# TRANSACTIONAL DATA STRUCTURES

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2011

By
Kimberley Jarvis
Computer Science

# Contents

Word count 70552

# List of Tables

# List of Figures

# Abstract

Concurrent programming is difficult and the effort is rarely rewarded by faster execution. The concurrency problem arises because information cannot pass instantly between processors resulting in temporal uncertainty.

This thesis explores the idea that immutable data and distributed concurrency control can be combined to allow scalable concurrent execution and make concurrent programming easier. A concurrent system that does not impose a global ordering on events lends itself to a scalable distributed implementation. A concurrent programming environment in which the ordering of events affecting an object is enforced locally has intuitive concurrent semantics.

This thesis introduces Transactional Data Structures which are data structures that permit access to past versions, although not all accesses succeed. These data structures form the basis of a concurrent programming solution that supports database type transactions in memory. Transactional Data Structures permit non-blocking concurrent access to familiar abstract data types such as deques, maps, vectors and priority queues. Using these data structures a programmer can write a concurrent program in C without having to reason about locks.

The solution is evaluated by comparing the performance of a concurrent algorithm to calculate the minimum spanning tree of a graph with that of a similar algorithm which uses Transactional Memory and by comparing a non-blocking Producer Consumer Queue with its blocking counterpart.

Kimberley Jarvis
Transactional Data Structures
Doctor of Philosophy
The University of Manchester
11 November 2011

# Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

# Copyright

# Acknowledgements

I have been privileged to work with my supervisor Chris Kirkham and my advisor Ian Watson. They have given me an enormous amount of valuable advice and guided me in the preparation of this thesis.

I owe a great debt of gratitude to my colleagues from the Advanced Processor Technologies group at the University of Manchester where I have had the pleasure of working closely with Mohammad Ansari, Behram Kahn, Christos Kotselidis, Mikel Luján, Ian Rogers and Jeremy Singer.

I am grateful to my father James Jarvis for improving the readability of this thesis.

# Chapter 1

# Introduction

## 1.1 Speculative execution

Programs that deliver scalable speed-up from parallel execution on Chip Multi-Processors are difficult to develop. As the number of processing cores in a Chip Multi-Processor increases so does the potential speed-up from parallel execution but there are few programs that actually achieve scalable speed-up when executing on a Chip Multi-Processor.

This thesis examines the problem of getting the processors of a Chip Multi-Processor to work together on a single program and complete the program in less time than it would take a single processor working alone. The program can be divided into tasks which are simultaneously executed by the processors and these tasks may or may not be interdependent. This thesis focuses on the case where task dependencies are not known until the program is executed.

### 1.1.1 Mutual exclusion

The most commonly used mechanism to support concurrent execution is mutual exclusion. Mutual exclusion does not permit tasks with possible dependencies to execute in overlapping periods of time. On a Chip Multi-Processor mutual exclusion can be used to permit those sections of a program in which there are known to be no conflicting operations to be executed in parallel and ensure that the critical sections of the program, that may have dependencies, execute serially.

The use of mutual exclusion limits the scalability of a concurrent program and makes it susceptible to progress pathologies such as deadlock.

Section 1.1.5 describes why concurrent programming using mutual exclusion is difficult.

This thesis is primarily concerned with the problem of developing scalable concurrent programs for Chip Multi-Processors and it focuses on the case where task dependencies are not known until the program is executed.

Section 1.1.6 describes the scope of the problem addressed by this thesis.

### 1.1.2 Speculative execution

Speculative execution is a programming technique that can be used as an alternative to mutual exclusion. It facilitates scalable concurrent execution by permitting the simultaneous execution of tasks that may be interdependent. Speculative execution permits tasks to be executed optimistically. A concurrent program can safely speculate that a task is not affected by tasks running on other processors, provided it has a mechanism to re-execute the task should that speculation prove incorrect.

To discover the dependencies between tasks, information must pass between the processors performing them, while the program executes. Processors cannot pass information to each other instantaneously, so each task has a slightly delayed view of the progress that tasks on other processors are making. This delay necessitates speculative execution because if a processor were to wait for a task, on which it is possibly dependent, to complete then there would be little benefit from executing on multiple processors.

This thesis develops programming techniques that support speculative execution on a Chip Multi-Processor.

Section 1.1.3 outlines the techniques presented in this thesis.

This thesis describes software that enables the use of these techniques in an application.

Section 1.1.4 describes the software we develop.

### 1.1.3 Techniques to support speculative execution

This thesis develops programming techniques that allow applications to access shared state speculatively. These techniques include mechanisms for maintaining shared state and mechanisms for detecting conflicting accesses during program execution.

This thesis presents a number of data structure representations and algorithms and it describes features which make them suitable for concurrent execution. These features include the use of a sentinel to distinguish between a non-existent and an empty data structure and the use of additional annotations in vertices so that traversals do not need to examine all of the children of a vertex when determining a path through the structure.

A technique for composing data structures which can be used to implement a low-overhead checkpointing and backtracking mechanism is presented. This thesis explores and evaluates the use of the technique by developing a concurrent graph algorithm.

A technique for detecting conflicting concurrent accesses to a data structure is presented. Conflicts are detected by mapping relative positions within the data structure to the variables considered by a concurrency control protocol.

This thesis explores the use of distributed concurrency control to ensure the correctness of concurrent accesses to a data structure. A technique for incorporating a distributed transaction manager into the access functions of a data structure is presented.

A technique for developing non-blocking algorithms that permits the non-blocking execution of data structure access functions is presented. It is implemented by a non-blocking scheduling routine. This thesis explores and evaluates the use of the technique by developing a non-blocking queue.

### 1.1.4 Infrastructure to support speculative execution

We develop software to demonstrate the programming techniques described in this thesis. This software supports our experiments and allows us to evaluate these techniques. It takes the form of a C++ header library which implements the data structures and a set of routines that implement the mechanisms which permit speculative execution.

A concurrent application that makes use of our techniques accesses shared state through the functions of a data structure. These data structures are implemented by a C++ header library. This library implements common Abstract Data Types (ADTs) in a way that permits the speculative execution of their functions. The library implements a mechanism for maintaining speculative state within a data structure and a mechanism for discovering conflicting accesses at execution time.

A concurrent application that makes use of our techniques is divided into tasks that may be executed in parallel on a Chip Multi-Processor. The software includes routines to distribute and schedule concurrent tasks. These routines are written in C++ and implemented using Intel's Threading Building Block product [Int09].

### 1.1.5 The problem with locks

The correctness of the concurrent execution of critical sections can be ensured by locking. In the simplest scenario, a processor acquires the exclusive ownership of a lock before executing the code within a critical section and relinquishes ownership once the execution of the program code within that critical section is complete. The lock is said to protect the critical section.

Mutual exclusion ensures the serial execution of critical sections regardless of how often dependencies between tasks actually arise. If there is the slightest possibility of a dependency between tasks then they must always be executed serially. As the number of processes is increased, the execution time of the program code within the critical sections dominates and the benefit of parallel execution is diminished. This effect is a consequence of Amdahl's law. [Amd67] [HM08].

The relationship between a lock and the critical section that it protects is abstract. This relationship is not necessarily reflected in the program code or its associated documentation. For example, the code in an existing critical section may not contain information about a newly developed critical section which is protected by the same lock.

Blocking occurs when a delay in the execution of a section of program code delays the execution of other code sections. Mutual exclusion ensures the correctness of the execution of a code section by ensuring that only one processor executes it at any given moment. The progress of processors that attempt to access the code section simultaneously is blocked. A program that blocks is prone to the progress pathology of deadlock which prevents it from completing. For example, a concurrent program in which two critical sections acquire the same locks but in a different order may deadlock.

Sections of program code are said to be composable if they can be combined without examining or altering their implementation. Critical sections are not composable because they have the potential to block the progress of processors that attempt to execute them. Consequently, it is not possible to encapsulate the

concerns of mutual exclusion. For example, the order of lock acquisition in an existing critical section must be examined when developing a new critical section protected by the same locks.

Critical sections that are protected by a common lock have shared performance considerations. For example, a low-latency task may be blocked by a long running task protected by the same lock.

Critical sections that are protected by a common lock must be tested to the same level of integrity regardless of the importance of their functionality. For example, a trivial task must be tested to the same level of integrity as an important task protected by the same lock.

### 1.1.6 The concurrency problem

This thesis addresses the problem of developing scalable concurrent programs for execution on Chip Multi-Processors. We focus exclusively on the execution of tasks from a single program on Chip Multi-Processor hardware that supports parallel execution. We are particularly interested in the case where the dependencies between tasks are unknown until their execution is complete. We regard the concurrency problem as the problem of obtaining speed-up from the parallel execution of tasks from a single program on a Chip Multi-Processor when task dependencies are unknown until their execution is complete.

A Chip Multi-Processor supports parallel execution, which is the simultaneous execution of tasks on different hardware processors. A system may support concurrent execution, which is the execution of tasks that have the *potential* to execute simultaneously. In this thesis we are concerned with internal concurrency which is the potential to execute related tasks, that may interact through memory and which are part of a single program, simultaneously. We do not consider external concurrency which occurs when a program, such as an operating system, needs to perform several possibly unrelated tasks at the same time.

Many applications have the potential to benefit from parallel execution on Chip Multi-Processors but are unable to realise this benefit because their execution is dominated by frequent accesses to shared data. These applications include low-latency trading, travel reservations, on-line inventory, on-line gaming, payment verification and graph traversal algorithms. For these applications the concurrency problem has two related components. Firstly, the development costs associated with ensuring the correctness of concurrent accesses to shared

data using mutual exclusion exceed the commercial benefits of parallel execution on a Chip Multi-Processor. Secondly, the scaling restriction imposed by using mutual exclusion precludes significant speed-up from parallel execution on a Chip Multi-Processor.

This thesis focuses on programs written in C++ because applications that may benefit from parallel execution on a Chip Multi-Processor are frequently written in this language. However, the techniques we describe are not restricted to a particular programming language.

Only small regions of many programs may benefit from parallel execution. We focus on techniques which can be applied locally, without disrupting those regions of an application that do not benefit. We seek solutions that can be applied to existing programs. However, the techniques we describe can also be used to develop new programs. The programs we develop to support our evaluation are entirely new.

## 1.2 Memory Transactions

Transactional Memory is a programming technique that promises to make concurrent programming easier and concurrent programs more scalable. However, Transactional Memory systems rarely deliver on these promises. The design choices made during the development of Transactional Memory systems are motivated by a common set of priorities. These priorities need to be re-evaluated and different design choices considered.

The main contribution of this section is the identification of priorities that motivate Transactional Memory designs. This section focuses on whether Transactional Memory systems really deliver on their promise of making scalable concurrent programs easier to write.

### 1.2.1 Transactional Memory systems

Transactional Memory is a technique to support speculative execution that can be used as an alternative to mutual exclusion. It facilitates scalable concurrent execution by allowing the simultaneous execution of tasks that may be interdependent.

Section 1.2.4 introduces Transactional Memory.

Software Transactional Memory systems provide a software framework for programmers to construct concurrent programs that can be executed speculatively. However, the overheads of supporting speculative execution entirely in software often exceed the benefits of concurrent execution.

Section 1.2.5 discusses the claim that Software Transactional Memory makes concurrent programming easier.

Hardware Transactional Memory systems support concurrent execution by providing a hardware environment in which concurrent programs can be executed speculatively. The engineering challenges that must be overcome by Hardware Transactional Memory are significant and the commercial barriers to adoption are high.

Section 1.2.6 discusses the claim that Hardware Transactional Memory makes concurrent programming easier.

Transactional Memory can make programming easier by freeing the programmer from having to reason about locks, but concurrent programming using Memory Transactions is not necessarily easier than concurrent programming using

mutual exclusion.

Section 1.2.7 discusses the claim that Transactional Memory makes concurrent programming easier.

Research carried out in both the private and public sectors has yet to produce convincing evidence that Transactional Memory systems are making progress towards delivering on their promise of scalability, because the overheads of supporting concurrent execution exceed the benefits of concurrent execution. They also fail to deliver on their promise of improved programmer productivity, because concurrent programming using Memory Transactions is no easier than concurrent programming using mutual exclusion.

### 1.2.2 Design priorities

Transactional Memory research is founded on the premise that speculative execution is necessary to support scalable concurrent execution on Chip Multi-Processors and it has the goal of making concurrent programming easier. This thesis does not doubt this premise nor question this laudable goal, but it does question the priorities that motivate the design of Transactional Memory systems.

Transactional Memory proposals prioritise some aspects of system design at the expense of others:

- They focus on the speculative execution of programs, at the expense of the interaction with external systems.
- They choose to buffer speculative state, at the expense of increased memory bandwidth.
- They weaken transactional isolation, at the expense of semantic simplicity.
- They centralise the responsibility for transaction management, at the expense of scalability.
- They focus on ease of programming per se, at the expense of total productivity across the software development cycle.

Given the disappointing progress of Transactional Memory systems to date it is reasonable to suggest that some of the priorities should be re-assessed and that designs based on a different set of priorities should be considered.

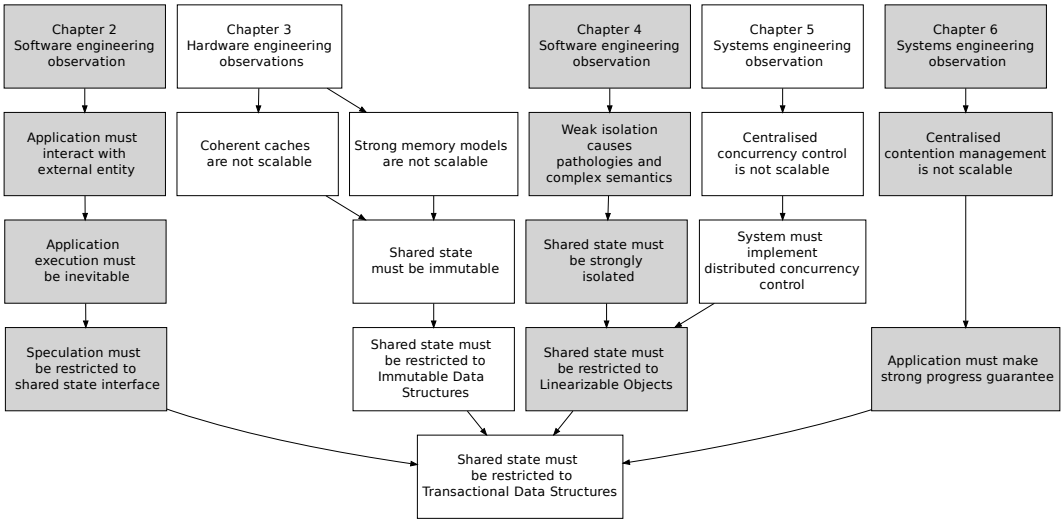



Figure 1.1: **Observations about scalable concurrent systems** led to the development of the techniques described in this thesis.

### 1.2.3 Design choices

This thesis is divided into chapters, each of which considers one aspect of the design of a scalable concurrent system. The first section of each chapter examines how the design priorities have influenced the development of Transactional Memory systems. Subsequent sections develop an alternative approach based on a different interpretation of the design priorities.

Figure 1.1 illustrates the organisation of this thesis.

Some aspects of the design of a concurrent system that need to be considered are:

#### How to interact with entities outside the concurrent system?

A useful concurrent application must interact with external entities, but it cannot do so while executing speculatively. This software engineering observation suggests that application execution must be inevitable.

Chapter 2 identifies the interaction with external entities as a primary design concern. It presents a critique of the approach taken by Transactional Memory systems and explores an alternative in which application execution is inevitable and speculative execution is restricted to the interface with shared state.

**How to maintain shared state and support speculative execution?**

Strong memory models and coherent caches are not scalable. These hardware engineering observations suggest that shared state in a concurrent system must be immutable.

Chapter 3 identifies the restrictions on the scalability of hardware as a primary design concern. It examines the approaches to maintaining shared state employed by Transactional Memory systems and develops an alternative in which shared state is immutable.

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

Weakly isolated transactional systems have complex concurrent semantics and are prone to pathologies. This software engineering observation suggests that concurrent systems must enforce strong isolation.

Chapter 4 identifies weak isolation as a source of both programming complexity and pathologies. It presents a critique of the approach taken by Transactional Memory systems and develops an alternative in which shared state is maintained in data structures.

**How to guarantee correct concurrent execution?**

Centralised concurrency control is not scalable. This systems engineering observation suggests that concurrency control must not be centralised.

Chapter 5 presents a critique of the centralised approach to concurrency control adopted by many Transactional Memory systems and develops an alternative approach based on distributed concurrency control.

**How to implement contention management to eliminate progress pathologies?**

Centralised contention management is not scalable. This systems engineering observation suggests that a scalable application must make a strong guarantee of progress.

Chapter 6 explores the progress guarantees offered by concurrent systems and suggests that the need for centralised contention management can be alleviated

by ensuring that the interface to shared state does not block the progress of the application.

### 1.2.4 The development of Transactional Memory

Research into Transactional Memory is comprehensively described in a book entitled "Transactional Memory" [HLR10].

The following are some of the significant developments in the history of Transactional Memory:

Lomet proposed the use of transactions within programs [Lom77].

Weihl and Liskov proposed the use of transactions to support concurrent programming [WL83].

Stone, Heidelberger and Turek recognised that Memory Transactions are discontiguous multi-word atomic operations [SSHT93]. Computer hardware typically provides atomic operations that act on a single word or a contiguous double-word in memory.

Herlihy and Moss proposed Hardware Transactional Memory [HM93]. A Hardware Transactional Memory implements Memory Transactions by using modified hardware to support speculative execution.

Shavit and Touitou proposed the conventional model of Software Transactional Memory [ST95]. A Software Transactional Memory implements Memory Transactions entirely in software by buffering speculative state in core or in a log.

Lie and Asanovic proposed Hybrid Transactional Memory [LA04]. Recent Hardware Transactional Memory systems are typically hybrids involving both compiler and run-time support for Memory Transactions executing on modified hardware.

### 1.2.5 Software Transactional Memory

Software Transactional Memory systems provide a framework and a run-time system to support the speculative execution of Memory Transactions. Dependencies between tasks are checked at run-time. If conflicting operations are found then the tasks containing them are re-executed resulting in wasted work.

The influential paper "Software Transactional Memory: Why is it only a research toy?" was written by the team responsible for IBM's Software Transactional Memory system [CBM+08]. They compared the performance of their

Software Transactional Memory with comparable systems from Intel and Sun [ART08] [DDS06]. They examined the performance of programs from the STAMP benchmark suite [CMCKO08]. This benchmark suite contains programs that are good candidates for concurrent execution. The team does not discuss the interaction with the Network or Operating System because the benchmark programs are monolithic.

The team found that none of the Software Transactional Memory systems they examined overcame the overhead of supporting Memory Transactions. They also found that, as more processors were added to the concurrent system, the overheads of concurrent execution increased faster than the benefits, so the Software Transactional Memory systems they studied did not exhibit scalable concurrent execution.

The team concluded that complex concurrent semantics, weak atomicity, transactional pathologies, the interaction with serial code, memory reclamation and the support for legacy binaries are all major barriers to the development of Software Transactional Memory.

The conclusion of the IBM paper is that Software Transactional Memory does not achieve its goal of supporting scalable concurrent execution. Soon after publication of the IBM paper Microsoft cancelled its Software Transactional Memory research project without publishing any results [Duf10].

This thesis examines why Software Transactional Memory fails to achieve this goal and explores alternatives that might make the goal achievable in the future.

### 1.2.6 Hardware Transactional Memory

Hardware Transactional Memory systems use a combination of techniques including run-time systems, modified programs and modified hardware to support speculative execution. The original goal of Hardware Transactional Memory was to facilitate the concurrent execution of critical sections in programs written for mutual exclusion without program modification. Unfortunately, programs written for mutual exclusion rarely contain enough information about dependent variables for this to be achievable.

The influential paper "Early experience with a commercial Hardware Transactional Memory implementation" was written by the team responsible for SUN's ROCK processor [DLMN09]. The ROCK processor is a Chip Multi-Processor which aims to support concurrent processing. The processor contained hardware

support for speculative execution and explicit support for Memory Transactions. The team evaluated the system using programs from the STAMP benchmark suite [CMCKO08].

Prior to the ROCK processor Hardware Transactional Memory research was restricted to architectural simulation. A hardware architecture can be simulated in software by a virtual machine on which the target program runs. Typically, the proposed hardware support for Hardware Transactional Memory is included in the virtual machine and the execution time for benchmark applications is determined by simulation. The main problem with architectural simulation is that one cannot be sure that the results will be similar to those that would be obtained were the proposed modifications to be implemented in physical hardware. This is especially true of simulated Chip Multi-Processors because cycle accurate simulation of the shared memory subsystem is extremely difficult to achieve [Jac09]. The ROCK processor was seen, by the Transactional Memory research community, as the most significant Hardware Transactional Memory design to be implemented in real hardware.

The team reported some speed-up on the benchmarks they tested. They also demonstrated some scalability. However, the speed-ups were not very impressive and a great deal of program adaptation was required to obtain them. The team concluded that Hardware Transactional Memory is a promising area of research.

The ROCK project was cancelled a few months after publication of the paper. The reasons for the cancellation were not made public, but the cancellation may suggest that support for Hardware Transactional Memory in commercial Chip Multi-Processors was not found to be economically viable [Van09]. It might also suggest that the benchmark results obtained from the hardware implementation were disappointing when compared to those of architectural simulation [And09].

Many Transactional Memory research papers are able to demonstrate scalable speed-up from the concurrent execution of benchmark applications. There is no doubt that Hardware Transactional Memory implementations can speed-up the concurrent execution of Transactional Memory benchmark programs, such as those in the STAMP or DaCapo benchmark suites [CMCKO08] [BGH+06]. However, these benchmark applications are not general applications. Performance results obtained using them are not indicative of the performance one might expect from the concurrent execution of an operating system or a game.

The conclusion of the ROCK paper is that Hardware Transactional Memory

does not achieve its goal of supporting scalable concurrent execution.

This thesis examines why Hardware Transactional Memory fails to achieve this goal and explores alternatives that might make the goal achievable in the future.

### 1.2.7 Ease of programming

Concurrent programs are difficult to write because using mutual exclusion to serialise access to shared data is error prone. Concurrent programs must implement mutual exclusion, correctly, to avoid the run-time problem of data races and the pathology of deadlock. Concurrent programming must be done at a very low level of abstraction, because locks are not composable, so the fundamental interaction with the program cannot be hidden by abstraction [HMPH05].

Ease of programming is a subjective criterion for determining the efficacy of Transactional Memory systems. Writing a concurrent program using Memory Transactions can be just as difficult as writing one using mutual exclusion as it is often very difficult to break an algorithm into transactions of sufficient size to be worth scheduling. Transactional Memory systems are also prone to run-time pathologies such as livelock and priority inversion.

Research publications often claim that programming concurrent systems using Transactional Memory is somehow easier than writing the same algorithm using locks. In a sample of 25 papers chosen at random from the on-line transactional memory bibliography we found that 17 asserted that Transactional Memory made concurrent programming easier [JBR10]. However, none of the papers in our sample contained any explicit justification for this claim and most referred to it only in the introductory section. Indeed, many of the papers contained descriptions of syntax and semantics that would indicate precisely the opposite to be the case.

Many Transactional Memory research papers claim that concurrent programming using Memory Transactions is easier than using mutual exclusion. However, very few papers support this assertion with quantitative analysis or empirical results. In an exceptional paper Rossbach, Hofmann and Witchel describe experiments that showed that students found programming a concurrent algorithm using Software Transactional Memory was just as difficult as constructing the same algorithm using mutual exclusion [RHW09]. Rossbach was not able to show that Memory Transactions were easier to use than mutual exclusion.

The claim that transactional programming is easier than using mutual exclusion is largely based on experience of programmers using transactions to program Relational Database systems which is undoubtedly easier than accessing a database using mutual exclusion. Relational Database systems that support serialisable transactional execution have largely replaced earlier systems, such as CICS, in which the programmer is responsible for enforcing mutual exclusion [GR92]. However, Relational Database systems and Transactional Memory systems have very little in common as a Memory Transaction coded using an atomic section is very different from a database transaction specified as a Structured Query Language (SQL) statement. Transactional programming is easier in the context of a database system but this does not necessarily mean that using Memory Transactions makes concurrent programming easier in the context of a Chip Multi-Processor.

If we compare the specification of TCC [HCW+04], an early Transactional Memory system, with that of openTM [BMT+07], which is a more recent system from the same institution, then we find the more recent specification to be more complex. So, the claim that Transactional Memory systems have the *potential* to make concurrent programming easier does not appear to be based on an extrapolation of the current trend.

# Chapter 2

# Concurrent Programming

A concurrent program cannot communicate with an external entity or with a component running on another processor while executing speculatively. The Transactional Memory programming model does not offer a satisfactory solution to the problem of communicating with entities outside the program nor does it offer a way of avoiding the complexity inherent in coordinating concurrent actions. This chapter examines the aspects of the database programming and functional programming models that can be usefully incorporated into a concurrent programming model.

Section 2.1 identifies the choice of mechanism to support Input and Output (IO) and operating system interaction as one of the key decisions when designing a concurrent system.

Section 2.2 identifies uncertainty about the passage of time as the source of the complexity in a concurrent system.

Section 2.3 identifies the characteristics that a solution to the concurrency problem must have.

## 2.1 Concurrent IO

The difficulty of presenting a consistent view of shared state to entities outside the control of a program is central to the problem of concurrent programming. A solution to this problem defines the structure of the concurrent program and the nature of the interface with shared state. Concurrent programs do not exist in isolation, they interact with the Operating System, the Network and human interfaces. The requirements for the interaction with entities outside the control of the system should be a primary design concern of any concurrent system.

Concurrent programs that conform to the database programming model present a consistent view of shared state to their external interfaces. A concurrent programming model for Chip Multi-Processors can be developed from the database programming model.

The main contribution of this section is the identification of the features of the database programming model that facilitate interaction with external entities. This section focuses on adapting these features to create a concurrent programming model for a Chip Multi-Processor.

### 2.1.1 The interaction with external entities

Transactional Memory systems do not treat the interaction with external entities as a primary design priority. In fact many Transactional Memory systems do not present any solution for interaction with external entities, other than to support the output of a result at the end of program execution.

A concurrent program must conduct a serial interaction to each outside entity. There must *actually* be a causal, "happens before", relationship between events and their responses and concurrent programs must be built around this causal relationship. It is not possible for a concurrent program to interact with an external entity while it is executing speculatively. Speculative execution may be aborted and restarted, its effects on shared state are speculative and can be undone. However, the interaction with an external entity cannot be undone so this interaction must be restricted to those parts of a concurrent program that are executed inevitably.

A concurrent program presents the appearance of a serialisable interaction with shared state. There must *appear* to be a causal, "happens before", relationship between events and their responses from the point of view of a particular

external entity.

The requirements of the interaction with external entities dictate both the structure of the concurrent program and the nature of its interface with shared state. The causal relationships required by external interaction cannot easily be engineered into a system designed with other priorities.

Transactional Memory systems execute some program code speculatively within an atomic section, which prevents the program from interacting freely with external entities.

Section 2.1.3 discusses atomic sections.

The problem of presenting a consistent view of shared state to entities outside the control of the concurrent program has been successfully solved by the database programming model which describes how a concurrent system should interact with external entities.

Both the Database programming model and the Transactional Memory programming model rely on a transactional approach to concurrent processing. They are targeted at different problem areas and assign different priorities to design criteria. The main difference between them is that the database model regards the concurrent interaction with the client as the primary design concern and regards concurrency as a performance enhancement, whereas Transactional Memory regards concurrent performance as the primary design goal.

Many Chip Multi-Processor systems, such as those used in embedded systems, are heterogeneous. A heterogeneous Chip Multi-Processor consists of processor cores and caches of varying size and complexity linked to each other by an on-chip interconnect. This arrangement of communicating components internalises the problem of interaction.

Section 2.1.6 discusses the difficulty of writing concurrent applications for heterogeneous Chip Multi-Processors.

### 2.1.2 The database programming model

This section identifies features of the database programming model that facilitate external interaction and adapts them to create a concurrent programming model for Chip Multi-Processors. It suggests that a program that executes inevitably can present a consistent view of shared state to external entities.

A database application program executes inevitably and restricts speculation to the interaction with the database, whilst allowing the application to interact

freely with external entities. A database system isolates shared state from an application by implementing the client server model, thereby allowing the interaction with shared state to be treated as a transaction.

Section 2.1.5 discusses the Client Server Database programming model.

We adapt the Client Server Database programming model and apply it to the concurrent execution of a program on a Chip Multi-Processor. The model permits application programs to interact freely with external entities because they execute inevitably.

In our model Memory Transactions are specified in terms of an Application Programming Interface (API) to shared memory. Concurrent programs execute inevitably and speculative execution is restricted to the interaction with shared state. Shared state is stored in objects that are isolated from local state. The interfaces to these shared objects supports Memory Transactions. The shared state interface executes speculatively, but this speculative execution is encapsulated within transactions that present a consistent view of shared state to the application. This allows the program to present a consistent view of shared state to external entities.

### 2.1.3 Atomic sections

An atomic section is a programming idiom that supports the development of concurrent programs. An atomic section is a section of program code that appears to be performed atomically and in isolation as a transaction [CCG08]. An atomic section differs from a critical section because the instructions within the atomic section can be simultaneously executed by more than one processor, whereas a critical section guarantees that only a single processor executes program code within the section at any particular moment in time.

Speculative lock elision is an execution technique that permits the simultaneous speculative execution of program code within a critical section [RG01]. It permits a concurrent program written using mutual exclusion to be interpreted as a program containing atomic sections. Within a section all memory writes are considered speculative and when a conflict occurs the speculative state is rolled back and corrective action is taken. The rationale behind speculative lock elision is that conflicts are rare and that execution of the section is unnecessarily serialised by mutual exclusion.

To detect conflicts it is necessary to distinguish variables that are shared from

variables that are local to a section. In programming languages that allow the use of pointers, such as C, the locality of a variable is not explicitly defined by the program. This limits the utility of atomic sections in general and speculative lock elision in particular. In programming languages that do not allow the use of pointers, such as Java, a system can attempt to determine the locality of variables within a section using techniques such as escape analysis [SR01].

Implementations of atomic sections require the programmer to indicate the locality of variables to the run-time system in some way. However, the object oriented programming model encourages programmers to place logically connected variables with different access characteristics together in the same object. The object oriented model is orthogonal to a model in which the locality of each variable is considered individually.

The apparent simplicity of the use of the *atomic* keyword to identify an atomic section belies the subtle complexities of the use of atomic sections [MBL06]. This complexity arises when atomic sections compromise isolation or are implemented by locks which block the progress of other processors [CGE08].

Database systems support transactions without explicitly supporting atomic sections [WA02]. However, it is informative to consider applying the programming model adopted by Transactional Memory to the programming of a Relational Database system. SQL is a complete functional programming language so complex routines can be written as single SQL statements rather like atomic sections.

Not surprisingly, a database program written in this way has many of the negative characteristics of a program written for Transactional Memory. SQL does not have an IO mechanism so interaction with external systems is restricted. SQL requires that each shared variable must be specified in the database schema so such a program would be tedious to write. For these reasons Database programmers rarely write programs in this style and do not generally express transactions as atomic sections.

The original proponents of Hardware Transactional Memory envisaged a hardware system that would be able to execute programs written for mutual exclusion by concurrently executing critical sections as atomic sections. They imagined that this hardware system could implement transactions transparently and that critical sections could be converted into atomic sections so that applications would not have to be changed. Today, few believe that this is achievable. Transactional

Memory systems require that an application program is significantly modified to support Memory Transactions. Atomic sections seem at odds with modern networked and object oriented applications. Despite this, the basic approach of expressing Memory Transactions as atomic sections has remained the same since Transactional Memory was first proposed.

### 2.1.4 Previous work

It is common for Transactional Memory systems to treat IO and Operating System interaction as engineering problems to be addressed at a late stage in the implementation. However, it is difficult to engineer a serial interaction with external entities into a system primarily designed around the requirements of concurrent execution. This section describes attempts to engineer support for external interaction into Transactional Memory systems.

The main reference book on Transactional Memory describes how Transactional Memory systems perform IO and interact with the Operating System [HLR10]. However, the limited coverage of the topic suggests that the interaction with external systems is not the primary design concern when developing a Transactional Memory system nor is it the main focus of Transactional Memory research.

Transactional Memory systems take three general approaches to interaction. Firstly, they delay interaction by buffering the output produced within an atomic section. The buffer can be discarded if the atomic section is restarted. Secondly, they undo interaction with the Operating System. A memory allocation within an atomic section can be undone should the atomic section be restarted. Thirdly, they stop concurrent execution before interacting with an external entity.

xCall is a Transactional Memory aware API that has been proposed for handling system calls [VTG+09]. xCall addresses the problem of performing IO while executing speculatively. It also addresses the problem that the atomicity and isolation guarantees made by the transactional system do not apply to the Operating System kernel.

xCall provides output facilities to Memory Transactions by buffering IO operations until a transaction has committed. This buffered output can be discarded if speculation fails. The technique makes writing monolithic programs easier as output can be built up as the program runs.

xCall improves the concurrent semantics of some system calls by undoing their

effect when the transaction is aborted. This technique works well for memory allocation but not all Operating System calls are reversible.

Applications in the STAMP benchmark suite stop all concurrent execution before initiating output [CMCKO08]. Software Transactional Memory systems generally approach Operating System interaction in the same way as output. They stop all concurrent execution before making a call to the Operating System.

Operating System interaction complicates the implementation of Hardware Transactional Memory systems and a great deal of engineering effort is required to support it. Many Operating System calls involve a context switch. The state of the transaction prior to the context switch must be preserved and this state must be restored after the Operating System call is complete [KHLW10]. For Hardware Transactional Memory systems that buffer speculative state in cache the context switch associated with Operating System calls is particularly problematic. The Hardware Transactional Memory system must ensure that speculative state held in cache is not flushed during the Operating System call.

In-memory databases implement the database programming model [Gra02]. In-memory database systems execute programs inevitably and present a consistent view of shared state to external entities. However, many of the features of in-memory databases, such as abstract query language and relational tables are not suitable as a model of shared state for concurrent programming. A concurrent programming solution should adopt only those features of the database model that are relevant to supporting the interaction with external entities.

### 2.1.5 The Client Server Database Model

The Client Server Database model addresses a similar problem to Transactional Memory and it shares the goals of supporting scalable concurrent execution and ease of programming. The reason why the programming styles and supporting systems appear so different is that database programs treat the interaction with external entities as the primary design concern and this affects every aspect of the program and supporting system.

The Client Server Database model is a software engineering concept in which the application processing and the management of shared data are regarded as distinct processing tiers. These tiers do not share access to each other's data and the interaction between the tiers is restricted to passing messages between them.

The Client Server Database model provides the appearance of serial execution

to entities outside the control of the system. This is achieved by isolating and serialising the interaction with any particular external entity through a client server relationship. The processing of the interaction with each client is treated as an independent task. These tasks can be executed concurrently while each client experiences a serial interaction with the program.

In the Client Server Database model applications execute inevitably with speculative execution restricted to the accesses to shared data. The execution of a program can be regarded as serial because it is isolated from concurrently executing programs and because the access to shared data is serialised. In the Client Server Database programming model, output is only contingent on committed state so all speculative execution related to the output values must be committed before output can start.

The Client Server Database model describes how shared state should be restricted so that external entities experience a consistent view of shared state. This is achieved by giving the appearance of a serialised interaction with shared state to any particular external entity by using a database as the exclusive repository of shared state.

A Client Server Database system treats state local to an application and state shared between applications completely differently. State shared between processes is restricted exclusively to values in the database, which can only be accessed through the interface provided by the database, whereas state local to an application can be accessed by the usual memory operations.

In the Client Server Database model all state shared between users is restricted exclusively to the database. Data related to one client is isolated from data related to any other client. Output must be based on committed shared state. Typically, a database server will implement some kind of memory protection or address space restriction to prevent instances of concurrently executing programs affecting each other's execution.

### 2.1.6 Heterogeneous processors

Message passing is the predominant model for programming heterogeneous Chip Multi-Processors. The message passing model restricts shared state to the internals of the message passing interface. The program must pass all shared values in messages. When message passing is orchestrated, as it is in a parallel processor, it can be a very efficient way of sharing data, but when messages must be

marshalled, as they are in an embedded Chip Multi-Processor, the overheads of routing messages can be very high.

A communications protocol is used to pass messages between processors. A programmer must be careful to abide by the rules of this protocol and handle all conditions relating to the transmission of the message. It is possible to implement layers of abstraction over message passing protocols but the fundamental interaction with the program cannot be abstracted away [Zim81]. The usual approach to programming heterogeneous systems is to avoid sharing any state at all by using a programming language such as Erlang [Arm07]. There is almost universal agreement that concurrent programs for heterogeneous Chip Multi-Processors are difficult to write [DL09].

The reason why Transactional Memory has not been proposed as a technique for making the programming of heterogeneous Chip Multi-Processors easier is that heterogeneous processors do not have mechanisms for ensuring the consistency of shared memory. Heterogeneous Chip Multi-Processors do not implement mechanisms, such as cache coherency, which would allow them to share state.

The solution to the problems of allowing heterogeneous systems concurrent access to shared data are solved by Client Server Databases which are naturally heterogeneous. The mechanisms used to maintain shared state in a database environment could serve as a model for heterogeneous Chip Multi-Processors.

## 2.2 Parallelism

It is difficult to express an algorithm, within an existing imperative program, in such a way that a computer can execute it concurrently. The key to making this easier is to remove the concept of shared mutable state from both the expression of the program and its execution. This can be achieved by incorporating pure functions and immutable data into the imperative programming paradigm. This relieves the programmer of having to reason about dependencies and coordinate concurrent execution.

Software is becoming more and more complex. Much of this complexity is incidental, arising from the way problems are solved, rather than the problems themselves. Unfortunately, the mechanisms required to utilise the concurrency afforded by Chip Multi-Processors introduce even more incidental complexity. The functional programming and transactional programming paradigms offer ways to reduce the incidental complexity arising from the utilisation of concurrent execution.

The main contribution of this section is to identify concepts fundamental to the support of concurrent programming within the functional and transactional programming paradigms. This section focuses on combining the concepts of pure functions and immutable data within the context of an existing imperative programming language.

### 2.2.1 Temporal uncertainty

The imperative programming paradigm does not offer a satisfactory solution to the problem of coordinating the concurrent actions of multiple processors as it relies on the, essentially serial, concepts of impure functions and mutable state.

For many organisations the investment in existing software is too large to contemplate entirely re-writing working programs just to gain a performance benefit from concurrency. Only a small region of a program benefits from concurrent execution. Finding a way to support the concurrent execution of performance-critical regions of existing imperative programs is of great commercial importance.

This thesis focuses on the problem of expressing concurrency, within an existing imperative program, in such a way that a Chip Multi-Processor can obtain speed-up from concurrent execution. Our approach is to combine aspects of functional programming and transactional programming within the imperative

programming paradigm.

The aim is to reduce the incidental complexity introduced into an algorithm when it is expressed in such a way that it can execute concurrently on a Chip Multi-Processor. This incidental complexity is not restricted to the additional code required to allow a routine to execute concurrently. Mechanisms to support concurrency also make it more complex to design, code, test, debug and maintain a concurrent algorithm than an equivalent serial algorithm.

The source of this complexity is the uncertainty about the passage of time perceived by the concurrently executing components and this temporal uncertainty originates from uncertainty about the dependencies between functions and the interleaving of memory operations.

Functional programming overcomes the incidental complexity of determining whether concurrently executing functions are dependent on each other as it emphasises the use of pure functions in which all dependencies are explicit. This raises the question of how to express pure functions in such a way that programmers can incorporate them into existing imperative programs easily?

Section 2.2.3 discusses this problem in detail.

Functional programming eliminates the incidental complexity inherent in the management of mutable shared state. Functional programming emphasises the use of immutable data which can be safely shared between processors. This raises the question of how to express immutable data in such a way that programmers can incorporate it into existing imperative programs easily?

Section 2.2.4 discusses this problem in detail.

Transactional programming reduces the incidental complexity of coordinating concurrent actions between processors. Transactions permit the simultaneous speculative execution of functions in the absence of complete information about their dependencies. This raises the question of how to express Memory Transactions in such a way that programmers can incorporate them into existing imperative programs easily?

Section 2.2.5 discusses this problem in detail.

### 2.2.2 Minimising temporal uncertainty

The problem of supporting concurrency in an imperative programming language can be broken down into the problems of determining dependencies between functions, managing shared state and coordinating concurrent actions. The functional and transactional programming paradigms offer solutions to these problems. Functional programming languages emphasise the use of pure functions and immutable data as a means of identifying dependencies and managing shared state respectively. Transactional programming emphasises the use of transactions as a means of coordinating concurrent actions. However, the concurrency problem is not solved by translating a concept from one programming paradigm into another. Solutions should be found to the problem of balancing concurrent work while ensuring that operations appear to occur in the correct semantic order.

Certainty about the dependencies between functions permit a functional program to include atomic sections which can be evaluated speculatively.

Section 2.2.7 discusses the speculative evaluation of functional programs.

Certainty about the isolation of atomic sections permit those sections to be executed speculatively without blocking the execution of other processors. The non-blocking property permits atomic sections to be composed. The ability to compose atomic sections raises the level of abstraction offered to programmers freeing them from concerns about locks, lock acquisition order and deadlock. It permits the combination of abstractions without knowing their implementations.

Section 2.2.8 discusses the composition of atomic sections implemented by functional programming languages.

Pure functions, immutable data and Memory Transactions are difficult to incorporate into the imperative programming paradigm, independently. However, there are synergies to be gained by incorporating these concepts into the imperative programming paradigm in an integrated manner. The challenge is to combine these concepts in a way that makes concurrent programming easier within the imperative programming paradigm.

The proposal developed in this thesis is to decompose programs into functions acting on immutable data and to execute those functions speculatively as Memory Transactions. In this way the transactional and functional programming paradigms can be combined to support concurrent execution.

### 2.2.3 Functional dependencies

A function that uses a value produced by another function is said to be dependent on that function. The dependency implies that the functions should be executed in a particular order called the precedence order of the functions. Functions that are not dependent on each other may be executed concurrently. Precedence is usually a weak ordering offering many opportunities for concurrent execution. When there is uncertainty about the dependencies between functions it is not safe to execute them concurrently but when the uncertainty about dependencies is reduced the opportunities for exploiting concurrency increase.

An impure function is one that does not necessarily produce the same return value each time it is executed with a given set of parameters. Impure functions may have side effects. They read the state of memory in addition to their parameter list and they can modify the state of memory in addition to returning a value. These side effects introduce dependencies between functions which are not expressed in the function's parameter list or return value. The dependencies between functions should be known if they are to be executed concurrently.

Imperative programming languages are not usually expressive enough to allow the identification of all functional dependencies. Consequently, it is often not possible to identify sets of routines that do not contain dependencies and that can safely be executed concurrently.

Functional programming is a style of programming that emphasises the use of pure functions. Pure functions do not have side effects. The dependencies of pure functions are easily determined because they are restricted to their parameters. The effects of pure functions are restricted to the return values of the function.

An expression is said to be referentially transparent if it can be replaced with its value without changing the behaviour of a program. A referentially transparent expression corresponds to an expression in pure mathematics; it is a timeless statement of truth.

