Contents

7.1 The flow of time

The concurrency problem makes it difficult to write a program that executes efficiently on a Chip Multi-Processor. This problem arises because information cannot pass instantly between the processors so each has a different view of the flow of time. Multi-Processor Systems that treat the flow of time as a global phenomenon are difficult to program, prone to pathologies and do not scale well, whereas those that treat time as a local phenomenon have intuitive concurrent semantics, freedom from progress pathologies and few barriers to scalability.

Our commonsense notion of time is that it flows and that some changes are simultaneous while others form an ordered sequence and it is a global phenomenon experienced everywhere in the same way. In this section we consider whether it is necessary or desirable to enforce this commonsense notion of the flow of time on a concurrent system.

7.1.1 The notion of the flow of time as a global phenomenon

Concurrent systems attempt to impose a global view of the flow of time by enforcing a global ordering on state transitions and by preventing simultaneity.

Speculation about global state transitions

As time passes, events that were once in the future occur in the present moment and are then relegated to the past. The present moment is the temporal boundary between the uncertain future and the fixed past. This notion is referred to as the passage of time.

In a uni-processor system the present moment in time is represented by the state of memory. However, there is no global present moment in a Multi-Processor System because information cannot pass instantly between processors.

In a Transactional Memory system speculation centres on the future state of shared memory. The speculation is that a putative future state created in isolation does not conflict with any other putative state created by another processor. Transactional Memory systems weaken isolation to facilitate value sharing and transactional composition and this blurs the boundary between speculative and committed state making it difficult to impose a global present moment.

The difficulty of imposing a global temporal boundary between speculative and shared state is the source of the complex semantics of concurrent systems.

Preventing simultaneity

Simultaneous state transformations must appear simultaneous to all observers. This notion is referred to as absolute simultaneity.

In a uni-processor system the lack of coherence between components in the memory hierarchy goes unnoticed by the application. However, in a Multi-Processor System it is difficult to guarantee that state transformations, that may appear simultaneous to an application executing on some processors, appear simultaneous to all processors.

Mutual exclusion prevents processors from simultaneously accessing the same memory location by blocking the execution of some processors, but this obstructs progress and is the source of progress pathologies.

A Transactional Memory system prevents processors from simultaneously accessing the same memory location by ensuring that only one of the conflicting transactions succeeds. To achieve this the system must maintain both the speculative and a committed version of the same memory location which increases the effective memory bandwidth of the application.

The difficulty of imposing absolute simultaneity is a source of both the progress pathologies and the high memory bandwidth requirement of concurrent applications.

Enforcing a global ordering on state transitions

Events form a uni-directional sequence in time which is a consequence of the second law of thermodynamics. The arrow of time denotes an asymmetry between the future and the past that imposes a global ordering on state transformations.

In a uni-processor system successive states of memory form a uni-directional sequence, so execution can be seen as an ordered sequence of state transitions. However, there is no global ordering of state transformations in a Multi-Processor System because information cannot pass instantly between its components.

A Transactional Memory system implements a concurrency control protocol to impose a global order on state transitions so that their effect on shared state is equivalent to a serial execution. A centralised transaction manager is required to impose a global ordering and this restricts scalability.

The difficulty of imposing a global ordering on state transitions is the source of the scaling restrictions on concurrent systems.

Memory Transactions are not like database transactions

Few people in the computer architecture community believe that strong models of memory consistency are scalable. Modern Chip Multi-Processors impose neither the concept of a global present moment nor the concept of absolute simultaneity, except when processing instructions with associated memory barriers. Weakly consistent memory models, such as total store ordering, remove the need to impose a global ordering of events [AG95]. However, most database systems implement strong consistency models and many people in the database community believe that a global ordering of events is essential for programmers to write concurrent programs. Imposing a global view of the flow of time is the primary purpose of the transaction manager in a database system [GR93].

Transactional Memory has inherited the idea that a framework for speculative execution must impose a commonsense notion of the flow of time. The idea is so pervasive that few have questioned it. The conclusion of this thesis is that it is neither necessary nor desirable to enforce a global view of the flow of time on a concurrent system.

7.1.2 The notion of the flow of time as a local phenomenon

This thesis proposes that a Multi-Processor System should treat the flow of time as a local phenomenon because information cannot pass instantly from one place to another. A local notion of time does not invoke the concept of a global present moment and only requires that simultaneity is relative and that events and observations are only ordered in relation to each other.

