelism in transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 163–174, New York, NY, USA, 2008. ACM.

[BBG$^+$95] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. *SIGMOD Rec.*, 24:1–10, May 1995.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[Her08] Maurice Herlihy. Linearizability. In *Encyclopedia of Algorithms.* Springer, 2008.

[HK08] Maurice Herlihy and Eric Koskinen. Checkpoints and continuations instead of nested transactions. In *TRANSACT '08: 3rd Workshop on Transactional Computing*, February 2008.

[HLR10] Tim Harris, James R. Larus, and Ravi Rajwar. *Transactional Memory, 2nd edition.* Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[HMPJH05] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[ISO92] ISO. *SQL Specification.* ISO, 1992.

[Lam97] Leslie Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Comput.*, 46(7):779–782, 1997.

[MH06] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: model and architecture sketches. *Sci. Comput. Program.*, 63(2):186–201, 2006.

[NMAT$^+$07] Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *PPoPP '07: Proc. 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 68–78, mar 2007.