Pure functions have many advantages over the impure functions typical of imperative programming languages. The use of the functional program style in imperative programs is explored in [HM97]. However, the functional program style is in many ways orthogonal to the style in which imperative programs are written. The difficulty of using pure functions in an imperative context arises because imperative programming languages lack the expressiveness necessary to enforce purity through the use of the type system.

### 2.2.4 Mutable shared state

When a function modifies data that is shared it must ensure that no function executing on another processor is accessing that data at the same moment in time. A function can only be certain about mutable data that is never shared. The conventional approach to ensuring that a function has exclusive access to shared data is to serialise access to it using mutual exclusion. An alternative approach is to eliminate mutable shared data altogether and share only immutable data. Both approaches increase the opportunities for exploiting concurrency by reducing uncertainty about the order of access to shared data.

Imperative programming languages permit mutable shared data in the form of variables, objects and data structures. Shared data cannot be simultaneously modified by multiple processors safely. To prevent simultaneous modification imperative programming languages implement mutual exclusion which serialises the execution of a code section accessing shared data. The association between a serialised code section and the shared data which it protects is a convention. It is not expressed in, and is not enforced by, the programming language.

Functional programming emphasises the use of immutable data. Immutable memory locations are unreachable before they have been written and cannot be modified after they have been written. Immutable data can be organised into Immutable Data Structures. A data structure is an Immutable Data Structure if all of the memory locations within it are immutable. Immutable implementations of many common data structures are described in the literature [Oka98]. These Immutable Data Structures can have access times and space requirements similar to their mutable counterparts.

Immutable Data Structures have received little attention outside the field of functional programming languages and there are no publicly available libraries of Immutable Data Structures implemented in imperative programming languages. Immutable Data Structure are traditionally regarded as more difficult to implement than their ephemeral counterparts.

The use of immutable data is in many ways orthogonal to the imperative program paradigm. In general, the use of Immutable Data Structures does not make imperative programming easier and very few imperative programs make use of them.

### 2.2.5 Coordinating concurrent actions

An algorithm may be decomposed into tasks that can be executed concurrently on multiple processors. The actions of these tasks should be coordinated. However, imperative programming languages do not offer a general solution to the problem of coordinating actions on multiple processors.

Transactional programming is a style of programming that emphasises the use of speculative execution. Transactions permit speculation by allowing their effects to be undone should speculation prove incorrect. Transactions permit the separation of the *actual* order of execution from the order in which operations *appear* to have executed. It is the separation of the actual and apparent order of execution which permits speculation.

Concurrent actions are easier to coordinate if their effects are restricted to transactions. Transactions permit reactive and optimistic coordination, so conflicts can be detected after they happen, and can be corrected. Without transactions coordination must be preemptive and pessimistic, so conflicting events occurring on different processors must be anticipated and avoided.

The support for Memory Transactions within imperative programming languages is discussed extensively in this thesis. A central problem is how to express a Memory Transaction within the imperative programming paradigm without making extensive changes to existing applications?

### 2.2.6 Previous work

Harris, Marlow, Peyton-Jones and Herlihy develop a Software Transactional Memory system called STM Haskell that is based on the functional programming language Haskell [HMPH05].

Haskell is a pure functional programming language. It prevents a programmer from using impure functions or mutable state, so the choice of Haskell as a base for developing a concurrent programming language eliminates uncertainty about both functional dependencies and the interleaving of memory operations during concurrent execution.

STM Haskell permits the use of atomic sections that can be evaluated speculatively as Memory Transactions. STM Haskell's type system ensures that these

Memory Transactions are strongly isolated. STM Haskell's run time system ensures that these Memory Transactions do not block each others progress. Consequently, Memory Transactions implemented in STM Haskell are composable and are not prone to deadlock.

Harris describes the desirable properties of the functional and transactional programming paradigms and attempts to combine them. Harris's system combines pure functions, immutable data and Memory Transactions to support concurrent execution. Our Transactional Data Structures also combine these elements. However, Harris chooses to regard Memory Transactions as atomic sections, whereas we choose to regard them as speculative access to shared data.

### 2.2.7 The speculative execution of functional programs

Concurrent Haskell adds support for concurrent execution to the Haskell language by introducing explicitly forked threads and a mechanism for communicating between them [PGF96]. To support the interaction of concurrent programs with external entities Concurrent Haskell introduces functions with side effects to the otherwise pure language. Side effects are implemented by a mechanism called a monad which is incorporated into the type system [PW93].

A monad permits a value of a specific type to have an associated side effect. When a value of a monadic type is accessed it performs some action before yielding its value. Side effects can be combined by a monadic bind combinator to produce a sequence of actions. For example, the IO monad has the side effect of performing input and output operations. An access to a value of a monadic IO type reads a value of that type from an input stream or writes a value to the output stream. These accesses can be combined to produce a sequence of input and output actions.

STM Haskell extends Concurrent Haskell to support speculative execution by introducing mutable variables called transactional variables and introducing a monad to support speculative access to them [HMPH05]. This monad facilitates the creation of atomic sections. Within an atomic section operations on transactional variables are speculative and are not visible to atomic sections executing concurrently.

The speculative values of transactional variables can be combined by a monadic bind combinator to form the speculative state of an atomic section. STM Haskell introduces a function that atomically commits the speculative values of all the

transactional variables written by an atomic section thereby making them visible to other atomic sections. Atomic sections implemented in STM Haskell can be regarded as Memory Transactions because they are isolated and they commit atomically.

The speculative values of all of the transactional variables accessed by an atomic section are validated and committed at the end of a transaction. The validation process ensures that no concurrent transactions have committed conflicting updates to any of the transactional variables accessed by the atomic section. If validation is successful then the state of the transactional variables is updated atomically and can be observed by subsequent Memory Transactions, otherwise the speculative state of the transaction is discarded and the atomic section is automatically retried.

STM Haskell introduces a mechanism for explicitly retrying an atomic section. Explicit retry aborts the transaction and discards the speculative values of the transactional variables. Explicit retry occurs under the control of the programmer but it is otherwise similar to the automatic retry that occurs after an unsuccessful validation. Automatic retry repeats until no conflicting operations occur, whereas explicit retry can be used to retry until an event, signalled by the value of a transactional variable, occurs. To improve efficiency both types of retry are initiated only when at least one of the transactional variables accessed by the atomic section is modified by another processor.

Haskell has a static type system which allows the compiler to infer a precise type for all values automatically. The type system prevents a speculatively executing Memory Transaction from interacting with an external entity by ensuring that only pure functions and speculative actions on transactional variables are permitted within an atomic section. The type system also ensures that Memory Transactions are isolated from each other and from non-transactional execution by preventing transactional variables from being accessed outside an atomic section.

STM Haskell provides mechanisms to support atomic and isolated Memory Transactions. Harris and Peyton-Jones describe how STM Haskell can be extended to include mechanisms for ensuring the consistency of accesses to data structures [HP06a].

### 2.2.8 Composable atomic sections

Harris, Marlow, Peyton-Jones and Herlihy describe how STM Haskell permits the composition of Memory Transactions [HMPH05]. Peyton-Jones provides an accessible introduction to the modular construction of concurrent programs in STM Haskell [Pey07].

Sections of program code that ensure their correctness by blocking the progress of other code sections cannot be composed because correct code sections may fail when combined. To ensure that two code sections do not deadlock the order in which their locks are acquired must be examined, so code sections that block do not encapsulate their implementation details. However, atomic sections implemented by STM Haskell do not block, so they are composable and they can encapsulate the details of their implementation.

A useful concurrency mechanism should provide a means by which a processor can wait for an event to occur on another processor. STM Haskell provides a mechanism whereby an atomic section can wait for an event by explicitly retrying until the event occurs. An atomic section *speculatively executes* the code section that will be completed when an event, which is signalled by the value of a transactional variable, occurs. The atomic section may be retried repeatedly until the event occurs without blocking the progress of the processor. Atomic sections that wait for an event in this way can be composed because it is not necessary to examine or modify their internals when combining them.

A processor can wait for an event to occur on another processor by blocking the progress of the processor until notification of the event is received. A lock may block the progress of a processor to *protect* the code section that will be completed when it is released. It is the act of blocking progress that prevents code sections from being composed. Code sections that block the progress of a processor are not composable because it is necessary to examine their internals to ensure that deadlock does not occur.

STM Haskell provides a mechanism whereby two atomic sections can be sequentially composed to create a composite atomic section. The composite atomic section commits as a single atomic action. It is not necessary to examine or modify the internals of the atomic section during composition, so atomic sections are sequentially composable.

Critical sections protected by locks cannot be sequentially composed. To sequentially combine two critical sections the locks around each section must be

replaced by a single lock to create a combined critical section. This requires the examination and modification of the internals of the critical sections, so critical sections are not sequentially composable.

STM Haskell provides a mechanism whereby two atomic sections can be composed as alternatives, so that either one or the other is eventually executed. If validation of one atomic section fails then, instead of retrying the same atomic section, the other can be tried. If validation of the alternative fails then the entire composition is retried. Atomic sections that act as alternatives in this way can be composed because it is not necessary to examine or modify their internals when combining them.

Critical sections protected by locks cannot be composed as alternatives. To combine two alternative critical sections they must be combined to form a critical section that contains both alternatives. This requires the examination and modification of the internals of the critical sections, so critical sections are not composable as alternatives.

The mechanisms described here facilitate the construction of non-blocking algorithms. Discolo et al. describe how STM Haskell can be used to implement a non-blocking queue [DHM$^+$06].

## 2.3 Compatibility

A solution to the concurrency problem should be compatible with existing programs and software development processes. Unfortunately, the changes needed to support concurrent execution are not always confined to the performance critical regions of the program. This section examines how the compatibility criterion restricts the design space of concurrent programming solutions.

The main reason for writing a parallel program is to obtain speed-up from the performance-critical regions of the program that can benefit from parallel execution. These regions are only a small part of most programs and a solution to the concurrency problem should only apply to these regions.

The main contribution of this section is the recognition that the potential benefits of concurrent execution are rarely compelling enough to justify disrupting existing software development processes or completely re-writing existing programs. This section focuses on defining the scope of possible solutions to the concurrency problem that are compatible with existing software.

### 2.3.1 Disruptive changes to existing software

The benefit of exploiting concurrency must exceed the costs associated with implementing it.

Section 2.3.3 explains why a worthwhile concurrent programming solution must improve total software development productivity.

Research into concurrent programming tends to focus on obtaining speed-up from the concurrent execution of those regions of a program that benefit from it while giving little consideration to the impact on those regions of a program that do not. To be compatible with existing software a concurrent programming solution should only affect the regions of a program that benefit from concurrent execution.

To stand a realistic chance of adoption a concurrent programming solution should be compatible with existing software, libraries, operating systems, development tools and hardware.

Regions with unknown dependencies can occur in applications that also contain regions that are known to be independent. Mechanisms to support the speculative execution of tasks with unknown dependencies should not adversely

affect mechanisms that support the execution of tasks that are known to be independent.

During the unit testing phase of application development it must be possible to reproduce a problem for debugging purposes, during the acceptance testing phase it must be possible to stimulate all possible program behaviours and in production it must be possible to capture a program's behaviour so that errors can be reproduced. To be compatible with existing software a concurrent application must exhibit reproducible behaviour, so that it can be integrated into existing testing methodologies.

Thus, a concurrent programming solution must be locally applicable, compatible with existing software and development processes, compatible with a parallel programming solution and compatible with existing testing methodologies.

### 2.3.2 Compatibility with existing software

A concurrent programming methodology should be applicable locally and it should not be necessary to structure a program around the requirements of those regions that it is beneficial to execute concurrently. We focus on restricting the locality of program changes to those routines that benefit most from concurrent execution.

A concurrent programming solution should be implemented in software, without requiring changes to the compiler, the operating system or the software development tool chain. We focus on developing a concurrent programming solution for a conventional imperative language to minimise the impact on existing programming methodologies.

A concurrent programming solution should permit the parallel execution of tasks that are known to be independent and tasks with possible dependencies. We focus on providing compatibility with the Threading Building Blocks library which is an integrated parallel programming solution for Chip Multi-Processors [Int09].

A concurrent execution environment should ensure reproducible application behaviour. We focus on using time stamps to ensure the correctness of concurrent execution. Time stamps can be used to ensure reproducible behaviour and to determine the relationship between tasks during the problem solving process.

### 2.3.3 Making concurrent programs easier to write

The goal of research into concurrent programming is to make it easier to create scalable concurrent programs. To achieve this goal, the benefit from the reduction in the execution time of a concurrent program, relative to an equivalent serial program, must exceed the total cost associated with making that program execute concurrently.

A technique that makes program coding easier might make a program more difficult to debug offsetting any programmer productivity gains. Any proposal to make programming easier should improve productivity when amortised over the entire development process including: program design, coding, debugging, testing, operation and maintenance. The benefits of a new programming technique must also exceed the costs associated with learning it and the cost of rectifying mistakes made when it is applied incorrectly.

Regions of many types of application may benefit from concurrent execution, so the challenge is to integrate techniques to support concurrency into existing programming environments in such a way that utilising concurrency in those regions is worthwhile.

# Chapter 3

# Maintaining State

A concurrent program that uses mutable speculative and shared state is not scalable. Transactional Memory systems buffer speculative and shared state at the expense of increased memory bandwidth which limits scalability. This chapter examines how immutable data offers a means of maintaining both speculative and shared state that permits a concurrent program to scale.

Section 3.1 identifies the choice of where to store speculative state as one of the central design decisions of a Transactional Memory system. The section reviews the mechanisms that Transactional Memory systems employ to support shared and speculative state and it proposes an alternative approach in which both speculative and shared state are stored immutably.

The remainder of the chapter focuses on an implementation of an Immutable Data Structure suitable for use in a concurrent execution environment.

Section 3.2 describes how Immutable Data Structures can be used to store speculative state.

Section 3.3 describes techniques for implementing Immutable Data Structures.

Section 3.4 describes an Immutable Data Structure that we call the Canonical Binary Tree.

Section 3.5 describes how the Canonical Binary Tree can be specialised to implement common ADTs.

Section 3.6 describes how the Canonical Binary Tree can be balanced to minimise access time.

## 3.1 Speculative State

The memory wall is an obstacle to obtaining scalable speed-up from the execution of a program on a Chip Multi-Processor. Transactional Memory systems promise to speed-up concurrent execution by removing the barriers to scalability imposed by mutual exclusion, but concurrent speed-up has only been demonstrated in a few applications, because the buffering of speculative state increases the memory bandwidth requirement of a concurrent program, restricting scalability. The use of immutable memory permits concurrent programs to scale to greater numbers of processors before hitting the memory wall.

The effective memory bandwidth of a scalable concurrent program must be independent of the number of processors participating in its execution.

The main contribution of this section is an examination of why the demonstrable concurrent speed-up of general applications has remained elusive. This section focuses on identifying the impact of the buffering of speculative state on memory bandwidth as a factor limiting the speed-up that can be achieved.

### 3.1.1 The memory wall

For execution-bound programs there is a potential for speed-up from concurrent execution on a Chip Multi-Processor. For such programs a barrier to concurrent speed-up is mutual exclusion as described by Amdahl's law. Speculative execution avoids the need for mutual exclusion and alleviates the scaling restrictions of Amdahl's law. However, the scaling of a concurrent program is bounded by restrictions imposed by both memory latency and memory bandwidth. Wulf and McKee describe these restrictions which are collectively known as the memory wall [WM95].

The connection between the processors of a Chip Multi-Processor and main memory has a finite bandwidth that is shared by all of the processors. The connection consists of the caches, the memory controller and the wiring between the processor chip and main memory. Contention in the common components of the path to memory affects the speed at which memory requests can be serviced.

A program has a memory bandwidth requirement which is the bandwidth, expressed in bytes per second, that it consumes. An increase in the memory bandwidth requirement leads to an increase in the latency of individual memory requests and an increase in the elapsed execution time of the program [HP06b].

In a Chip Multi-Processor a finite memory bandwidth is shared amongst all of the processors and this limits the speed-up that can be obtained from concurrent execution. Increasing the available memory bandwidth is a much more difficult engineering challenge than increasing the number of processors in a Chip Multi-Processor so memory bandwidth tends to increase more slowly than aggregate processing power.

Section 3.1.3 describes the limiting effect of memory bandwidth on execution time and the difficulty of increasing the available bandwidth.

Buffering speculative state increases the memory bandwidth of a concurrent program. Data is written twice, once as isolated speculative values and again as shared committed values. Bookkeeping information, required to ensure correct concurrent execution, is also written to memory. Wasted work from failed speculation also contributes to the volume of data written to memory. Together these factors cause a concurrent program to have a much higher memory bandwidth requirement than the equivalent serial program.

Section 3.1.4 describes how storing speculative state increases the memory bandwidth requirement of a program.

For many applications the memory wall is a constraint on the speed of serial execution and such applications are known as memory-bound. It is reasonable to expect that memory bandwidth will also be the main barrier to obtaining concurrent speed-up on Chip Multi-Processors. The use of multiple processors does little to alleviate the memory wall problem, instead Chip Multi-Processors make the memory bandwidth problem more acute.

Section 3.1.5 describes how concurrent programming transforms an execution-bound program into a program bounded by memory bandwidth.

Chip Multi-Processors enforce a cache coherency protocol to keep caches coherent but mechanisms to ensure cache coherency do not scale well. The overheads associated with maintaining coherent caches reduce the effectiveness of caching and thus increase the effective memory bandwidth of a concurrent application. The engineering difficulty of scaling cache coherency mechanisms is a barrier to increasing the number of processors in a Chip Multi-Processor design.

Section 3.1.6 describes the difficulty of scaling the mechanisms that ensure cache coherence.

### 3.1.2 Immutable memory

When only immutable data is used to represent shared state, the amount of shared data that is either read or written to main memory by a program is independent of the number of processors involved in its concurrent execution.

A concurrent program should maintain both speculative and shared state immutably in memory. Immutable values are written just once so immutable data satisfies the requirement that it does not increase memory bandwidth of a program. Immutable values cannot change so cached copies are always coherent.

Immutability is a memory usage convention. A memory location is said to be immutable if its value is written just once and cannot be changed thereafter. Prior to writing the value the memory location cannot be reached by the program, so the program cannot read memory locations that have not already been written and cannot write to those locations that have already been written.

An immutable object is an object whose state cannot be modified once it has been created. It can be regarded as a set of constant values. An object reference associates an identifier with a location in memory where the object can be found. An immutable object cannot be modified but a reference to it may be mutable, so an identifier can be associated with different versions of an immutable object by modifying its reference. A concurrent program that maintains state immutably requires mutable memory to maintain both unshared state and shared references to immutable objects.

Immutable objects can be relocated while retaining the property of immutability. To relocate an immutable object in memory a copy of the object is made at another location. Values can be inserted into and deleted from an immutable object during the copy operation. It is possible to create an immutable object with identical properties to any mutable object by implementing all of the object's mutating methods as constructors of new copies of the object. A serial program that maintains state in immutable objects may have a different memory bandwidth requirement from a similar program that uses mutable objects but in many cases immutable objects can be implemented just as efficiently as their mutable counterparts. It is not necessary to perform a full copy of an object every time a mutating method is called to preserve the property of immutability.

Immutable data is written just once so an immutable value written speculatively does not need to be written again when it is shared. A concurrent program that maintains shared state immutably scales without increasing its

memory bandwidth requirement as the total amount of data both written to and read from memory is unaffected by the number of processors participating in its execution.

An immutable object can never go stale in cache because its value cannot be changed so it is not necessary to ensure that the cached copies of an immutable object are coherent. However, a mechanism to enforce cache coherency is required to ensure that all processors observe an up to date copy of the reference to the immutable data.

Immutable data frees Chip Multi-Processors from the scaling restrictions of cache coherency in two ways. Firstly, it is not necessary for the processor design to enforce a cache coherency protocol for all memory locations, allowing the design to be more scalable. Secondly, the cache pathologies of cache coherency misses and false sharing do not occur and this increases the effective memory bandwidth of the cache.

### 3.1.3 Memory bandwidth

A program executing in parallel on two processors requires twice the memory bandwidth of an equivalent program executing on one. The bandwidth requirement for processors executing general applications is around 1GB/s per core. Desktop and server Chip Multi-Processors use single or dual DDRx memory systems. The maximum bandwidth of such an arrangement is less than 10GB/s. Jacob offers a reason why four physical core Chip Multi-Processors are common and eight core systems have yet to appear which is that, unless the memory system is upgraded, an eight core system would perform no better than a four core system [Jac09].

A solution to the problem of restricted bandwidth is to increase the memory bandwidth of the processor. Historically, memory bandwidth has increased more slowly than processor frequency for physical reasons, such as the difficulty of scaling the number of off-chip pins. Increasing the number of off-chip pins is challenging because of their energy requirements and because it increases the complexity of printed circuit boards. Currently, processor frequency is static and the number of processors on a chip is increasing. Jacob describes why the number of concurrent memory operations that a processor's memory controllers can support is much harder to scale than the number of processors on the chip [Jac09].

Memory bandwidth can be increased to match the number of cores, but at significant design cost. A Chip Multi-Processor saturates its memory subsystem once the number of cores multiplied by the bandwidth of the program executing on them reaches a maximum sustainable bandwidth. Jacob finds that the 32 core Niagara Chip Multi-Processor has a memory subsystem that saturates at 25GB/s, so the Niagara processor has a memory bandwidth of less than 1GB/s per core [Jac09].

Memory bandwidth is limited by physical factors and dramatic increases in bandwidth are unlikely in the near future. Consequently, proposals to support concurrent programming should focus on decreasing the effective memory bandwidth requirement of programs.

### 3.1.4 The effect of speculation

Transactional Memory systems take several different approaches to storing speculative state. Each of these approaches has its own relative merits, which are discussed in detail in the main reference book on Transactional Memory [HLR10]. However, each approach involves writing values to more than one location or writing additional meta-data to memory. The additional memory writes tend to increase the memory bandwidth requirement of the program.

Maintaining state in a recovery log is a common technique in Software Transactional Memory systems. Logging state increases memory bandwidth as each shared value must be written to main memory at least twice. Typically, a system will write the old value of a location to a log before storing the new value. For example, the logTM Software Transactional Memory system maintains the committed state of memory locations that have been written speculatively in a log [MBM+06]. This technique is known as eager versioning. The amount of state written to the log is equal to the amount of speculative state written by the program. The latency of a memory write operation can be reduced by caching the log but, eventually, both the old and new values must be written to main memory as a result of the operation thus increasing the memory bandwidth requirement of the program.

Maintaining speculative state in cache is a technique adopted by some Hardware Transactional Memory systems. For example, the Hardware Transactional Memory proposal of Herlihy and Moss maintains speculative state in a dedicated transactional cache [HM93]. Speculative values are eventually written to main

memory in addition to committed values so the caching of speculative state increases the memory bandwidth of a program. When cache contains both the speculative and committed state of an object the number of distinct objects that it can contain is reduced so the caching of speculative state also increases the memory bandwidth of a program by reducing the effectiveness of cache.

Maintaining speculative state in a buffer is a technique adopted by many Hybrid and Software Transactional Memory systems. Buffering shared state increases memory bandwidth because objects must be copied when they are written. Typically, a buffering Transactional Memory system will copy an entire object to a new location when one of its fields is modified speculatively. The operation usually has low latency because it occurs in cache, but the whole of the copied object must eventually be written to main memory as a result of the operation. Object copying increases the memory bandwidth of the program.

Each of these techniques require additional bookkeeping information to ensure the correct concurrent execution of the program. This information will eventually be written to main memory, increasing the effective memory bandwidth of the program.

Speculative execution necessitates that some transactions will be aborted and the work they did will be wasted. Memory operations performed by this wasted work also increases the effective bandwidth of the concurrent program.

Transactional Memory increases the memory bandwidth requirement of the program. In many cases the overhead of buffering speculative state is the main factor limiting the speed-up that can be achieved from the concurrent execution [Olu07].

### 3.1.5 Moving the bottleneck

The number of processing cores that it is possible to fit into a single Chip Multi-Processor is expected to increase in future. As the number of cores increases so does the potential speed advantage of concurrent programs over their serial counterparts. Concurrent programming is universally accepted to be difficult but at some point the speed advantage of concurrent execution will make the effort of writing concurrent programs worthwhile.

This familiar argument is based on two questionable assumptions. Firstly, that the difficult of writing concurrent programs is a major obstacle to the adoption of concurrent programming. Secondly, that a concurrent program has the potential

to execute faster on a Chip Multi-Processor than the equivalent serial program.

In many application programming environments, such as the computer games industry, there are enormous financial incentives to improve concurrent performance. In such environments no programmer effort is spared in utilising concurrent execution. The difficult of writing concurrent programs can be overcome by applying many programmers to the task and requiring each of them to think very hard. The real problem is that their efforts are so rarely rewarded by improved performance of the program.

The elapsed execution time of a memory-bound program on a Chip Multi-Processor is equal to or greater than the serial execution time, no matter how many processors are applied to the problem. Only execution-bound programs have the potential for a concurrent implementation executing on a Chip Multi-Processor to execute faster than a serial implementation.

For execution-bound programs there is a potential speed-up from concurrent execution. The first obstacle to realising this speed-up is that executing on multiple processors increases the bandwidth of the program causing it to become memory-bound. The second obstacle is that instrumentation to support speculative execution increases the effective memory latency and bandwidth of the program causing it to become memory-bound.

At best Transactional Memory converts a concurrent program with speed-up restricted by mutual exclusion into a concurrent program with speed-up restricted by the memory wall. Transactional Memory systems increase the memory bandwidth of the program and this lowers the amount of scaling possible before a concurrent program hits the memory wall. Programs that have a low memory bandwidth requirement tend to scale well when the restrictions of mutual exclusion are removed and these are the programs that Transactional Memory research focuses on [PW10].

### 3.1.6 Cache coherency

Small memories are generally faster than large memories because they contain shorter wires. Processors maintain a hierarchy of caches of different sizes to reduce memory latency and increase memory bandwidth. Chip Multi-Processors maintain both shared and unshared caches. Typically, each processor has a small local cache that is not shared and if a memory access cannot be satisfied from this cache an attempt is made to satisfy it from a larger slower cache shared between

all of the processors of the Chip Multi-Processor.

To present a consistent view of memory to each processor a Chip Multi-Processor implements a cache coherency mechanism which enforces a cache coherency protocol. A snoop-based cache coherency mechanism broadcasts the address of memory locations that have been modified to all caches and a directory-based mechanism records where all of the copies of a particular location reside. Chip Multi-Processors generally enforce snoop-based protocols to avoid the additional latency of accessing a centralised directory.

The implementation complexity of snoop-based cache coherency protocols increases with processor count because the number of processors that can access a memory bus is physically limited, so designers face the challenge of maintaining coherency without the benefit of a single bus to serialise events [Sto06].

A coherency cache miss is a cache miss required to maintain coherency between processor caches. When a cached location is modified by a processor all of the copies of that location held in the local caches of the other processors must either be updated or discarded. Typically, a snoop-based protocol regards the copies held by the other processors as stale and marks them as invalid so the next access to the location will result in a cache miss. Coherency cache misses tend to increase with the processor count and are unaffected by cache size. They have a detrimental effect on performance as each cache miss increases the effective memory bandwidth of the program.

The messages sent between processors to maintain coherent caches are known as coherency bus traffic. Coherency bus traffic increases with processor count and is unaffected by cache size. Congestion on the bus has a detrimental effect on memory latency and additional bus traffic increases the effective memory bandwidth of the program [HP06b].

## 3.2 Immutable Data Structures

To support scalable execution a concurrent system should support speculation without increasing the effective memory bandwidth of the program. A solution should facilitate concurrent access to shared data while requiring that values are written to main memory only once. Immutable data is necessarily written to main memory only once so we propose that Immutable Data Structures can act as repositories of both speculative and shared state. Immutable Data Structures have not previously been considered in the context of concurrent execution so support for them must be developed before this proposal can be evaluated.

The problem is to find a mechanism for maintaining speculative and shared state in memory. The solution should support the isolation of speculative state and the atomic transformation of speculative state into shared state. It should also support simultaneous access to shared state and require that data values be written to main memory once only.

This section identifies Immutable Data Structures as candidate repositories of shared state in concurrent systems and examines techniques for maintaining both speculative and shared state in Immutable Data Structures.

### 3.2.1 Supporting speculation

To support speculation a mechanism to isolate speculative state and permit its atomic transformation into shared state is required. This mechanism should afford scalable concurrency without increasing the effective memory bandwidth of the program.

The mechanism should support the isolation of speculative state from other functions executing concurrently. Only the process that wrote the state speculatively should be able to observe it. The mechanism should also ensure that a function observes a consistent view of shared state. Consistency criteria must be met at the point speculative state becomes shared state.

The mechanism should support the atomic transformation of speculative state into shared state. In a Chip Multi-Processor the only mechanism for performing an atomic action is an atomic instruction, so the transformation of speculative state into shared state must be implemented by an atomic instruction.

Typically, atomic instructions act on only one word in memory. The atomic transformation of isolated multi-word values into shared values can be achieved by

atomically updating a reference to those values instead of the values themselves. To enable atomic transformation, to shared state, speculative state should be identified by a single reference and this reference should be modified by an atomic instruction.

An atomic instruction typically implements a memory barrier to ensure that any memory writes, buffered by the processor, are completed and that caches are coherent during the execution of the atomic instruction. The memory barrier ensures that the speculative state identified by the reference appears to be atomically transformed into shared state.

### 3.2.2 Immutable Data Structures

Immutable Data Structures provide a solution to the problem of maintaining both speculative and shared state. Paths within an Immutable Data Structure can be isolated until the mutable reference to the data structure is modified by an atomic instruction so functions acting on Immutable Data Structure can benefit from isolation and atomicity provided by the structures themselves.

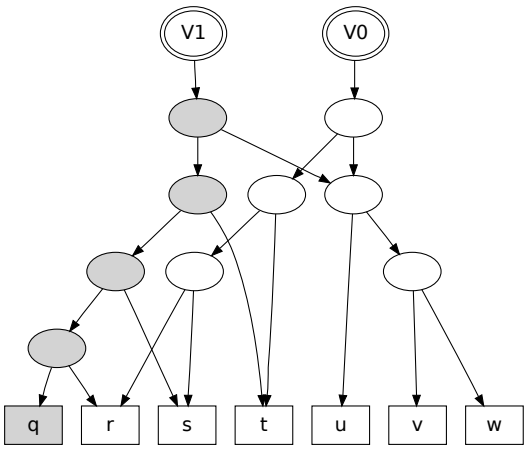
Figure 3.1 illustrates the insertion and removal of an element in an immutable binary tree. The functions cause a new path to be created within the data structure but do not change any of the existing values. A version of a data structure is identified by a mutable reference. The data structure does not change per se. Instead, a new version is created by copying data and modifying the reference.

### 3.2.3 Immutability and concurrency

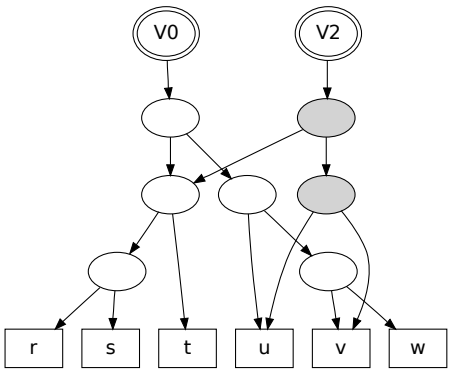
In this section we describe how certainty that shared data within an Immutable Data Structure is immutable enables a program to access it concurrently.

Immutable Data Structures provide a medium for maintaining immutable shared state within the data structure itself. Immutable Data Structures also provide a medium for maintaining isolated speculative state, in the form of the values written by an access function. The mutable reference to the data structure is modified by an atomic instruction and this causes the speculative state, created in isolation by the access function, to be transformed atomically into shared state.

Concurrent accesses to mutable data structures must be coordinated for two reasons. Firstly to protect the integrity of the data structure itself and secondly




(a)




(b)

Figure 3.1: **Insertion and deletion from an immutable binary tree.** The shaded vertices represent the path created by the operation. An ellipse with a double border represents a mutable reference to a version of the Immutable Data Structure. Version V0 of the immutable binary tree contains the elements $\{r, s, t, u, v, w\}$.
(a) Insertion of an element $q$ into an immutable binary tree creates version V1 containing the elements $\{q, r, s, t, u, v, w\}$.
(b) Removal of the element $w$ from an immutable binary tree creates version V2 containing the elements $\{r, s, t, u, v\}$.

to ensure the correct semantic order of operation. An Immutable Data Structure distinguishes between the structural consistency criteria of the data structure and the semantic consistency criteria of the application data. However, Immutable Data Structures do not offer a mechanism for ensuring the correct ordering of the effects of concurrent operations. A mechanism to ensure this ordering is presented in subsequent chapters.

## 3.3 Path Copying

We wish to determine the appropriate technique for implementing Immutable Data Structure in a concurrent execution environment. In a serial execution environment ease of implementation and performance are important considerations. However, in a concurrent execution environment ensuring the consistency of the data structure is the primary consideration. This section reviews the techniques for implementing Immutable Data Structures described in the literature.

The main contribution of this section is an examination of techniques for implementing Immutable Data Structures. This section focuses on the applicability of each technique in a concurrent execution environment.

### 3.3.1 Implementing Immutable Data Structures

Driscoll, Sarnak, Sleator, and Tarjan describe techniques for implementing Immutable Data Structures [DSST86]. This section examines how a complete immutable binary tree, with values on leaves, can be implemented using these techniques.

#### Full copying

The easiest technique for making a data structure immutable is to copy the entire data structure including the application values when any change is made. This technique is called naïve copying. A similar technique called full copying causes the structure to be copied while leaving the application values in place.

Figure 3.2 illustrates how the full copy technique is used to maintain a complete immutable binary tree.

The children of an immutable node should be copied before the node itself. Typically, a full copy of an immutable tree is implemented using a post-order traversal of the nodes. The performance overheads of full copying are significant so the technique has received little attention.

#### Path copying

The copying of the data structure can be restricted to the copying of those nodes that are modified by the operation. A path is a set of connected nodes linking the root with one or more leaves. A path copy operation creates a new path within

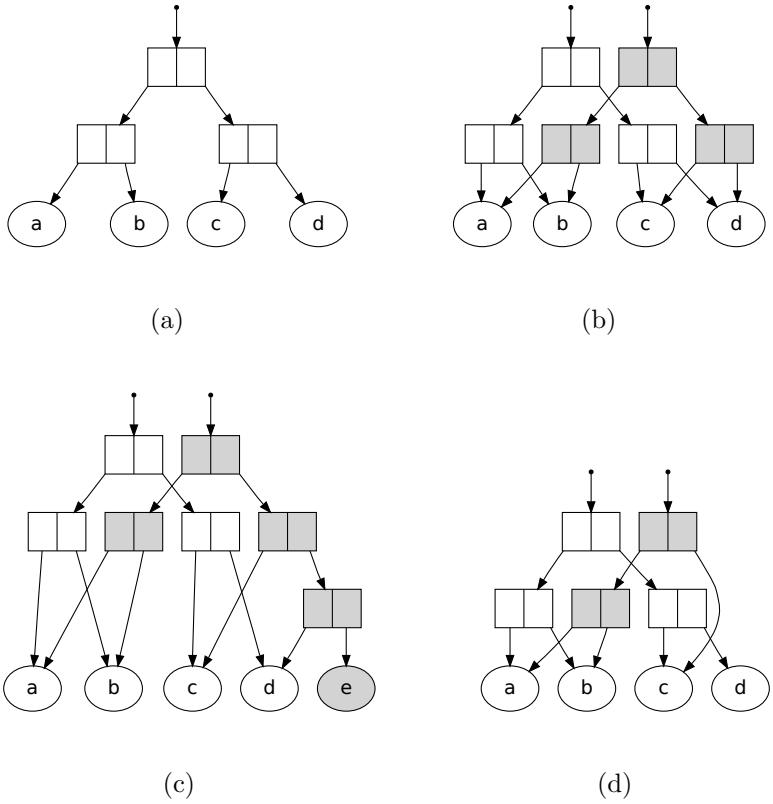



Figure 3.2: **Full copying technique.**
(a) Original complete binary tree.
(b) A full copy duplicates the data structure. The nodes created during the operation are shaded. Each node of the data structure is copied to create a new structure.
(c) The leaf $e$ is inserted into the data structure by copying all of the nodes and adding a new node.
(d) The leaf $d$ is removed from the data structure by copying all of the nodes except the parent of the leaf being removed.

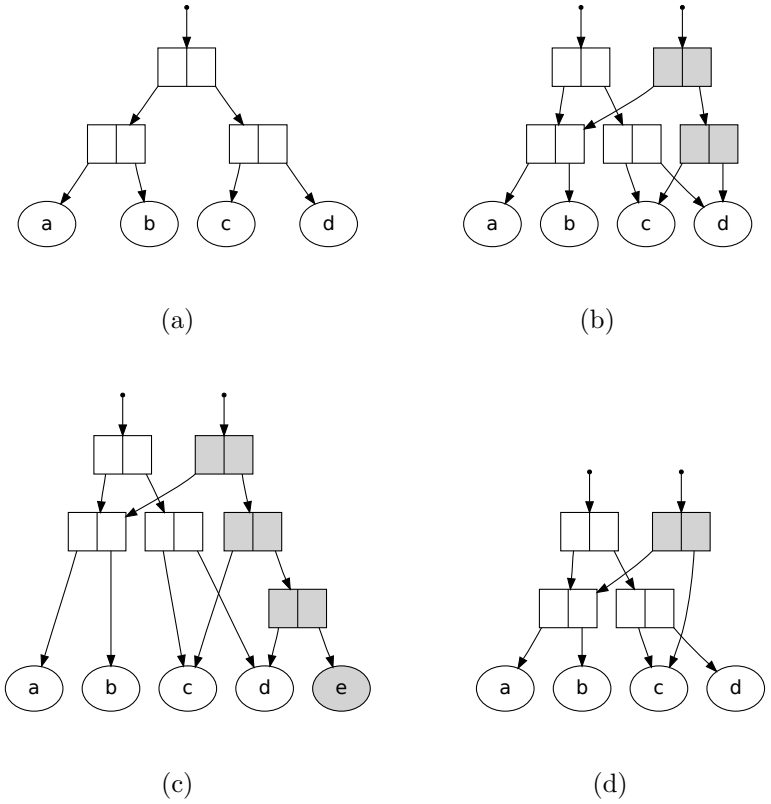



Figure 3.3: **Path copying technique.**
(a) Original complete binary tree.
(b) A leaf to root path copy. The nodes created during the operation are shaded. Each node on the path is copied to make a new data structure that has nodes in common with the original data structure.
(c) The leaf $e$ is inserted into the data structure by copying the path to a leaf and adding a new node.
(d) The leaf $d$ is removed from the data structure by copying the path to the parent of a leaf.

the data structure. A path copy operation may create a new path by copying an existing path from leaf to root or by selectively copying nodes in some other way.

Figure 3.3 illustrates how the leaf to root path copying technique is used to maintain a complete immutable binary tree.

In the context of functional programming languages an Immutable Data Structure maintained using the path copying technique is called a purely functional data structure.

Section 3.3.3 discusses path copying in more detail.

### Fat node

The fat node technique is based on the idea of recording changes to the data structure within the nodes themselves. The nodes modified by an operation are regarded as alternatives. Fat nodes can be implemented either by a list of alternative values for a node or by a list of alternative values for each pointer within a node.

Figure 3.4 illustrates how the fat node technique is used to maintain a complete immutable binary tree.

The fat node technique can be implemented by augmenting a node with an additional reference. If this reference is null then the node is the most recent and its children are interrogated to determine the path. If the reference is not null then the node or pointer has been superseded by a new value so the reference should be followed instead.

An access function determines the correct alternative based on a version number. Driscoll describes how alternatives can be associated with a range of version numbers [DSST86].

The fat node technique requires less copying than the path copy technique, because a path copy copies all of the nodes that the fat node copies together with additional nodes to the root.

The fat node technique is fairly easy to implement. However, long lists of alternative nodes can build up and these lists can degrade performance. To improve performance the lists can be split using the path copying technique but this comes at the cost of additional implementation complexity.

The fat node technique has received attention as the basis of maintaining persistent data structures in computational geometry [WR08].

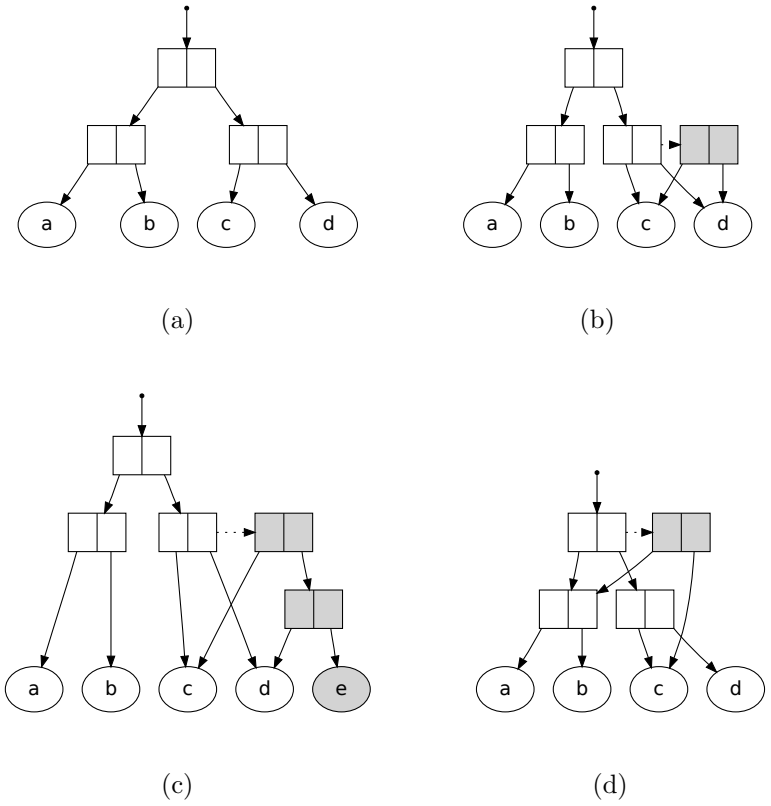



Figure 3.4: **Fat node technique.**
(a) Original complete binary tree.
(b) A node is added to the fat node. New nodes are created as needed. The nodes joined by the horizontal dotted line are regarded as part of a single fat node. The fat node is a list of past values for the node. The nodes created during the operation are shaded.
(c) The leaf $e$ is inserted into the data structure by the creation of an alternative value for the parent of the leaf $c$.
(d) The leaf $d$ is removed from the data structure by the creation of an alternative value for its grandparent.

#### Node copying

The node copying technique is similar to the fat node technique except that a finite number of alternatives are maintained within a node instead of in a list. These can either be alternative values for pointers or alternative values for the node. When the node becomes full it may be split. Cole describes the node copying technique in relation to persistent data structures [Col86].

Figure 3.5 illustrates how the node copying technique is used to maintain a complete immutable binary tree.

A node contains alternative values for the references to its children which are initialised to null and used if set. Each node in the data structure contains a set of alternative values. Typically, the alternative pointers are set to null when the node is created and subsequently modified when a path is modified. When a node becomes full it is split into multiple nodes by creating new paths to nodes in a similar manner to the path copying technique.