Davies provides an accessible introduction to the distinction between a local and a global concept of the flow of time [Dav02].

Speculation about events and observations

If we accept that the passage of time is a local phenomenon affecting events and observations rather than states then there is no global present moment and speculation can be restricted to events and their observation.

This thesis describes a concurrent system in which there is no concept of a global present moment separating the past from the future. When this concept is removed speculation can centre on events and their observation, rather than about states. The speculation is that an event does not change an observation that has already been made. When speculation is restricted to events it is not necessary to impose a global temporal boundary between speculative and shared state.

The access functions of a Transactional Data Structure execute speculatively and the speculation is that the execution of the access function does not affect any value that has already been returned to the application. The concurrent semantics of the access functions of a Transactional Data Structure are intuitive because the functions acting on the Transactional Data Structure are strongly isolated from each other and their effects on the structure are strictly serialisable.

By speculating about events and observations affecting a single object concurrent systems with intuitive concurrent semantics can be constructed.

Permitting simultaneity

If we accept that simultaneity is relative, and that events that occur at the same moment in time when observed from one frame of reference may occur at different moments if viewed from another, then there is no requirement to either restrict simultaneity or to enforce it.

This thesis describes a concurrent system in which simultaneity is relative and this differs from a system that restricts or enforces simultaneous state transitions. When simultaneity is relative it is neither necessary to ensure that a mutation is observed simultaneously by all processors nor prevent multiple processors from simultaneously accessing the same object.

The access functions of a Transactional Data Structure permit simultaneous access to data because that data is immutable. Immutable data is timeless and it can be simultaneously accessed by multiple processors safely. Immutable data is written just once so speculation does not increase the memory bandwidth of the application. The access functions of a Transactional Data Structure do not restrict simultaneous events by blocking the execution of another processor so they do not restrict progress and are not prone to progress pathologies.

By accepting that simultaneity is relative it is possible to construct a concurrent program that is free from progress pathologies and that does not have an increased memory bandwidth requirement when executing on multiple processors.

Speculation about a local order of events

If we accept that the arrow of time is a local phenomenon referring to the relationship between an event and its observation then there is no concept of a global ordering of events so order can be enforced locally.

This thesis describes a concurrent system which imposes a local ordering on the events affecting a particular object and this differs from the imposition of a global ordering on state transitions. A locally serialisable ordering of events affecting a particular object can be ensured by making that object linearizable.

The access functions of a Transactional Data Structure enforce an ordering on the events that affect the data structure. A concurrent system that does not impose a global ordering of events lends itself to a distributed implementation and permits scalability.

By implementing distributed concurrency control it is possible to construct a scalable concurrent system.

7.2 Making scalable concurrent programs easier to write

This thesis proposes that pure functions, immutable data and Memory Transactions can be combined to create a programming model that makes scalable concurrent programs easier to write. It is grounded on the observation that a scalable concurrent program must be able to interact with an external entity and it must guarantee progress. It also makes the observation that a scalable concurrent system must not rely on coherent caches, strong memory models or centralised transaction management.

Application programmers expect their programs to execute inevitably. However, a scalable concurrent system must support speculative execution so speculation must be restricted to the interface with shared state. There are many ways to present a speculative interface to shared state but familiar ADTs are the interface that programmers expect, so the design of the shared interface follows from programmer expectations.

The memory bandwidth of a scalable concurrent application must be independent of the number of processors participating in its execution. Engineering barriers, such as the difficulty of scaling coherent caches, restrict the scalability of systems that support mutable shared state, so in a scalable concurrent system shared state must be immutable. There are many ways to implement immutable data, but Immutable Data Structures based on the Canonical Binary Tree are the simplest.

Distributed transaction management is scalable, whereas centralised transaction management is not. Programmers must abandon the concept of globally serialisable state transitions because distributed concurrency control can only ensure the correctness of concurrent accesses to individual objects. Linearizability is a correctness condition for objects which provides behaviour that programmers expect. Concurrent applications must make strong progress guarantees, because contention management is not scalable. The requirement that transaction management must be distributed dictates both the nature of the interface to shared state and the progress guarantees offered by the application.

Figure 7.1 illustrates how observations about scalable concurrent systems led to the development of Transactional Data Structures.

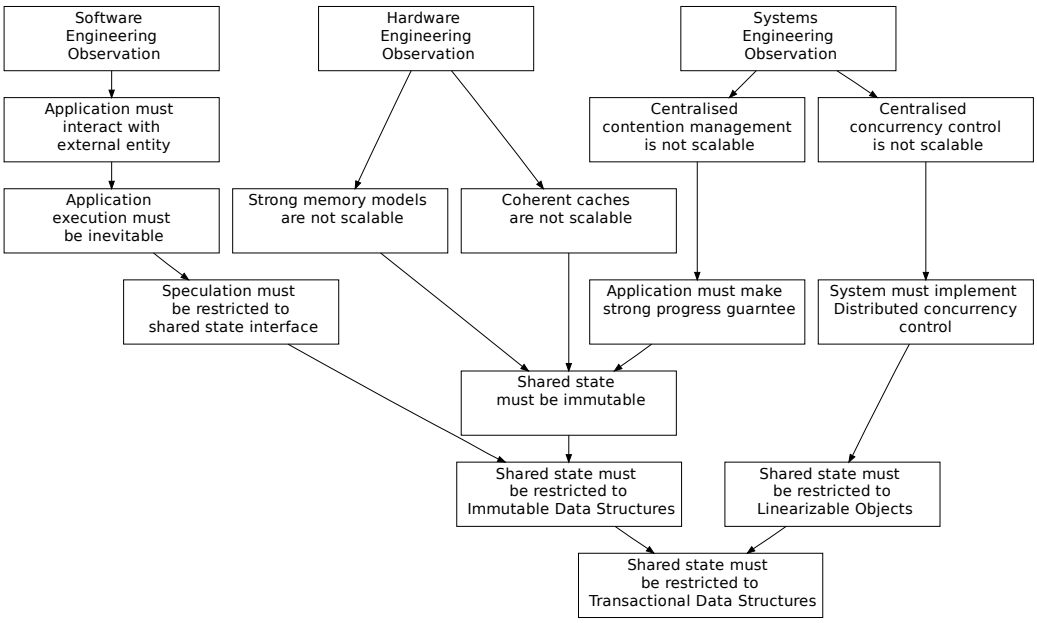

Figure 7.1: **Observations about scalable concurrent systems** led to the development of Transactional Data Structures.

These design choices are co-dependent and may be difficult to implement independently. For example, it is difficult to envisage a concurrent system that implements distributed transaction management efficiently without requiring that shared state is immutable. Both the strong progress guarantees, required by a concurrent system that does not implement centralised contention management, and the appearance of atomic transformations, required by distributed concurrency control, are very difficult to implement efficiently unless shared state is immutable.

7.3 Future work

Transactional Memory designs are based on a common set of priorities, such as the support for atomic sections, and approaches, such as centralised transaction management. We identified seven design decisions that are dependent on these priorities and examined alternative approaches.

How to interact with entities outside the concurrent system?

A useful concurrent application should be able to interact with external entities. This thesis explores the idea that the interface to shared state should be presented to the application as an ADT so that an application program can execute inevitably. We have found that developing concurrent applications using our interface is easier than using atomic sections. However, we did not have the opportunity to evaluate whether our proposal facilitates external communication in concurrent systems.

Heterogeneous systems are constructed from communicating components so a programming model for them must support interaction. Message passing is the predominant concurrent programming model for embedded systems and in this model processors do not share state.

We suggest that the use of Transactional Data Structures as a state sharing mechanism for heterogeneous embedded systems should be investigated.

This thesis originated as an investigation into the use of Transactional Memory as a state sharing mechanism for embedded Chip Multi-Processors without coherent caches. The original proposal was that shared state could be maintained in tightly coupled memory and that distributed concurrency control could be used to ensure its correctness.

We now suggest that Transactional Data Structures can be used to maintain shared state in Chip Multi-Processors without coherent caches and that distributed concurrency control can be used as an alternative to a cache coherence protocol.

We suggest that Transactional Memory systems should support database type transactions in memory rather than atomic sections. However, the choice of programming interface is fundamental to Transactional Memory design, so we are not optimistic that there is an evolutionary development path from existing Transactional Memory systems to concurrent systems that permit an application

to interact freely with external entities.

How to maintain shared state and support speculative execution?

A scalable concurrent system should maintain both shared state and isolated speculative state without increasing the memory bandwidth requirement of the application. This thesis explores the idea that both shared state and isolated speculative state can be maintained in an Immutable Data Structure and that doing so does not increase the memory bandwidth requirement of the application because immutable values are written only once. There are many ways to implement Immutable Data Structures and many optimisations that can be applied to improve their performance, but we were only able to explore a single approach in any depth.