The path copying technique requires that a new path containing copies of all the nodes from leaf is created during each operation, whereas the node copying technique copies only part of the path, so the node copying technique often performs better than the path copying technique. Typically, two alternative values are stored in a single cache line so the overhead of checking whether the alternative value is in use is low.

The node copying technique is more difficult to implement than the fat node technique because node splitting requires that the path copying technique must also be implemented.

The node copying technique has also received attention as the basis of maintaining persistent data structures in computational geometry [ST86].

### 3.3.2 Supporting concurrent access

It is useful to distinguish between an immutable memory location and one that is singly-assigned. An immutable memory location is always observed to contain a constant value and it is unreachable until it is assigned. A singly-assigned memory location can be observed to contain either no value or a value that is constant once set.

In the context of concurrent execution an immutable memory location can be safely written because it is unreachable in its uninitialised state, whereas a shared

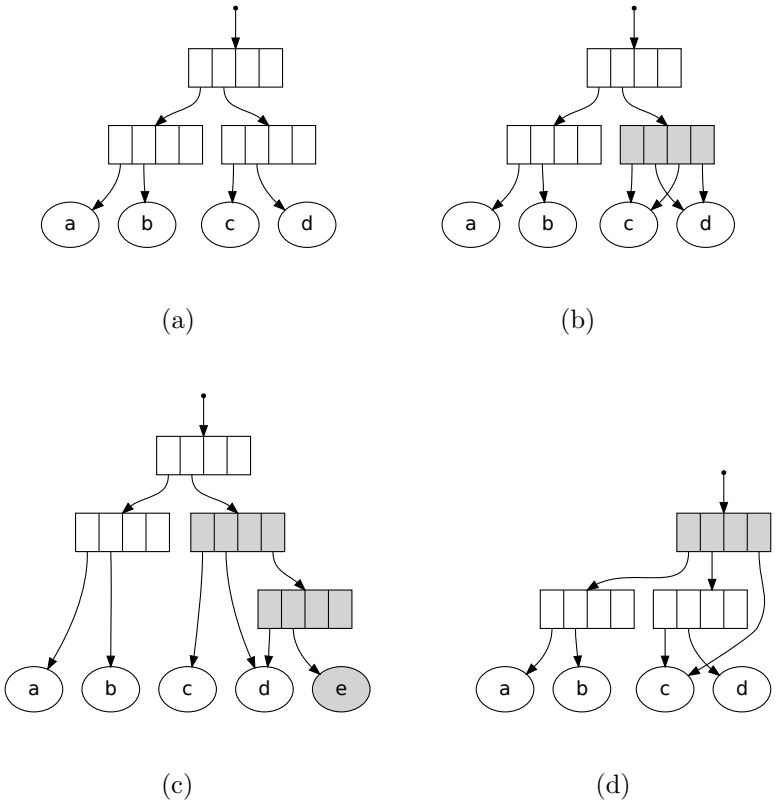



Figure 3.5: **Node copying technique.**
(a) Original complete binary tree.
(b) A node is copied. The nodes affected by the operation are shaded. A node contains a left and right pointer and an alternative left and right pointer. In order to find a leaf a test is done to determine whether the alternative pointers are null and if not the new path is taken.
(c) The leaf $e$ is inserted into the data structure by using the alternative right pointer of the parent of the leaf $c$.
(d) The leaf $d$ is removed from the data structure by using the alternative right pointer of its grandparent.

singly-assigned memory location must always be written by an atomic instruction to prevent race conditions.

An immutable memory location can be safely cached by multiple processors without requiring any mechanism to ensure cache coherence, whereas a singly-assigned memory location may not because the location can be accessed and cached in an uninitialised state and is, in effect, mutable. A cache coherency mechanism is required to ensure that all processors observe the correct value of a singly assigned memory location.

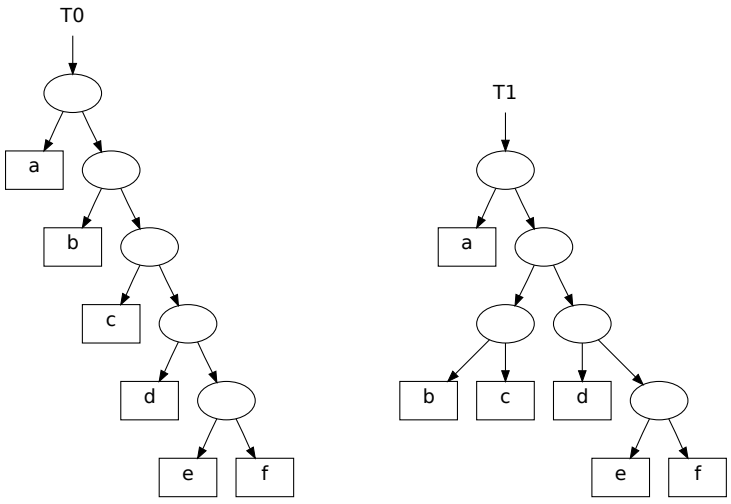
The fat node and node copying techniques both require that a singly-assigned memory location is checked for a null value within the data structure. The fat node technique checks whether the alternative value is null and the node copying technique checks whether a reference to an alternative node is null. In a concurrent execution environment the fat node and node copying techniques require that these shared singly-assigned locations must be modified by an atomic hardware instruction. Coherent caches and memory barriers are also required to support these singly-assigned values. These restrictions mean that the fat node and node copying techniques are not suitable for implementing Immutable Data Structures in a concurrent execution environment.

The full copy and path copy techniques do not rely on singly-assigned memory locations. In the context of concurrent execution the path copying and full copying techniques share the advantage, over the other techniques, that modifications to the data structure can be made to appear atomic because a new version of the data structure only becomes visible to other processors when the root node is written. They also share the advantage that all values are immutable so they can be cached in processor-unique caches without requiring that these caches implement a coherency protocol.

### 3.3.3 Path copying transformations

Okasaki relates trees to number systems [Oka98]. We can use this relationship to show that the path copying technique can be used to transform the topology of an immutable binary tree arbitrarily.

Let S be the set of expressions {{a.b.c.d.e.f},{a.(b.c).d.e.f},...} in which the letters a to f are in the same order. The set S corresponds to the set of possible topologies of the binary tree in which the leaves are in the same order from left to right. If the binary operator '.' is regarded as being associative then all of the




(a) (b)

Figure 3.6: **Bracket operations.**
(a) A degenerate tree T0 corresponding to the expression {a.b.c.d.e.f} in which the binary operator '.' acts on the values a to f, which are represented by the leaves in order from left right.
(b) A tree T1 corresponding to an expression in which the brackets have been re-arranged to cause some operations to take precedence over others. The operation acting on b and c takes precedence and the expression can be written as {a.(b.c).d.e.f)}.

expressions in the set S are equivalent. This topological equivalence is the basis of tree balancing.

For any expression in S there is a transformation, essentially removing brackets, that maps it onto the expression {a.b.c.d.e.f}. Similarly, for any expression in S there is a transformation, essentially adding brackets, that maps the expression {a.b.c.d.e.f} onto it.

Figure 3.6 illustrates bracket operations acting on a degenerate tree.

For any tree topology there is a transformation that maps it onto an equivalent degenerate tree. Similarly, for any tree topology there is a transformation that maps the degenerate tree onto it. These transformations can be broken down into a finite sequence of steps, each of which corresponds to adding or removing brackets from the expression.

By using a similar informal argument we can show that a value can be added

to or removed from an expression and that this corresponds to the insertion and deletion of leaves in a degenerate tree.

### Implementing topological changes using path copying

The operations that add or remove brackets from an expression correspond to topological transformations of the tree. These transformations can be implemented using the path copying technique.

Figure 3.7 illustrates bracket operations implemented by path copying.

Given a degenerate tree it is possible to transform it into an equivalent tree with any topology without altering the original by using the path copying technique. Similarly, it is possible to transform a tree with any topology into an equivalent degenerate tree without altering the original.

It is interesting to note that these trees really are equivalent. The trees T0 and T1 in figure 3.6 are equivalent in the vague sense that an in-order traversal returns the same values in the same order, whereas the trees T2 and T3 in figure 3.7 are equivalent because the same leaves are shared by both trees, they are in the same order and they exist at the same moment in time.

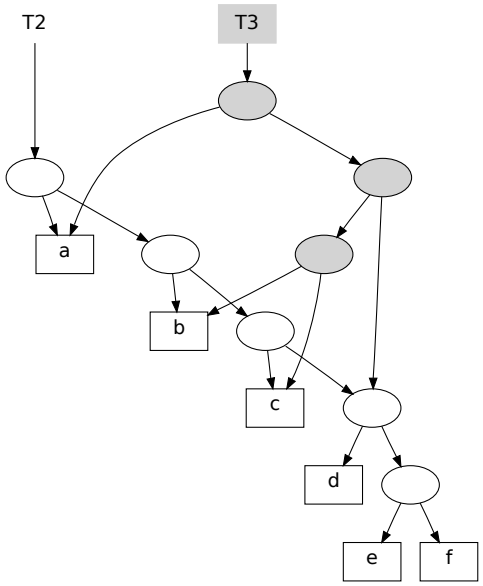
### Implementing structural changes using path copying

The operations that add or remove values from an expression correspond to structural transformations of the tree.

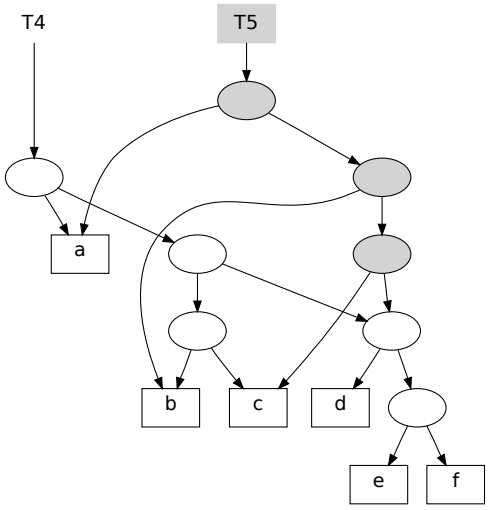
A value can be inserted into an expression at any point and correspondingly a leaf can be inserted into the degenerate tree at any point to create a new degenerate tree. A value can be removed from an expression at any point and correspondingly a leaf can be removed from a degenerate tree at any point to create a new degenerate tree.

Figure 3.8 illustrates how these transformations can be implemented using the path copying technique.

By using the path copying technique it is possible to make structural changes to a tree, by inserting or removing an arbitrary number of leaves, without altering the original.

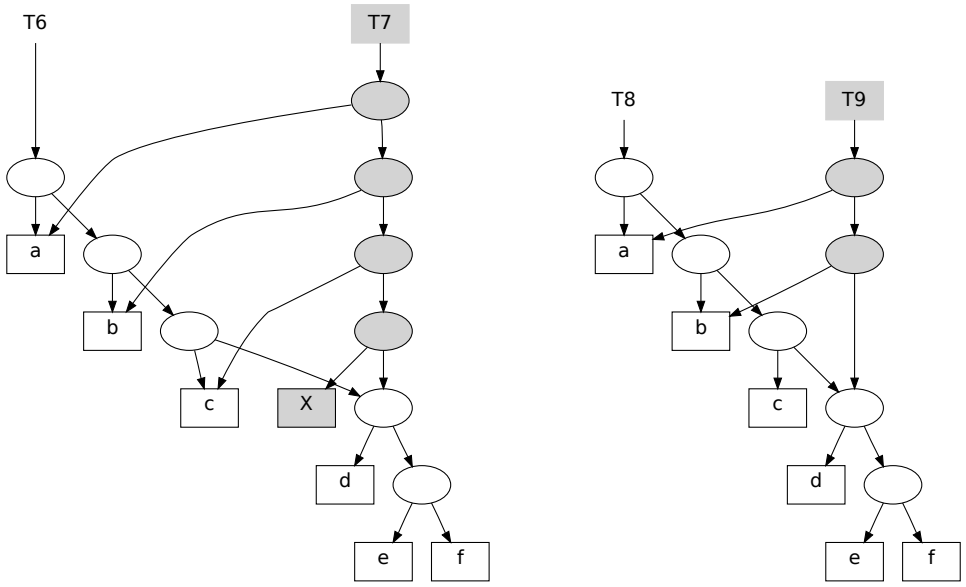



(a)




(b)

Figure 3.7: **Immutable add bracket and remove bracket operations.**
(a) A tree corresponding to an expression in which the precedence of one operation has been elevated. Degenerate tree T2 representing the expression {a.b.c.d.e.f} and the tree T3 representing the expression {a.(b.c).d.e.f}.
(b) A tree corresponding to an expression in which the precedence of one operation has been reduced. Tree T4 representing the expression {a.(b.c).d.e.f} and the degenerate tree T5 representing the expression {a.b.c.d.e.f}.




(a) (b)

Figure 3.8: **Immutable insert and delete operations.**
(a) Insertion of an item into the degenerate binary tree. Degenerate tree T6 representing the expression {a.b.c.d.e.f} and the degenerate tree T7 representing the expression {a.b.c.X.d.e.f}.
(b) Removal of an item from the degenerate binary tree. Degenerate tree T8 representing the expression {a.b.c.d.e.f} and the degenerate tree T9 representing the expression {a.b.d.e.f}.

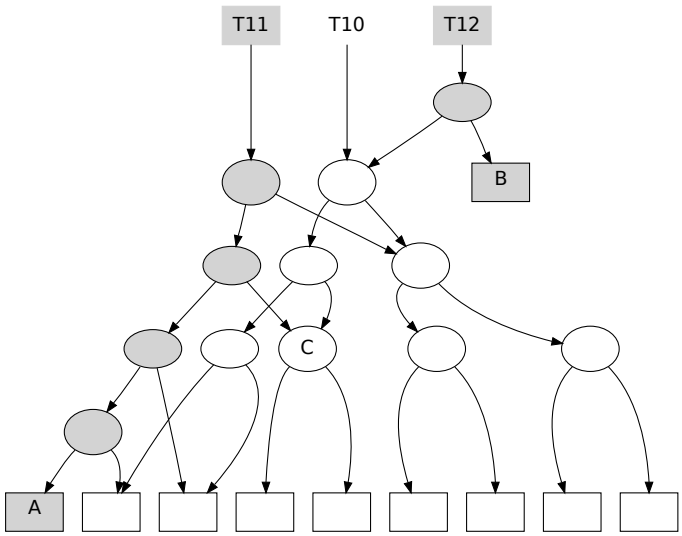



Figure 3.9: **The leaf to root path copying technique** is distinct from other path copying techniques because it preserves the position of nodes relative to the root.
A new leaf A is added to tree T10 by a leaf to root path copy to create tree T11. The path copy preserves the position of nodes that are not copied relative to the root. For example, the node C is the right child of the root's left child in both versions.
A new leaf B is added to tree T10 by another path copying technique to create tree T12. The relative positions of the existing nodes are not preserved.

### 3.3.4 Previous work

A leaf to root path copy operation preserves the position of existing nodes relative to the root, but not all path copy operations preserve the relative position of existing nodes.

The literature does not make a distinction between a leaf to root path copy or the creation of a new version through the construction of a new path in some other way. Driscoll describes path copying without detailed consideration of how it is achieved [DSST86]. Okasaki describes many copying optimisations which are not leaf to root copies [Oka98].

Figure 3.9 illustrates the distinction between leaf to root path copy and other path copying techniques.

## 3.4 Binary Trees

There are no publicly available libraries of Immutable Data Structures implemented in imperative programming languages. This section describes the design of a general purpose Immutable Data Structure. This flexible design can be specialised to implement a variety of ADTs. The topology of the data structure is hidden from the program so it can be balanced independent of the ADT that it implements.

In a functional programming language immutable values are maintained in purely functional data structures, such as those described by Okasaki [Oka04]. Purely functional data structures might appear to be a starting point for developing Immutable Data Structures. However, functional programming languages permit the expression of a function in terms of immutable data, whereas the evaluation of a function typically relies on mutable data. In some cases a functional programming language compiler implements a purely functional data structure as a mutable structure. In a concurrent execution environment it is the *actual* immutability of values used during the execution that matters rather than the *appearance* of immutability presented to the programmer by the programming language.

The main contribution of this section is the development of a general Immutable Data Structure that can be used to maintain speculative and shared state. This section focuses on design flexibility and subsequent sections show how the structure can be specialised to conform to a variety of ADTs.

### 3.4.1 A flexible Immutable Data Structure design

A general purpose Immutable Data Structure should be flexible enough to present a variety of familiar ADTs to the program. Design flexibility is a vague term but we take it to mean three things. Firstly, we prefer the simplest most general solution. In practice, this means designs that contain no special cases. Secondly, we prefer to hide details of the data structure implementation from the application. In practice, this means making details of the topology inaccessible to the ADT. Thirdly, we delay performance optimisations until the final stages of implementation.

### 3.4.2 The Canonical Binary Tree

The proposed solution is an immutable binary tree that can be specialised to conform to a particular ADT. We call this structure the Canonical Binary Tree because it is an immutable binary tree reduced to the simplest and most significant form possible without loss of generality. This section states the decisions on which the design of the Canonical Binary Tree is based.

#### Why a binary tree?

A tree in which each node has many children can be shallower than a binary tree containing the same number of leaves. It is common for purely functional data structures to be based on shallow trees so that access times are minimised.

For example, the Clojure language implements a number of purely functional data structures internally. These structures are based on a 32 bit hash array mapped trie. Each node of the trie has up to 32 children so the structure is shallow and permits fast access. Bagwell describes the implementation of these data structures in detail [Bag01]. A hash array mapped trie is a complex structure optimised for good performance on modern computer hardware, but it is difficult to implement. Clojure offers just a few Immutable Data Structures as primitives and the language offers no control over the implementation of the underlying data structure. At the time of writing a new ADT is under development by the Clojure community. Hickey describes the performance benefits of the hash array mapped trie and the significant work involved in implementing ADTs based upon it [Hic11].

In a concurrent execution environment access time is not the most important design consideration and optimisation can be deferred to a later stage in the design process. The binary tree has the simplest possible structure and offers the greatest design and implementation flexibility.

#### Why associate values exclusively with leaves?

A Canonical Binary Tree contains both structural information and application values. Structural information is necessarily associated with the nodes but application values can be associated either exclusively with leaves or with both leaves and nodes, which we refer to as vertices. A tree that associates application values exclusively with leaves can hide its topology.

For example, the priority queue ADT associates a priority with an application value. It is common for a priority queue to be implemented as a binary heap in which both a priority and an application value are associated with each vertex. Both the ephemeral priority queue considered by Sedgewick [Sed98] and the purely functional priority queue considered by Okasaki [Oka04] associate an application value with a vertex because a vertex can be accessed more quickly than a leaf. When a priority queue is implemented by a binary heap the highest priority vertex can be accessed in $O(1)$ time and insertion into the queue takes $O(log_2(n))$ time. However, when a priority queue is implemented by a tree with application values associated exclusively with leaves the access time for all operations is $O(log_2(n))$.

The Canonical Binary Tree associates application values exclusively with leaves. All functions access leaves so the amortised access time is:

$$O(log_2(n))$$

This amortised access time is identical to that of an ephemeral binary tree with application values maintained exclusively by leaves.

### Why separate keys and annotations?

A key is an argument to a function of a data structure, whereas a vertex annotation is a value used to navigate a path through the tree. Usually, annotations and keys are of the same type and annotations are accessible to the program.

The Canonical Binary Tree design separates the concepts of keys and annotations. Annotations are not accessible to programs so the topology of the tree can be altered independent of the ADT being implemented.

For example, an associative data structure in which all values are reachable, such as a map, is typically distinguished from one in which not all application values are reachable, such as a deque. However, the front and back functions of a deque can be regarded as a query function that takes as its access argument a binary key indicating which end of the queue it acts upon.

By separating the concept of the annotation from the key all ADTs can be regarded as associative. The Canonical Binary Tree treats all ADTs as associative and hides the details of the annotations from the calling program.

### Why fix the comparison function?

The function that determines the annotation of a node given the annotations of its children is referred to as the annotator and the operation that determines which of the children of a node is on the path is called the comparison. The annotator function specialises the Canonical Binary Tree so that it conforms to a particular ADT. The Canonical Binary Tree uses the same comparison function for every ADT.

For example, a path through a Binary Search Tree can be determined by a comparison function that causes the right child of a node to be selected if the access argument is greater than its annotation. This causes an in-order traversal to return application values in ascending order of the access argument used to insert them. The order of the elements returned by an in-order traversal can be reversed either by using a different comparison function or by inserting the values using a different annotator.

The Canonical Binary Tree fixes the comparison function to reduce the amount of information that must be specified to specialise it to conform to a particular ADT.

### Why maintain a sentinel leaf?

There is a distinction between a data structure that is empty and a data structure that does not exist. This is particularly important for data structures that are accessed concurrently.

An empty Canonical Binary Tree contains a sentinel leaf which is always present within the tree. We adopt the convention that the sentinel is always the right-most leaf of the tree.

### Which access functions should the Canonical Binary Tree implement?

The Canonical Binary Tree implements only the access functions: *create*(), *insert*(), *query*(), *delete*() and *empty*(). The interface functions required by common ADTs, such as $Top()$, $Front()$ etc. are implemented by wrapper functions.

The *create*() function creates a new data structure containing only the sentinel. Its parameters specify the appropriate sentinel annotation for the ADT being implemented and an application value. The function allocates storage for the root and returns a reference to it. The root is initialised with a reference

to the sentinel. References to the root and the sentinel are maintained by the program.

The *query*() function returns an application value. The function accepts an access argument, a reference to the root and a reference to the sentinel as its parameters. It is ADT agnostic and does not require a specialising function as a parameter. Its access argument will always match a single leaf within the tree. When the tree is empty it returns the application value of the sentinel.

The *insert*() function always succeeds in inserting a leaf into the tree and has no return value. The function accepts a specialising annotator function, an access argument, a reference to the root and a reference to the sentinel as its parameters.

The *delete*() removes a leaf from the tree unless it is empty and has no return value. The function accepts a specialising annotator function, an access argument, a reference to the root and a reference to the sentinel as its parameters. The sentinel cannot be deleted and an instance of the Canonical Binary Tree persists until the program terminates, so there is no function to delete an entire data structure.

The *empty*() function compares the address of the sentinel with the address of the root node, both of which are maintained by the program and passed as parameters.

### 3.4.3 Previous work

Sedgewick provides a comprehensive guide to important ephemeral data structures [Sed98]. Okasaki provides a comprehensive guide to purely functional data structures [Oka98].

Tarjan describes methods of amortised time analysis called the Banker's and Physicist's methods [Tar85]. Okasaki adapts these analyses to purely functional data structures [Oka98]. The Banker's method associates credits and debits with short and long paths in the data structure respectively. The analysis balances the debits and credits to determine the effective cost of an operation. The Physicist's method describes a function mapping each element in the data structure onto a real number called its potential. The analysis balances the positive and negative potential of accesses to particular elements to determine the effective charge of an operation. These analyses are more complicated than ours because the ADTs presented are tightly coupled to the data structures that implement them.

Okasaki focuses on the path copying technique and the diagrams in the book imply that the programmer should visualise path copying when thinking about the structures. However, in a functional programming language a data structure is specified at a high level of abstraction and how the language compiler implements the structure is not specified. In some cases path copying is used by the generated code but this is compiler dependent. A structure that appears immutable when described in a functional programming language might be compiled to a mutable structure to improve performance.

Moss describes a set of benchmark applications that can be used to assess the performance of purely functional data structures [Mos99].

Prior to this thesis there were no publicly available libraries of Immutable Data Structures implemented in an imperative programming language. We do not know of any previous attempts to produce such a library.

Persistent Data Structures implemented in an imperative programming language are typically bespoke solutions to problems in algebraic geometry or version control. Sarnak and Tarjan describe how a persistent data structure can be used to solve the planar point location problem in computational geometry [ST86]. Pluquet, Langerman, Marot and Wuyts describe how to construct a partially persistent data structure in C++ to solve the same planar point location problem [PLMW08]. These persistent data structures use the fat node technique so they are not immutable.

Parrish et al. describe a class based implementation of persistence in C++ [PDC+98]. The problem that Parrish addresses is one of transforming a general application class into a persistent class. The resulting data structure is immutable but new versions can only be created by copying the entire object.

The C++ Standard Template Library (STL) contains several associative ADTs that are usually implemented by a balanced red-black tree [Jos99]. The STL separates the concerns of the ADT from those of the data structure that implements it. The STL separates the ADT from the balancing process. STL iterators separate the ADT from the process of traversing the tree. STL allocators separate the ADT from the memory management processes so the data structure implements a container.

Hinze and Paterson describe how a similar separation of concerns can be applied to a purely functional data structure [HP05]. Hinze describes a general technique for creating Immutable Data Structures in a functional programming

language. This technique has not previously been explored in the context of imperative programming. Hinze reduces the amortised access time of a binary tree by adding a central spine, to create a so-called finger tree. However, the spine is just an access time optimisation. Hinze describes how a specialising function can be used to make an immutable binary tree conform to a particular ADT. Hinze shows how monoid functions, which are associative functions with an identity, can be used to specialise a binary tree.

A finger tree is statically specialised to conform to a particular ADT, whereas the Canonical Binary Tree is dynamically specialised. The set of access functions associated with each structure implemented by a finger tree is ADT dependent, whereas the Canonical Binary Tree presents a basic set of functions that can be adapted to implement a particular ADT.

Hinze's design is based on an Immutable Data Structure that requires both a function to determine the annotation of a node given its children and a comparison operation to determine the path, whereas the Canonical Binary Tree requires only one specialising function.

Finally, Hinze does not make a distinction between an empty tree and a non-existent tree, whereas the Canonical Binary Tree maintains a sentinel to make this distinction.

## 3.5 Abstract Data Types for Immutable Data

The design of a data structure is normally tightly coupled with the ADT being implemented. The property of immutability permits the development of a general technique for implementing an ADT. The technique has not previously been explored in the context of an imperative programming language. The Canonical Binary Tree can be made to conform to many different ADTs by specifying a specialising function as a first order parameter. Functions acting on the Canonical Binary Tree, including those supporting concurrent execution, can be implemented independent of the ADT.

The main contribution of this section is the development of an Immutable Data Structure that separates the concerns of the structure from those of the ADT to which it conforms. This section focuses on techniques for specialising the Canonical Binary Tree so that a mechanism to allow concurrent access can be implemented independent of the ADT.

### 3.5.1 Priority queue

A priority queue associates a priority with a data value so that the value associated with the highest priority can be recovered. Priority queues are used to schedule operating system tasks and to solve the selection problem, which is to return the $k$th largest element from a set of elements.

A priority queue has a $Push()$ function to insert a value with an associated priority into the structure. It has a $Top()$ function that returns the value with the highest priority and a $Pop()$ function that removes that value. It is conventional to regard low numbers as high priorities.

Hinze and Paterson describe an implementation of a purely functional priority queue based on a min-tree [HP05]. The min-tree is a type of tournament tree in which the annotation of a leaf is the priority and the annotation of a node is the minimum annotation of its children. This property causes the annotation of the root node to be equal to the lowest priority of any leaf. A path from the root to the leaf with the highest priority is found by examining the annotation of the root node and then repeatedly choosing the child node with matching priority until a leaf is reached.

Figure 3.10 illustrates an example of a min-tree.

The min-tree corresponds to a mathematical expression in which the minimum

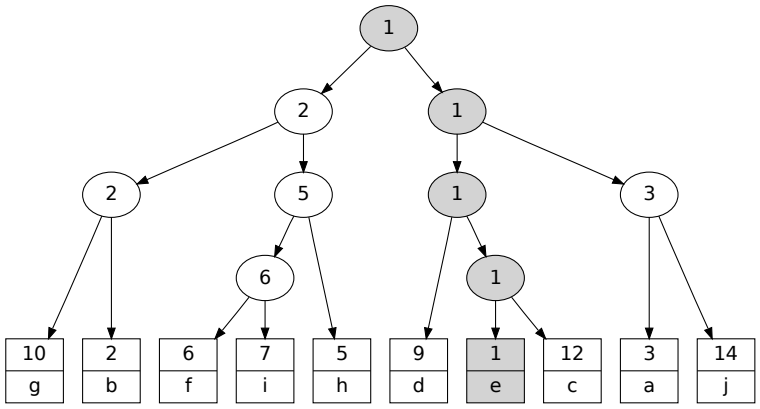



Figure 3.10: **Example Min-tree** containing the priority value pairs $\{1 \mapsto e, 2 \mapsto b, 3 \mapsto a, 5 \mapsto h, 6 \mapsto f, 7 \mapsto i, 9 \mapsto d, 10 \mapsto g, 12 \mapsto c, 14 \mapsto j\}$. The shaded vertices illustrate the path to the value with the highest priority.

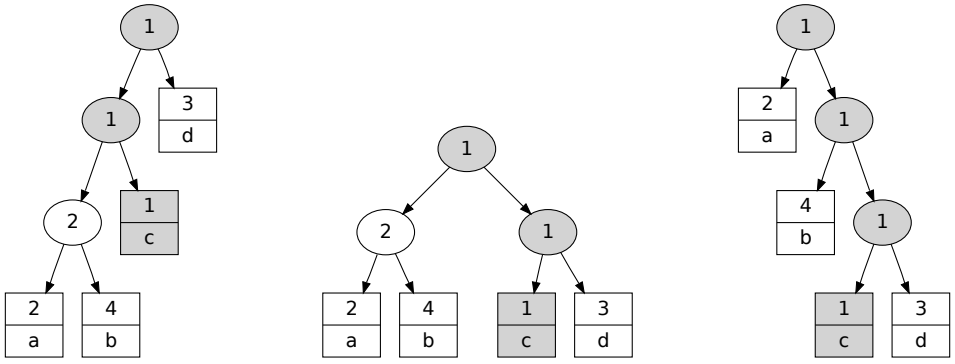
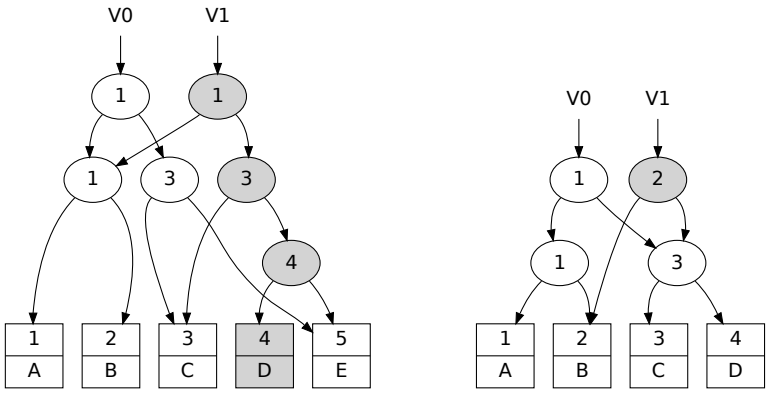



Figure 3.11: **Associativity property of a min-tree.** Min-trees with different topologies maintain the property that the root node is annotated with the minimum annotation of any leaf. The shaded vertices illustrate the path to the highest priority element.




(a) (b)

Figure 3.12: **Insertion and removal of an element in a min-tree.**
(a) Insertion of an element into an immutable min-tree. Version V0 contains the priority value pairs $\{1 \mapsto A, 2 \mapsto B, 3 \mapsto C, 5 \mapsto E\}$. The operation $Push(4 \mapsto D)$ creates version V1 containing the priority value pairs $\{1 \mapsto A, 2 \mapsto B, 3 \mapsto C, 4 \mapsto D, 5 \mapsto E\}$. The path created by the operation is shaded.
(b) Removal of an element from an immutable min-tree. Version V0 contains the priority value pairs $\{1 \mapsto A, 2 \mapsto B, 3 \mapsto C, 4 \mapsto D\}$. The operation $Pop()$ creates version V1 containing the priority value pairs $\{2 \mapsto B, 3 \mapsto C, 4 \mapsto D\}$. The path created by the operation is shaded.

function is applied to the priorities. The minimum function is both associative and commutative so the min-tree maintains the property that the annotation of a node is equal to the minimum priority of any leaf in the subtree that it suspends, regardless of the topology of that subtree.

Figure 3.11 illustrates the associativity property of the minimum function.

The $Push()$ function inserts a value into the min-tree by creating a new leaf containing the value and annotated by the priority. A new path from the root to this leaf is created by path copying. The annotation of each node on the path is set to be the minimum of the annotations of its children. Path copying creates an entirely new path so the annotations of existing nodes are unaffected by the operation. The $Push()$ function can insert a leaf anywhere in the tree because the minimum function is commutative.

The $Pop()$ function removes the value with the highest priority from the immutable min-tree by creating a new path which makes the leaf with the highest priority unreachable. The root node is annotated with the next highest priority.

Figure 3.12 illustrates the insertion and removal of an element from a min-tree.

Figure 3.13 illustrates the growth of an immutable min-tree. Successive leaves are added through a process of path copying. The properties of the min-tree are preserved by each version.

The min-tree requires that the children of a node are examined when determining the path. If the path could be determined without accessing the children then the number of nodes accessed when traversing a path would be approximately halved.

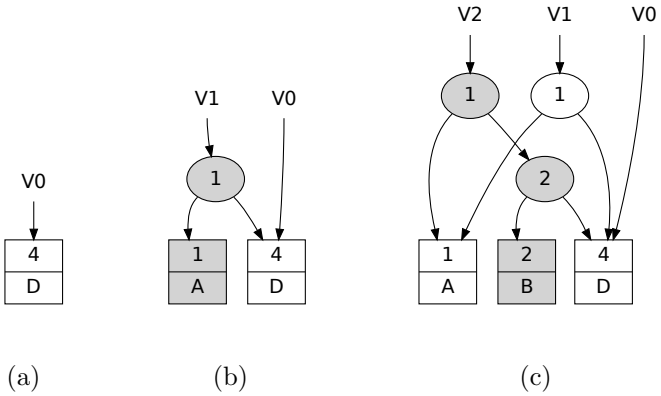
In the context of concurrent execution the benefit of determining the path without accessing the children is significant because nodes that are read while traversing the path must be recorded to ensure correct concurrent execution. It is therefore beneficial to restrict node access to those nodes actually on the path.

The min-tree requires that both the comparison and the annotator function are supplied as specialising functions. The annotation of the root node is followed to the leaf. This requires a special comparison operation to reach the highest priority element because the value of the annotation of the root node must be retained while following the path. If the path could be determined without specialising the comparison operation then the amount of information required to describe the data structure would be reduced.

The min-tree does not specify a representation of the empty priority queue. If a representation of the empty priority queue were specified it would be possible to distinguish an empty priority queue from a non-existent queue.

In a typical priority queue implementation the $Pop()$ function behaves differently when removing the last remaining element in a data structure because the data structure is subsequently empty. In the concurrent execution environment the status of a data structure between function calls is unknown so it is necessary that the data structure represents and includes checks for an empty queue in access function.

The functions $Top()$, $Push()$ and $Pop()$ are specific to the priority queue ADT. If these functions could be specified as adaptations of the access functions of the Canonical Binary Tree then it would be possible to abstract the priority queue ADT from the data structure that implements it.



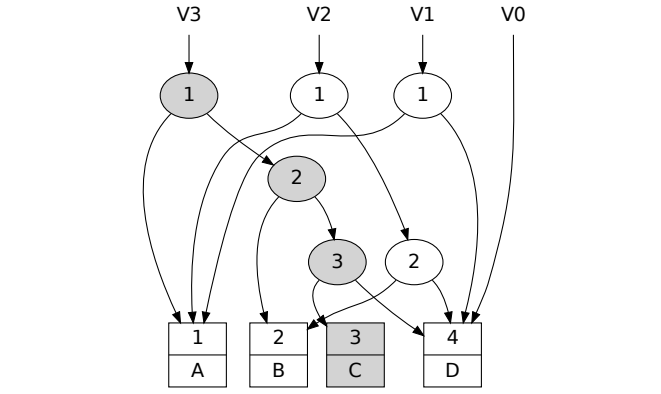




Figure 3.13: **Animation showing the growth of a min-tree** through a series of insertions. A new version of the data structure is created by each operation. In each case the path created by the operation is shaded.
(a) Initial data structure containing the priority value pair $4 \mapsto D$.
(b) After $Push(1 \mapsto A)$
(c) After $Push(2 \mapsto B)$
(d) After $Push(3 \mapsto C)$.

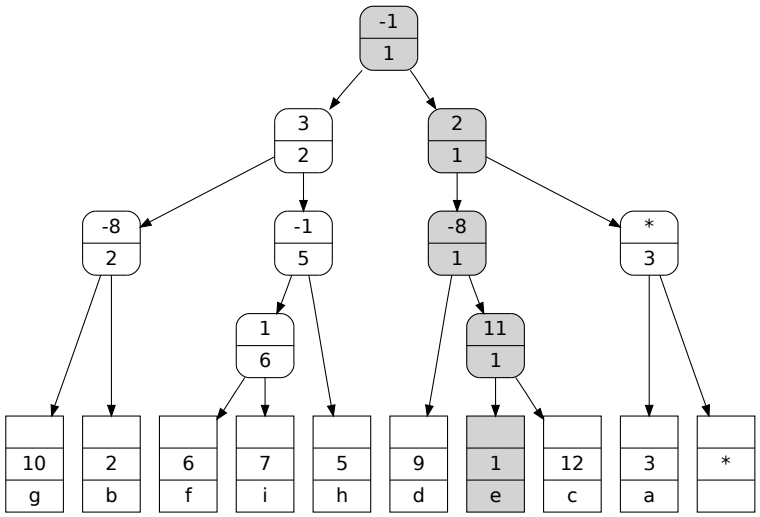



Figure 3.14: **Example Directed min-tree** containing the priority value pairs $\{1 \mapsto e, 2 \mapsto b, 3 \mapsto a, 5 \mapsto h, 6 \mapsto f, 7 \mapsto i, 9 \mapsto d, 10 \mapsto g, 12 \mapsto c\}$. The shaded vertices illustrate the path to the highest priority element. The sentinel is the right-most leaf of the tree. The first annotation is shown above the second annotation.

### 3.5.2 Directed min-tree

A new data structure, the directed min-tree, implements the priority queue ADT and addresses the shortcomings of the min-tree.

The min-tree suffers from the problem that the annotations of both of the children must be examined to determine the path. The directed min-tree solves this problem by regarding the annotation as a pair of values. One value contains the minimum value of the child annotations and the other contains an indicator as to whether the left or right child of a node has a lower annotation value.

Figure 3.14 illustrates the annotations of a directed min-tree. The annotation pair contains two values that we call the first and second annotations of the node. In the figure the first annotation is shown above the second annotation. The first annotation is calculated by subtracting the second annotation of the left child from the second annotation of the right child. The second annotation is the minimum of the second annotations of the children. The first annotation of a leaf is not used and its second annotation is the priority associated with the application value.

Only the first annotation of a node is examined when traversing the path. This annotation indicates which child has the minimum second annotation. The path to the leaf with the highest priority can be found by comparing the first annotation of each node with zero. If it is greater than zero then the left child is on the path, so to determine the path it is only necessary to examine the annotations of nodes on the path.

The min-tree suffers from the problem that both the comparison operation and the annotator function must be supplied as specialising functions, whereas the directed min-tree requires only that the annotator function be specified. The comparison function is regarded as a feature of the Canonical Binary Tree common to all ADTs.

The comparison function and the path determination process are the same regardless of the ADT being implemented, so the *query*() function is ADT agnostic. For example, the *Top*() function is implemented as a Canonical Binary Tree *query*() function with an access parameter of zero. Path determination is a common feature of the *query*(), *insert*() and *delete*() functions so the implementation of each function is simplified by making path determination ADT agnostic.

The min-tree suffers from the problem that the annotation of the root node must be an argument to the comparison function for every node on the path, whereas the directed min-tree does not treat the root node as special and does not require an annotation to be retained while determining the path.

The min-tree suffers from the problem that it does not specify a representation of the empty priority queue, whereas the directed min-tree contains a sentinel that can be used to distinguish an empty data structure from a non-existent data structure. The sentinel is annotated in such a way that it cannot be removed from the tree.

The access parameter of the *insert*() function identifies a leaf in the data structure. When a new leaf is inserted to the min-tree it can be inserted either to the left or the right of this leaf because the minimum function is commutative so it does not matter on which side of the path the insertion takes place. However, the Canonical Binary Tree requires that the sentinel is always the right most leaf. To ensure this, the insert function always inserts a new leaf to the left of the path identified by the access parameter. When the tree is created the sentinel is the only leaf and the *insert*() function always inserts leaves to the left of the path,

| Canonical Binary Tree specialisation | |
|---|---|
| $annotator(< a, b >, < c, d >)$ | $< d - b, min(b, d) >$ |
| $identity$ | $<, \infty >$ |

| API function | Canonical Binary Tree access function |
|---|---|
| $Push(priority)$ | $insert(priority)$ |
| $Pop()$ | $delete(0)$ |
| $Top()$ | $query(0)$ |

Table 3.1: **Directed min-tree implementation.** The Canonical Binary Tree can be specialised to implement a directed min-tree and its access functions can be adapted to present a priority queue ADT to the application.

so the sentinel will always remain the right-most leaf of the tree.

The sentinel must be annotated in such a way that the left child of its parent is always chosen by the $query()$ and $delete()$ functions because the sentinel is unreachable by $query()$ and cannot be removed by $delete()$. The second annotation of the sentinel is infinity which causes its parent to have a first annotation value of infinity so a path through the directed min-tree will include the sentinel only when the tree is empty. In practice, the sentinel is annotated with the highest value of the data type of the annotation.

The min-tree suffers from the problem that the ADT cannot be completely abstracted from the data structure that implements it, whereas the directed min-tree can be implemented by specifying access arguments to adapt the functions of the Canonical Binary Tree.

Table 3.1 contains all of the information needed to specialise the Canonical Binary Tree so that it implements the priority queue ADT. The annotator function returns the annotation of a node given the annotations of its children. The identity is the annotation of the sentinel. The $Push()$ function is implemented by the $insert()$ function of the Canonical Binary Tree. The $Pop()$ function is implemented by the $delete()$ function, The value with the highest priority will always be found by specifying an access argument of zero as the access parameter of the $delete()$ function. Similarly, the $Top()$ function is implemented by a $query()$ with an access argument of zero.

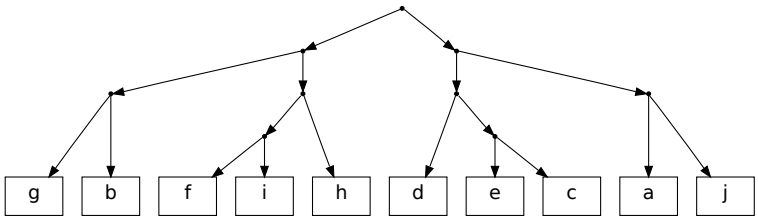



Figure 3.15: **Example Deque** containing the values $\{g, b, f, i, h, d, e, c, a, j\}$.

### 3.5.3 Deque

A deque data structure contains an ordered list of elements and only permits access to those elements at either end of the list. The functionality of the data structure can be further restricted to implement a queue or stack.