The Immutable Data Structure infrastructure developed to support the evaluation is both original and interesting. The purpose of the infrastructure is to support Concurrent Memory Transactions without requiring a centralised transaction manager. However, a system that supports Immutable Data Structures in an imperative programming environment can have uses outside the area of concurrent programming. For example, data structures that permit backtracking have many useful applications in combinatorics.

We suggest that the use of Immutable Data Structures in an imperative programming environment is a fruitful area of research.

The Canonical Binary Tree permits a separation of the concerns of the data structure from those of the ADT so that the performance of the data structure can be optimised independent of the ADT that it implements. The performance of the access functions of Immutable Data Structures can be improved by using shallower trees with more children per node. The techniques used to develop the Canonical Binary Tree could be applied to trees with fast merge functions, such as binomial heaps, to improve the performance of the meld function.

Section **??** describes how the Canonical Binary Tree may be optimised by both reducing the size of the node and reducing the number of nodes accessed by common operations. We have not had opportunity to implement these optimisations.

We suggest that the performance of the Canonical Binary Tree implementation can easily be improved.

It is not necessary to enforce a cache coherency protocol on immutable data.

However, current Chip Multi-Processor hardware ensures that the entire address space is cache coherent. When an immutable value is written a cache invalidate message is sent to all processors unnecessarily. These messages increase the effective memory bandwidth of the application. Hardware designed specifically to realise the benefits of immutability might partition memory into non-coherent regions suitable for maintaining local and immutable data and cache coherent regions suitable for maintaining the roots of Immutable Data Structures.

We suggest that hardware designed specifically to realise the benefits of immutability can improve the performance of concurrent systems.

Immutable Data Structures consume the memory address space very quickly. The memory occupied by a leaf of an Immutable Data Structure cannot be returned immediately when it is deleted by the application. Instead, it can be returned when it becomes unreachable. The vertices that cannot be reached from the root are potential candidates for return but some of these vertices cannot be returned because they are reachable by tardy functions.

The task of determining whether vertices in an Immutable Data Structure can be reached and the process of returning them, while the structure is being accessed, is similar to the task of garbage collection. Jones describes the process of garbage collecting managed memories [JL96]. In our implementation the memory occupied by the vertices of an Immutable Data Structure is returned by periodically compacting the structure. The return of unreachable values in an Immutable Data Structure is discussed on our website [Jar11].

We suggest that the management of immutable memory needs to be improved before Immutable Data Structure can be used in production software.

We suggest that the use of immutable data in existing Transactional Memory systems should be investigated. However, the choice of the mechanism for maintaining shared and speculative state is fundamental to a Transactional Memory design, so we are not optimistic that there is an evolutionary development path from existing Transactional Memory systems to concurrent systems that support speculation without increasing the effective memory bandwidth of the application.

How to provide access to shared state with intuitive concurrent semantics?

Shared state should present an intuitive interface to an application to make concurrent programming easier. This thesis explores the idea that shared state can be encapsulated by linearizable objects and that Immutable Data Structures can be composed by Entanglement. We evaluated this idea by implementing a concurrent algorithm to determine the minimum spanning tree of a graph. We concluded that the intuitive concurrent semantics of linearizable objects and Immutable Data Structures have the potential to make the process of developing concurrent applications easier. We were able to investigate some of the properties of confluently persistent data structures during our evaluation of a minimum spanning tree algorithm, but we did not have opportunity the investigate partially persistent data structures.

We suggest that the properties of partially persistent Transactional Data Structures should be investigated.

How to implement concurrency control to guarantee correct concurrent execution?

A scalable concurrent system should implement distributed concurrency control. This thesis explored the idea that the Time Stamp Ordering concurrency control protocol can ensure the serialisability of functions acting simultaneously on an Immutable Data Structure. We were only able to investigate one of the many ways of imposing a distributed concurrency control protocol on Memory Transactions. We found that implementing concurrency control locally by serialising simultaneous accesses to a single object is much easier than implementing centralised concurrency control.

We suggest that the use of distributed concurrency control in existing Transactional Memory systems should be investigated. However, the choice of the concurrency control mechanism is fundamental to a Transactional Memory design, so we are not optimistic that there is an evolutionary development path from existing Transactional Memory systems to concurrent systems that implement scalable distributed concurrency control.