A deque is regarded as having a front and a back. The $Push\_front()$ function inserts a value onto the front of the deque. The $Front()$ function returns that value. The $Pop\_front()$ function removes the value at the front of the deque. The corresponding functions $Push\_back()$, $Back()$ and $Pop\_back()$ affect the back of the deque.

Hinze and Paterson describe an implementation of an immutable deque based on the ordering of leaves of a binary tree [HP05].

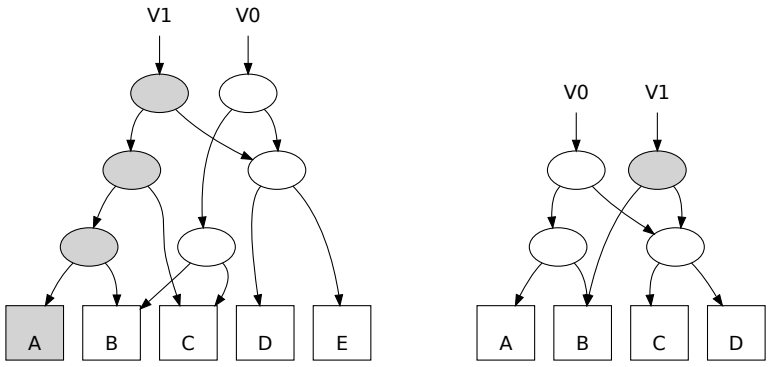
Figure 3.15 illustrates an example of a deque.

The vertices of the tree are not annotated. The front of the deque is found by choosing the left child of each node starting from the root node. The order of the leaves is preserved.

Figure 3.16 illustrates the insertion and removal of an element in a deque.

Figure 3.17 illustrates the growth of the immutable deque. Successive leaves are added through a process of path copying.

The deque corresponds to an expression in which the list concatenation function is applied to the values. The concatenation function is associative so the deque maintains the property that the annotation of a node is equal to the concatenation of the values of the leaves in the subtree that it suspends. It is not necessary to annotate the nodes with the value of the concatenation. List concatenation is not commutative so the order of leaves must be maintained during any transformation of the tree.




(a) (b)

Figure 3.16: **Insertion and removal of an element in a deque.**
(a) Insertion of an element into an immutable deque. Version V0 contains the values $\{B, C, D, E\}$. The operation $Push\_front(A)$ creates version V1 containing the values $\{A, B, C, D, E\}$. The path created by the operation is shaded.
(b) Removal of an element from an immutable deque. Version V0 contains the values $\{A, B, C, D\}$. The operation $Pop\_front()$ creates version V1 containing the values $\{B, C, D\}$. The path created by the operation is shaded.

Special comparison functions are required to reach the front and back of the deque. One of the comparison functions creates a path to the front of the queue by always selecting the left child. The other comparison function accesses the back of the queue.

This deque suffers from some of the same shortcomings as the priority queue, it requires the implementation of access functions that are specific to the deque ADT, it requires multiple comparison functions and it does not distinguish an empty deque from a non-existent deque.

### 3.5.4 Directed deque

A new data structure, the directed deque, addresses the shortcomings of the deque. It supports a sentinel and fully abstracts the ADT implementation from the functions of the Canonical Binary Tree.

The nodes are annotated with a pair formed from the second annotation of the child on the right and the second annotation of the child on the left. The first annotation of a leaf is not used and the second annotation is zero. The second

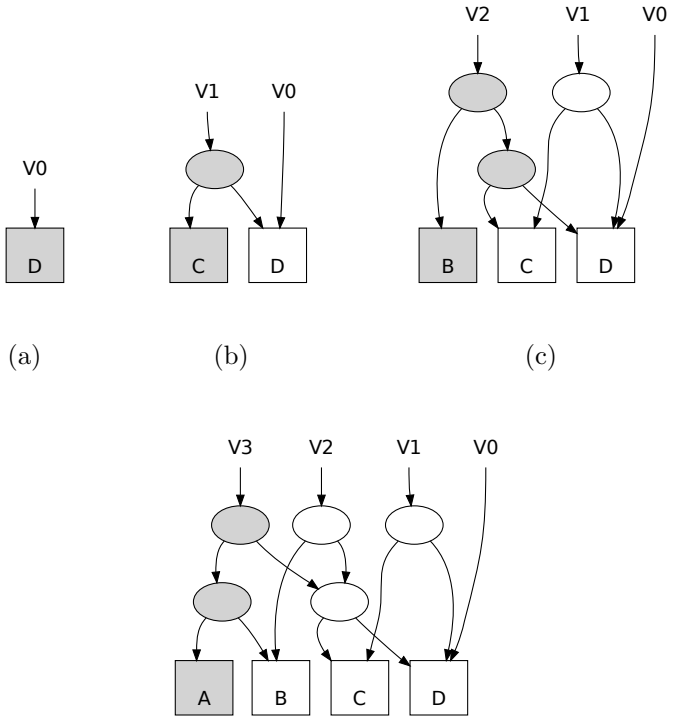



Figure 3.17: **Animation showing the growth of an immutable deque** through a series of insertions. New versions of the data structure are created by each operation. In each case the path created by the operation is shaded.
(a) Initial deque.
(b) After $Push\_front(C)$
(c) After $Push\_front(B)$
(d) After $Push\_front(A)$

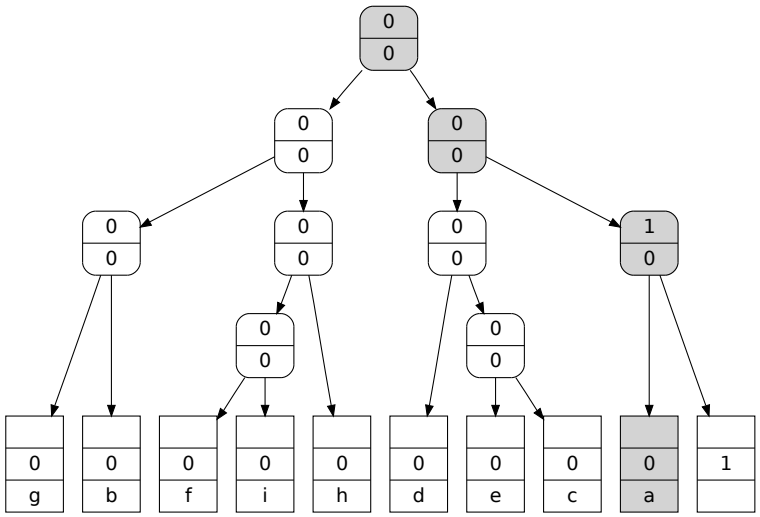



Figure 3.18: **Example Directed deque** containing the values $\{g, b, f, i, h, d, e, c, a\}$. The shaded vertices illustrate the path to the back of the queue. The sentinel is the right-most leaf of the tree. The first annotation is shown above the second annotation.

annotation of the sentinel is one.

Figure 3.18 illustrates an example of a Directed deque.

The sentinel is annotated in such a way that it cannot be removed from the tree and leaves cannot be inserted to the right of the sentinel. Using the annotation scheme three leaves are reachable, they are the left-most leaf, the sentinel and the leaf to the left of the sentinel.

The $Push\_front()$, $Front()$ and $Pop\_front()$ functions are implemented by the Canonical Binary Tree functions $insert()$, $query()$ and $delete()$ each called with an access parameter of zero which causes the path to the front of the queue to be selected. The $Push\_back()$ function is implemented by the $insert()$ function with an access parameter of infinity, which causes a path to the sentinel to be selected. Insertion takes place to the left of the sentinel which causes an element to be added to the back of the queue. The $Back()$ and $Pop\_back()$ functions are implemented by the $query()$ and $delete()$ functions with an access parameter of one which causes a path to the back of the queue to be selected.

Table 3.2 contains all of the information needed to specialise the Canonical Binary Tree so that it implements the directed deque ADT.

| Canonical Binary Tree specialisation | |
|---|---|
| $annotator(< a, b >, < c, d >)$ | $< d, b >$ |
| $identity$ | $<, 1 >$ |

| API function | Canonical Binary Tree access function |
|---|---|
| $Push\_front()$ | $insert(0)$ |
| $Pop\_front()$ | $delete(0)$ |
| $Front()$ | $query(0)$ |
| $Push\_back()$ | $insert(\infty)$ |
| $Pop\_back()$ | $delete(1)$ |
| $Back()$ | $query(1)$ |

Table 3.2: **Directed deque implementation.** The Canonical Binary Tree can be specialised to implement a Directed deque and its access functions can be adapted to present a deque ADT to the application.

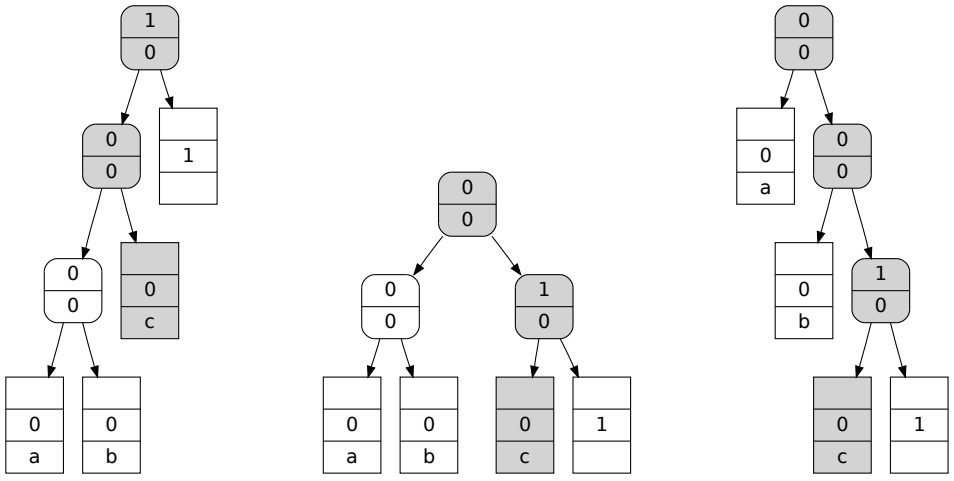



Figure 3.19: **Associativity property of a directed deque.** Directed deques with different topologies maintain the order of their leaves. The path to the back of the queue is shaded.

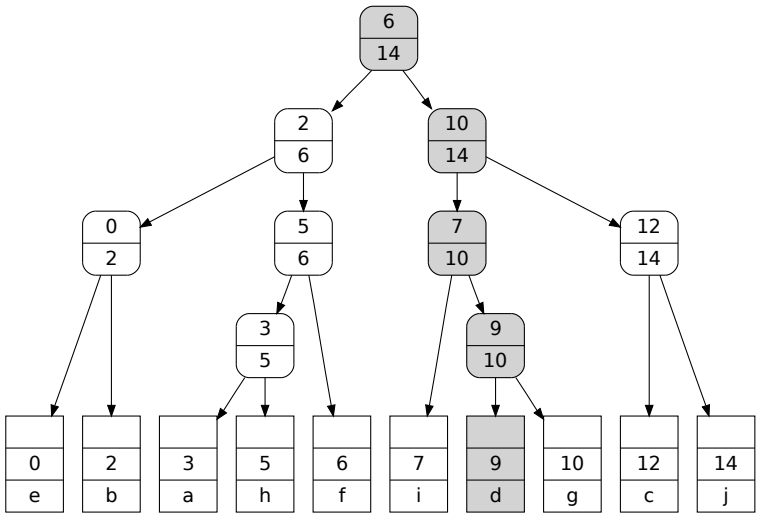



Figure 3.20: **Example interval tree** containing the key-value pairs $\{0 \mapsto e, 2 \mapsto b, 3 \mapsto a, 5 \mapsto h, 6 \mapsto f, 7 \mapsto i, 9 \mapsto d, 10 \mapsto g, 12 \mapsto c, 14 \mapsto j\ \}$. The shaded path illustrates the mapping $9 \mapsto d$.

Figure 3.19 illustrates the associativity property of the Directed deque. The associativity property allows the topology of the data structure to be modified without affecting the functionality provided by the ADT.

### 3.5.5 Map

A map is a sorted associative data structure that provides access to a set of key-value pairs. It also supports in-order traversal of leaves in sorted order. The functionality that the map ADT provides is similar to that of a C++ STL map [Jos99].

A map has an *Insert*() function that inserts a key-value pair, a *Query*() function that retrieves an application value given its key and a *Remove*() function that removes the key-value pair from the map.

Hinze and Paterson describe an implementation of an immutable map using an interval tree [HP05].

Figure 3.20 illustrates an interval tree.

An interval tree corresponds to a mathematical expression in which the maximum and minimum functions are applied to the annotations. The first annotation

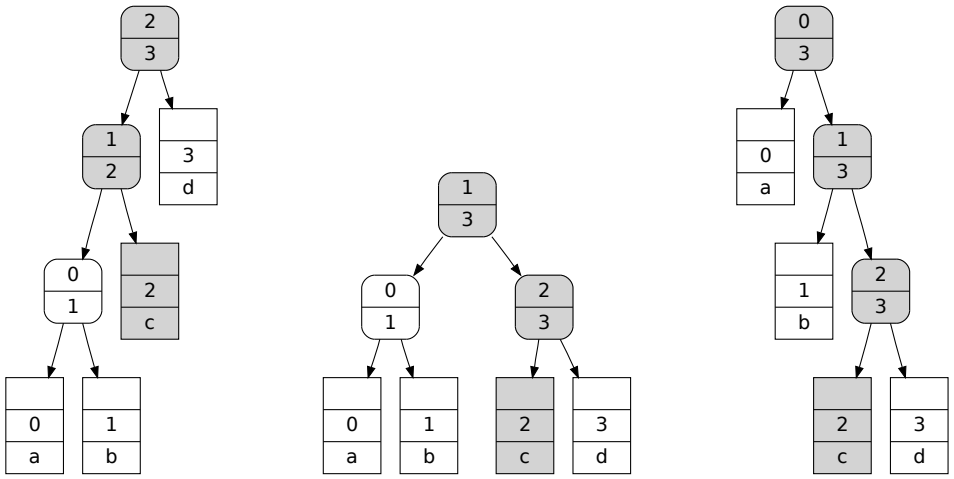


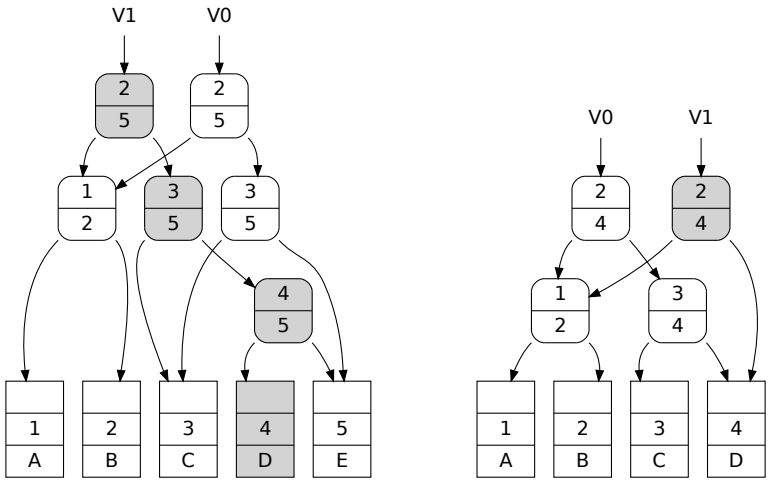

Figure 3.21: **Associativity property of an interval tree.** Interval trees with different topologies maintain the property that the first annotation of a node is the highest first annotation in the subtree suspended on the left and the second annotation is the highest first annotation in the subtree suspended on the right.

of a node is the minimum of the second annotations of its children. The second annotation of a node is the maximum of the second annotations of its children. The minimum and maximum functions are associative, so during topological transformations the interval tree maintains the property that the first annotation of a node is the maximum key in the sub-tree suspended by its left child and the second annotation of a node is the maximum key in the sub-tree suspended by its right child.

Figure 3.21 illustrates the associativity property of the interval tree.

The key is the access parameter for the functions of the data structure. A path from the root to a leaf with a given key is found by repeatedly checking for a leaf and then comparing the key to the first annotation of the node. If the key is greater than the first annotation then the right child of the node is on the path. If a leaf with a given key is not present in the interval tree then a leaf with a different key will be found. It is not necessary to access the children of a node in order to determine the path.

Figure 3.22 illustrates the insertion and removal of an element in an interval tree.




(a) (b)

Figure 3.22: **Insertion and removal of an element in an interval tree.**
(a) Insertion of an element into an immutable interval tree. Version V0 contains the key-value pairs $\{1 \mapsto A, 2 \mapsto B, 3 \mapsto C, 5 \mapsto E\}$. The operation $Insert(4 \mapsto D)$ creates version V1 containing the key-value pairs $\{1 \mapsto A, 2 \mapsto B, 3 \mapsto C, 4 \mapsto D, 5 \mapsto E\}$. The path created by the operation is shaded.
(b) Removal of an element from an immutable interval tree. Version V0 contains the key-value pairs $\{1 \mapsto A, 2 \mapsto B, 3 \mapsto C, 4 \mapsto D\ \}$. The operation $Remove(3)$ creates version V1 containing the key-value pairs $\{1 \mapsto A, 2 \mapsto B, 4 \mapsto D\}$. The path created by the operation is shaded.

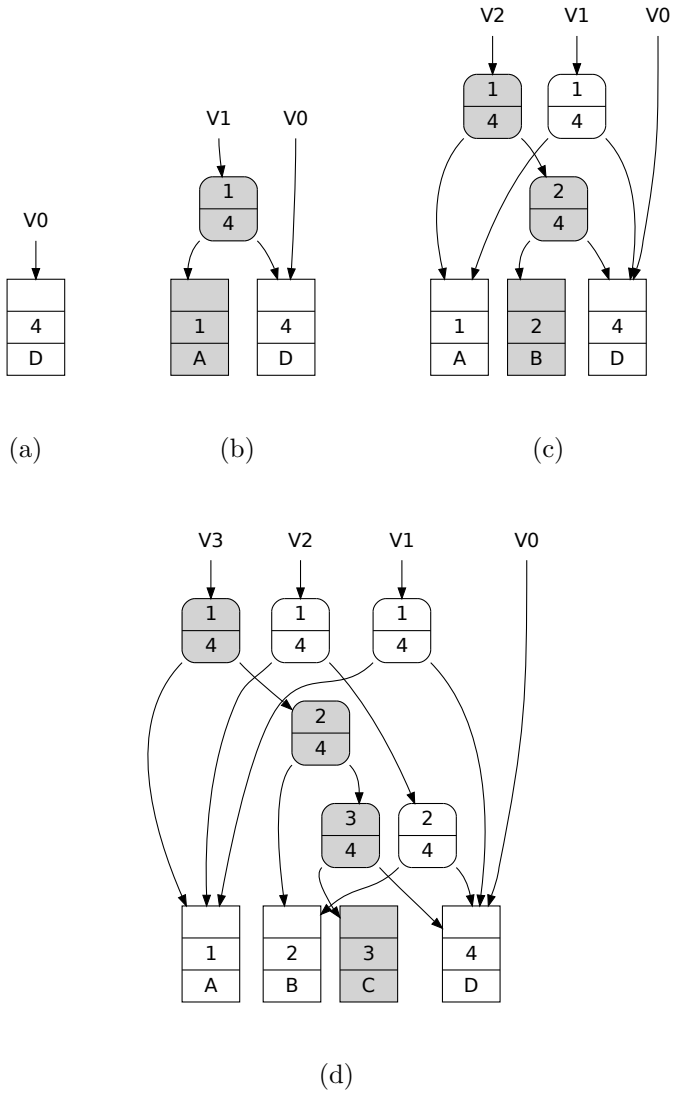



Figure 3.23: **Animation showing the growth of an interval tree** through a series of insertions. New versions of the data structure are created by each operation. In each case the path created by the operation is shaded.
(a) The initial data structure containing the key-value pair $4 \mapsto D$.
(b) After $Insert(1 \mapsto A)$
(c) After $Insert(2 \mapsto B)$
(d) After $Insert(3 \mapsto C)$

| Canonical Binary Tree specialisation | |
|---|---|
| $annotator(< a, b >, < c, d >)$ | $< min(b, d), max(b, d) >$ |
| $identity$ | $<, \infty >$ |

| API function | Canonical Binary Tree access function |
|---|---|
| $Insert(key)$ | $insert(key)$ |
| $Remove(key)$ | $delete(key)$ |
| $Query(key)$ | $query(key)$ |

Table 3.3: **Map implementation.** The Canonical Binary Tree can be specialised to implement an interval tree and its access functions can be adapted to present a map ADT to the application.

Figure 3.23 illustrates the growth of the immutable interval tree.

The interval tree suffers from the problem that it does not specify a representation of the empty map.

### 3.5.6 Interval tree with sentinel

The Canonical Binary Tree can be adapted to implement an interval tree with a sentinel. The first annotation of the sentinel is not used and the second annotation is infinity. In practice, the sentinel is annotated with the maximum value of its data type.

Table 3.3 contains all of the information needed to specialise the Canonical Binary Tree so that it implements the map ADT.

The $Query()$ function returns a value for every possible value of the access parameter, even when the access parameter does not match a key. This is not the behaviour typically expected of a map. The ADT wrapper functions can implement checks to ensure that the value retrieved by a query corresponds to the key and that duplicate keys are handled appropriately.

To check that the value retrieved by a query corresponds to a key it is necessary to store the value of the key as part of the application value. In a concurrent execution environment the annotation of a leaf cannot be used for this purpose as it is regarded as structural information and is not accessible through the functions of the Canonical Binary Tree.

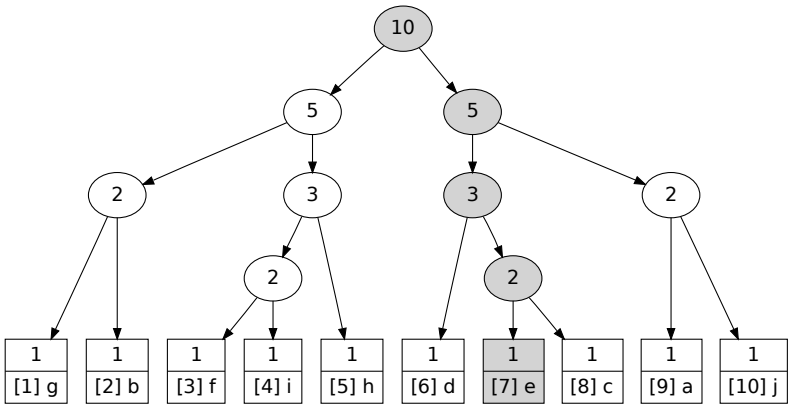



Figure 3.24: **Example Sequence tree** containing the values $\{[1]g, [2]b, [3]f, [4]i, [5]h, [6]d, [7]e, [8]c, [9]a, [10]j\}$

A set ADT can be implemented by an interval tree that does not permit duplicate values. To implement the set ADT the $Insert()$ function should call $query()$ to ensure the uniqueness of a value before calling $insert()$. In a concurrent execution environment the Canonical Binary Tree functions should be referentially transparent so the $insert()$ function, which alters the Canonical Binary Tree, cannot give any indication of success.

### 3.5.7 Vector

A vector is an ordered set of values that supports random access based on ordinal number. The vector ADT provides a similar set of functions to the deque ADT in addition to random access to ordinals within the sequence. The functionality that the vector ADT provides is similar to that of a C++ STL map [Jos99].

The $Insert()$ function inserts an ordinal value pair into the vector. The $Query()$ function is supplied with an ordinal as the access parameter. The function returns the value associated with the ordinal. The $Remove()$ function deletes an ordinal value pair from the data structure. An in-order traversal of the vector returns values in the order given by their ordinal number.

Hinze and Paterson describe an implementation of an immutable vector based on a sequence tree [HP05].

A node of the tree is annotated with the sum of the annotations of its children.

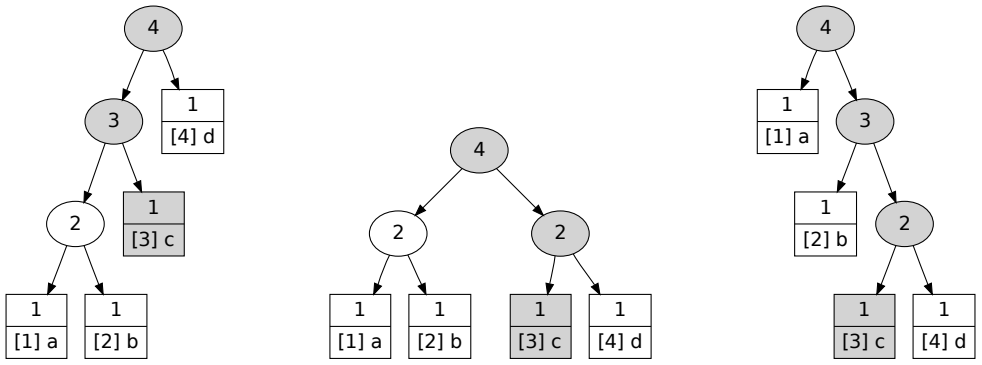



Figure 3.25: **Associativity property of a sequence tree.** Sequence trees with different topologies maintain the property that a node is annotated with the sum of the annotations of its children.

A leaf of the tree is annotated with a value of one.

Figure 3.24 illustrates the sequence tree. The ordinal numbers shown in square brackets are for illustration purposes only and are not part of the data structure.

A vector corresponds to a mathematical expression in which the addition function is applied to a value of one. Addition is both associative and commutative, so the vector maintains the property that the annotation of a node contains the sum of the number of leaves in the subtree that it suspends, regardless of the topology of that subtree.

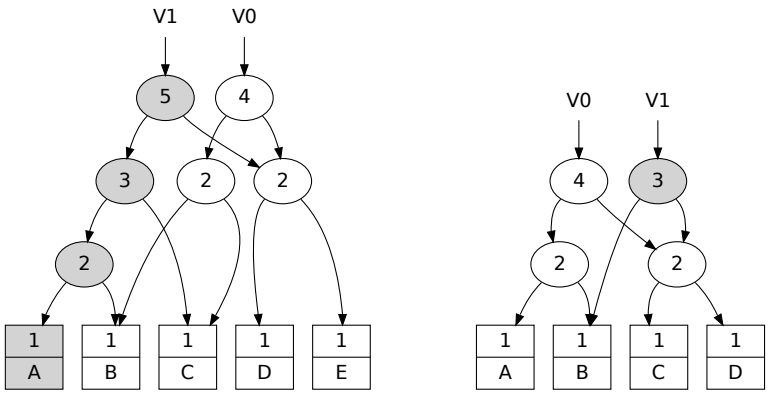
Figure 3.25 illustrates the associativity property

To locate a leaf with a given target ordinal the annotations of the children of the root node are examined. If the target ordinal is greater than or equal to the annotation of the left child then the right child is on the path. If the right path is chosen then the annotation of the left path is subtracted from the target ordinal number. If the left path is chosen then the target ordinal is unchanged. The comparison process continues at each node until a leaf is reached.

Figure 3.26 illustrates the insertion and removal of an element in an immutable sequence tree.

Figure 3.27 illustrates the growth of the immutable sequence tree.

The vector ADT can be restricted to implement an immutable array. To implement an immutable array a vector is populated with values before normal




(a) (b)

Figure 3.26: **Insertion and removal of an element in an immutable sequence tree.**
(a) Insertion of an element into an immutable sequence. Version V0 contains the sequence $\{[1]B, [2]C, [3]D, [4]E\}$. The operation $Insert([1]A)$ creates version V1 containing the sequence $\{[1]A, [2]B, [3]C, [4]D, [5]E\}$. The path created by the operation is shaded.
(b) Removal of an element from an immutable sequence. Version V0 contains the sequence $\{[1]A, [2]B, [3]C, [4]D\}$ The operation $Remove([1])$ creates version V1 containing the sequence $\{[1]B, [2]C, [3]D\}$. The path created by the operation is shaded.

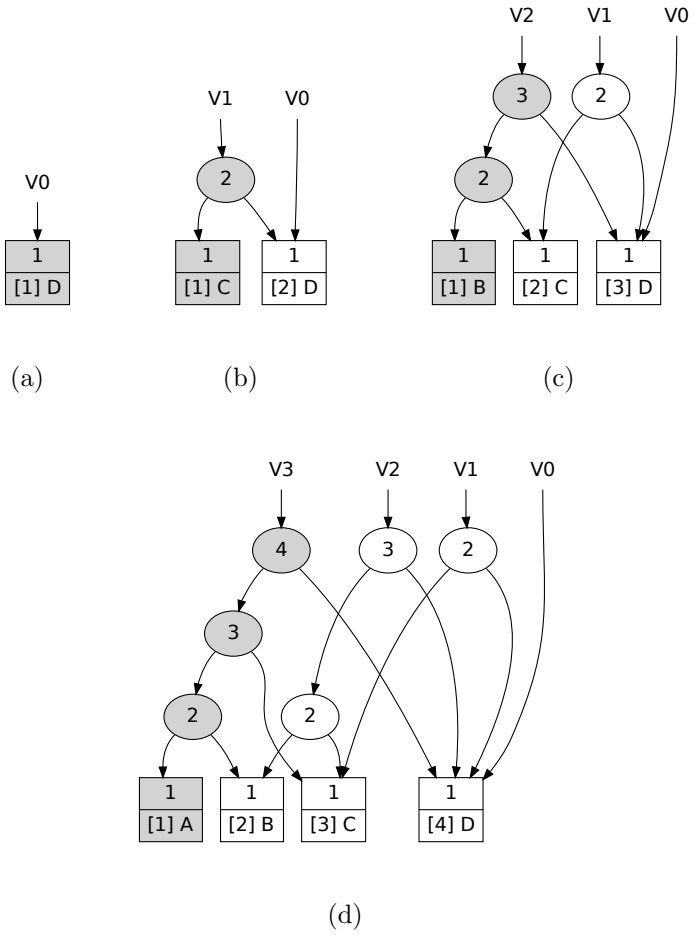



Figure 3.27: **Animation showing the growth of an immutable sequence tree** through a series of insertions. New versions of the data structure are created by each operation. In each case the path created by the operation is shaded. Version V3 represents the sequence $\{[1]A, [2]B, [3]C, [4]D\}$
(a) Initial data structure containing the value $D$.
(b) After $Insert([1]C)$
(c) After $Insert([1]B)$
(d) After $Insert([1]A)$

access is permitted. Replacement is a compound operation formed by an insert and a remove operation acting on elements with the same ordinal number. It is the only operation normally permitted by a vector implementing an array.

The sequence suffers from some of the same shortcomings as the priority queue, it requires the implementation of access functions which are specific to the vector ADT and it does not distinguish an empty vector from a non-existent vector.

### 3.5.8 Directed sequence

A new data structure, the directed sequence, addresses the shortcomings of the sequence tree. The comparison function of the sequence accesses only the node annotation. The directed sequence supports a sentinel and fully abstracts the ADT implementation from the functions of the Canonical Binary Tree.

The sequence requires that the annotations of the children of a node are examined to determine the path and this results in unnecessary accesses to nodes that are not on the path. To avoid these accesses a node should be annotated in such a way that the direction of the path can be determined without accessing the annotations of its children. This can be achieved by a regarding the annotation as a pair.

The second annotation, of the directed sequence, is the sum of the second annotations of the left and right children. The first annotation is set to the value of the second annotation of the left child. The first annotation is used to determine the path.

Figure 3.28 illustrates an example of a directed sequence.

To locate a leaf with a given target ordinal that ordinal is compared with the first annotation of the root node. If it is greater the right child is chosen otherwise the left child is chosen. This process is repeated until a leaf is reached. When a right child is chosen the second annotation of the child is subtracted from the target ordinal.

The sentinel is always the right-most leaf of the Canonical Binary Tree. To support the $Back()$ and $Push\_back()$ functions the sentinel and the leaf to the left of the sentinel must be annotated in such a way that they can be found without specifying an ordinal. The ADT specifies a special value which causes the second annotation of the root to be used as the access argument. The second value of the root is the number of elements in the sequence, including the sentinel.

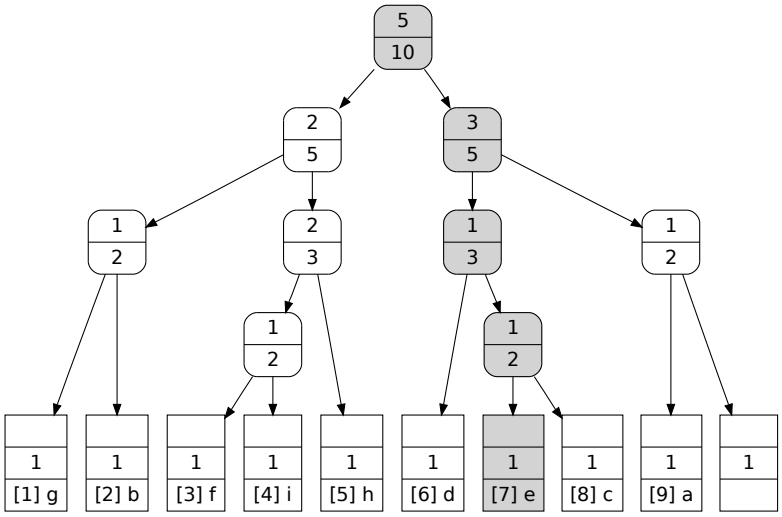



Figure 3.28: **Example Directed sequence** containing the values: $\{[1]g, [2]b, [3]f, [4]i, [5]h, [6]d, [7]e, [8]c, [9]a\}$. The shaded path illustrates the access to the leaf with an ordinal of seven. The sentinel is the right-most leaf. The first annotation is shown above the second annotation.

| Canonical Binary Tree specialisation | |
|---|---|
| $annotator(< a, b >, < c, d >)$ | $< b, b + d >$ |
| $identity$ | $<, 1 >$ |

| API function | Canonical Binary Tree access function |
|---|---|
| $Insert(ordinal)$ | $insert(ordinal)$ |
| $Remove(ordinal)$ | $delete(ordinal)$ |
| $Query(ordinal)$ | $query(ordinal)$ |
| $Push\_front()$ | $insert(0)$ |
| $Pop\_front()$ | $delete(0)$ |
| $Front()$ | $query(0)$ |
| $Push\_back()$ | $insert(\star)$ |
| $Pop\_back()$ | $delete(\star - 1)$ |
| $Back()$ | $query(\star - 1)$ |

Table 3.4: **Directed sequence implementation.** The Canonical Binary Tree can be specialised to implement a directed sequence and its access functions can be adapted to present a vector ADT to the application. The second annotation of the root is represented by a star.

Table 3.4 contains all of the information required to specialise the Canonical Binary Tree so that it implements the vector ADT. The value of the second annotation of the root node is represented by a star.

The functions of the Canonical Binary Tree are complete so all values of the access parameter are valid arguments. An ordinal value less than or equal to one refers to ordinal number one. However, an ordinal number equal to or higher than the number of elements in the sequence refers to the sentinel. The $Remove()$ function verifies that its access argument is less than the second annotation of the root.

The directed sequence permits access to all leaves using their ordinal number. The ordinal is relative to the start of the sequence. Array indexes map to ordinals which start at one. For example, the $Query(0)$, $Query(1)$ and $Front()$ functions have the same effect.

An immutable array can be created by restricting the functions of the vector.

The sequence implementation requires that a value is retained and decremented while determining the path, so the function that determines the path through the data structure is specific to the vector ADT. This is unfortunate as some of the generality of the Canonical Binary Tree must be sacrificed to support the sequence. In our implementation the Canonical Binary Tree functions are supplied with an additional parameter which alters the mechanism for determining the path when implementing a sequence.

The sequence implementation requires that the ADT has access to the second annotation of root node. This is unfortunate as annotations are structural information that should not be exposed to the application.

### 3.5.9 Previous work

Andersson and Nilsson describe how the comparison operations used to determine the path through an interval tree can be confined to those nodes that are actually on the path [AN95]. We extend this idea and apply it to immutable min-trees, deques and sequence trees.

## 3.6 Balancing

An immutable binary tree can be balanced to minimise the time taken to access the data within it. A number of schemes for balancing binary trees are described in the literature. This section describes the implementation of a scheme for balancing the Canonical Binary Tree. The balancing process is independent of the ADT implemented by the Canonical Binary Tree.

The problem is to adapt a balancing scheme, implemented in an imperative programming language, so that it can be used to balance the immutable Canonical Binary Tree. Balancing reduces the average time required to access the data associated with a leaf. Balancing improves the performance of the priority queue, deque, map and vector implemented by the ADT by reducing the average access time of all of the Canonical Binary Tree operations.

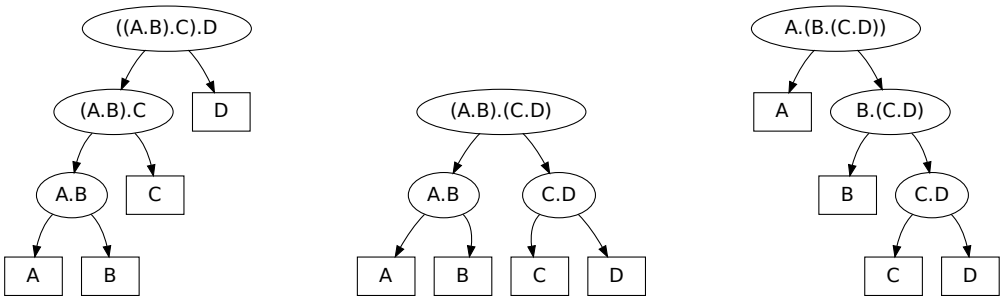
The main contribution of this section is the development of an Immutable Data Structure that separates the performance characteristics of the structure from the ADT to which it conforms. This section focuses on techniques for minimising the average access time and the amortised access time of Immutable Data Structures implemented in imperative programming languages.

### 3.6.1 Balancing schemes

The Canonical Binary Tree described in the previous section can be balanced independent of the ADT that it implements.

Figure 3.29 illustrates how the associativity property of the annotator function permits the topological transformations required to balance the tree regardless of the ADT that it implements.

A balancing scheme works by implementing a set of balancing invariants. When the invariants are compromised the tree is restructured so that its topology conforms to the invariants. Restructuring takes the form of topological transformations called rotations. A balancing scheme can be characterised by a set of invariants and a set of rotations. These are combined into a set of cases. Each case is characterised by a configuration of nodes that violates the invariants. These nodes are made to conform to the invariants by applying the rotations. Typically, balancing algorithms restructure the tree during a mutation, guaranteeing balancing invariants as post conditions to each mutating function. The post condition is enforced by applying the checks to each node altered by the




(a) (b) (c)

Figure 3.29: **The associativity property permits balancing.** The mean number of nodes that are traversed to reach a leaf of the balanced tree (b) is two, whereas for the unbalanced trees (a) and (c) it is 2.25. The associativity property of the annotator function '.' allows the topology of the tree to be modified, in order to reduce the average access time, without affecting the functionality of the ADT being implemented.

mutation.

Many tree balancing schemes are based on B-Trees [BM72]. In these schemes the invariants and balancing information are expressed in terms of a binary B-Tree composed from the nodes of the binary tree. The B-Tree nodes are referred to as pseudo-nodes. Each B-Tree pseudo-node contains a number of binary tree nodes. The maximum number of children of a pseudo-node is referred to as the order of the B-tree. Nodes in the same pseudo-node are said to be joined by horizontal edges. Edges between pseudo-nodes are said to be vertical. Balancing information in the node indicates whether each edge is horizontal or vertical. Pseudo-nodes can be joined or split by changing the orientation of the edges. A balancing algorithm joins and splits the B-tree nodes so that the B-tree remains perfectly balanced. This places a limit on the imbalance of the underlying binary tree.

A red-black tree is a self balancing binary B-tree of order four. Guibas and Sedgewick describe the implementation of an ephemeral red-black tree in detail [GS78]. For every red-black tree there is at least one corresponding 2-4 B-Tree with elements in the same order. The invariants are described in terms of the colours red and black. As a post condition to insertion each node on the path is examined to ensure that it complies with the invariants. Compromised invariants

are restored by a series of rotations. For each node on the path of an insertion there are five possible cases. There are six possible cases for each node during a deletion.

Okasaki describes an immutable implementation of a red-black tree [Oka98] in the functional programming languages Haskell and ML. The implementation relies on repeated pattern matching. Each node on the path is compared with one of the cases and compromised invariants are fixed by applying rotations. Each case corresponds to a single program line in both functional programming languages making the implementation both brief and easy to follow.

The C++ STL contains an implementation of a red-black tree in an imperative programming language [Jos99]. The imperative implementation relies on explicit testing of each of the cases for each node on the path. Rotations are implemented by pointer manipulation. As a result the imperative implementation appears much more involved than the functional implementation. The high number of cases and the complexity of the rotations makes the implementation of red-black balancing in an imperative programming language both long winded and opaque.

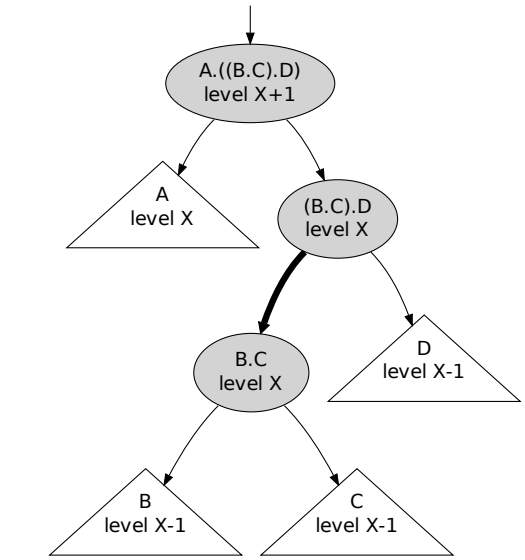
To apply a balancing scheme to the Canonical Binary Tree the rotations must be implemented using path copy. This promises to make the implementation even more unwieldy, so to ease the implementation effort a simpler balancing scheme than red-black is required.

### 3.6.2 Balancing the Canonical Binary Tree

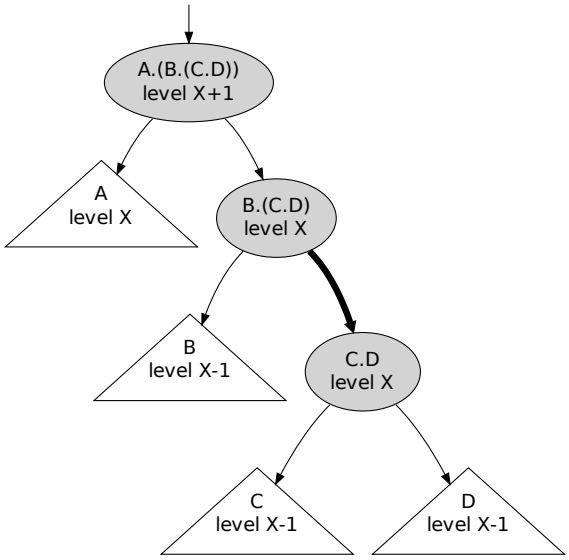
This section describes how the Canonical Binary Tree can be balanced using the AA-tree balancing scheme. The AA-tree has fewer cases and simpler rotations than a red-black tree making it simpler to implement. The rotations are easily expressed in terms of path copy operations.

An AA-tree is a self balancing binary B-tree of order three. Andersson describe the implementation of an ephemeral AA-tree in detail [And93]. For every AA-tree there is at least one corresponding 2-3 B-Tree with elements in the same order. A 2-3 B-Tree consists of pseudo-nodes containing either one or two binary tree nodes. Two binary tree nodes joined by a horizontal edge are regarded as forming a pseudo-node. Restructuring operations implement rotations that maintain a perfectly balanced 2-3 B-tree. This places a limit on the imbalance of the underlying binary tree.

Each tree node maintains balancing information in the form of a level number.

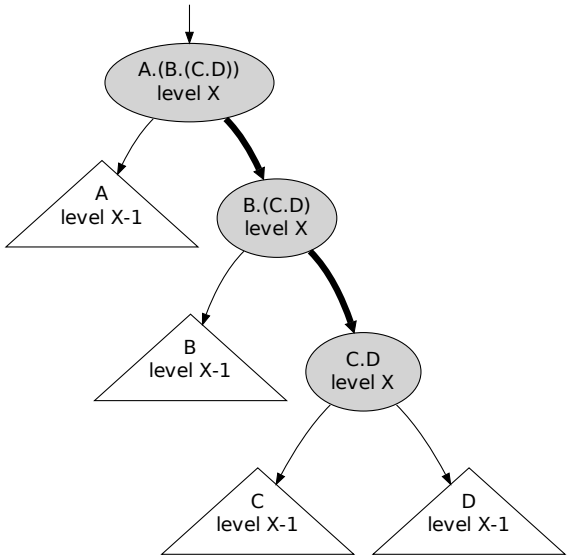



(a)

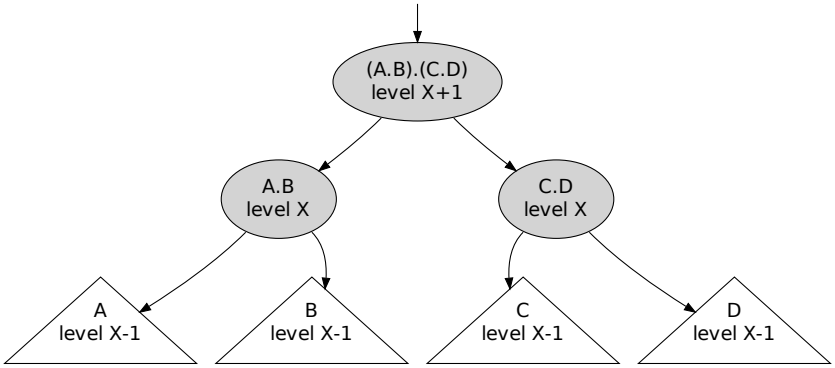



(b)

Figure 3.30: **A skew balancing rotation** corrects violations of the invariant that only right edges are horizontal. Horizontal edges linking the nodes within a pseudo-node are shown in bold. The level number adjustment is also indicated. (a) A subtree that violates the invariant because it has a horizontal left edge. (b) A right rotation transforms the horizontal left edge into a horizontal right edge.




(a)




(b)

Figure 3.31: **Split rotation** corrects violations of the invariant that the largest pseudo-node has three children. Horizontal edges linking the nodes within a pseudo-node are shown in bold. The level number adjustment is also indicated.
(a) A subtree that violates the invariant because it forms a pseudo-node with four children.
(b) A left rotation is performed to transform a pseudo-node with four children into three pseudo-nodes with two children each. The split balancing rotation reduces the size of a pseudo-node by elevating the middle node.

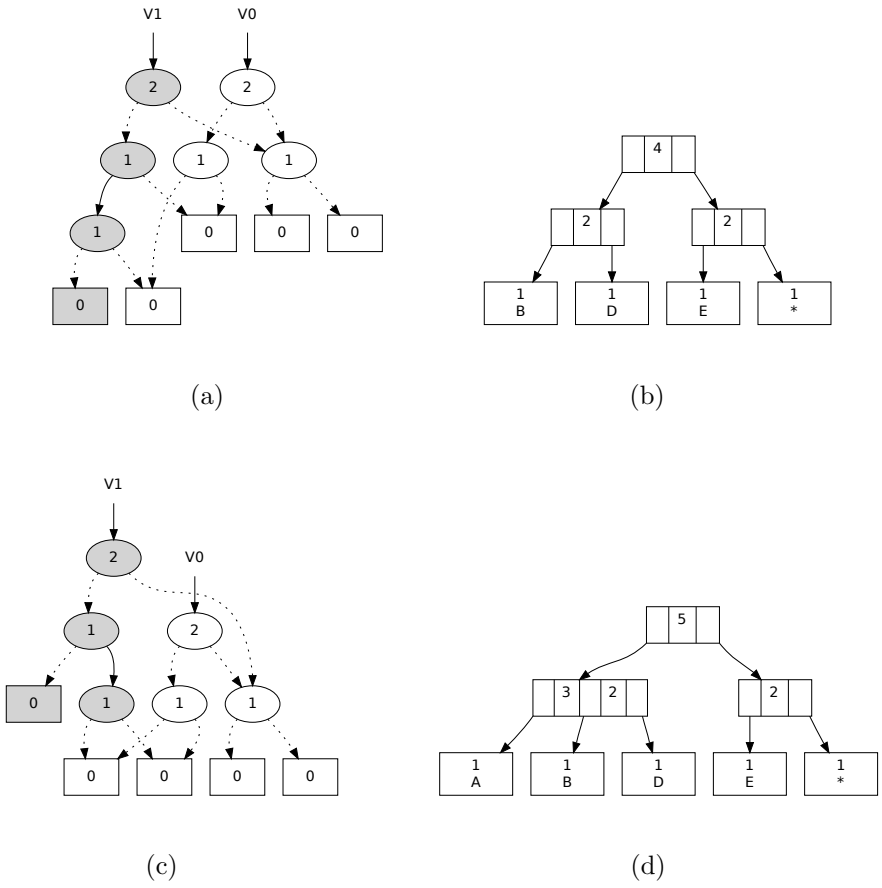



Figure 3.32: **Example of a skew balancing rotation acting on a vector.**
(a) A Canonical Binary Tree implementing a vector. The level number of each node is shown. Vertical edges are dotted. Horizontal edges are bold. The insertion of an element, with a value of A, creates a new version which is shaded. Insertion without balance creates a horizontal left edge that violates the invariant that only right edges may be horizontal so a skew balancing rotation is performed.
(b) Version V0 of a vector viewed as a 2-3 B-tree.
(c) The Canonical Binary Tree implementing this vector after inserting the value A and performing a skew balancing rotation. The skew balancing rotation transforms a potential horizontal left edge into a horizontal right edge.
(d) Version V1 of the vector viewed as a 2-3 B-tree.

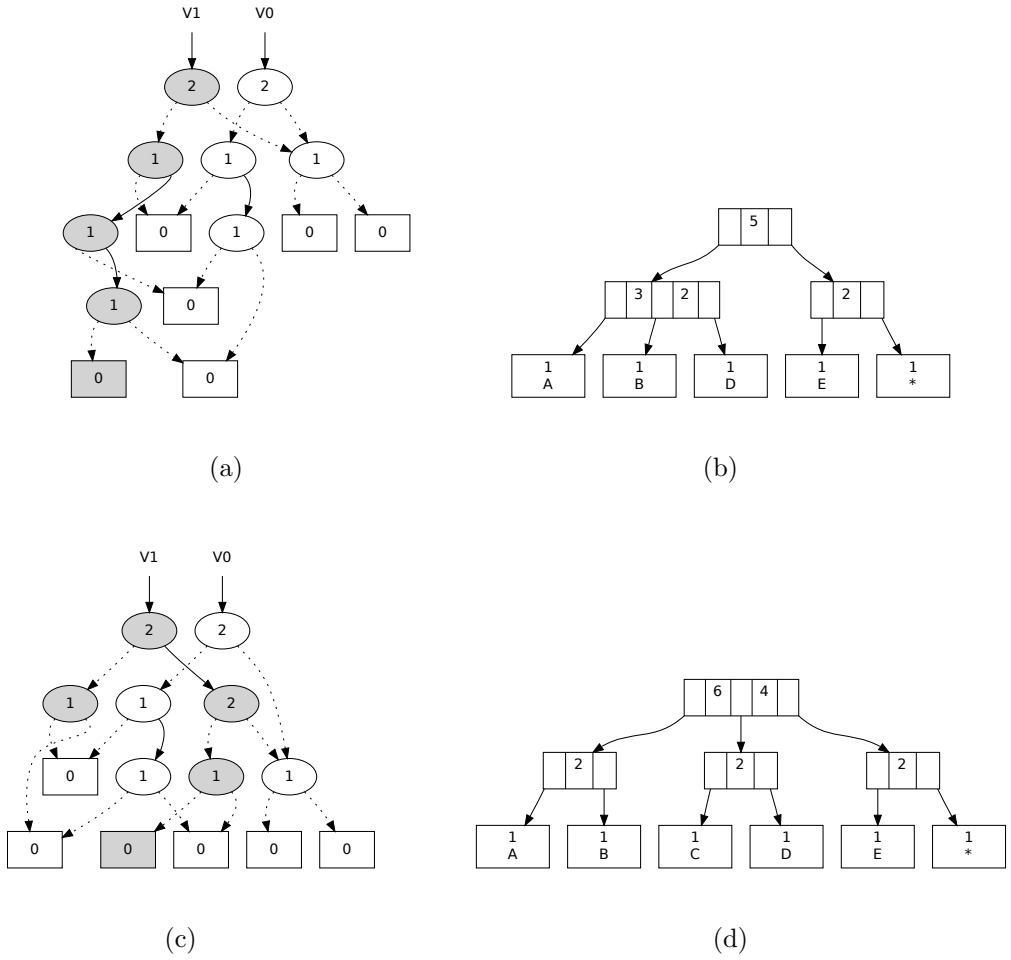



Figure 3.33: **Example of a split balancing rotation acting on a vector.**
(a) A Canonical Binary Tree implementing a vector. The level number of each node is shown. Vertical edges are dotted. Horizontal edges are bold. The insertion of an element, with a value of C, creates a new version which is shaded. Insertion without balance creates two adjacent horizontal right edges representing an overly full pseudo-node that violates the invariant that a pseudo node has at most three children.
(b) Version V0 of a vector viewed as a 2-3 B-tree.
(c) The Canonical Binary Tree implementing this vector after inserting the value C and performing a skew followed by a split balancing rotation. The split balancing rotation transforms the potential overly full pseudo-node by raising the level of the middle node. This creates a horizontal left edge which must be transformed into a horizontal right edge by a skew rotation.
(d) Version V1 of the vector viewed as a 2-3 B-tree.

This number corresponds to the level of the pseudo-node within the B-Tree. The root pseudo-node has the highest level number. Pseudo-nodes at the bottom of the tree are on level 1 and leaves are on level 0. A binary tree edge which connects nodes with equal level numbers is regarded as a horizontal edge and the nodes joined by a horizontal edge form a pseudo-node. A binary tree edge connecting nodes with different level numbers is regarded as a vertical edge and these edges connect pseudo-nodes to form the B-tree.

The AA-Tree maintains the following invariants:

- Only right edges are horizontal
- Pseudo-nodes have at most three children
- There are no breaks in the level numbers

The invariants can be implemented by examining three cases for insert operations and five cases for delete operations. Restructuring operations consisting of combinations of just two balancing rotations, called skew and split, are required to maintain the invariants. The skew and split balancing rotations change the topology of the Canonical Binary Tree. The associativity of the annotator function ensures that the annotations preserve the functionality of the ADT that it implements.

A 2-3 B-tree preserves the invariant that all right edges are horizontal. A horizontal left edge is transformed into a horizontal right edge by a right rotation called a skew.

Figure 3.30 illustrates a skew balancing rotation.

A 2-3 B-tree preserves the invariant that a pseudo-node has at most three children. A pseudo-node with four children contains two horizontal right edges. These are transformed into a left and a right vertical edge by a left rotation called a split. The split balancing rotation raises the level of the middle vertex.

Figure 3.31 illustrates a split balancing rotation.

Figure 3.32 illustrates an example of a skew balancing rotation acting on a vector.

Figure 3.33 illustrates an example of a split balancing rotation acting on a vector.

A tree can be balanced at any time. In our implementation balancing takes place during insert and delete operations. These operations create a new path

within the Canonical Binary Tree and this path can cause the balancing invariants to be compromised. Each node on the path is checked by comparing the configuration of neighbouring nodes to ensure the invariants are preserved. The process of checking the balancing invariants starts at the leaf and ends at root. The addresses of the vertices on the path created by the operation are maintained on a stack so that the path can be accessed in reverse order. An operation that would cause an invariant to be compromised is made compliant by restructuring the tree during the path copy.

### 3.6.3 Previous work

Tree balancing schemes reduce access time by ensuring that paths within a tree are of similar lengths so that the tree remains balanced. Guibas and Sedgewick describe a commonly used balancing scheme for ephemeral trees called the red-black tree [GS78]. Balancing schemes are applicable to both ephemeral and Immutable Data Structures. Okasaki applies the red-black scheme to balance immutable binary trees implemented by a purely functional data structure [Oka98]. Okasaki also describes a purely functional implementation of the AVL tree balancing algorithm. However, the balancing of an immutable binary tree implemented in an imperative programming language has not previously been considered.

### 3.6.4 Utility functions

The Canonical Binary Tree provides an ADT-agnostic in-order traversal of the data structure that returns the values of the leaves. The first value returned is that of the left most leaf of the tree and the last value returned is that of the sentinel.

The Canonical Binary Tree also provides a full copy utility that creates a copy of the tree which shares leaves with the tree being copied and a naïve copy utility that creates an entirely new copy of the tree. The full copy utility can be used to both copy and compress a tree. The naïve copy can be used to both copy a tree and to convert between different ADTs.

### 3.6.5 Optimisation

To reduce the number of paths created by the balance process a copy of a path can be balanced in mutable storage and then copied to immutable storage. This

optimisation reduces the amount of storage consumed by the Immutable Data Structure at the cost of additional copying.

Anderson describes a 2-3 B-tree in which the balancing information takes the form of a level number and the invariants act on this number. This formulation is known as an AA-tree. However, the level number is just an implementation convenience. The invariants of the tree can also be expressed in terms of just horizontal and vertical edges. Such a formulation results in a smaller node size as a single bit indicator can be used to indicate the orientation of an edge.

A node may contain information about the subtree it suspends because that subtree is immutable. This information can reduce the need to access the subtree during balancing operations. It is possible to record within the node itself whether or not a child of a node is a leaf. It is also possible to record whether or not the link to a child's left child is horizontal or vertical within the parent node. These optimisations reduce the number of nodes that must be accessed when testing the balancing invariants. Using these optimisations it is only necessary to access nodes on the path to balance the tree.

The size of a node can have a significant effect on performance as larger node sizes reduce the effectiveness of caching. In our implementation the data type of the annotation is supplied as a template parameter. The smallest data type that can accommodate the expected range of annotation values is chosen to minimise the node size. The nature of the references to a child node also affects the size of the node. Memory displacements or ordinal numbers may be used instead of pointers to reduce the node size when the memory to be used by the data structure is pre-allocated in a contiguous chunk.

A tree with multiple children per node improves access times by making the tree shallower but this optimisation comes at the cost of increased implementation complexity. A binary tree can be mapped onto a tree with larger nodes by making some of the parent-child relationships implicit. When paths are written to contiguous memory the relationship between a node and one of its children can be implied, because this child resides in a consecutive memory location, so it is only necessary that a node contain a reference to one of its children and a bit indicating whether that child is to the left or right. An immutable path is typically written into contiguous memory leaf first, so the implicit child of a node immediately precedes it in storage. By restricting the implicit parent-child relationship to nodes on a path it is possible to achieve the effect of a large

node with multiple children without significantly increasing the implementation complexity.

When implementing Immutable Data Structures using C or C++ it is tempting to allocate individual nodes by using the functions *malloc* or *new*, but we found that this leads to poor performance. The effect is particularly pronounced in a concurrent execution environment where memory allocation is typically serialised. We found that allocating nodes on a cache line boundary and fitting an integral number of nodes into a cache line improved performance. In our implementation we used the Threading Building Blocks scalable memory allocator to pre-allocate cache aligned contiguous chunks of storage for our data structures [Int09].

### 3.6.6 Amortised analysis

Amortised analysis is a method of analysing the performance of a function acting on a data structure. The idea is that costly operations occur rarely and that their effect on performance is offset by the occurrence of cheaper, more frequent operations. The analysis proceeds by establishing the worst case time bound of the function and how frequently this worst case occurs. The amortised time per operation is the worst case time bound on a series of $m$ operations divided by $m$. Goodrich and Tamassia give a detailed explanation of amortised time analysis for many common ephemeral data structures [GT09].

Amortised analysis has two distinct purposes. Firstly it guides data structure design and allows comparison between equivalent operations on different data structures. For example, the amortised time for the insertion of a value with associated priority into a priority queue ADT implemented by two different data structures can be compared. Secondly, it guides application development because it indicates the relative cost of different operations.

Published amortised access times for functions acting on a data structure are an important guide for developing high performance applications. However, access times are a characteristic of the data structure implementation rather than the ADT. If the application programmer needs to know the amortised access time for a series of operations then the details of the implementation of that data structure have not been completely hidden by abstraction. For example, both a tree and a hash table can implement a map ADT with identical access functions, however the access time of a tree is logarithmic and that of a hash

table is constant. It is not possible to exchange the implementations without affecting the performance of the access functions, so the implementation details have not been fully encapsulated. Using the results of amortised analysis as a guide to developing applications is orthogonal to the principle of encapsulating the implementation details of a data structure.

The amortised times for all functions of the Canonical Binary Tree are identical as each function requires that a single leaf be accessed. The worst case time bound is $2log_2(n)$ as the longest path to a leaf in a balanced AA-tree is twice as long as the shortest [And93].

## 3.7 Memory Management

An Immutable Data Structure can grow to consume all available memory which prevents an application from completing. The Immutable Data Structures presented in this thesis are unbounded and persist for the duration of the application. The memory occupied by vertices must be allocated and returned in a way that does not serialise the concurrent application and vertices that are no longer in use must be identified and reclaimed.

This thesis does not propose a solution to the problem of managing the memory used by Immutable Data Structures, but a robust solution is necessary before they can be used in production. This section focuses on techniques for allocating and reclaiming memory from an Immutable Data Structure in a concurrent execution environment.

### 3.7.1 Memory allocation and reclamation

The memory management problem can be decomposed into two components. The concurrent allocation problem, which is the problem of allocating and returning memory locations without serialising the application, and the concurrent memory reclamation problem, which is the problem of identifying memory for reclamation.

The concurrent allocation problem can be solved by using a scalable memory allocator which can allocate and return the memory occupied by vertices of an Immutable Data Structure without serialising the concurrent application using it. The Threading Building Blocks scalable memory allocator can be employed to allocate memory for a concurrent application. Reinders describes this allocator in detail [Rei07].

The memory reclamation problem centres on determining reachability. Immutable shared memory can be reclaimed if and only if it is unreachable from all processors participating in the concurrent application's execution. The problem is to determine which vertices of the Immutable Data Structure are reachable and then return all those vertices that are not.

Vertices allocated in isolation become unreachable when the function that allocated them is unsuccessful in updating the root, so the memory occupied by vertices allocated speculatively can be returned when speculation proves unsuccessful.

The reachability of vertices within an Immutable Data Structure is dependent

on the reachability of versions. The reachable vertices of an Immutable Data Structure are those vertices that form the transitive closure of the reachable versions.

One approach to the memory reclamation problem is to quiesce all concurrent execution so that only the most recent version of the Immutable Data Structure is reachable. A full copy of this version is made. The entire Immutable Data Structure can then be discarded and the memory it occupies can be reclaimed. After the copy operation is complete concurrent execution can resume using the copy as the base version of a new Immutable Data Structure. This approach is particularly effective when the memory for the vertices is pre-allocated as a contiguous chunk.

Another approach is to determine the set of vertices within the transitive closure of the reachable versions and then to return all vertices not included in this set. To achieve this, the reclamation process first traverses the transitive closure of reachable versions to determine the set of reachable vertices. It then performs a post-order traversal of the transitive closure of all versions returning those vertices not included in the set of reachable vertices.

### 3.7.2 Previous work

Jones and Lins describe the problem of garbage collecting managed memories in a comprehensive book [JL96]. The problem of reclaiming memory from committed versions of the Canonical Binary Tree is simpler than the garbage collection problem in managed memory systems. The Canonical Binary Tree simplifies traversal because functions acting on the data structure preserve its structural properties. It simplifies the problem of determining reachability because vertices are only reachable by a single link. It also simplifies the problem of coalescing reclaimed memory because vertices are of uniform size and may be allocated contiguously.

# Chapter 4

# Accessing State

Transactional Memory systems have complex concurrent semantics and are prone to pathologies which make their run-time behaviour unpredictable. In a concurrent system shared state should be isolated so that functions accessing it have intuitive concurrent semantics and avoid pathologies. This chapter describes how isolating shared state in linearizable objects provides a concurrent programming model that has intuitive concurrent semantics and that is not prone to pathologies.

Section 4.1 makes the observation that linearizable objects have intuitive concurrent semantics and are not prone to pathologies.

Section 4.2 describes how Immutable Data Structures can implement linearizable objects.

Section 4.3 describes a checkpointing technique which relies on the composition of Immutable Data Structures.

Section 4.4 compares a concurrent application which calculates the minimum spanning tree of a graph with a similar application which uses Transactional Memory.

## 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

A database system is said to guarantee transactional isolation if for every pair of transactions $T_i$ and $T_j$, it appears to $T_i$ that either $T_j$ finished execution before $T_i$ started, or $T_j$ started its execution after $T_i$ finished. When this condition is true each transaction is unaware of other transactions executing concurrently in the system [SK86].

The definition of isolation in a database context does not consider the effects of non-transactional execution. A transactional system is said to exhibit strong isolation if transactions are isolated from both other transactions and concurrent non-transactional execution [DS09]. A transactional system that is not strongly isolated is said to exhibit weak isolation. A weakly isolated system is one in which transactions may be affected either other transactions or concurrent non-transactional execution or both.

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 deadlock.

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 values 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 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 serialisable 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 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 [HMPH05]. 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 [GR92]. Nested database transactions can reduce the overhead of transactional execution. Nesting facilitates the checkpointing 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 [GR92].

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.

## 4.2 Persistent Data Structures

Linearizability is a desirable property for the objects that encapsulate shared state in a concurrent system. However, we have proposed that shared state should be maintained in Immutable Data Structures. This section considers how the access functions of an Immutable Data Structure can be made linearizable. Linearizability endows the functions that access shared state with intuitive concurrent semantics.

A persistent data structure permits access to past versions but in the context of serial execution Immutable Data Structures do not. During concurrent execution access to a past version occurs when the root of the structure is modified by a function executing on another processor. This causes the version being accessed to become a past version. The tardiness of a function permits access to a past version. An Immutable Data Structure that permits access to past versions is persistent. This section considers how access to past versions can be controlled to ensure the property of linearizability.

The main contribution of this section is the implementation of Immutable Data Structures as linearizable objects. This section focuses on mechanisms to restrict access to past versions of an Immutable Data Structure.

### 4.2.1 Accessing past versions

The property of immutability permits a separation of concerns about the integrity of a data structure from concerns about the semantic order of the functions acting on it. A Canonical Binary Tree preserves its structural invariants during concurrent operation. However, it does not provide any mechanism for ensuring the semantic ordering or the invariants of the functions acting upon it.

For example, consider a set implemented by a Canonical Binary Tree. A function inserts a value into the set if and only if the value is not already present. Structural integrity can be described by a set of invariants related to the connectedness of the tree. During concurrent execution the structure of the binary tree is guaranteed because its structural invariants are the pre and post conditions of the path copy operation which implements the function. However, the uniqueness of values in the set is guaranteed by pre and post conditions of the function. The semantic invariants are guaranteed by the ADT and the structural invariants are guaranteed by the data structure.

A function can ensure both the structural and the semantic invariants of a data structure by enforcing mutual exclusion. The use of mutual exclusion is so pervasive that programmers do not usually distinguish between the semantic and structural invariants.

A proof of the linearizability of an object typically requires the identification of the linearization point of each method acting on the object [HW90]. The linearization point of a method is some moment in time between the invocation of the method and its response. Prior to this moment in time the pre-conditions of the method are true and after it the post-conditions are true. To be linearizable the execution of the method must appear to take place atomically at its linearization point. An Immutable Data Structure has both structural and semantic invariants that can be considered separately.

For example, two processors might attempt to insert the same value into an Immutable Data Structure concurrently. Both of the operations complete eventually and the structural invariants of the data structure are preserved. Each of the instances of the function appears to modify the structure at a unique moment in time so the data structure has a linearization point and is structurally linearizable.

Now consider the semantic order of operations on the set. If two instances of a function attempt to insert the same value concurrently both succeed. The resulting data structure contains a duplicate value that violates the invariants of the function. There is no instant in time when the operation can be considered to have taken place. Consequently, there is no semantic linearization point so the Immutable Data Structure is not semantically linearizable.

To imbue an Immutable Data Structure with the desirable properties of a linearizable object all of its access functions must be linearizable. Structural linearizability is the concern of the Canonical Binary Tree and the semantic linearizability is the concern of the ADT. The access functions of an Immutable Data Structure must have both semantic and structural linearization points in order that the Immutable Data Structure is linearizable.

### 4.2.2 Persistence

An Immutable Data Structure can be made structurally linearizable by requiring that the root is modified by an atomic instruction. Immutable Data Structures can be made semantically linearizable by recording the root at the moment of

functional invocation and validating that the root has the same value at the moment of response. If the value of the root has changed then the function is invalid.

The root can be modified while a function is active. When this occurs two functions can be reading different versions of the same data structure concurrently. The version referenced by the current value of the root is regarded as the current version. The version that was referenced by the root at some point in the past can be regarded as a past version. An Immutable Data Structure that permits access to past versions is called a persistent data structure.

An Immutable Data Structure in which all access functions record the root when they start and validate it before replacement is ephemeral. Functions acting on versions other than the current version are not successful. An ephemeral Immutable Data Structure is semantically linearizable.

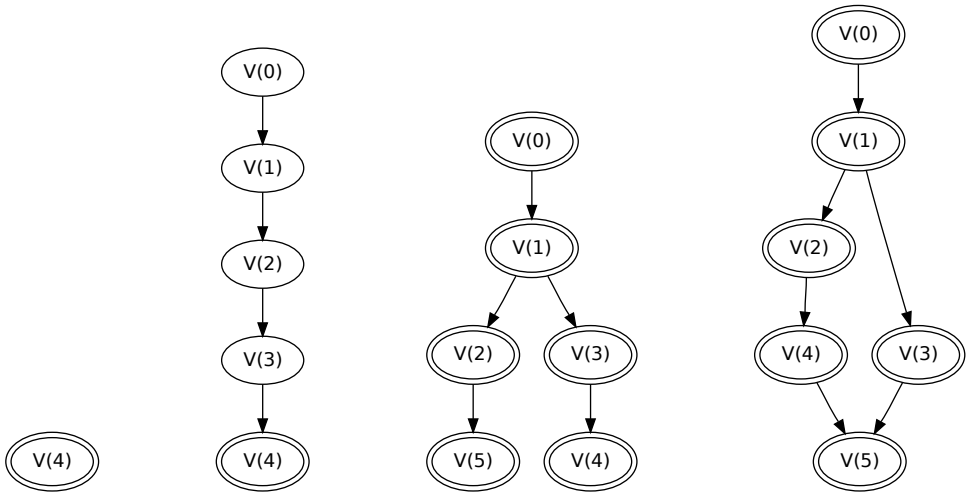
An Immutable Data Structure in which only mutating access functions record the root when they start and validate it before replacement is partially persistent. Past versions can be accessed by tardy readers but successful mutations always act on the most recent version. Mutations acting on past versions are not successful. A partially persistent Immutable Data Structure is structurally linearizable.

Section 4.2.3 describes the classification of persistent data structures.

Concurrent applications implemented using mutual exclusion always access the most recent version of a data structure, because it is the only version available. Programmers are not used to questioning whether the most up to date version is required. In many application domains the very latest version is not always required. For example, an on-line reservation system does not need to present the customer with the most recent version of inventory while they are browsing, but an up to date inventory is required when the customer is making a purchase. A partially persistent data structure is suitable for such a purpose because it ensures that mutations are serialised while permitting concurrent read accesses.

### 4.2.3 The classification of persistent data structures

Most of the data structures encountered by programmers are both mutable and ephemeral. A data structure is called a persistent data structure if it permits access to past versions and it is called ephemeral otherwise. The data structure is partially persistent if all versions can be accessed but only the most recent can be modified. The structure is fully persistent if every version can be both




(a) Ephemeral (b) Partial persistence (c) Full persistence (d) Confluent persistence

Figure 4.1: **Version graphs** for various types of persistent data structure. Versions which can be modified are represented by an ellipse with a double border.
(a) An ephemeral data structure restricts both read and write access to the most recent version.
(b) A partially persistent data structure restricts write access to the most recent version but permits read access to past versions.
(c) A fully persistent data structure permits both read and write access to all versions.
(d) A confluently persistent data structure permits both read and write access to all versions and provides a meld function that can combine past versions.

accessed and modified. The data structure is confluently persistent if it is fully persistent and has a meld operation which combines more than one version. The evolution of a persistent data structure can be represented by a directed graph in which each vertex represents a version and each edge a transformation between versions [Kap04].

A version graph is a representation of the evolution of a data structure.

Figure 4.1 illustrates the version graphs of common types of persistent data structure.

A persistence type is a restraint on the access to past versions. There are many possible restraints and therefore many persistence types in addition to the common types illustrated. For example, Conchon and Filliâtre describe a semi-persistent data structure which permits access only to those past versions that are ancestors of the most recent version [CF08].

An Immutable Data Structure is composed entirely of immutable values. Informally, we describe a persistent data structure as an Immutable Data Structure that possesses a look-up function capable of accessing past versions. However, not all persistent data structures are Immutable Data Structures. Persistent Data Structures constructed using the Fat Node or the Node Copying techniques contain singly assigned values that are not immutable. Persistent Data Structures constructed using the full copying or path copying technique are Immutable Data Structures.

### 4.2.4 Previous work

Kaplan provides an accessible introduction to persistent data structures [Kap04]. Driscoll, Sarnak, Sleator, and Tarjan describe persistent data structures in detail [DSST86].

Interest in persistent data structures originated from the development of text editors. Reps, Teitelbaum and Demers proposed that persistence can be used to create a text editor with an undo operation [RTD83].

Sarnak and Tarjan describe how persistent data structures can be applied to solve problems in computational geometry [ST86]. Computational geometry is a branch of computer science concerned with algorithms that can be stated in terms of geometry. Computation geometry is relevant to the subject of computer graphics and much of the early development of persistent data structures occurred in the mid 1980's when computer graphics became commercially important.

| **Persistence type** | **Permitted read access** | **Permitted write access** |
|---|---|---|
| Ephemeral | Most recent only | Most recent only |
| Partially persistent | Tardy reads | Most recent only |
| Fully persistent | Tardy reads | Tardy writes |
| Confluently persistent | Non-conflicting reads | Non-conflicting writes |

Table 4.1: **Persistence types for Transactional Data Structures** are characterised by the access to past versions that they permit.

The challenges of computational geometry required complex data structures so the first persistent data structures to be developed were certainly not the simplest. The subject of persistent data structure has always focused on highly optimised solutions to challenging problems whilst the implementation of simple persistent data structures has been somewhat neglected.

The research literature does not make a distinction between persistent data structures and Immutable Data Structures. The use of a persistent data structure in a concurrent execution environment and the access to a past version by a tardy reader have not been considered before.

To permit access to past versions a persistent data structure must implement some look-up mechanism. Generally, the research literature describing persistent data structures does not describe the version look-up function in detail. The focus of research is on the accessibility of past versions rather than the mechanisms to support that access. Previous work either neglects the details of the look-up function or assumes that versions are accessed by indexing an array mapping a version number to the root address of the corresponding version.

### 4.2.5 The classification of Transactional Data Structures

This thesis introduces Transactional Data Structures which are data structures that permit access to past versions, although not all accesses are successful.

For each type of persistent data structure there is a corresponding Transactional Data Structure. A Transactional Data Structure with linearizable access functions is ephemeral because accesses to past versions are not successful. A tardy reader is a function that accesses a past version of a persistent data structure which was the most recent version at the moment the function started its

execution. A Transactional Data Structure that permits tardy readers but prevents writers from successfully accessing all but the most recent version is partially persistent. A fully persistent Transactional Data Structure permits both tardy readers and mutations. A Transactional Data Structure which permits simultaneous accesses while ensuring serialisability is confluently persistent. It has a validate function that causes conflicting functions to be unsuccessful and a meld function that unites past versions.

Table 4.1 summarises the persistence types for Transactional Data Structures.

A Transactional Data Structure is necessarily an Immutable Data Structure because it permits simultaneous access to values.

## 4.3 Entanglement

Algorithms with irregular fine-grained parallelism are difficult to compose into transactions that are large enough to be worth executing concurrently. The solution is to compose such work into larger transactions that can be rolled-back to a previous state when conflicts are detected. Checkpointing reduces the amount of work wasted when a conflict occurs. Checkpointing and roll-back mechanisms enable the efficient concurrent execution of algorithms with irregular fine-grained parallelism.

The state of an algorithm can be represented by multiple data structures. To permit roll-back they must be checkpointed at a moment in time when they are mutually consistent.

The main contribution of this section is a technique for composing Immutable Data Structures to support checkpointing and roll-back. This section focuses on composing Immutable Data Structures so that they record a history of the algorithm they implement.

### 4.3.1 Fine grained irregular parallelism

The overhead of scheduling concurrent execution places a lower bound on the size of a transaction that is worth scheduling. Many algorithms exhibit irregular parallelism which is fine-grained and they appear not to be worth executing concurrently. A solution to this problem is to compose the work into transactions that are large enough to execute concurrently, but this increases the likelihood of conflicts and increases the amount of work wasted when conflicts do occur.

When composing fine-grained work into large transactions it is often desirable to create a checkpoint to reduce the amount of work wasted when conflicts occur. The amount of wasted work is reduced by rolling-back to a state prior to the conflict instead of entirely aborting a transaction. A mechanism for this checkpoints a consistent state of the algorithm, backtracks through previous states of the algorithm when a conflict is detected and rolls-back to a consistent state of the algorithm.

For example, consider an algorithm that removes an item from a queue, performs a function on that item, which can conflict with an instance of the function executing on another processor, and then places the result in a second queue. This is typical of a wide range of problems that exhibit fine-grained parallelism,

but for which no efficient concurrent algorithm is known. The operations on an item may be regarded as a single transaction, but such a transaction can be too small to be worth scheduling. The presence of an item in one and only one of the queues is an invariant of the algorithm. The algorithm is in a consistent state only when this invariant is true. Checkpoints should be taken at moments in time when the invariants are preserved.

When executing an algorithm speculatively it might be necessary to discard the speculative state, which can be represented by more than one data structure, and re-start execution from some consistent past state of the algorithm. The roll-back mechanism must roll-back so that it is not possible to observe an intermediate state in which one data structure involved in the algorithm is rolled-back and another not. One problem is to find a checkpointing mechanism which can ensure that all of the data structures involved in the algorithm are consistent at the moment the checkpoint is taken. Another problem is to find a backtracking mechanism that can backtrack through states of the algorithm to a previously checkpointed state. Another problem is to ensure that there is the appearance of instantaneous state transition during the roll-back.

### 4.3.2 The composition of Immutable Data Structures

The composition of fine-grained work into larger transactions can be achieved by composing the Immutable Data Structures involved in the algorithm into a single data structure. We call this technique Entanglement. In graph theory the Entanglement of a directed graph is a measure of how strongly the cycles of the graph are intertwined. In the context of Immutable Data Structures we take this to mean the composition of multiple Immutable Data Structures into one Immutable Data Structure through a process of adding links. Entanglement is achieved by referencing the root address of one Immutable Data Structure from the leaf of another Immutable Data Structure.

When applied to a single data structure entanglement is a look-up mechanism which makes a data structure persistent. A data structure that allows access to past versions only through entanglement is semi-persistent because only ancestors of the most recent version may be accessed. When applied to multiple data structures entanglement permits the composition of two data structures, which may not be persistent, to form a persistent data structure.

Expanding on our example, consider a process that removes lower case letters

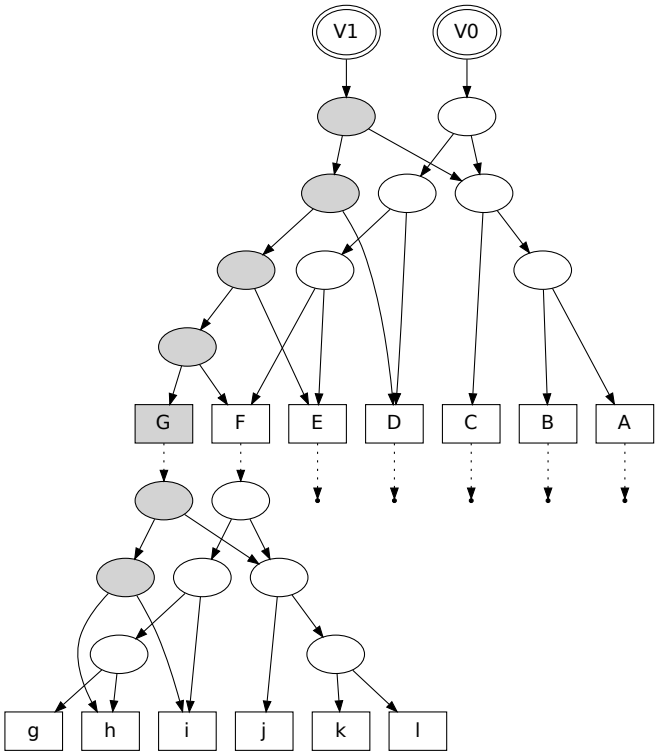



Figure 4.2: **A pair of entangled queues** is created by referencing the root of one queue from the leaf of another. In this example letters are removed from the parameter queue, shown in the lower part of the figure, converted to upper case and inserted into the result queue, shown in the upper part of the figure. In version V0 of the entangled data structure the parameter queue contains the lower case letter $g$. In version V1 the letter $g$ has been removed from the parameter queue and the upper case letter $G$ has been added to the result queue. The new root of the parameter queue is stored together with the converted letter in the result queue. In this figure prior versions of the parameter queue are hidden for clarity. The links from the leaves to the roots are dotted and the most recently created path is shaded.

from a queue, which we call the parameter queue, converts them to upper case and places them on another queue, which we call the result queue. The presence of a letter in one or other of the queues, but not both, is an invariant of the algorithm. The queues are said to be in a consistent state when this invariant is true. A consistent state of the algorithm can be checkpointed by recording a reference to one of the data structures in the leaf of the other. When a conflict is detected the reference in the leaf affected by the conflict indicates the root of the entangled structure at the moment in time prior to the conflict. The data structures can be rolled back to a consistent state by restoring this root. When the root is restored the invariants of both data structures are preserved. To ensure that the roll-back appears instantaneous the root of the entangled queues is modified atomically.

Figure 4.2 illustrates an operation on the two logically separate data structures which have been combined into a single structure by Entanglement.

Back tracking through past versions of an entangled data structure can be achieved by examining only the most recent version. A leaf created by a conflicting operation will contain a reference to the root of the entangled structure at the moment in time prior to the conflict.

At their most basic, Memory Transactions allow the atomic modification of discontinuous memory locations. Immutable Data Structures permit the atomic modification of discontinuous memory locations which are part of the same data structure and Entanglement extends this to locations which are not part of the same logical data structure but which are affected by the same algorithm. Linearizability is a composable property so the functions of our combined Immutable Data Structures may also be linearizable.

Entanglement is a low overhead checkpointing technique which works by recording the execution of an algorithm immutably instead of logging state changes. Entanglement satisfies our requirements for a solution to the problem presented by fine-grained parallelism as it permits backtracking through the entangled data structures and atomic roll-back to a state in which all the data structures involved in the algorithm are consistent.

### 4.3.3 Entanglement and Persistence

An Immutable Data Structure can be entangled with a past version by adding a reference to the root node of the version it was path copied from. This creates a

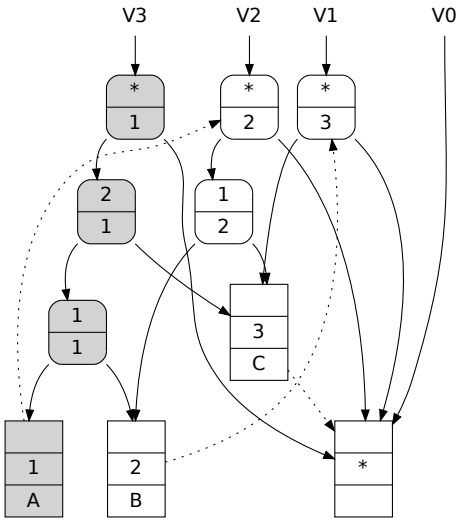



Figure 4.3: **A persistent Directed min-tree** is created by referencing the root of a version from the leaf of the past version from which the path was copied. In this example each leaf is linked to the version of the data structure from which it was created by path copy. The links from the leaves to the roots are dotted and the most recently created path is shaded. The Immutable Data Structure is persistent because past versions are accessible by a look-up function which follows these links.

link between a version of the data structure and a past version. A self-entangled Immutable Data Structure is persistent because past versions can be accessed by a look-up function which follows the links.

Figure 4.3 illustrates an immutable directed min-tree which is entangled in such a way that past versions can be accessed.

### 4.3.4 Previous work

Checkpointing and roll-back mechanisms have been proposed as solutions to the problem of fine-grained concurrency many times before [RW02],[HK08],[WS08]. The primary drawback of each of these mechanisms is the overhead of checkpointing and of backtracking.

One of the main applications of persistent data structures is their use in backtracking algorithms. Conchon and Filliâtre describe a semi-persistent data structure which only permits ancestors of the most recent version to be accessed

or updated. Conchon show how a semi-persistent data structure can be used for checkpointing and roll-back [CF08].

### 4.3.5 Low overhead checkpointing

In a typical implementation of checkpointing, changes are logged and values are written to memory more than once. The sources of the overhead of checkpointing are similar to those of maintaining duplicate copies of shared state. In our implementation data is written to memory once, so the overhead of checkpointing is reduced to that of storing a root address in each leaf of the entangled data structure. Entanglement provides a mechanism for checkpointing at very little additional cost because immutable data is written just once.

In a typical implementation, backtracking is a serial process which takes place while progress of the algorithm on other processors is halted. In our implementation the examination of past versions and the detection of conflicting operations can take place in parallel with the actions of the algorithm itself.

In a typical implementation, roll-back requires that the progress of the algorithm is halted so roll-back does not have concurrent semantics. In our implementation roll-back only affects those transactions involved in the conflicting operation. Entanglement provides a checkpointing mechanism with intuitive concurrent semantics because events occurring during the execution of the algorithm are checkpointed rather than system states.

## 4.4 Minimum Spanning Tree

The problem of finding the minimum spanning tree of a graph is typical of combinatorial problems which exhibit fine-grained irregular parallelism. Researchers have been frustrated in their attempts to exploit this parallelism and for many types of graph the fastest known algorithms are serial. To evaluate our checkpointing technique we measure the performance of a concurrent implementation of a minimum spanning tree algorithm that uses entangled Immutable Data Structures. We find that our concurrent implementation does not perform as well as a serial implementation.