How to implement contention management to eliminate progress pathologies?

Strong progress guarantees alleviate the need for centralised contention management. This thesis explored the idea that functions acting on an Immutable Data Structure can guarantee lock-free progress. We evaluated an implementation of a non-blocking Producer Consumer Queue and we found that progress pathologies were eliminated and that centralised contention management was unnecessary.

A scalable concurrent application must make a strong progress guarantee because centralised contention management is not scalable, but non-blocking algorithms that rely on mutable shared data are difficult to write. We found that the development of non-blocking algorithms is made easier by requiring that shared data is immutable.

We suggest that non-blocking algorithms that focus on immutable data should be investigated.

How to marshall work and schedule concurrent execution?

A scalable concurrent system has few scheduling requirements. This thesis explores the idea that the responsibility of the scheduler should be restricted exclusively to that of load-balancing concurrent work and that a scheduler intended for a parallel workload can be used to schedule Memory Transactions. During our evaluation of a Producer Consumer Queue we used a parallel scheduler and found that it was both easy to use and effective.

A system which makes the distinction between a parallel workload, in which conflicts are statically known not to occur, and a concurrent workload, in which conflicts are detected dynamically, is not generally useful as both types of workload occur in a typical application. A concurrent programming solution should be capable of scheduling an application containing both parallel and concurrent work.

We suggest that schedulers intended for parallel work should be used to permit workload flexibility in concurrent systems.

Functional programming permits concurrent execution because it supports both parallelism and speculation. However, the problem of dynamically load-balancing parallel execution remains to be solved. Immutable Data Structures in the form of purely functional data structures are widely used in the expression of

a functional program but they could also be used to maintain the abstract syntax tree of a functional program during its execution.

We suggest that the use of Immutable Data Structures as a potential solution to the dynamic load-balancing problem in functional programming should be investigated.

How to integrate a concurrent programming solution into the software development cycle?

A concurrent programming solution should make it economically viable to develop concurrent applications. This thesis explores the idea that concurrent applications can be developed using conventional imperative languages, compilers and tools so as to minimise the impact on existing software and methodologies. We found that, by focusing on the shared state interface and developing concurrent applications, rather than transactional systems, we were able to restrict the locality of changes to those routines that benefit most from concurrent execution.

We suggest that a C++ STL compatible user interface for Immutable Data Structures should be developed so that programmers can easily integrate these structures into existing concurrent applications.

7.4 Summary

"The overarching goal [of parallel programming research] should be to make it easy to write programs that execute efficiently on highly parallel computing systems" [ABC+06].

We observed that a concurrent program must execute inevitably in order to communicate, so speculative execution must be restricted to the interface with shared state. Neither coherent caches nor strong models of memory consistency scale, so shared state must be immutable. Centralised concurrency control restricts scalability, so a scalable concurrent program must implement distributed concurrency control, and centralised contention management restricts scalability, so a scalable concurrent program must guarantee progress.

These observations indicate that scalable concurrent programs are confined to sharing only immutable data and that scalable concurrent systems are bound to ensure the correctness of concurrent execution on a per object basis.

We conjectured that a concurrent program that shares only immutable data and which executes in a system which implements distributed concurrency control will be both easier to write and more scalable than an equivalent program that uses mutual exclusion.

We proposed Transactional Data Structures which are an interface to shared state that permit strongly isolated speculation while allowing programs to execute inevitably. Transactional Data Structures do not rely on coherent caches or strong memory consistency models, they are compatible with existing software and software development processes, they require only localised changes to performance critical regions of existing programs and they facilitate the sharing of immutable data while ensuring correct concurrent execution and guaranteeing progress.

We evaluated our proposal and concluded that the use of Transactional Data Structures facilitates both the development of scalable check pointing algorithms and the construction of simple non-blocking algorithms.

Further research is required before we can determine whether Transactional Data Structures will make it easy to write programs that execute efficiently on highly parallel computing systems, but the work we have done so far seems to indicate that they will.

Bibliography

[ABC+06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.

[AG95] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29:66–76, 1995.

[Dav02] Paul Davies. That mysterious flow. *Scientific American*, pages 40–47, September 2002.

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

[Jar11] Kim Jarvis. Transactional Data Structures. http://transactionalmemory.com, June 2011.

[JL96] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.