The problem is to determine the minimum spanning tree of a connected undirected graph with weighted edges. The minimum spanning tree of a graph is an acyclic sub-graph which connects all of the vertices and has the minimum weight.

Sedgewick explains the problem in detail and describes a number of serial algorithms for computing the minimum spanning tree [Sed02]. The minimum spanning tree problem is one of the most important in combinatorics. Ahuja describes how many problems in network routing and linear programming are related to the problem of finding the minimum spanning tree [AMO93].

A minimum spanning tree is the tree of edges $T \in G(V, E)$ with minimal weight:

$$W(T) = \sum_{(u,v)\in T} W((u, v))$$

where $W((u, v))$ is the weight of an edge $(u, v)$.

The main contribution of this section is the evaluation of an algorithm which uses entangled Immutable Data Structures to facilitate the checkpointing of speculative execution. This section focuses on comparing the times taken to determine the minimum spanning tree of an undirected planar graph.

### 4.4.1 Experiment

Prim describes an algorithm to determine the minimum spanning tree of a graph [Pri57]. To evaluate our checkpointing technique we compare the performance of a concurrent implementation that uses entangled Immutable Data Structures with a concurrent implementation that uses Software Transactional Memory. We also compare these concurrent implementations with their serial counterparts.

Prim's algorithm is typically implemented using a mutable adjacency list to represent the graph and its minimum spanning tree and a priority queue from which minimally weighted edges are chosen. The implementation records whether edges belong to the minimum spanning tree by storing a value, which is usually referred to as a colour, as an edge property in the adjacency list. We call this a *Serial Graph Colouring Implementation* of Prim's algorithm. We use the adjacency list and the Serial Graph Colouring Implementation of Prim's algorithm from the Boost graph library. Siek, Lee and Lumsdaine describe the format of the adjacency list in detail [SLL01].

Section 4.4.3 describes the experimental set up.

Section 4.4.4 describes the Serial Graph Colouring Implementation of Prim's algorithm.

We develop an implementation of Prim's algorithm that uses a set, instead of graph colouring, to represent the minimum spanning tree. We call this a *Serial No-Colouring Implementation* of Prim's algorithm. This serial implementation is used to measure the effect that maintaining the minimum spanning tree in a set, rather than in the adjacency list, has on the execution time of the algorithm. We use a data structure from the C++ standard template library to implement the set of edges representing the minimum spanning tree and we also use a priority queue from the standard library [Jos99]. The graph is implemented by an immutable adjacency list from the Boost library.

Section 4.4.5 describes the Serial No-Colouring Implementation of Prim's algorithm.

A Concurrent Graph Colouring Implementation of Prim's algorithm must ensure the correctness of concurrent accesses to the edge colours.

Section 4.4.6 explains why a Concurrent Graph Colouring Implementation of Prim's algorithm that executes efficiently on a Chip Multi-Processor is difficult to construct.

Kang and Bader developed a concurrent implementation of Prim's algorithm using Software Transactional Memory [KB09]. We call this a *Concurrent Graph Colouring Implementation* of Prim's algorithm. The implementation allows some speculative execution by lazily detecting conflicting accesses to the graph colours.

Section 4.4.7 describes Kang's Concurrent Graph Colouring Implementation of Prim's algorithm.

We develop a concurrent implementation of Prim's algorithm which uses entangled Immutable Data Structure to allow checkpointing, backtracking and rollback to a previous state of the algorithm. We call this a *Concurrent No-Colouring Implementation* of Prim's algorithm. The implementation uses an immutable set, to represent the minimum spanning tree, and an immutable priority queue, from which minimally weighted edges are chosen. Both of these data structure are specialisations of the Canonical Binary Tree. The data structures are entangled to facilitate checkpointing. The graph is implemented by an immutable adjacency list from the Boost library.

Section 4.4.9 describes the Concurrent No-Colouring Implementation of Prim's algorithm.

### 4.4.2 Results

Our experiment shows that the Concurrent No-Colouring Implementation of Prim's algorithm takes longer to determine the minimum spanning tree of a graph than either the Serial Graph Colouring Implementation or the Serial No-Colouring Implementation for all graph sizes. The Serial No-Colouring Implementation of the algorithm takes about twice as long as the Serial Graph Colouring Implementation for all graph sizes.

Figure 4.4 illustrates a comparison of the elapsed time taken to determine the minimum spanning tree of a graph.

The Concurrent No-Colouring Implementation does not return the memory used by the Immutable Data Structures because they are persistent. Only 32 GB of memory are available to contain the persistent data structures on the evaluation hardware and this limited the maximum size of the graph whose minimum spanning tree could be determined to $2^{19}$ vertices.

The topology of the graphs representing the road maps of urban states differs from those of more rural states. This accounts for some of the variation in elapsed time taken to calculate the minimum spanning tree of states with similar numbers of vertices.

This thesis does not make any claims about the absolute performance of Immutable Data Structures. However, even when using 8 hardware threads the Concurrent No-Colouring Implementation takes longer to calculate the minimum spanning tree than either serial algorithm.

Section 4.4.10 describes how the performance of the Concurrent No-Colouring

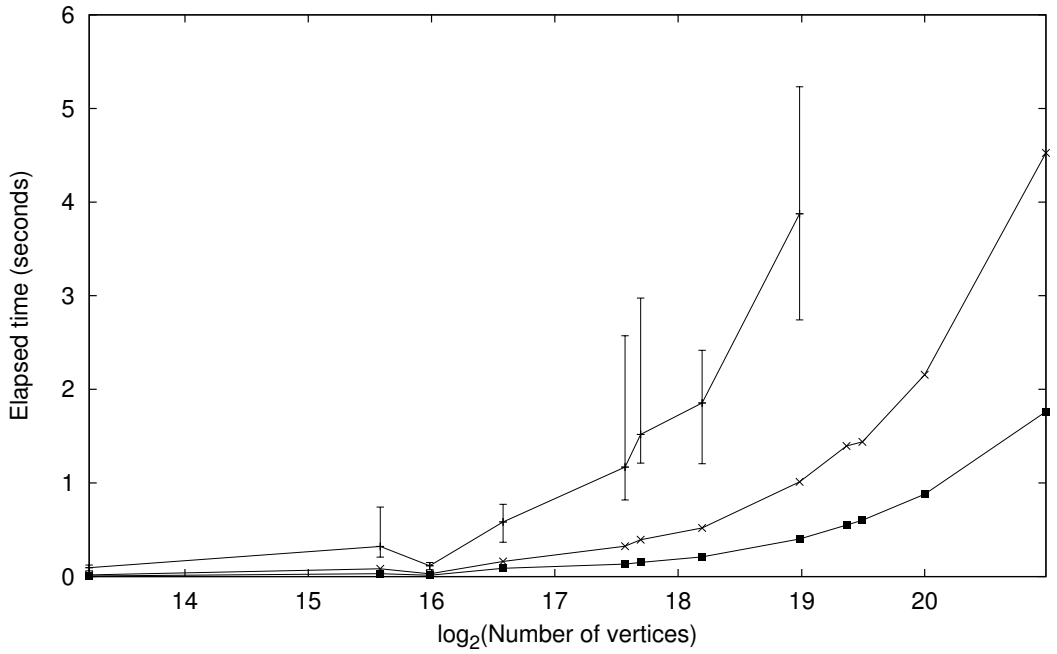



Figure 4.4: **Comparison of the elapsed time taken to calculate the minimum spanning tree** of planar undirected graphs representing road maps of US states.
The elapsed time taken by the Serial Graph Colouring Implementation ( ■ ), the Serial No-Colouring Implementation (x) and the Concurrent No-Colouring Implementation (+) is plotted against varying graph sizes. We uses a log scale to represent the number of vertices in the graph.
Eight hardware threads participate in the concurrent execution. Each hardware thread executes on a dedicated processor.
Figures given are the mean of 10 measurements. Bars indicate the range of elapsed times take by the concurrent implementation. The elapsed time taken by the serial implementations did not vary significantly.

Implementation can be improved.

Kang provided results for a Concurrent Graph Colouring Implementation which uses Software Transactional Memory [KB09]. Kang measured the elapsed time taken to calculate the minimum spanning tree of a planar graph with $2^{22}$ edges. Unfortunately, we were not able to calculate the minimum spanning tree of a graph of this size so we cannot make a direct comparison with Kang's result.

When a single hardware thread was used the elapsed time taken to determine the minimum spanning tree was 1143 seconds. When using 8 hardware threads, on the same core, a 14X speed-up was achieved. Kang attributed this super-linear speed-up to the sharing of cache by the hardware threads. When using 64 hardware threads, on 8 cores, a speed-up of 61.5X was achieved.

Kang attributed 98.5% of the execution time of the Concurrent Graph Colouring Implementation to the overhead of Software Transactional Memory. To overcome this overhead 64 hardware threads were applied to the problem. Kang concluded that "even with this level of scalability, our parallel algorithm runs only at the comparable speed to the single-threaded case which does not incur the Software Transactional Memory overhead" [KB09].

This thesis claims that the use of Immutable Data Structures can make concurrent programming easier. Kang described the difficulty of ensuring the correctness of the Concurrent Graph Colouring Implementation which uses locks to ensure serialisable access to the graph colours "this [acquiring locks] can lead to many complex scenarios that can cause race conditions, deadlocks, or other complications, and it is far from trivial to write correct and scalable code" [KB09]. Our Concurrent No-Colouring Implementation of Prim's algorithm requires no synchronisation. It is not prone to race conditions, because all shared data is immutable and it is not prone to deadlock because it does not block. Our Concurrent No-Colouring Implementation of Prim's algorithm was simpler to develop than the Concurrent Graph Colouring Implementation described by Kang.

### 4.4.3 Method

Demetrescu describes a set of planar graphs, representing road maps, which were used during the DIMACS implementation challenge competition [DGJe09]. These graphs are widely accepted as benchmarks for evaluating minimum spanning tree algorithms. Each graph node represents an intersection between roads and each graph edge is weighted with the distance between intersections. The graphs have

an average of 2.7 edges per vertex.

We evaluate our algorithm using a Sun Ultra Sparc T2 server [vRV+09]. The server contains a single Niagara Chip Multi-Processor which implements simultaneous multi-threading, it has 64 hardware threads and 8 physical processors.

Both the hardware and the graph data sets used in our evaluation are identical to those used by Kang [KB09].

We use an immutable priority queue implemented by a Directed min-tree specialisation of the Canonical Binary Tree. The Canonical Binary Tree is balanced but none of the optimisations, suggested in section 3.6.5 are implemented. We also use an immutable set implemented by an interval tree specialisation of the Canonical Binary Tree.

### 4.4.4 Serial Graph Colouring Implementation

Prim's algorithm is based on an observation known as the graph cut property [Pri57]. A graph cut partitions the vertices of a graph into two disjoint sets. Given a cut in the graph, any edge between the two sets which has a minimum weight belongs to some minimum spanning tree of the graph.

Prim's algorithm grows a minimum spanning tree iteratively from a single graph edge. The algorithm maintains a set of crossing edges, called the fringe, which is the set of edges with one vertex in the growing minimum spanning tree and one outside it.

Initially, both the minimum spanning tree and the fringe are empty. A single vertex is added to the minimum spanning tree and all of the edges from this vertex are added to the fringe. At each iteration the minimally weighted edge is removed from the fringe and added to the growing minimum spanning tree. This adds a new vertex to the minimum spanning tree and all of the edges from this vertex to vertices that are not already in the minimum spanning tree are then added to the fringe. The resulting fringe is processed by the next iteration. The algorithm completes when the set of vertices outside the minimum spanning tree is empty and edges in the minimum spanning tree connect all of the nodes in the graph.

Prim's algorithm can be implemented by using an adjacency list to represent the graph. Each graph edge has an associated weight. The minimum spanning tree is represented by a colour indicator associated with each edge. The fringe is represented by a priority queue which contains references to edges in the adjacency

list. The priority associated with an edge is the weight of that edge.

Serial Graph Colouring Implementations of Prim's algorithm are among the fastest known. Typically, high performance Serial Graph Colouring Implementations focus on improving the performance of the priority queue.

### 4.4.5 Serial No-Colouring Implementation

Prim's algorithm can also be implemented by using a set to maintain the growing minimum spanning tree instead of colouring graph edges. We call such an implementation a Serial No-Colouring Implementation because the adjacency list does not have any mutable properties. The set contains references to edges in the adjacency list and is used to determine whether an edge is part of the minimum spanning tree.

The Serial No-Colouring Implementation of Prim's algorithm uses three data structures. The graph is represented by a constant adjacency list with weighted edges. The fringe is represented by a priority queue which orders edges by weight. The minimum spanning tree is represented by a set of edges. In our implementation the priority queue is implemented using a vector from the standard library and the set is implemented using the standard map [Jos99].

The algorithm starts from a single node and adds edges to the minimum spanning tree iteratively. The edge with the minimum weight is removed from the priority queue and added to the set to indicate that it is part of the minimum spanning tree. This adds a new vertex to the minimum spanning tree. The edges including this vertex, which are not already present in the set, are added to the priority queue to complete the iteration.

To determine whether an edge is part of the minimum spanning tree a set look-up is performed. Typically, set look-up takes $O(log(n))$ time whereas accessing a mutable graph edge takes $O(1)$ time, so there is a significant overhead associated with looking up edges in a set. Consequently, it is not common to implement Prim's algorithm in this way. Our Concurrent No-Colouring Implementation maintains the minimum spanning tree an immutable set and uses a constant adjacency list to avoid sharing mutable data. We implement the Serial No-Colouring Implementation algorithm so that we can measure the effect of maintaining the growing minimum spanning tree in a set.

### 4.4.6 The concurrent implementation of Prim's algorithm

A concurrent implementation of Prim's algorithm may attempt to combine the minimum spanning trees of sub-graphs, produced by multiple processors, to form a larger minimum spanning tree.

Two sub-graphs are disjoint if they do not have any vertices in common. Two sub-graphs are adjacent if they are disjoint and there is at least one edge joining vertices which belong to different sub-graphs. The minimum spanning trees of adjacent sub-graphs can be combined by including the minimally weighted joining edge in the graph formed by their union. The minimum spanning trees of disjoint sub-graphs can be combined easily only by growing them until they are adjacent. The minimum spanning trees of overlapping sub-graphs are difficult to combine. Ideally, the minimum spanning trees of adjacent sub-graphs should be identified and combined.

The graph cannot be decomposed into disjoint sub-graphs before the algorithm starts because this problem is more difficult than the minimum spanning tree problem itself. Dor and Tarsi prove that the problem of decomposing a graph into disjoint sub-graphs with no common edges is NP-Complete [DT92].

To permit the combination of minimum spanning trees created concurrently an algorithm can check that their sub-graphs are disjoint each time a vertex is added. This imposes a synchronisation overhead on the concurrent implementation. The implementation can speculate that sub-graphs are disjoint to reduce the synchronisation overhead. However, it must be prepared to roll-back the algorithm to the point at which adjacency first occurs.

A concurrent algorithm has the potential to demonstrate speed-up provided it is faster to check and combine the minimum spanning tree of sub-graphs with the growing minimum spanning tree than to grow the minimum spanning tree by the corresponding amount. Once checked, the merging of minimum spanning trees is straightforward. The problem is to create minimum spanning trees in such a way that they can be combined without incurring a significant synchronisation overhead.

### 4.4.7 Concurrent Graph Colouring Implementation

Kang describes a Concurrent Graph Colouring Implementation of Prim's algorithm which uses Software Transactional Memory [KB09]. Kang's implementation is an application which implements Memory Transactions rather than an application developed within an existing Software Transactional Memory framework.

Memory operations acting on the graph colours can cause race conditions, so simultaneous access must be restricted. A naïve implementation might serialise every access to the colours but this effectively serialises the entire algorithm, negating any benefit from concurrent execution. Kang uses Software Transactional Memory to support speculation by buffering memory operations and detecting conflicts. Conflicting accesses to the graph colours are rare so it can be beneficial to speculate that a conflict did not occur.

Kang's Concurrent Graph Colouring Implementation relies on the semantics of memory transactions to avoid data races. Each thread colours the vertices of its own minimum spanning tree and also colours all of the neighbours of the marked vertices with a unique colour. The process of picking one vertex and then applying an operation to its neighbours is encapsulated in a Memory Transaction. Conflicts are detected by checking the colour of vertices in graph to determine whether another processor has included the node in its minimum spanning tree.

Our experimental results are directly comparable to those of Kang because we use identical graph data sets and identical hardware. Unfortunately, Kang was not able to report concurrent speed-up because the overheads associated with Software Transactional Memory exceed the benefits of concurrent execution.

### 4.4.8 Previous work

Borůvka described a concurrent algorithm to determine the minimum spanning tree of a graph nearly a century ago [NMN01]. However, realising speed-up from concurrent execution has proved difficult. Chazelle describes an algorithm which has the minimal amortised time [Cha00]. Vineet, Harish, Patidar and Narayanan describe an algorithm that makes use of a graphics processing unit. This speeds-up the calculation of some very large minimum spanning trees by an order of magnitude when compared with a serial implementation [VHPN09]. In practice, the fastest methods for finding the minimum spanning tree of a dense graph are

based on a serial implementation of Prim's algorithm. Bazlamacci and Hindi present a survey of high performance minimum spanning tree algorithms [BH01]. In the fastest, the fringe is represented by a priority queue based on a Fibonacci heap. Weiss describes the implementation of a Fibonacci heap data structure in detail [Wei93].

Dice, Lev, Moir and Nussbaum describe a Concurrent Graph Colouring Implementation of Prim's algorithm which uses Hardware Transactional Memory [DLMN09]. Dice uses a form of Hardware Transactional Memory known as speculative lock elision, which permits speculative access to the graph colours, while relying on hardware to detect conflicting accesses [RG01]. Dice implements this algorithm on the Sun ROCK processor [CCE+09]. The implementation difficulty and the modest speed-up observed may have been factors contributing the cancellation of the ROCK processor, which we described in section 1.2.6.

### 4.4.9 Concurrent No-Colouring Implementation

We develop a Concurrent No-Colouring Implementation of Prim's algorithm. One processor is designated as the main processor and it grows the minimum spanning tree of the entire graph. The other processors are designated as helper processors and they build the minimum spanning trees of sub-graphs. The main processor occasionally checks whether the minimum spanning trees of these sub-graphs overlap with the growing minimum spanning tree. When overlap is detected the sub-graphs produced by the helper processors are rolled-back to the state they were when they were adjacent to the growing minimum spanning tree. They are then combined with the growing minimum spanning tree and their fringes are added to the fringe of the growing minimum spanning tree. The helper processors contribute to reducing the elapsed time taken to calculate the minimum spanning tree.

We use the Serial No-Colouring Implementation of Prim's algorithm on the main processor because it makes the merging of the sub-graphs built by the helper processors easier. However, we could have chosen a Serial Graph Colouring Implementation, in which case only the main processor would access the graph colours.

The helper processors create minimum spanning trees in such a way that the execution of their algorithm can be rolled back to a previous state. The algorithm executing on the helper processors is also a Serial No-Colouring Implementation

of Prim's algorithm. It uses an immutable set to identify edges in the minimum spanning tree and an immutable priority queue to represent the fringe. Both of these Immutable Data Structures are specialisations of the Canonical Binary Tree. The immutable set and the priority queue are entangled so that they can both be rolled-back to a mutually consistent state.

The Immutable Data Structures are entangled by storing the root of the immutable priority queue in the leaves of the immutable set. Each leaf contains an edge and a reference to the root of the past version of the priority queue from which it was removed. The leaf also contains the corresponding root of the past version of the set. This entanglement checkpoints the state of the algorithm at the start of each iteration. The checkpoint allows the set and the priority queue to be restored to a consistent state in which the set represents a minimum spanning tree and the priority queue its fringe.

At some moment in time the minimum spanning trees of sub-graphs built in isolation are checked by the main processor and possibly combined with the growing minimum spanning tree. The frequency at which the minimum spanning trees of sub-graphs are checked is a heuristic of the algorithm which does not affect its correctness.

If necessary, the main processor examines the set produced by a helper processor and backtracks through past versions by traversing the leaves of the set. The algorithm must backtrack to the moment in time when the first common edge was added. An ordinal number is used to determine the first common edge. Each iteration of Prim's algorithm performed by the helper processor causes a process-unique ordinal number to be incremented. This ordinal number is stored in a leaf of the set. The first common edge is the common edge with the lowest ordinal number.

The main processor does not block the execution of the helper processors while backtracking. The data structures produced by the helper processors are immutable and can be examined by the main processor without requiring synchronisation.

When backtracking detects overlap the state of the algorithm is rolled-back to the point at which the first common edge was added. The traversal finds the leaf containing a common edge with the lowest ordinal number. This leaf contains a reference to the past version of the set which represents the minimum spanning tree of a sub-graph which is adjacent to the growing minimum spanning

tree. This leaf also contains a reference to the version of the priority queue which represents the fringe of the minimum spanning tree of the sub-graph.

A merge process combines the past version of the minimum spanning tree of the sub-graph with the growing minimum spanning tree and the past version of the fringe of the sub-graph with the fringe of the growing minimum spanning tree. The helper processor is stopped after the merge to reduce contention in the path to memory.

### 4.4.10 The performance of the Concurrent No-Colouring Implementation

The performance of a concurrent implementation of Prim's algorithm is dependent on both the topology of the graph and the choice of starting vertices. Heuristics can guide the choice of starting vertices used by the helper processors. Our concurrent implementation chooses the starting vertices for each processor at random. We have focused exclusively on planar graphs, so the performance of the Concurrent No-Colouring Implementation when applied to other graph topologies remains to be investigated.

The elapsed time taken by the Concurrent No-Colouring Implementation of Prim's algorithm is dependent on heuristics such as the frequency at which the main processor checks whether the minimum spanning trees of sub-graphs, produced by the helpers, overlap with the growing minimum spanning tree. The checking process is performed by the main processor. There is a trade off between the frequency of checking and the benefit from merging a minimum spanning tree produced by a helper. We found that the best results were obtained when the main processor checked for overlap infrequently. Our implementation adds $2^{10}$ nodes to the growing minimum spanning tree between checks. There is little chance of overlap when the minimum spanning trees are small and little to be gained from merging sub-graphs when the growing minimum spanning tree is near completion. Checking for overlap is probably most advantageous when about one quarter of the planar graph is covered by the minimum spanning trees of sub-graphs. However, we did not attempt to find the optimum interval because it is dependent on the topology of the graph.

When processing a large graph a high proportion of memory accesses result in cache misses. Our implementation uses only 8 hardware threads, one on each

physical processor. We found that using additional hardware threads did not reduce the elapsed time taken to determine the minimum spanning tree. This indicates that the elapsed time is bound by the performance of the memory subsystem. Jacob describes how the Niagara processor in the Sun Ultra Sparc T2 server has four memory controllers and uses fully buffered DIMM memory to permit fast processing for frequent cache misses [Jac09]. However, we also carried out experiments using an Intel core i7 system. We found that for graphs of less than $2^{18}$ vertices the elapsed time taken by the Intel system was less than that obtained by processing the same graph on the Sun Ultra Sparc T2. The restricted memory available on the Intel system prevented a comparison for larger graphs.

The Sun Ultra Sparc T2 server has 32 GB of main memory. Our algorithm does not return any memory so the size of graphs considered during the evaluation are restrained by the available memory. There are many ways that the memory restraint could be lifted. For example, the helper processors could return the memory occupied by a sub-tree after it is merged with the growing minimum spanning tree.

The size of the node used to implement the Canonical Binary Tree and the number of nodes accessed while balancing of the tree are important factors affecting the performance of the No-Colouring Implementations because they contribute to the effective memory bandwidth of the implementation.

Section 6.3.7 describes how node size affects the performance of algorithms which use specialisations of the Canonical Binary Tree.

# Chapter 5

# Concurrency Control

Distributed concurrency control is scalable but centralised concurrency control is not. Transactional Memory systems use centralised concurrency control to ensure consistent access to shared state but a scalable concurrent system should use distributed concurrency control to ensure consistent access to a particular object. This chapter describes how a distributed concurrency control protocol can serialise access to an Immutable Data Structure.

Section 5.1 identifies the choice of concurrency control mechanism as one of the most significant decisions taken when designing a concurrent system.

Section 5.2 describes how functions acting simultaneously on an Immutable Data Structure can be serialised.

Section 5.3 describes how an Immutable Data Structure can be made confluently persistent.

## 5.1 Distributed Concurrency Control

Transactional Memory systems require the application program to interact with a centralised transaction manager but this interaction makes programming difficult and restricts scalability. This section proposes using distributed transaction management to ensure the correct concurrent execution of Memory Transactions. Distributed transaction management makes concurrent programming easier and concurrent systems more scalable.

The main run-time component of a Transactional Memory system is the transaction manager which ensures the correct concurrent execution of Memory Transactions. Correctness is usually taken to mean that the result of the concurrent execution is equivalent to the result obtained by executing the transactions in some serial order. A transaction manager ensures serialisability by enforcing a concurrency control protocol and the choice of protocol dictates the design of the transaction manager.

Section 5.1.3 introduces Transaction management.

The main contribution of this section is the observation that the correct concurrent execution of Memory Transactions can be ensured without centralised transaction management. This section focuses on ensuring the serialisable execution of functions acting on an Immutable Data Structure.

### 5.1.1 Centralised concurrency control

Centralised transaction management restricts the scalability of a concurrent system as some part of the management processing is necessarily serialised. As the number of concurrent processors increases the time spent within the serialised part grows and eventually dominates the execution time of the concurrent system. Amdahl's law imposes restrictions on the scalability of a system with centralised transaction management.

A concurrent application communicates with the transaction manager to signal that it is ready to commit a transaction and the transaction manager then responds. This two way communication cannot be easily hidden by abstraction. The orchestration of communication with the transaction manager makes concurrent programming difficult.

Centralised transaction management makes it difficult for programmers to use

Memory Transactions in existing programs. To make use of Memory Transactions a programmer must adapt a program to fit into a transaction processing framework. This is an obstacle to the integration of Memory Transactions into existing software and it is a barrier to the adoption of Transactional Memory.

The solution to these difficulties should ensure the serialisability of concurrent Memory Transactions without requiring a centralised transaction manager.

### 5.1.2 Distributed concurrency control

Distributed transaction management is scalable because it does not require a centralised mechanism to enforce concurrency control. It makes concurrent programming easier because programmers do not need to coordinate the application's interaction with a centralised system and it makes the use of Memory Transactions in existing applications easier by alleviating the need to integrate a concurrent application into a centralised transaction management framework.

A distributed transaction manager can make the decision whether to commit or abort a transaction independent of operations taking place on other processors because a distributed concurrency control protocol requires only information local to a processor. It does not depend on any information about concurrently active transactions so in a distributed system it is not necessary to orchestrate the interaction of transactions on multiple processors. Each processor can implement transaction management independently.

A distributed transaction manager can make the decision whether to commit or abort a transaction using only local information about the transactions that affect an object. It does not depend on information about accesses to any other objects so in a distributed system each transaction manager can maintain information about the objects that it manages and go about making its decisions independent of the action of other transaction managers. Each object can implement transaction management independently.

A distributed transaction manager does not attempt to serialise access to multiple objects. Groups of objects that require mutually consistent access are logically connected and should be combined into a single object for the purposes of concurrency control.

A fully distributed concurrency control protocol requires no communication between transaction managers whatsoever as it can be implemented on a per processor per object basis.

### 5.1.3 Transaction management

Database systems divide transaction management into three distinct tasks: concurrency control, contention management and scheduling. Concurrency control is the task of ensuring correct concurrent execution by enforcing serialisability. Contention management is the task of guaranteeing progress. Scheduling is the task of load-balancing the execution between processors. We make a distinction between these tasks and consider each independently. However, Transactional Memory systems tend not to treat these aspects of transaction management as distinct. Consequently, transaction management in Transactional Memory systems tends to be difficult to characterise.

A transaction manager ensures that concurrent execution is correct by ensuring that it is equivalent to a serial execution. Determining whether a concurrent execution is serialisable is a NP-Complete problem [Pap79]. A transaction manager enforces a concurrency control protocol which ensures that all conforming transaction schedules are serialisable.

A transaction manager applies the rules of the concurrency control protocol to determine whether a transaction can commit or not. A concurrency control protocol can be viewed as a set of invariants and a binary function which ensures them. In the Transactional Memory literature the action of this function is referred to as validation.

A concurrency control protocol can be enforced either pessimistically, by a scheduler which checks that each operation conforms to the invariants of the concurrency control protocol before it is executed, or optimistically, by a certifier that enforces the concurrency control protocol when a transaction commits. The Transactional Memory literature refers to pessimistic concurrency control as eager validation and optimistic concurrency control as lazy validation. Many Transactional Memory systems employ mixed protocols detecting some types of conflict eagerly and others lazily.

A concurrency control protocol considers conflicting read and write operations acting on variables. These conflicts can be either between a read and a write or between two writes. Different concurrency control protocols can be applied independently to each type of conflict. A concurrency control protocol considers conflicts between these operations without regard to the values of the variables. Transactional Memory systems can be roughly divided into those which regard the variables as objects and those which regard them as memory words.

A transaction certifier requires a record of the read and write operations on variables and the transactions that issued them. The association between variables and transactions can be maintained by placing a transaction identifier within each affected object. It can also be maintained by associating a transaction with a list of addresses or object identifiers representing its read and write set. A certifier also requires meta-data, such as time stamps, relating to the operations on each variable.

The interaction between weakly isolated transactions is complex so concurrency control is simplified by strong isolation. The validation process is made simpler if it is known that all the values read by a transaction were written by transactions that have already committed.

### 5.1.4 Previous work

Bernstein, Hadzilacos and Goodman describe concurrency control and transaction management in a book entitled "Concurrency Control and Recovery in Database Systems" [BHG87]. Özsu and Valduriez describe distributed transaction management and distributed concurrency control in database systems [ÖV99].

Kotselidis et al. develop the idea of distributing Memory Transactions across a computing cluster [KAJ+07]. Hammond et al. describe the TCC protocol which is a centralised broadcast based concurrency control protocol enforced by a centralised transaction manager [HCW+04]. Kotselidis describes a centralised broadcast concurrency control protocol based on the TCC protocol which ensures the serialisability of transactions both within a Chip Multi-Processor and across the cluster. However, in a computing cluster the latency and bandwidth restrictions of Inter-Processor Communication are more severe and the problems created by centralised transaction management are more apparent than in a Chip Multi-Processor. Kotselidis et al. found that the centralised nature of transaction management made concurrent programming difficult and restricted the scalability of the system [KAJ+08]. These problems were not easily overcome despite a significant engineering effort.

### 5.1.5 Time Stamp Ordering

There are several distributed concurrency control protocols described in the literature and each can be applied independently to different types of conflict. Both

the Time Stamp Ordering protocol and Reed's Multi-version Time Stamp Ordering protocol can be implemented without blocking so a distributed transaction manager can enforce either concurrency control protocol [BHG87] [Ree79].

Pessimistic concurrency control requires fine-grained memory serialisation and a strongly coherent memory model. As the number of processors on a Chip Multi-Processor increases the overhead of implementing fine-grained memory serialisation in hardware increases [HP06b]. The Transactional Memory literature therefore makes a strong case for optimistic concurrency control [HLR10].

The Time Stamp Ordering concurrency control protocol can be enforced optimistically by a Time Stamp Ordering certifier which associates each transaction with a unique monotonically increasing time stamp. The certifier maintains a set containing the variables read and written by a transaction and also associates each variable with the time stamp of the transaction that wrote the variable and the highest time stamp of any transaction to have read the variable. When a transaction commits the certifier examines the read and write time stamps of all of the variables affected by the transaction and if the operations conform to the protocol then the transaction can commit, otherwise it must be aborted.

### 5.1.6 Programmer productivity

Ease of problem diagnosis is an important contributor to overall programmer productivity. It is often very difficult to diagnose problems in a concurrent system where concurrency control is enforced by a locking protocol because it can be difficult to determine which transaction wrote a particular value to a variable. When Time Stamp Ordering is used as a concurrency control protocol transactions appear to occur in the order of their starting time stamps. The order in which transactions are executed can be recorded and this aids the diagnosis of any problems that occur when a transactional system is executing concurrently. The order of the memory operations at the time the problem occurred can be determined using the read and write time stamps associated with variables so it is possible to diagnose a problem from a core dump taken at the moment in time that a problem occurred.

Ease of problem reproduction is an important contributor to overall programmer productivity. It is often very difficult to reproduce a problem in a concurrent system where concurrency control is enforced by a locking protocol because the serial order, to which the execution should be equivalent, may be unknown. When

Time Stamp Ordering is used as the concurrency control protocol the serial order is given by the order of the transaction time stamps so it is possible to reproduce problems by executing the transactions serially in the order given by their time stamps.

## 5.2 Serialisability

A transaction manager should ensure that the effects of functions accessing an Immutable Data Structure are equivalent to those of a serialisable execution. Functions acting on an Immutable Data Structure can be mapped onto abstract read and write operations on variables and a concurrency control protocol can be enforced on these operations to ensure serialisability. The protocol permits functions to act on an Immutable Data Structure simultaneously, although not all of them succeed.

Functions acting concurrently on an Immutable Data Structure can be made linearizable but enforcement of this property restricts scalability because when two functions simultaneously act on the same data structure only one of them is successful. To improve scalability functions should be able to act on the same data structure simultaneously. The problem is how to ensure the serialisability of functions that simultaneously act on a data structure?

The main contribution of this section is a technique for making functions simultaneously acting on an Immutable Data Structure serialisable. This section focuses on mapping these functions onto abstract read and write operations on the variables considered by a concurrency control protocol.

### 5.2.1 Simultaneous access

This section considers how two functions can be permitted to act simultaneously on an Immutable Data Structure.

When functions simultaneously access a semantically linearizable Immutable Data Structure only one of them succeeds.

Section 5.2.3 discusses the semantics of functions concurrently accessing an Immutable Data Structure.

The property of immutability allows the implementation of a mechanism that permits simultaneous access while ensuring that the actions of one function appear to precede those of the other.

Section 5.2.4 discusses the semantics of functions simultaneously accessing an Immutable Data Structure.

The problem of ensuring the serialisability of functions which simultaneously access shared data has been successfully solved in the context of database systems. A database system ensures the correct concurrent semantics of transactions

simultaneously acting on a relational table by serialising the file operations on the records that implement it. The file operations on these records are mapped to abstract read and write operations and the transaction manager enforces a concurrency control protocol on these operations to ensure that their effect is equivalent to a serial execution. In a database system the variables on which the concurrency control protocol acts are records and the operations that it considers are file operations. The records are, typically, the leaves of a B+tree data structure which maintains the application data. There is a layer of abstraction between a database table and the B+tree which implements it, so there is a complex relationship between a transaction expressed in SQL and the abstract read and write operations considered by the concurrency control protocol [GR92].

### 5.2.2 Implementing concurrency control

The problem of ensuring the serialisability of functions simultaneously accessing an Immutable Data Structure is one of mapping the functions onto a concurrency control protocol and enforcing that protocol.

A concurrency control protocol is normally expressed in terms of a history of abstract read and write operations on a system of variables so the functions must first be mapped to operations on a set of variables.

Section 5.2.6 describes how an Immutable Data Structure is mapped onto variables for the purposes of concurrency control.

Section 5.2.7 describes how the functions acting on the Immutable Data Structure are mapped onto abstract read and write operations on variables.

The concurrency control protocol is enforced by a validate function that ensures that conflicting operations conform to the protocol. Functions which contain non-conforming operations are rejected.

Section 5.2.8 describes how conflicting operations can be detected by a validate function.

A concurrency control protocol can be expressed as a set of invariants on meta-data associated with abstract read and write operations. The Time Stamp Ordering concurrency control protocol requires that these abstract read and write operations are associated with time stamp meta-data which the functions collect and record.

Section 5.2.9 describes how information about the operations can be recorded as meta-data within an Immutable Data Structure and how the Time Stamp

Ordering concurrency control protocol can be enforced.

### 5.2.3 Concurrent semantics

Immutable Data Structures have the property of structural linearizability. Structural modifications take place in isolation and appear to be atomic so no matter which functions are concurrently applied to the data structure the resulting structure is always a valid structure. However, the property of structural linearizability does not endow the ADT presented by the data structure with any meaningful concurrent semantics. The concurrent behaviour of a structurally linearizable data structure is uncertain because it may not reflect the action of all of the functions that have acted upon it.

An Immutable Data Structure can be made semantically linearizable which ensures that concurrently executing functions appear to take place at a single moment in time. No matter which functions are concurrently applied to the data structure the resulting structure is equivalent to some serial execution of those functions. The concurrent semantics of a semantically linearizable Immutable Data Structure are intuitive because functions appear to occur in some serial order. However, only one of the functions simultaneously accessing the Immutable Data Structure is successful and this limits scalability.

An Immutable Data Structure that permits tardy read access to past versions while ensuring the serialisability of mutating functions has the property of partial persistence. This ensures that mutations appear to take place at a single moment in time but the result of non-mutating functions do not necessarily reflect the latest version of the data structure. The concurrent semantics of a partially persistent data structure are easy to understand and can be useful. Some applications require that mutations are serialised to ensure that a data structure eventually reflects their effects, while permitting tardy read accesses. Partial persistence can improve the scalability of concurrent applications because mutating and non-mutating functions can execute at the same time.

For example, communication routers usually map symbolic names to IP addresses using a map based data structure called a PATRICIA trie [Mor68]. The map is read each time a message is processed, which occurs frequently, but it is only written when new IP addresses are added, which happens rarely. If the penalty for an incorrectly routed message is small then a partially persistent map can be appropriate. A partially persistent map permits read-only and write

accesses to take place simultaneously, while serialising writes. It separates the concerns about the structure of the data, which is ensured by serialising access to the root, from both concerns about the semantic order of modifications, which is ensured by serialising writes, and concerns about the routing of messages.

### 5.2.4 Simultaneous semantics

This section considers the behaviour programmers might expect from simultaneous accesses to an ADT.

#### Deque

The most desirable behaviour for a deque would be for it to permit accesses to both ends simultaneously. This behaviour can be described in terms of the serialisable access to two variables, each representing a different end of the queue.

For example, a Producer Consumer Queue is an application of a deque used to communicate between concurrent processes. One process inserts elements on one end of the queue and another process removes them from the other end. It is desirable that processes can simultaneously insert and remove elements.

#### Map

The most desirable behaviour for a map would be for it to permit simultaneous access to different groups of elements while ensuring that the accesses to a single group of elements are serialisable. This can be described in terms of the serialisable access to variables.

For example, fine-grained serialisability can be ensured by associating a small discrete group of elements with a variable and coarse-grained serialisability can be ensured by associating a larger discrete group with a variable.

#### Priority queue

The most desirable behaviour for a priority queue would be for it to permit simultaneous insertion of elements into the queue while ensuring that the highest priority element is removed in serialisable order. The desirable behaviour can be described in terms of the serialisable access to two variables, one representing the highest priority element and the other representing the rest of the priority queue.

For example, an event scheduler is an application of a priority queue used to communicate between concurrent processes. A process requesting an event inserts an element onto the priority queue. Another process services the queue by removing the highest priority element from the priority queue. It is desirable that elements can be inserted by one process while the highest priority element is removed by another. The insertion of elements onto the priority queue should be serialisable and the removal of the highest priority element should also be serialisable but it is not necessary to impose a serial order on all operations.

#### Vector

The most desirable behaviour for a vector is semantic linearizability. A mutating function has the potential to modify the relationship between the ordinal numbers and values of any element in the data structure so simultaneous access cannot be permitted. The desirable behaviour can be described in terms of the serialisable access to a single variable representing the entire vector.

### 5.2.5 Previous work

The problem of permitting simultaneous access to the data structures used in on-line gaming is commercially important and has received significant attention. Multi-player on-line game applications are usually constructed around a massive aggregate data structure called the game tree. The game tree contains information about all of the objects within the game such as players and weapons and their relationships. Actions in the game, such as a player dropping a weapon and another player picking it up, are represented by actions on the game tree. Access to the game tree is typically serialised by mutual exclusion. Sweeney identifies the serial nature of actions on the game tree as a significant obstacle restricting the performance of on-line games [Swe06]. Gajinov et al. describe how Transactional Memory can be used to improve the performance of an on-line game by allowing actions on the game tree to execute speculatively [GZU+09]. The challenge is to ensure correctness while permitting multiple functions to access the data structure simultaneously.

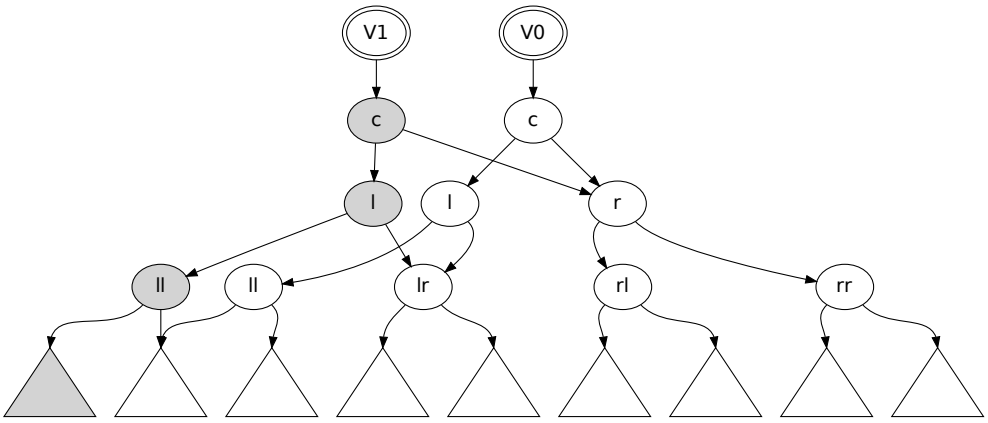



Figure 5.1: **Labelling of variables in the cap of an Immutable Data Structure.** The variables in the cap of an Immutable Data Structure are labelled **ll**, **l**, **lr**, **c**, **rl**, **r** and **rr**. The shaded path represents new instances of the variables. The triangles represent subtrees suspended by the cap.

### 5.2.6 Variables

For the purposes of concurrency control an Immutable Data Structure can be regarded as a system of variables. A concurrency control protocol ensures the correct concurrent semantics of abstract read and write operations acting on these variables. The functions implemented by the Canonical Binary Tree do not maintain any mutable state so variables must be maintained immutably within the Immutable Data Structure itself. The relative position of a vertex to the root can be regarded as a variable and the annotation of a vertex can be regarded as its value. A variable can have different values in each version of the data structure even though the vertices that implement it are immutable.

Figure 5.1 illustrates the labelling of variables within a tree. Each variable represents a position relative to the root. The value of a variable can only be altered by creating a new version of the tree.

For the purposes of concurrency control it is only necessary to consider the variables represented by a subset of the relative positions in the tree that we call the cap. A version of the Immutable Data Structure can be larger or smaller than the cap so a vertex may or may not correspond to a variable in cap. When the data structure is larger than the cap the variables represented by the leaves

of the cap act as proxies for the subtrees which they suspend.

The desirable behaviour of a deque can be described in terms of the serialisable access to three variables **l**, **r** and **c**. Variables **l** and **r** represent the front and back of the deque respectively and the variable **c** represents the empty queue.

A map can be represented either at a fine level of granularity, or at a coarse level of granularity. The size of the cap determines the level of granularity.

A priority queue can be represented by two variables. The variables **c** and **l** represent the highest priority element and the rest of the priority queue respectively.

A sequence can be represented by a single variable **c**.

### 5.2.7 Functions and operations

When a variable is read or written information about the operation is recorded in the data structure. A variable in the cap is either read, written or unaffected by a function. A function is implemented by a path copy which creates new nodes. A node records the type of abstract read and write operations that created it, along with the time stamp meta-data required to enforce the concurrency control protocol.

For the purposes of concurrency control the annotation of a node corresponds to the value of a variable. An operation is regarded as writing a variable if the annotation associated with its relative position in the tree changes. A read operation records an access to a variable which did not change its value. When the annotation associated with a relative position that is not in the cap changes a write operation is recorded as acting on the variable that corresponds to the node's most junior ancestor in the cap.

A *query*() function causes every variable on the path to be read but it is only necessary to record reads in nodes corresponding to variables in the cap. The nodes on the path read by the query function that correspond to variables in the cap are copied so that read operations can be recorded.

The *insert*() and *delete*() functions also read every variable on the path but they also cause the annotation of some of the variables on the path to change. The variables in the cap act as proxies for the variables in the subtrees they suspend so a change in the annotation of a node at some point on the path is represented by a write operation on a variable within the cap.

Figure 5.2 illustrates the abstract read and write operations on variables

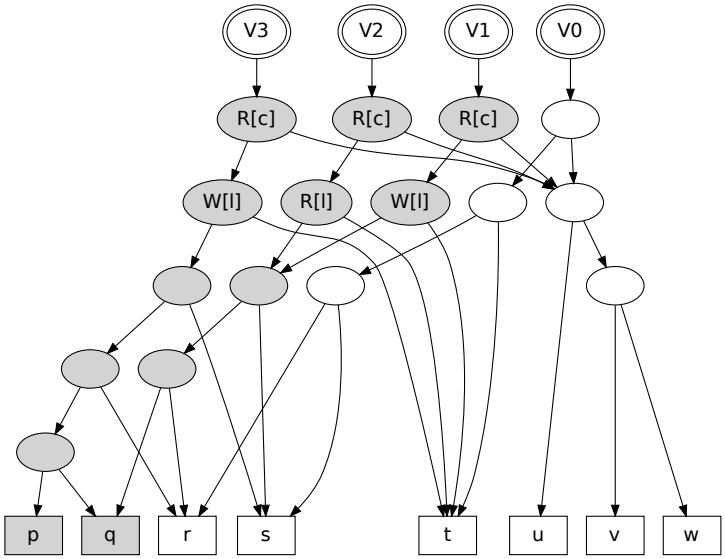



Figure 5.2: **Operations on variables in the cap of a deque.** The cap contains three variables **l**, **r** and **c**. The function $Push\_front(q)$ acts on version V0 containing $\{r, s, t, u, v, w\}$ to create version V1 containing $\{q, r, s, t, u, v, w\}$. The operations {**W**[**l**], **R**[**c**]} are recorded in the vertices corresponding to the cap. The function $Front()$ acts on version V1 to create version V2. The vertices corresponding to the cap are copied to record the operations {**R**[**l**], **R**[**c**]} performed by this non-mutating access. The function $Push\_front(p)$ acts on version V2 to create version V3 containing $\{p, q, r, s, t, u, v, w\}$. The operations {**W**[**l**], **R**[**c**]} are recorded in the vertices corresponding to the cap.

| **Cap** | **ADT** | **Semantics** | **Access** |
|---|---|---|---|
| $\emptyset$ | All | Structural linearizability | Uncontrolled |
| {**c**} | All | Semantic linearizability | Serialised |
| {**c**} | All | Partial persistence* | Tardy reads |
| {**l**,**c**,**r**} | Deque | Serialisable | Simultaneous |
| {**l**,**c**} | Priority queue | Serialisable | Simultaneous |
| {**l**,**c**,**r**} | Map | Fine grain serialisable | Simultaneous |
| {**ll**,**lr**,**l**,**c**, **rl**,**r**,**rr**} | Map | Coarse grain serialisable | Simultaneous |

Table 5.1: **Cap topology and granularity of concurrency.** The topology of the cap controls the granularity at which concurrency control is enforced. The variables represented by the cap are listed in the first column. The third column describes the semantics of the ADT. The permitted access is listed in the fourth column.
(*) Partial persistence is ensured by serialising mutating functions only.

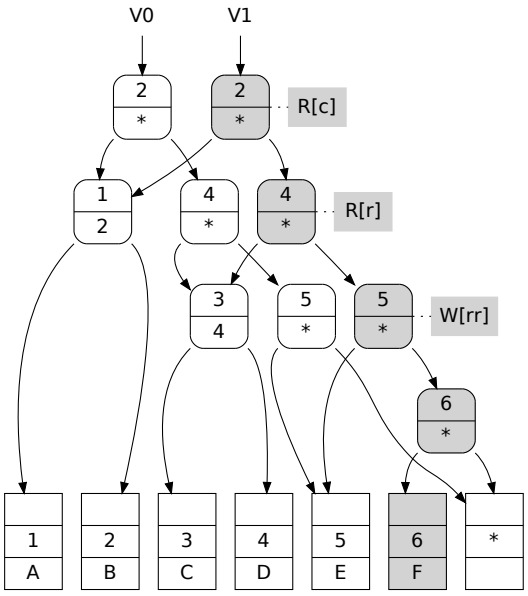
recorded in the cap of a deque. Nodes in the Immutable Data Structure record information about the operation that created them. Read and write operations on the right node, **r**, can be labelled **R**[**r**] and **W**[**r**] respectively.

Figure 5.3 illustrates the abstract read and write operations on variables recorded in the cap of a map.

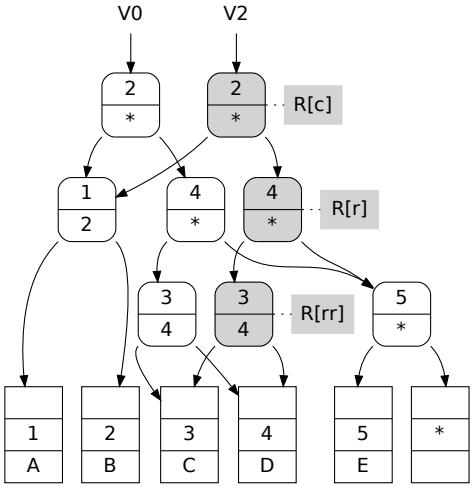
The cap can enforce serialisability at any level of granularity including making all functions accessing the data structure linearizable. A vector can be made linearizable by serialisable access to a single variable **c**.

Table 5.1 describes how the cap determines the granularity at which concurrency control is enforced.

The Canonical Binary Tree hides structural information from the application making the topology of the tree independent of the functions acting on it. The topology of a Canonical Binary Tree is not invariant because the tree may be balanced at any time. During balancing the topology of the tree is modified causing new nodes to be created. These new nodes must maintain information about the abstract read and write operations on the variables they represent. After a balancing rotation, information about abstract read and write operations is in the same positions relative to the root. The skew and split balancing rotations cause the annotation of a node to change resulting in an abstract write operation to the corresponding variable.




(a)




(b)

Figure 5.3: **Operations on variables in the cap of a map.** The cap contains the variables **ll**,**lr**,**l**,**c**,**rl**,**r** and **rr**.
(a) The path created by the function $Insert(6 \mapsto F)$ which creates version V1 is shaded. The operations $\{\mathbf{W}[\mathbf{rr}], \mathbf{R}[\mathbf{r}], \mathbf{R}[\mathbf{c}]\}$ are recorded in the vertices corresponding to the cap. The annotation of the variable **rr** does not change, but it is recorded as a write because there is a change in the subtree that it suspends.
(b) The $Query(4)$ operation creates a new version V2 of the data structure to record the operations $\{\mathbf{R}[\mathbf{rl}], \mathbf{R}[\mathbf{r}], \mathbf{R}[\mathbf{c}]\}$ in the vertices corresponding to the cap.

### 5.2.8 Validate function

The concurrency control protocol is enforced by the validate function that takes as its arguments two versions of the Immutable Data Structure. It considers the operations on variables in the caps of both versions. Operations conflict if they act on the same variables and one of them is a write. Conflicting operations may or may not conform to the concurrency control protocol. The validate function determines whether the versions contain conflicting operations that violate the protocol.

Figure 5.4 illustrates conflicting and non-conflicting operations on a deque.

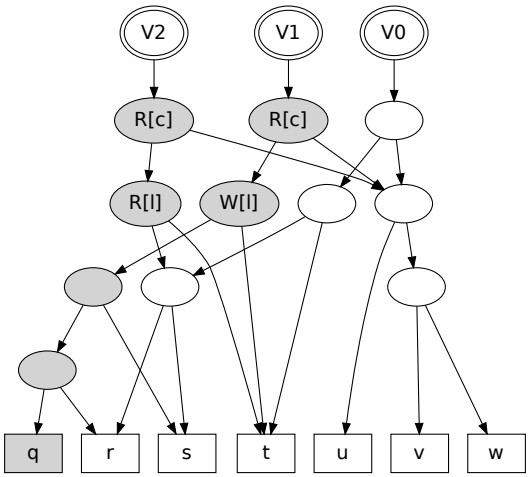
A value representing the topology of the cap is passed as a parameter to the validate function. The validate function traverses the nodes in the cap of both versions and compares the operations acting on nodes corresponding to the same variable. When conflicting operations are detected the time stamp meta-data is considered. The function returns a binary value which indicates whether or not the two versions contain conflicting operations that are not permitted by the protocol.

### 5.2.9 Meta-data

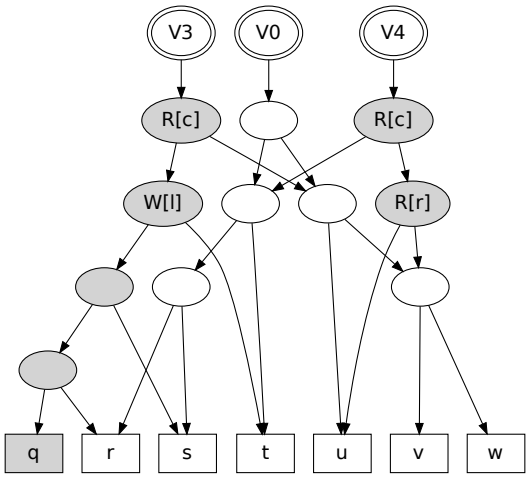
The two versions of the Immutable Data Structure considered by the validate function do not necessarily represent the application of single functions to a common ancestor version. If they did then conflict resolution would only be a matter of detecting conflicting abstract read and write operations on the same variable. Indeed, the paths considered by the validate function are of arbitrary complexity as they may represent the action of multiple functions applied to a common ancestor version. To resolve conflicts the Time Stamp Ordering concurrency control protocol is applied to the time stamp meta-data recorded in the nodes.

The Time Stamp Ordering protocol works by comparing the read and write time stamps of conflicting operations to determine whether the operations appear to occur in the order given by the time stamp of the functions. Details of the protocol and a proof that the operations that it permits are always equivalent to a serial execution can be found in Bernstein [BHG87].

The Time Stamp Ordering concurrency control protocol requires that a serialisable system maintains a unique monotonically increasing time stamp source and that a time stamp should be associated with all operations on variables. For




(a)




(b)

Figure 5.4: **Conflicting and non-conflicting operations on a deque.** The cap contains three variables **l**, **r** and **c**.
(a) Conflicting operations. The path created by the function *Push_front*() records the operations {**W**[**l**], **R**[**c**]} in the nodes corresponding to the cap. The function creates a new version V1 by path copying from version V0. The path created by the function *Front*() records the operations {**R**[**l**], **R**[**c**]}. The function creates a new version V2 by path copying from version V0. The functions conflict because they both act on variable **l** and one of them is a write.
(b) Non-conflicting operations. The path created by the function *Push_front*() records the operations {**W**[**l**], **R**[**c**]} in the nodes corresponding to the cap. The function creates a new version V3 by path copying from version V0. The path created by the function *Back*() records the operations {**R**[**r**], **R**[**c**]}. The function creates a new version V4 by path copying from version V0.

the purposes of concurrency control the serialisable system can be considered as a single Immutable Data Structure so a time stamp source is maintained independently by each data structure. A time stamp source can be implemented by an ordinal number using an atomic increment instruction.

A unique time stamp is associated with each function call. Each node retains the time stamp associated with the function that wrote the variable to which it corresponds. Each node also retains the highest time stamp of any function that reads the variable to which it corresponds.

The time stamps must be maintained in the correct positions relative to the root and this requirement dictates the implementation of the functions of the Canonical Binary Tree. Without this requirement the implementation of path copy is somewhat arbitrary. For example, an element can be inserted into a tree by creating a new root node whose children are the past version of the tree and a leaf containing the element. The *insert*() operation cannot be implemented in this way because it will alter the relative position of existing nodes. Instead, the path to an existing leaf must be copied when an element is inserted into the Canonical Binary Tree. The time stamps associated with each node on the path are copied to the new node corresponding to the variable it represents. For example, the second element in a Canonical Binary Tree must be inserted by a leaf to root path copy operation which creates two new nodes to maintain the relative position of time stamps.

Maintaining time stamps in the correct relative positions during balancing is straightforward. The relative position of a node in the subtree suspended by a pivotal node is altered by a balancing rotation. Both the skew and split balancing rotations can be regarded as writing to the variable corresponding to the pivotal node. It is not necessary to consider the time stamps of a node in the subtree suspended by the pivot because this write operation will conflict with any operation affecting a variable in this subtree.

## 5.3 Confluence

Functions acting simultaneously on an Immutable Data Structure each create different versions of the structure. These versions may be combined, provided they are not the result of conflicting functions, to create a new version which is equivalent to a serial execution of the functions. A function that combines past versions of a data structure is called a meld function. The existence of a meld function endows an Immutable Data Structure with the property of confluent persistence.

The validate function can ensure that two functions acting on an Immutable Data Structure do not contain conflicting operations and that combining the two versions produced in isolation will result in a single version which is equivalent to a serial execution of the functions. The problem is how to combine these versions into a single version?

The main contribution of this section is the description of a meld function that combines versions of an Immutable Data Structure produced in isolation. This section focuses on techniques for making Immutable Data Structures confluently persistent.

### 5.3.1 Simultaneous modifications

To make the functions of our Immutable Data Structure confluently persistent we need a meld function that can combine two non-conflicting versions of the Immutable Data Structure.

For example, consider two functions acting simultaneously on a deque. One function inserts an element onto the back of the queue and a second function, executing on another processor, simultaneously removes an element from the front of the queue. The functions do not conflict but they do result in two different versions of the queue which must be melded to produce a new version of the queue which is equivalent to a version produced by a serial execution of the functions.

### 5.3.2 Meld Function

The meld function takes two versions of the Canonical Binary Tree and creates a new version by full copying the nodes corresponding to variables in the cap. When used in conjunction with a validate function that rejects conflicting operations

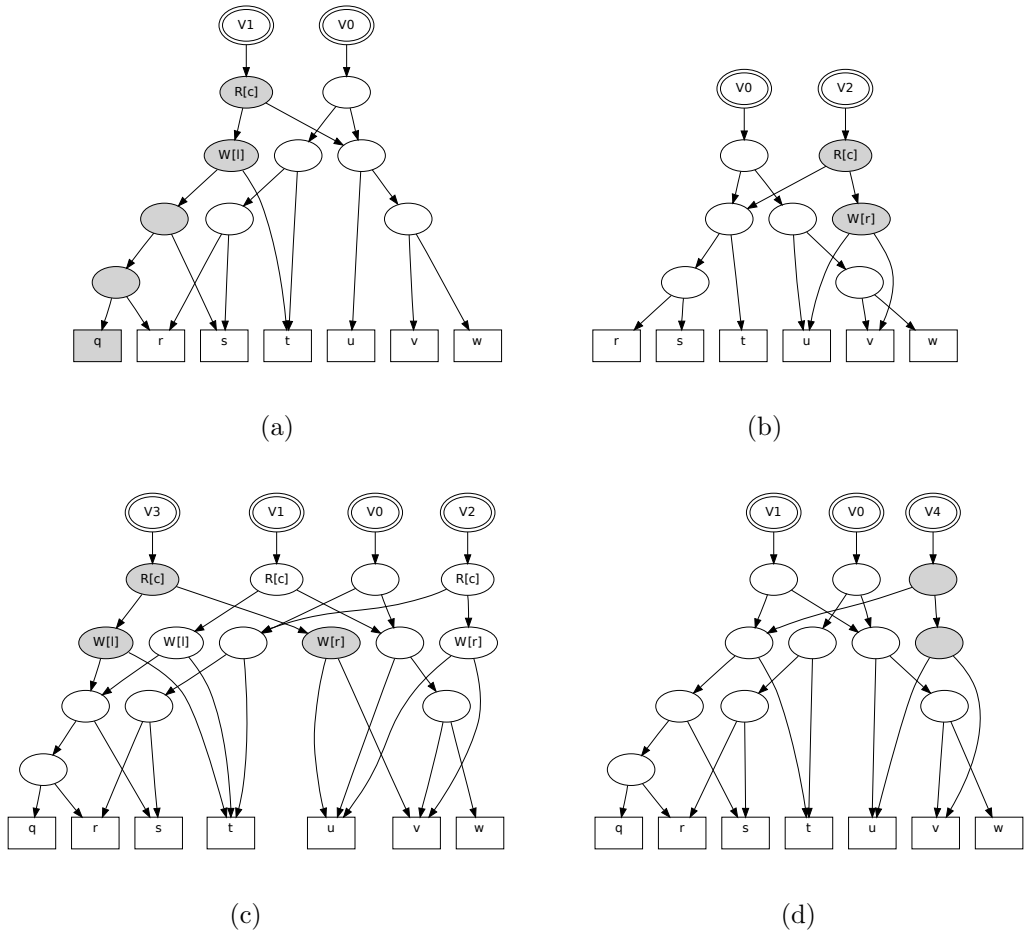



Figure 5.5: **Making a deque confluently persistent** by using a meld function that combines versions created in isolation. The cap of a deque contains three variables **l**, **r** and **c**.
(a) The function $Push\_front(q)$ acts on version V0 which contains values $\{r, s, t, u, v, w\}$. It creates version V1 which contains the values $\{q, r, s, t, u, v, w\}$. The operations $\{\mathbf{R}[\mathbf{c}], \mathbf{W}[\mathbf{l}]\}$ are recorded in the path.
(b) The function $Pop\_back()$ acts on version V0 to create version V2 which contains $\{r, s, t, u, v\}$. The operations $\{\mathbf{R}[\mathbf{c}], \mathbf{W}[\mathbf{r}]\}$ are recorded in the path.
(c) Versions V1 and V2 meld to produce a new version V3 which contains $\{q, r, s, t, u, v\}$. The meld selectively copies the nodes from versions V1 and V2.
(d) A serial execution of these functions would have created version V4 which contains $\{q, r, s, t, u, v\}$ and is equivalent to V3.

the meld function makes an Immutable Data Structure confluently persistent.

The meld function takes references to versions of arbitrary complexity as its parameters and returns a reference to a new version. It is specialised by a parameter representing the topology of the cap. The meld function traverses the nodes in the cap of both versions and compares the operations acting on nodes corresponding to the same variable. The meld function is a full copy operation that selectively copies nodes from the two versions using the operations and time stamps recorded in the nodes to determine which version to copy. For each variable in the cap a new node is created to represent it. The function returns a reference to a root node which represents a new version.

Figure 5.5 illustrates an example of how a deque can be made confluently persistent by a meld function that combines two versions simultaneously created in isolation.

The two versions in this example could have been combined by creating a new root node. However, in the general case it is necessary to copy all of the nodes which correspond to variables in the cap to ensure that the correct time stamps are recorded in the nodes and that the relationship between operations and variables is maintained.

The two versions in this example could have been combined without using time stamps because each version represents the action of a single function. However, in the general case it is necessary to consider the time stamps associated with each node while performing the full copy.

The deque is a particularly simple example, however all ADTs implemented by the Canonical Binary Tree can be made confluently persistent using the technique. The meld function is implemented by the Canonical Binary Tree, the topology of the cap is supplied as a parameter but the operation of the meld function is ADT agnostic.

### 5.3.3 Previous work

Driscoll, Sarnak, Sleator, and Tarjan describe confluently persistent data structures in detail [DSST86].

Versions of an Immutable Data Structure created by arbitrary transformations cannot always be combined because the functions that created them may conflict. Fiat and Kaplan consider that the problem of making a data structure confluently persistent is intractable in the general case [FK03]. When conflicting functions

are eliminated the problem of implementing a meld function becomes tractable, but even functions that do not conflict can transform the topology of a data structure in ways that are difficult to reconcile.

Version control systems for documents are well known applications of confluent persistence. Pilato, Collins-Sussman and Fitzpatrick describe a system called Subversion which enables multiple authors to modify a document concurrently [PCSF08]. Subversion provides a validate function that highlights any conflicting modifications to a document and a meld operation which combines multiple versions

# Chapter 6

# Contention Management

Concurrent programs that make strong progress guarantees are scalable but those that require centralised transaction management to ensure progress are not. Transactional Memory systems centralise the responsibility for scheduling and contention management, at the expense of scalability. This chapter explains why scalable concurrent programs should make strong progress guarantees. It also explains how a load-balancing scheduler, intended for use with a parallel workload, can be used to schedule Memory Transactions.

Section 6.1 identifies the choice of contention management mechanism as one of the most significant decisions taken when designing a concurrent system.

Section 6.2 describes how Immutable Data Structures can be used to implement non-blocking algorithms.

Section 6.3 compares a non-blocking Producer Consumer Queue with its blocking counterpart.

Section 6.4 describes how Memory Transactions can be load-balanced by a scheduler intended for a parallel workload.

## 6.1 Progress and Contention Management

Concurrent applications suffer from the progress pathologies of blocking, livelock and priority inversion. Progress pathologies can be alleviated by a contention manager. However, centralised contention management restricts the scalability of a concurrent system. This thesis proposes that a concurrent application should make strong progress guarantees to alleviate the need for contention management.

A concurrent application that guarantees that all of its constituent tasks complete in a finite period of time offers a progress guarantee, whereas an application that does not can suffer from a progress pathology. A useful concurrent application should make a strong guarantee of progress but it is difficult to write concurrent applications that guarantee progress.

The main contribution of this section is the observation that concurrent applications that make strong progress guarantees alleviate the need for contention management. This section focuses on guaranteeing that functions acting on an Immutable Data Structure eventually complete.

### 6.1.1 Blocking

A program that executes on a Transactional Memory system either blocks or guarantees obstruction-free progress. Obstruction-freedom is the guarantee that if a transaction is repeatedly re-tried and eventually encounters no interference from other transactions, it will complete. Obstruction-freedom does not guarantee that all of the transactions that constitute a concurrent program eventually complete. Programs that execute on a Transactional Memory system offer weak progress guarantees and are therefore prone to progress pathologies.

Section 6.1.3 discusses progress guarantees and progress pathologies.

Transactional Memory systems can implement a contention manager to alleviate progress pathologies and ensure the progress of the transactions executing in the concurrent system. The contention manager has an overview of the concurrent processing and can intervene to ensure that the application eventually completes. However, contention management is a necessarily centralised task so it restricts the scalability of the concurrent system.

### 6.1.2 Guaranteed progress

An application should guarantee lock-free progress to alleviate the need for a concurrent system to implement contention management. Centralised contention management is a fundamental barrier to the scalability of a concurrent system, whereas the difficulty of creating applications that guarantee progress is a problem that can be overcome. This thesis focuses on making it easier to write concurrent applications that guarantee progress.

Rajwar describes how concurrent applications based on the Time Stamp Ordering concurrency control protocol can be made lock-free [Raj02]. A lock-free concurrent application guarantees system-wide progress but permits individual operations to postpone indefinitely. A lock-free application is prone to the pathology of livelock and priority inversion. However, it will be argued that the use of Time Stamp Ordering and similarly sized transactions eliminates these pathologies in practice. So, a concurrent application that uses these techniques and guarantees lock-free progress does not require contention management.

Livelock can occur in lock-free applications. In practice, continual livelock is very unlikely to occur in a system that implements the Time Stamp Ordering concurrency control protocol. The unique monotonically increasing time stamp assigned to each transaction acts as a priority causing a single transaction to succeed eventually in any conflict between transactions.

Priority inversion can occur in lock-free applications. In practice, priority inversion is very unlikely to occur in a system in which all transactions execute for similar durations. Long running transactions do not occur when transactions are implemented at the granularity of accesses to a data structure.

Bernstein, Hadzilacos and Goodman explain in detail why database systems that implement Time Stamp Ordering do not require contention management [BHG87].

### 6.1.3 The Dining Philosophers

The dining philosophers problem can be used as an illustration of progress guarantees and progress pathologies. Five philosophers are sitting round a table dining on bowls of rice. Five chopsticks are placed between the bowls. Each philosopher sits in front of a bowl and can only reach the chopstick to his immediate left or right. A philosopher must have a pair of chopsticks in order to eat. The action

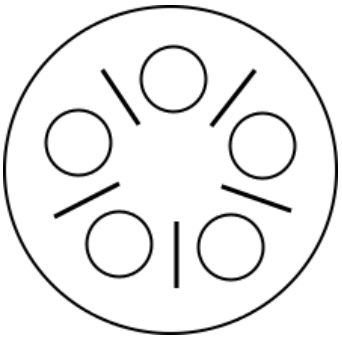


Figure 6.1: **The dining philosophers** each have a rice bowl but there are insufficient chopsticks for them all to eat at once.

of the philosophers is determined by a dining algorithm.

Figure 6.1 illustrates the arrangement of bowls and chopsticks.

Hoare reformulated a five computer synchronisation problem, originally posed by Dijkstra, as the dining philosophers problem [Hoa83]. Krishnaprasad describes how a number of synchronisation strategies can be expressed in terms of dining algorithms [Kri03].

The following discussion considers progress guarantees made by dining algorithms and the progress pathologies they are prone to.

Deadlock is a circular wait condition that occurs when each of the philosophers reaches for a second chopstick but finds that their neighbour has already taken it. The philosophers involved in the deadlock will starve because eating requires a pair of chopsticks.

Mutual exclusion is a convention that relies on blocking the progress of concurrent processes to prevent simultaneous execution. Deadlock can be prevented in a system in which mutual exclusion is enforced by serialising access to a single entity. To prevent deadlock a dining algorithm introduces a single napkin with the rule that only the philosopher in possession of the napkin can eat. The lack of a napkin blocks the other philosophers from eating. The napkin is placed on the table and all the philosophers try to possess it but only one is successful. When the successful philosopher has finished eating he places the napkin back on the table. A single philosopher can dominate the napkin causing the others to starve.

A non-blocking algorithm ensures that operations competing for a shared resource never have their progress indefinitely postponed by mutual exclusion. A non-blocking algorithm guarantees that a philosopher will not starve as a result

of mutual exclusion.

A non-blocking algorithm is obstruction-free if it guarantees that when an action is tried repeatedly and eventually encounters no interference from other actions it will complete successfully but it does not guarantee that such a situation will occur. An obstruction-free dining algorithm guarantees that a philosopher will be able to eat when the other philosophers are not attempting to eat.

Obstruction-free algorithms can suffer from the pathology of livelock. Livelock occurs when two or more competing operations cause each other to restart, preventing any of them making progress. A dining algorithm can livelock when each of the philosophers reaches for both chopsticks simultaneously but withdraws when he observe his neighbour behaving likewise. All of the philosophers involved in the livelock will starve.

Obstruction-free algorithms can also suffer from the pathology of priority inversion. Priority inversion occurs when a long running operation is preempted by an operation of brief duration. A dining algorithm in which a philosopher procrastinates when he has an opportunity to eat can suffer from priority inversion. The procrastinating philosopher might starve because he is continually interrupted by requests from other philosophers which prevent him from eating.

A non-blocking algorithm is lock-free if it guarantees that at least one action eventually completes. A dining algorithm is lock-free if it guarantees that at least one of the philosophers eventually eats. Lock-free algorithms can suffer from livelock and priority inversion but these pathologies do not prevent all operations from making progress. In practice, livelock and priority inversion are less likely to occur in an algorithm that guarantees lock-freedom than one that only guarantees obstruction-freedom.

A non-blocking algorithm is wait-free if it guarantees that eventually every action completes. A wait-free dining algorithm guarantees that all of the philosophers eventually get to eat. Wait-free algorithms are not prone to the pathologies of livelock and priority inversion. All concurrent algorithms can be converted into implementations that are wait-free but the overheads of the conversion are prohibitive [Her88] [FHS04].

At a philosophy conference philosophers have a choice of tables at which different dining algorithms are used. The wait-free table is the best because all of the philosophers are guaranteed to eat eventually and the blocking table is the worst because all the philosophers may starve because of deadlock. A lock-free

table is preferable to an obstruction-free table because at the lock-free table at least one philosopher does not starve. Wait-freedom is the strongest progress guarantee and the other guarantees are progressively weaker [HS08].

### 6.1.4 Previous work

Applications executing on Transactional Memory systems suffer from the progress pathologies of livelock and deadlock. In weakly isolated systems these pathologies can occur in combination with the isolation pathology of cascading aborts. Bobba et al. categorise Transactional Memory pathologies and describe them in detail [BMV+07].

Many Software Transactional Memory implementations block at some point in their execution. Blocking Software Transactional Memory systems are easier to design than non-blocking systems. A blocking Software Transactional Memory hides blocking from the application programmer by implementing it internally [DDS06] [HPST06] [SATH+06].

Several Software Transactional Memory systems guarantee obstruction-freedom [HLMS03] [SCKP07] [CRS05] [TMG+09]. These systems are based on the concept of exclusive but revocable object ownership.

In an obstruction-free Software Transactional Memory system each transaction is associated with a descriptor which indicates whether the transaction is active, committed or aborted. Objects are owned by transactions and have an associated pointer to their owner which is modified by an atomic instruction. When a transaction reads an object it checks the pointer to determine whether another transaction already owns it.

A transaction takes ownership of an object by modifying the pointer so that it references the transaction's descriptor. Once a transaction has taken ownership of all the objects it will access it can commit its changes to those objects. This is done by changing the transaction descriptor from live to committed. This action will atomically commit the changes to all affected objects.

An obstruction-free Software Transactional Memory system can suffer from progress pathologies. Concurrent transactions can prevent each other from owning all of the objects they require, causing livelock. Short running transactions can prevent long running transactions from obtaining all the objects they require, causing priority inversion.

Exclusive object ownership is a two-phase locking protocol which requires that

all the locks that will ever be required by a transaction are acquired before any are released. Bernstein, Hadzilacos and Goodman describe the two-phase locking concurrency control protocol in detail [BHG87]. Guerraoui and Kapałka explain that exclusive object ownership cannot provide the stronger guarantee of lock-freedom because systems based on two-phase locking cannot guarantee that at least one transaction will ever complete its operation, while other transactions are active [GK08].

Ennals argues that obstruction-free Software Transactional Memory systems are less scalable than their blocking counterparts [Enn06]. However, we believe that the best approach to overcoming scalability restrictions is to strengthen the progress guarantee, because a concurrent application that does not guarantee that all of its tasks eventually complete can hardly be described as scalable.

## 6.2 Non-blocking Algorithms

The construction of non-blocking algorithms is a challenging programming task. Non-blocking algorithms are scalable because they permit simultaneous access to a data structure and they offer strong progress guarantees without requiring centralised contention management. This section describes a simple technique for constructing non-blocking algorithms. Non-blocking functions acting on Immutable Data Structures are the foundation on which scalable concurrent programs can be built.

Non-blocking algorithms are scalable and relieve the programmer from having to reason about locks. A concurrent system in which semantically linearizable functions act concurrently on an Immutable Data Structure can guarantee lock-free progress. In order to simplify the construction of non-blocking algorithms concurrent systems should also ensure that serialisable functions guarantee lock-free progress.

The main contribution of this section is a general technique for implementing non-blocking algorithms. This section focuses on allowing simultaneous access to Immutable Data Structures.

### 6.2.1 Ensuring serialisability without blocking

The enforcement of serialisability is more involved than the enforcement of linearizability because conflict detection is performed by a function instead of an atomic hardware instruction.

Semantic linearizability is enforced by recording the value of the root of the Immutable Data Structure at the start of the execution of an access function and checking that its value has not changed before atomically replacing the root at the end of the execution. This replacement relies on an atomic compare-and-swap instruction which serialises access to the root and implements a memory barrier that ensures that the speculative path is atomically transformed into a new shared version of the data structure.

Semantic linearizability is lock-free because it guarantees that when the program runs for sufficiently long at least one function makes progress. An access function can be prevented from completing only if the value of the root changes whilst it is executing. However, if the value of the root changed then another function successfully completed so at least one function made progress.

Semantic linearizability does not achieve the stronger condition of wait-freedom. An algorithm is wait-free if every operation eventually completes. An access function can be prevented from completing if the value of the root changed whilst it was executing. The function can be re-tried indefinitely but there is no guarantee that it will eventually succeed.

The serialisability of simultaneous accesses to an Immutable Data Structure can be enforced by a validate function that implements a concurrency control protocol and a meld function that can combine two versions of the data structure making it confluently persistent. The problem is to combine these actions a way that guarantees progress.

### 6.2.2 Lock-free serialisability

An access function of an Immutable Data Structure includes both validate and meld functions which implement a simple distributed transaction manager. This transaction manager combines two forms of speculation. The first speculation is that the access function does not conflict with any functions accessing the data structure simultaneously. The second speculation is that the root of the Immutable Data Structure does not change while the validate and meld functions take place.

Figure 6.2 illustrates the execution of an access function in the presence of concurrent mutations.

An Immutable Data Structure access function records the value of the root of the Immutable Data Structure at the moment in time that it starts. This reference represents the starting version of the data structure. The function is applied to this starting version to produce a path in isolation. The function records the value of the root of the Immutable Data Structure at the moment in time that it completes the path, we call this version the first snapshot. This reference represents a version of the data structure which might have been arbitrarily mutated by concurrently executing functions. The validation function ensures that the execution of the function and the first mutation do not contain conflicting operations. The meld function combines the path with the first snapshot version to produce a putative version of the data structure. Both the validate and meld functions execute in isolation, their only inputs are the path and the first snapshot version. An atomic compare-and-swap instruction replaces the root with the putative version if and only if the root has the same value as the first snapshot.

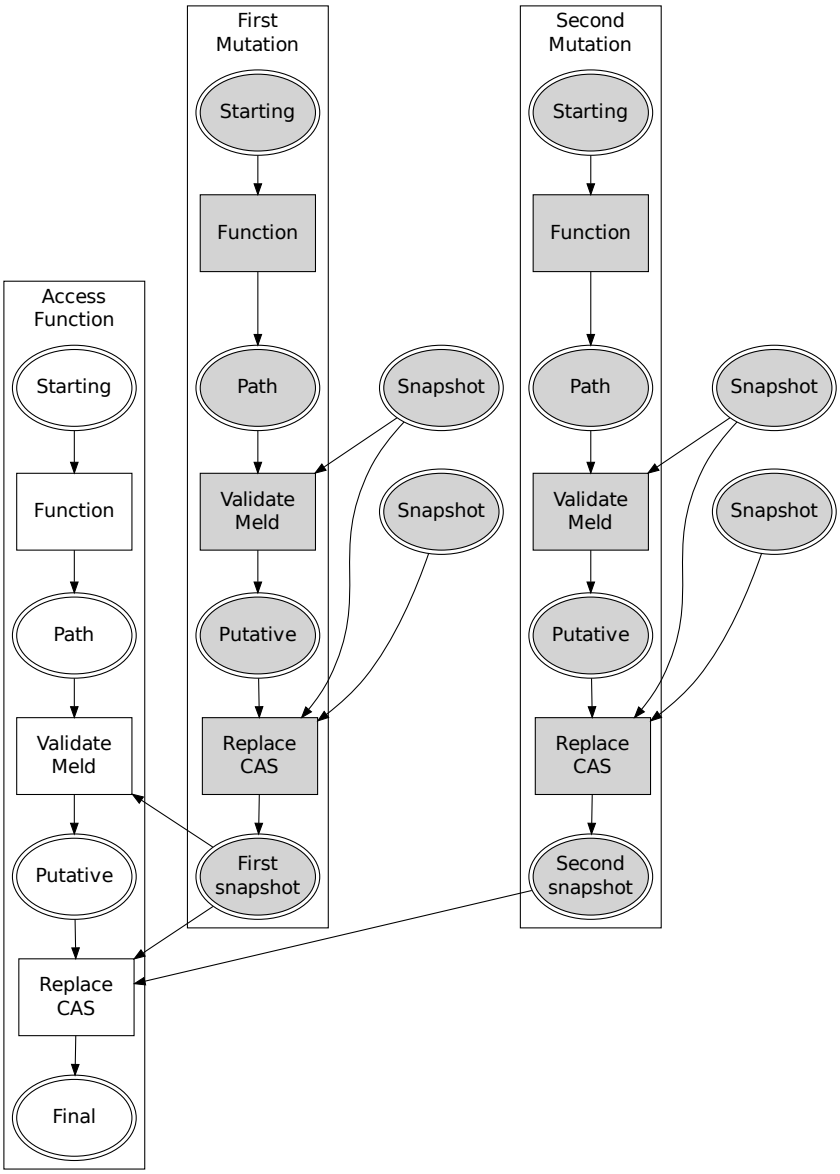



Figure 6.2: **The execution of an access function in the presence of concurrent mutations.** Each operation takes a version of the data structure, represented by an ellipse, as its argument and produces a new version. The operations executed by another processor are shaded.
The first mutation may successfully complete, while the access function is executing, creating the first snapshot version. If so, the path created by the access function is validated against this snapshot version and melded with it to create a putative path. A second mutation may complete, while this validation is taking place, creating the second snapshot version.

An Immutable Data Structure access function completes successfully if and only if both speculations are successful. The first speculation is that the access function does not conflict with the first mutation. The second speculation is that the second mutation does not complete successfully in the period between the first and second snapshot. In the first case a conflict is detected after the first mutation successfully modified the root which implies that the first mutation made progress. In the second case the value of the root only changes after the second mutation successfully modified it which implies that the second mutation made progress. Either the function or one of the mutations makes progress in each case. The execution is lock-free because it guarantees that when a program runs for sufficiently long at least one processor makes progress.

### 6.2.3 Previous work

Herlihy and Shavit describe the state of research into non-blocking algorithms in a book entitled "The art of multiprocessor programming" [HS08].

Non-blocking algorithms which access mutable values are complex, difficult to reason about and are usually regarded as the domain of expert programmers. Given an ADT there is no general technique for constructing a non-blocking algorithm that conforms to it.

A particularly difficult problem, the ABA problem, contributes significant complexity to the implementation of non-blocking algorithms. Fraser and Harris describe the ABA problem which is a pathology of the atomic compare-and-swap instruction that occurs when addresses are re-used [FH07]. The implementation of a non-blocking algorithm is simplified by ensuring that all the data it acts upon is immutable and that addresses are not re-used. The immutability of the vertices of an Immutable Data Structure ensures that the root cannot be assigned the same value more than once so the ABA problem cannot occur.

An atomic compare-and-swap instruction cannot modify two non-contiguous locations so non-blocking data structures with cycles, such as doubly linked lists, are difficult to construct. The ADTs presented by non-blocking algorithms are often similar to those presented by purely functional data structures which are also single pointer structures.

Goetz et al. examine the performance of the non-blocking algorithms included in `java.util.concurrent` library [GBB+06].

Allemany and Felten found that many non-blocking algorithms perform badly

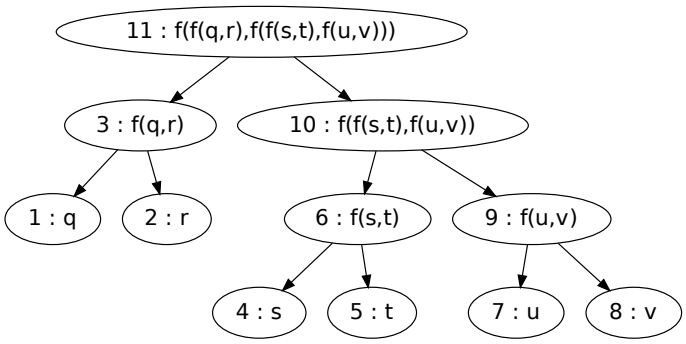



Figure 6.3: **The abstract syntax tree of an expression** in which each value is associated with a tag number. The expression is $f(f(q,r), f(f(s,t), f(u,v)))$.

on Chip Multi-Processors [AF92]. Allemany attributes this poor performance to resources wasted by operations that fail and the cost of data copying.

Non-blocking algorithms use the atomic compare-and-swap instruction which may take thousands of clock cycles to complete. Hennessy and Patterson provide an introduction to the complex performance issues surrounding the use of the atomic compare-and-swap instruction [HP06b].

### 6.2.4 Non-blocking evaluation

Non-blocking algorithms based on immutable data are more flexible than those based on mutable data. The following example illustrates how a non-blocking algorithm can be used to load-balance the evaluation of an arbitrary expression on multiple processors.

The expression $f(f(q,r), f(f(s,t), f(u,v)))$ can be described by an abstract syntax tree. The problem is to load balance the evaluation of the expression between processors. It is difficult to schedule the concurrent execution of this expression because the execution time of each function is not known. A solution is to schedule the execution of functions as their arguments become available dynamically.

Figure 6.3 illustrates the abstract syntax tree of the expression.

The evaluation may be dynamically load-balanced by recording intermediate values in an Immutable Data Structure. Initially, the data structure contains only the arguments of the expression. The final version contains all of the arguments

and intermediate values as well as the result. The data structure maintains an immutable record of the evaluation of the expression.

Figure 6.4 illustrates the initial and final versions of an Immutable Data Structure which represents the evaluation of the expression.

Figure 6.5 illustrates the non-blocking evaluation of the expression by multiple processors.

In the illustration, each function's arguments are available in the version of the Immutable Data Structure that it starts with. If the function is unable to find its arguments in the Immutable Data Structure then it is re-started with a new version. Eventually, all of the functions in the expression complete and the value of the expression can be obtained from the Immutable Data Structure.

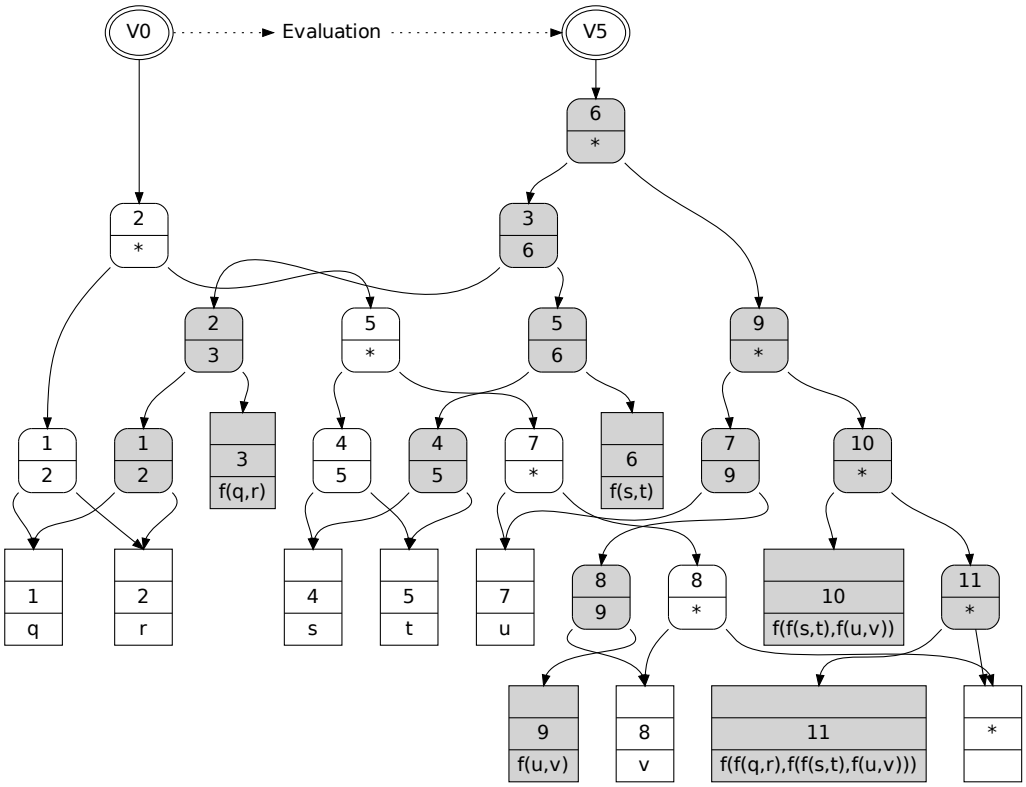



Figure 6.4: **An Immutable Data Structure representing the evaluation of an expression.** The Immutable Data Structure representing the evaluation of the expression $f(f(q,r), f(f(s,t), f(u,v)))$ is an interval tree, with a sentinel, which maps the tag number of a value in the abstract syntax tree to a leaf. The Immutable Data Structure contains all of the arguments and intermediate values as well as the result. Only the initial and final versions of the Immutable Data Structure are shown. The final version is shaded.

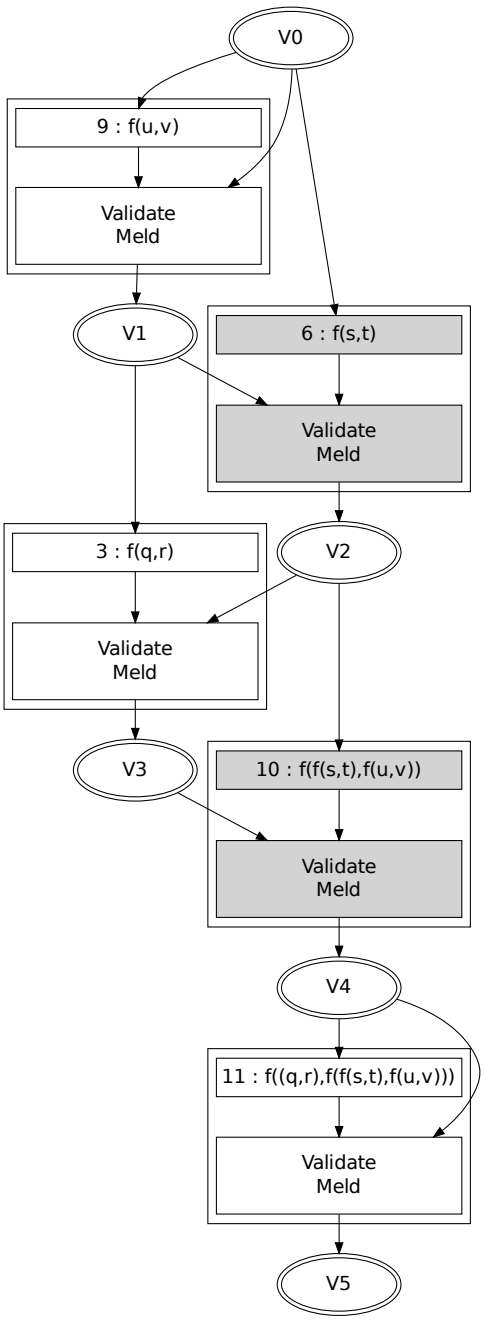



Figure 6.5: **The non-blocking evaluation** of the expression $f(f(q,r), f(f(s,t), f(u,v)))$ by multiple processors load-balances the work between them. Each operation takes a version of the data structure, represented by an ellipse, as its argument and produces a new version. The operations executed by another processor are shaded. Only validation against the first snapshot version is shown.

## 6.3 Producer Consumer Queue

To evaluate our technique for creating a non-blocking algorithm we compare the ease of use of a bounded non-blocking Producer Consumer Queue with that of a similar queue described in the literature. We also compare the performance of our queue with that of its counterpart implemented using mutual exclusion. We find that our queue is more flexible than the non-blocking queues described in the literature. We also find that our queue performs similarly to a queue implemented using mutual exclusion.

The Producer Consumer Queue is a concurrent design pattern. A bounded Producer Consumer Queue can act as a message queue for Inter-Processor Communication. Two classes of processors, the producers and the consumers, share a common buffer which acts as a queue of messages between them. A producer adds a message to the queue and a consumer removes it. The Producer Consumer Queue guarantees that a producer cannot add a message onto the queue when it is full and that a consumer cannot remove a message when it is empty and that each message is consumed exactly once.

The main contribution of this section is an evaluation of a Producer Consumer Queue implemented by an Immutable Data Structure. This section focuses on comparing: throughput, ease of implementation, ease of use, scalability and progress guarantees.

### 6.3.1 Experiment

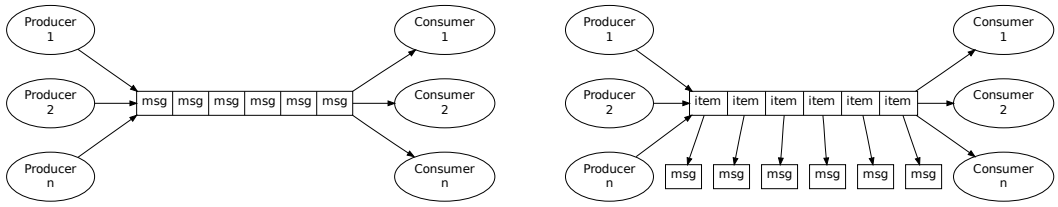
Our experiment compares the performance of a lock-free bounded Producer Consumer Queue implemented by an Immutable Data Structure with that of a blocking bounded Producer Consumer Queue implemented using mutual exclusion.

Section 6.3.3 describes the experimental set up.

We call a Producer Consumer Queue that transmits messages in a buffer a Mailbox Queue and a queue that transmits references to messages a Messaging Queue. We are primarily interested in transmitting messages between physical processors so each end of the queue is accessed by thread of execution on a different physical processor.

Figure 6.6 illustrates the Producer Consumer Queue design pattern.

The production and consumption of messages by an application affects the performance of the queue by introducing latency. Our experiment examines how




(a) Mailbox Queue (b) Messaging Queue

Figure 6.6: **The Producer Consumer Queue** design pattern.
(a) A Mailbox Queue acts as a buffer for fixed sized messages sent between producers and consumers.
(b) A Messaging Queue transmits messages referenced by items in the buffer between producers and consumers.

the throughput of the Messaging Queue varies depending upon this latency.

A Producer Consumer Queue based on a deque implemented by the Canonical Binary Tree is unbounded and it does not return any memory. Mechanisms are required to bound the queue and reclaim memory.

Section 6.3.4 describes the implementation of the Producer Consumer Queue.

A message workload is simulated so that the performance of the queues can be evaluated.

Section 6.3.5 describes a simulated message workload.

We examine whether our lock-free Producer Consumer Queue is easier to implement than a similar non-blocking queue. We also compare the ease of use, the scalability and the progress guarantees offered by our queue with those of other blocking and non-blocking queues.

## 6.3.2 Results

This thesis does not make any claims about the absolute performance of Transactional Data Structures. However, the results of our experiment show that the performance of the non-blocking Producer Consumer Queue was broadly similar to that of a queue implemented using mutual exclusion.

### Maximum throughput of the Mailbox Queue

The maximum throughput of a Mailbox Queue implemented by the non-blocking Producer Consumer Queue is compared with that of blocking queues from the

| Algorithm | Elapsed time (s) |
|---|---|
| **Non-blocking Producer Consumer Queue** | **0.24** |
| boost bounded circular buffer | 0.28 |
| boost bounded space optimised circular buffer | 0.30 |
| boost bounded std::deque container | 0.25 |
| boost bounded std::list container | 0.85 |

Table 6.1: **The maximum throughput of a Mailbox Queue.** The elapsed time taken to transmit one million mailbox messages between two processors for various queue types is listed. The experiment determines the maximum throughput of a Mailbox Queue with a capacity of one thousand 8 byte messages. Figures given are the mean of 10 observations.

Boost C++ library [Kar05].

Table 6.1 lists the elapsed time taken to transmit messages between two processors for various queue types.

Our non-blocking Producer Consumer Queue has the lowest overall execution time. The implementation based on a deque from the standard library has the lowest elapsed execution time of the blocking implementations.

Section 6.3.7 discusses the performance of the Mailbox Queue in detail.

We conclude that the maximum throughput of our mailbox Producer Consumer Queue is similar to that of its blocking counterpart.

### Maximum throughput of the Messaging Queue

The maximum throughput of a Messaging Queue implemented by the non-blocking Producer Consumer Queue is compared with that of a blocking queue from the Boost C++ library.

Figure 6.7 and figure 6.8 illustrate the elapsed time taken to transmit messages between processors while varying the latency of production and consumption.

The blocking queue has a lower elapsed time than the non-blocking queue, regardless of the latency incurred by either the producer or the consumer. The difference between the throughput of the queues becomes more pronounced as the latency increases. When there is an imbalance between the latency of the producer and that of the consumer the elapsed time taken by the non-blocking queue is significantly longer than that taken by the blocking queue.

Section 6.3.8 discusses the performance of the Messaging Queue in detail.

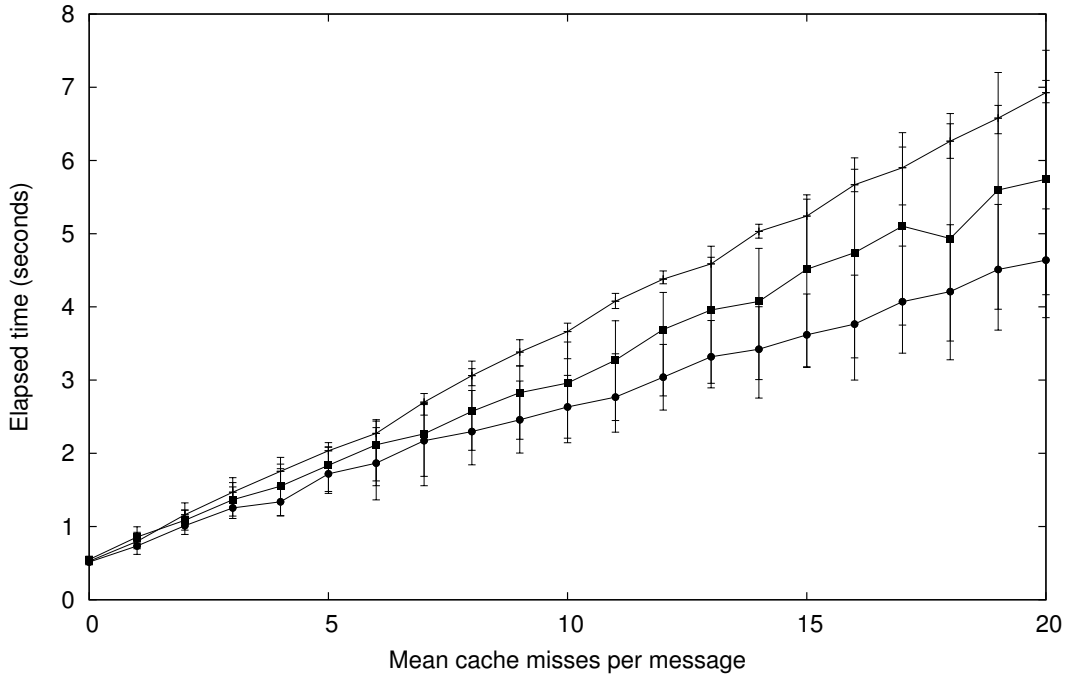



Figure 6.7: **The maximum throughput of a non-blocking bounded Messaging Queue implemented by a confluently persistent Immutable Data Structure.** The elapsed time taken to transmit one million messages between two processors is plotted against a varying number of forced cache misses incurred while: producing (■), consuming (●) and both producing and consuming the messages (+).
Figures given are the mean of 10 observations. Bars indicate the range of elapsed times observed.

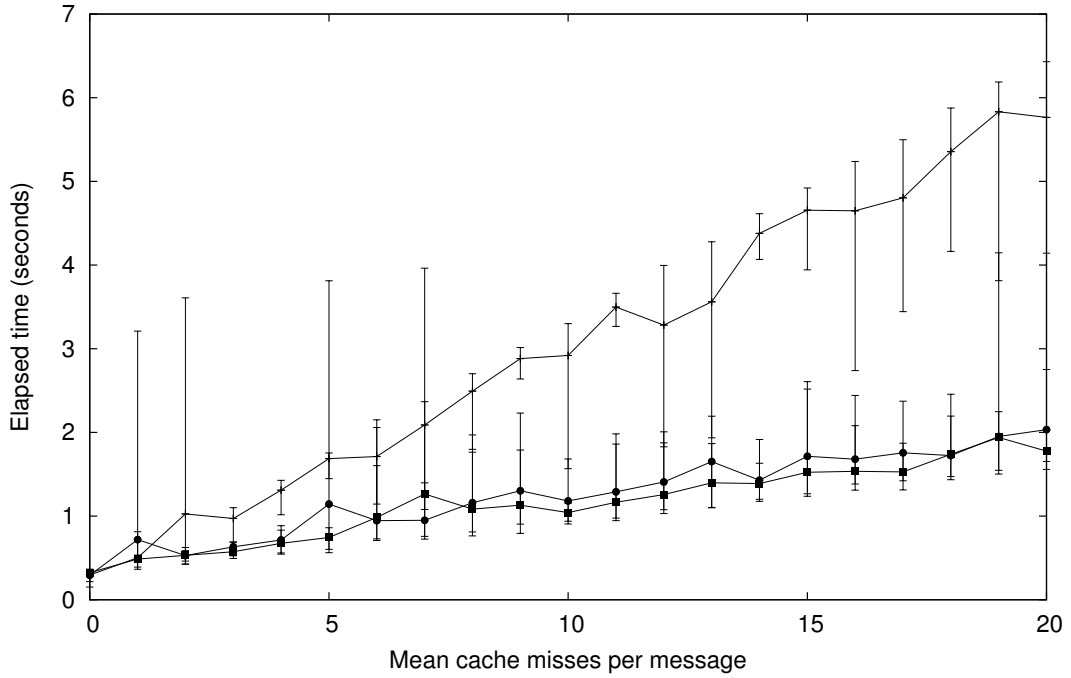



Figure 6.8: **The maximum throughput of a blocking Producer Consumer Queue from the Boost library, implemented by the std::deque container.** The elapsed time taken to transmit one million messages between two processors is plotted against a varying number of forced cache misses incurred while: producing ( ■ ), consuming ( ● ) and both producing and consuming the messages (+).
Figures given are the mean of 10 observations. Bars indicate the range of elapsed times observed.

We conclude that the maximum throughput of our messaging Producer Consumer Queue is lower than that of its blocking counterpart.

### Ease of implementation

The Canonical Binary Tree on which our Producer Consumer Queue is based is a general solution to problems in concurrency, whereas a non-blocking algorithm is typically a specialised solution to a particular problem.

Section 6.3.9 compares the ease of implementing our queue with that of other queues.

We conclude that our Producer Consumer Queue implementation is more flexible than either its blocking or non-blocking counterparts.

### Ease of use

This thesis claims that Transactional Data Structures make concurrent programs easier to write.

From the prospective of an application programmer our Producer Consumer Queue is as easy to use as either a blocking queue or the non-blocking Producer Consumer Queue developed by Scherer [SLS09].

Section 6.3.10 compares the ease of use of our queue with that of other queues.

We conclude that our Producer Consumer Queue implementation is as easy to use as either its blocking or non-blocking counterparts.

### Scalability

This thesis claims that Transactional Data Structures facilitate the development of scalable concurrent programs.

We found that the throughput of both our queue and a queue implemented using mutual exclusion were unaffected by the number of processors concurrently accessing them.

Section 6.3.11 compares the scalability of our queue with that of other queues.

We conclude that our Producer Consumer Queue implementation is as scalable as either its blocking or non-blocking counterparts.

**Progress guarantees**

This thesis claims that concurrent programs that use Transactional Data Structures can guarantee progress.

A Producer Consumer Queue implemented by mutual exclusion makes no progress guarantees, whereas our non-blocking Producer Consumer Queue guarantees lock-free progress.

Section 6.3.12 compares the progress guarantees offered by our queue with those offered by other queues.

We conclude that our Producer Consumer Queue implementation is preferable to its blocking counterpart because it offers a progress guarantee.

### 6.3.3 Method

Experiments are performed using a PC with an Intel Core i7 860 processor operating at 2.8GHz with 8 MB of cache and 4GB of DDR3 SDRAM running at 1333 MHz. The examples are compiled using the Intel 64 bit C++ compiler with the maximum optimisation level.

We use a bounded non-blocking Producer Consumer Queue based on a deque implemented by the Canonical Binary Tree. The Canonical Binary Tree is balanced but none of the optimisations, suggested in section 3.6.5 are implemented.

We compare the performance of our queue with that of the bounded blocking Producer Consumer Queue coding example from the Boost C++ library [Kar05]. The calls to the Boost Thread library are replaced by the corresponding calls to the Threading Building Blocks library and the scalable memory allocator from that library is used.

The queueing applications we compare behave differently because the access function of the blocking queue waits when the queue is empty, whereas the access function of the non-blocking queue may fail and must be re-tried. However, both applications transmit messages as fast as their queues allow.

### 6.3.4 Implementation

Data structures implemented by the Canonical Binary Tree are unbounded, so a non-blocking Producer Consumer Queue based on a deque implemented by the Canonical Binary Tree is also unbounded. To bound the queue a count of the number of items it contains can be associated with each version. To maintain this

association without blocking the count is incorporated into the data structure. An immutable management block containing the count is linked between the version root and the root node. Functions that create a new path in the Immutable Data Structure also create a new version of the immutable management block with an adjusted count.

The management block contains a count of the items in a version of the queue. When an item is pushed onto the queue a new management block with an increased count is created. Push functions fail when the count of items in the queue exceeds the bound. When items are pushed onto the queue concurrently each push function increases the same counter value but the validate operation ensures that only one succeeds. When items are pushed onto and popped from the queue concurrently the count is adjusted during the meld operation to indicate that the queue length is unaffected by the combined operations.

The data structures implemented by the Canonical Binary Tree do not return any memory. To implement a useful Producer Consumer Queue it is necessary to reclaim the memory occupied by unreachable vertices without introducing blocking. The memory reclamation process visits versions that cannot be reached by the data structure access functions. To achieve this, past versions are linked to the current version to form a version chain.

The root at the time an access function starts is the address of the most recently committed management block and this address is recorded in the new version of the management block created by the function. It links the management blocks created by functions that successfully update the root and forms a chain linking the root to all past versions of the Immutable Data Structure.

Memory is returned by a periodic reclamation process. This process follows the version chain examining past versions. It returns the memory occupied by unreachable vertices without blocking the progress of the access functions.

The management block contains additional immutable values used to control memory reclamation. These include the address of the last management block returned by the previous memory reclamation, which indicates where memory reclamation should stop, and a cyclic counter which indicates when memory reclamation should take place.

In our implementation memory reclamation relies on the observation that when the queue is empty all vertices in all past versions are unreachable. The reclamation process follows the version chain until it encounters a management

block with an item count of zero. It continues following the version chain releasing the memory occupied by past versions. This technique is sufficient for inter-processor queues which are frequently empty and is adequate for our experiments. However, there is no guarantee that the queue will ever be empty while in operation. A more robust solution to the memory management problem is required before our Producer Consumer Queue can be used in production.

### 6.3.5 Workload simulation

Inter-processor traffic is more difficult to characterise than network traffic. Standard protocols and benchmarks aid the evaluation of algorithms related to network traffic, whereas Inter-Processor Communication is generally based on bespoke protocols. Concurrent applications use a mixture of message sizes and perform varying amounts of work when preparing and processing messages.

The producer writes a message to memory and this write is cached. The atomic compare-and-swap instruction in both the blocking and non-blocking implementations forces an outstanding write operation buffered by the processor to be written to memory, so when the consumer reads the message from memory the operation is a cache miss. The elapsed time taken to transmit messages is dominated by the latency of these cache misses.

We simulate the work done during the production and consumption of messages by inducing cache misses, but it is not sufficient to assume that a fixed number of cache misses is associated with each message. Messages are created by applications which do a varying amount of work during the production and consumption of messages and this behaviour must also be simulated.

We assume that the program issues memory operations that result in a cache miss at random intervals and that the latency of these operations dominates the production and consumption of the message. The number of cache misses per message is modelled by a Poisson distribution. A Poisson distribution is a discrete probability distribution that expresses the probability of a number of independent events occurring in a fixed period of time. A cache miss can be induced by accessing an array much larger than the processor cache.

### 6.3.6 Previous work

The C programming language does not specify a memory model so a concurrent application written in C relies on the memory model implemented by the underlying hardware architecture, but memory models implemented by hardware architectures differ. Adve and Gharachorloo provide a comprehensive tutorial on shared memory consistency models [AG95]. Non-blocking structures implemented in C tend not to be portable between different hardware architectures because the memory models implemented by these architectures are different. For example, The Intel architecture software developer's manual describes how the memory models implemented by Intel IA-32 and Intel 64 bit processors differ [Int07]. It is difficult to construct a Non-blocking algorithm in C that is portable between the IA-32 and Intel 64 bit platforms.

Marginean describes a simple lock-free Producer Consumer Queue, implemented in C, in the mainstream magazine Dr Dobb's journal [Mar08]. This queue suffers from several problems including a misplaced memory barrier and false assumptions about the effect of atomic instructions on the iterators implemented by the Microsoft template library. The magazine published a revised version of the download code the following month but this too contained errors. Shutter described a working version of the queue, albeit with a restricted interface, four months after publication of the initial article [Shu08].

Non-blocking algorithms described in the literature may appear simple but getting them right is very difficult. Herlihy's book "The art of Multiprocessor Programming" unintentionally illustrates the difficulty of finding errors in non-blocking algorithms. This book has an extensive on-line errata, even though it was clearly written and reviewed by experts [HS08]. Erroneous non-blocking algorithms, such as double-checked locking, have even appeared in peer reviewed conference publications [BBB+06].

The Java programming language has a clearly defined memory model. Manson, Pugh and Adve describe the Java memory model in detail [MPA05]. The Java virtual machine for a particular hardware architecture implements memory barriers to ensure the correctness of functions in the Java libraries. Lea describes how portable concurrent programs can be constructed using the Java language [Lea06].

Scherer, Lea and Scott describe a lock-free unbounded Producer Consumer

Queue which is called a scalable synchronous queue [SLS09]. This queue outperformed the queue included in the Java SE 5.0 version of the `java.util.concurrent` library and was subsequently included in Java 6. This queue does not contain messages in the way that our Messaging Queue does. Instead, it queues instances of the producers and matches them to available consumers to allow the handover of a single message. Scherer's thesis lists the program code which implements the queue and describes its operation in detail [Sch06]. The program code required to implement this queue is much shorter than that of our Canonical Binary Tree implementation but this belies its complexity.

### 6.3.7 Mailbox Queue performance

The throughput of a Mailbox Queue implemented by mutual exclusion is dependent on the standard library data structure that implements it. The std::list container is implemented by nodes with both forward and backward pointers, whereas the std::deque is implemented in managed blocks of storage. The size of an element in a std::list is larger than that of the std::deque. A single atomic compare-and-swap instruction is performed by each operation and the amount of memory written by the synchronisation depends upon the implementation of the data structure. The memory written by the synchronisation results in coherency cache misses when it is read by the consumer. The latency of cache misses dominates the execution time so the throughput of Mailbox Queue is dependent on the size of the elements of the underlying data structure.

Each message is written to memory by the producer and then read by the consumer. We estimate that this operation takes 800 cycles to complete so with a processor speed of 2.8GHz one million operations take about $(1000000 * 800/2.8 * 10^9) = 0.29$ seconds to complete.

The throughput of both our Mailbox Queue and the blocking queue is similar because they are both bounded by the latency of a similar number of coherency cache misses per message. To verify this we increased the size of the node in our Canonical Binary Tree and found that the throughput of the mailbox queue was reduced.

We did not expect the node size to make such a large difference to the performance of Transactional Data Structures. This observation motivated the search for ways of optimising the performance of the Canonical Binary Tree by reducing both the size of the node and the number of nodes accessed. These optimisations

are described in section 3.6.5.

### 6.3.8 Messaging Queue performance

The throughput of the Messaging Queue is, like the Mailbox Queue, bounded by the latency of cache misses. However, some misses are a result of the simulated processing of the messages.

When the production and consumption of messages is balanced the throughput of the blocking and non-blocking Messaging Queues are broadly similar. However, when the production and consumption of messages is imbalanced the throughput of the blocking queue is higher than that of the non-blocking queue. When rate of production of messages is higher than the rate of consumption the instantaneous size of the queue is larger and consequently the path in the Canonical Binary Tree is longer.

The number of coherency cache misses incurred by each message processed by the non-blocking queue is dependent on the length of the path in the Canonical Binary Tree, whereas the number of coherency cache misses incurred by the blocking queue is independent of the size of the queue. Consequently the throughput of the non-blocking queue is dependent on the balance between the producer and the consumer, but the throughput of the blocking queue is not.

### 6.3.9 Ease of implementation

Both the non-blocking Producer Consumer Queue of Scherer and our Canonical Binary Tree took a similar amount of time to develop, so our non-blocking Producer Consumer Queue is no easier to implement from scratch than a comparable non-blocking queue [SLS09]. However, it is difficult to modify the ADT presented by the queue of Scherer without affecting its progress guarantee, whereas our queue can easily be tailored to the requirements of a particular application.

For example, a work stealing scheduler may be used to load-balance work among multiple consumers. A work stealing scheduler associates a unique Producer Consumer Queue with each consumer and it permits an idle consumer to remove messages from the back of a queue associated with a busy consumer to balance the work between consumers. Our Producer Consumer Queue can easily be adapted to permit equal access to both ends. It is more difficult to adapt the queue of Scherer to permit equal access to both ends.

### 6.3.10 Ease of programming

From the prospective of an application programmer our Producer Consumer Queue is as easy to use as either a blocking queue or the non-blocking Producer Consumer Queue of Scherer. However, Scherer's queue is more portable than our queue because it relies on the clearly defined Java memory model, whereas our queue is implemented in C which relies on the model implemented by the hardware architecture.

Our Producer Consumer Queue is more portable than other non-blocking queues implemented in C because it relies on a single atomic compare-and-swap instruction for synchronisation, whereas other non-blocking queues rely on separate memory barriers which are architecture dependent [Shu08].

For example, a windowing queue allows more than one message to be added or removed by a single operation. Our Producer Consumer Queue can easily be adapted to support windowing by applying concurrency control to a path created by more than one access function. However, windowing is difficult to implement using mutual exclusion and we were unable to find an open source implementation of a windowing queue to compare our implementation against.

Ease of programming is a vague concept but we found our Producer Consumer Queue to be both portable and adaptable. It is at least as easy to use as either its blocking or non-blocking counterparts.

### 6.3.11 Scalability

Non-blocking algorithms are preferable to blocking algorithms because they are potentially scalable, whereas the scalability of algorithms that use mutual exclusion is fundamentally limited by Amdahl's law. Even a non-blocking algorithm that performs poorly on a modern Chip Multi-Processor is preferable to its blocking counterpart because the non-blocking algorithm is potentially scalable, whereas a blocking algorithm has limited scalability on any future hardware.

Goetz et al. examine the scalability of the Producer Consumer Queues in the Java library [GBB+06]. Goetz found that the throughput of the queue is unaffected by the number of producers and consumers using it. We also found that the number of processors accessing a queue did not make any difference to its throughput.

### 6.3.12 Progress

Non-blocking algorithms are preferable to blocking algorithms because they offer a progress guarantee, whereas blocking algorithms do not. Even a non-blocking algorithm that performs less well than its blocking counterpart is preferable because the non-blocking algorithm guarantees progress, whereas its blocking counterpart has the potential to block indefinitely.

A lock-free queue may suffer from the progress pathology of livelock. This occurs when two processors repeatedly prevent each other from successfully accessing the queue. In practice, our queue is unlikely to suffer from this pathology because the Time Stamp Ordering concurrency control protocol ensures that one or other of the conflicting access functions takes precedence.

In practice, a Producer Consumer Queue is so simple and Chip Multi-Processors are so reliable that the lack of a progress guarantee makes little difference once the concurrent application is tested and shown to be working. However, programmers do not always get things right first time. During the development of a concurrent application a strong progress guarantee often makes the difference between an application that does not work correctly and one that requires a system restart to resolve deadlock.

## 6.4 Distribution and Scheduling

The benefits of concurrent execution come at the cost of distributing and scheduling work and detecting any conflicts. In a Transactional Memory system the scheduler is regarded as a component of the transaction manager. This section describes how the scheduling problem can be reduced to one of load-balancing concurrent execution. A two-level scheduler intended for a parallel workload can be utilised to load-balance concurrent execution.

The overhead associated with distributing parallel work on a Chip Multi-Processor is high and for many workloads the overhead of distribution exceeds the benefit of parallel execution. The overhead associated with distributing and scheduling concurrent Memory Transactions is significantly higher than that associated with distributing a similar amount of parallel work because of the additional effort required to ensure correct concurrent execution.

The main contribution of this section is observation that, once isolation and progress pathologies have been eliminated, the problem of scheduling Memory Transactions is similar to that of distributing parallel work. This section focuses on using an existing two-level scheduler to schedule Memory Transactions.

### 6.4.1 Scheduling

Transactional Memory systems implement transaction scheduling strategies that do not make a distinction among the transaction management tasks of concurrency control, contention management and load-balancing.

Transactional Memory systems may try to improve the efficiency of concurrency control by scheduling transactions to avoid conflicts and reduce the overhead of wasted work. These benefits should be balanced against the scalability restrictions of centralised concurrency control.

Transactional Memory systems that make weak progress guarantees may schedule transactions to avoid progress pathologies. The benefits of guaranteed progress should be balanced against the scalability restrictions of centralised contention management.

A Transactional Memory system should execute a workload that is known not to contain conflicting tasks without incurring the overhead of concurrency control.

The problem of scheduling parallel work on a Chip Multi-Processor is solved

by using a two-level scheduler.

Section 6.4.3 describes the scheduling of parallel work on a Chip Multi-Processor.

The overhead of scheduling work on a parallel system imposes a lower limit on the size of chunks of work that are worth scheduling and the additional overhead of concurrency control raises this limit further.

Section 6.4.5 describes how these limits influence the design of a transactional system.

An access function of an Immutable Data Structure is responsible for concurrency control, which alleviates the need for centralised concurrency control, and it also guarantees progress, which alleviates the need for centralised contention management. The only transaction management task that requires centralisation is scheduling.

A solution to the scheduling problem should isolate and simplify the scheduling component of transaction management and make it compatible with mechanisms for distributing parallel work.

### 6.4.2 Load-balance

The task of scheduling transactions can be simplified to the point that it is similar to that of load-balancing parallel work. This proposal satisfies the requirements because it isolates the scheduling task and provides a mechanism for scheduling tasks that are known not to conflict.

The overheads associated with scheduling concurrent work to reduce conflicts are difficult to justify through increased speed-up because scheduling around conflicts requires a centralised transaction manager and this restricts scalability.

The overheads associated with scheduling concurrent work to ensure progress are difficult to justify through increased speed-up because scheduling transactions to ensure progress requires a centralised view of contention management and this restricts scalability.

When these requirements are removed the problem of scheduling is reduced to one of load-balancing. If it is known that there are no dependencies between access functions then a parallel work scheduler can schedule them to be executed in parallel without the overhead of concurrency control.

The execution of an access function may be regarded as a Memory Transaction. The access functions of an Immutable Data Structure implement a distributed transaction manager internally. When a conflict is detected the transaction manager schedules the transaction for re-execution by adding it to the work-list of the scheduler.

The validate and meld functions are used to implement concurrency control in a Canonical Binary Tree. These functions can be wrapped by the functions which implement an ADT so a function acting on an Immutable Data Structure can be regarded as a chunk of work that can be scheduled by a two-level scheduler. If the validate function fails then the version of the Immutable Data Structure created by the function is discarded and the function may be re-tried. Re-try is implemented by placing the chunk of work back on the work-list.

### 6.4.3 Scheduling parallel work

The science of High Performance Computing focuses on the parallel execution of programs on supercomputers. Its main application is in the simulation of physical systems which evolve over time. Dowd provides a general introduction to High Performance Computing [Dow93]. Kumar, Grama, Gupta, and Karypis describe how schedules for executing parallel work can be determined statically, by the analysis of parallel algorithms [KGGK94]. Parallel algorithms focus on orchestrating the execution of discrete units of work which can be performed in parallel.

The scheduling of parallel work on a Chip Multi-Processor is different from orchestrating parallel work on a supercomputer. Chip Multi-Processors generally have a lower number of processors than Supercomputers and each processor shares a common cache and a common path to main memory. The effects of caching mean that the tasks scheduled on separate execution units affect each other in ways that are difficult to predict. Mattson, Sanders and Massingill describe common parallel application design patterns, which are very different from those of High Performance Computing [MSM04].

The problem of scheduling parallel work on a Chip Multi-Processor cannot be addressed by static analysis of algorithms alone, so parallel work should be orchestrated and load-balanced by a dynamic scheduler. A two-level scheduler implements a dynamic scheduling algorithm for parallel work. Two-level schedulers are designed to permit parallel workloads, such as the simulation of physical

systems, to be efficiently executed by a Chip Multi-Processor.

Blumofe introduces CILK which is a two-level scheduling system for parallel workloads [BJK+96]. CILK implements a run-time scheduler which frees the programmer from static scheduling considerations. The programmer specifies chunks of work which can be performed in parallel by describing them using the CILK programming language. The chunks are assigned to processors by the high-level scheduler. The low-level scheduler orchestrates the chunks to be performed by a particular processor.

The CILK scheduler implements a scheduling policy called work stealing. The low-level scheduler maintains a queue of chunks to be executed. It removes a chunk of work from the front of the queue and executes it. When the queue is exhausted the low-level scheduler steals chunks from the back of a queue belonging to another thread. This makes the scheduling task scalable, because the centralised high-level scheduler is only involved in the initial assignment of the chunks to the queues of each processor.

Intel's Threading Building Blocks product [Int09] is a parallel programming solution for Chip Multi-Processors. Threading Building Blocks applications are written in the C++ programming language and the Threading Building Blocks product is implemented as a library which is linked with the application. The product includes a two-level work stealing scheduler which dynamically schedules chunks of work provided to it on a work-list. This scheduler is similar to that provided by CILK. However, Threading Building Blocks frees the programmer from having to learn a new programming language in order to make use of a two-level scheduler. Reinders provides an accessible introduction to the features of the Threading Building Blocks product [Rei07].

### 6.4.4 Previous work

Ansari et al. propose a scheduling technique called Dynamic Transactional Reordering [ALK+09]. This technique reduces wasted work by re-trying conflicting transactions serially so that they do not repeatedly conflict. It also attempts to avoid both isolation and progress pathologies by implementing a transaction aware work stealing scheduler.

Ansari et al. propose a scheduling technique based on using information obtained by profiling transactional applications [AJK+09]. Profiling information can be used as input to a scheduler which anticipates conflicting transactions and

schedules them to execute serially. Ansari notes that the speed-up obtained by reducing wasted work does not always overcome the scheduling overheads.

The high overhead associated with the distribution and scheduling of parallel work can be contrasted with the low overhead of scheduling Memory Transactions assumed in the Transactional Memory literature. Warg and Stenström describe how the overhead of thread creation prevents fine grained speculative execution on a Chip Multi-Processor from being worthwhile [WS01]. However, some studies of speculative execution assume that the time required to create and schedule a thread is lower than the access time to the second level cache. Quiñones et al. describe an infrastructure for speculative execution which assumes a thread creation time of ten clock cycles [QnMS+05]

### 6.4.5 Transaction granularity

The overhead of scheduling concurrent work places a lower bound on the granularity of transactions that are worth scheduling. Transaction granularity influences many aspects of Transactional Memory system design. Assumptions about transaction granularity influence the style of transactional programming a system permits. For example, at a fine level of transaction granularity it is possible for a compiler to analyse the instructions within a Memory Transaction, whereas at a coarser level of granularity the compiler is less able to reason about the execution.

At a fine level of transaction granularity the amount of speculative state produced by a Memory Transaction is small and the probability that transactions conflict is small, so the amount of work wasted when a conflict is detected is small and the likelihood of work being wasted is low. However, at a coarse level of transaction granularity large amounts of speculative state are produced and the probability of conflict is high, so the amount of work wasted when a conflict is detected is large and the likelihood of work being wasted is high.

To make use of a two-level scheduler the programmer divides an application into chunks of work that are large enough to be worth scheduling. If the chunks are too small then the overheads associated with scheduling each chunk can outweigh the benefits of executing it in parallel with other chunks. If the chunks are too large then the scheduler may not be able to load-balance the work evenly among processors. In practice, it is often difficult to divide an application into suitably sized chunks because it is the expected execution time that determines chunk size. The execution time of the chunks is typically dominated by the

latency of cache misses, which are difficult to predict.

The overhead of scheduling concurrent work is high. Threading Building Blocks requires that a chunk of work should contain at least 10,000 instructions [Int09]. The documentation does not define an instruction in this context but assuming that an instruction completes each cycle, a chunk of work should have an elapsed execution time of at least 10,000 clock cycles to be worth scheduling. In practice, it is difficult to divide an application into transactions which take at least 10,000 clock cycles to execute.

The number of clock cycles required to perform a data structure access is normally dominated by the latency of cache misses. Jacob found that the latency of a single cache miss is around 200 clock cycles and that the latency of consecutive cache misses to dis-contiguous locations is considerably longer [Jac09]. A function acting on a large memory resident data structure may require thousands of clock cycles to execute so functions that access a data structure are potentially worth scheduling for concurrent execution.

# Chapter 7

# Conclusion

## 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

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 [GR92].

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 progress of other processors.

By accepting that simultaneity is relative it is possible to construct a concurrent program that does not have an increased memory bandwidth requirement when executing on multiple processors and that is able to guarantee the lock-free progress of the processors participating in the concurrent execution.

### 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 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 3.6.5 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 reclaimed immediately when it is deleted by the application. Instead, it can be reclaimed only when it becomes unreachable. The vertices that cannot be reached from the root are potential candidates for reclamation but some of these vertices cannot be reclaimed because they are reachable by tardy functions.

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 orchestrate 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.

## 7.3 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 checkpointing 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.

# Glossary

**AA-tree** An AA-tree is a self balancing binary B-tree of order three. 112

**ADT** Abstract Data Type. 17

**amortised analysis** Amortised analysis is a method of analysing the performance of a function acting on a data structure. The idea is that costly operations occur rarely and that their effect on performance is offset by the occurrence of cheaper, more frequent operations. 120

**annotation** The annotation of a vertex is a value used to navigate a path through a tree. An annotation is distinct from a key, which is an argument to a function of the data structure. 80

**API** Application Programming Interface. 33

**atomic section** An atomic section is a section of program code that appears to be performed atomically and in isolation as a transaction [CCG08]. 33

**blocking** A section of program code is said to block when a delay in its execution can delay the execution of other sections. 18

**cascading aborts** A transaction schedule in which a transaction is permitted to read uncommitted values can suffer from cascading aborts. Cascading aborts are an isolation pathology that causes unpredictable run-time behaviour. 130

**composable** Sections of program code are composable if they can be combined without examining or altering their implementation. 18

**concurrency problem** We regard the concurrency problem as the problem of obtaining speed-up from the parallel execution of tasks from a single program on a Chip Multi-Processor when task dependencies are unknown until their execution is complete. 19

**concurrent execution** Concurrent execution refers to the execution of tasks that have the *potential* to execute simultaneously. In this thesis we are concerned with internal concurrency which is the potential to execute related tasks, which may interact through memory and which are part of a single program, simultaneously. We do not consider external concurrency which occurs when a program, such as an operating system, needs to perform several possibly unrelated tasks at the same time. 19

**confluently persistent** A data structure is confluently persistent if it is fully persistent and has a meld operation which combines more than one version. 137

**critical section** A critical section is a section of program code that is protected from simultaneous execution by mutual exclusion. 15

**deadlock** Deadlock is a progress pathology of transactional systems that occurs when each transaction in a set of transactions is blocked waiting for another transaction in the set, and therefore none will become unblocked unless there is external intervention. 186

**dirty read** A dirty read is an access to the uncommitted state of another transaction. 128

**Entanglement** Entanglement is the composition of multiple Immutable Data Structure into one Immutable Data Structure through a process of adding links. Entanglement is achieved by referencing the root address of one Immutable Data Structure from the leaf of another Immutable Data Structure. 141

**fat node** Fat node is a technique for creating new versions of a persistent data structure. 68

**full copying** Full copying is a technique for creating new versions of an Immutable Data Structure. 65

**fully persistent** A data structure is said to be fully persistent if every past version can be both accessed and modified. 135

**immutable** Immutable memory locations are unreachable before they have been written and can be accessed but not modified after they have been written. Immutable memory locations are distinct from singly assigned locations which are reachable before they have been written. 43

**Immutable Data Structure** A data structure is called an immutable data structure if all of the memory locations within it are immutable and it is called a mutable data structure otherwise. 43

**IO** Input and Output. 30

**isolation** Transactional isolation is a guarantee relating only to the actions occurring within transactions. "The system guarantees that for every pair of transactions $T_i$ and $T_j$, it appears to $T_i$ that either $T_j$ finished execution before $T_i$ started, or $T_j$ started its execution after $T_i$ finished. Thus, each transaction is unaware of other transactions executing concurrently in the system [SK86]". 125

**isolation pathology** Isolation pathologies arise when scheduling is applied to enforce reasonable behaviour on weakly isolated transactions. 130

**leaf to root path copy** A leaf to root path copy operation is a type of path copy operation that preserves the position of existing nodes relative to the root node. 77

**linearizability** 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]. 126

**livelock** Livelock is a progress pathology of transactional systems that occurs when two or more competing operations cause each other to restart, preventing any of them making progress. 187

**lock-free** A non-blocking algorithm is lock-free if it guarantees that at least one action completes eventually. "A concurrent object implementation is lock-free if it guarantees that infinitely often some method call finishes in a finite number of steps [HS08]". 187

**marshall** Work assigned to a particular processor under direct application control is described as being marshalled. 38

**meld function** A function that combines past versions of a data structure is called a meld function. 179

**mutual exclusion** Mutual exclusion is a coordination protocol that ensures that only one thread executes within a section of program code at a time. The protocol is said to protect a section of code which we call a critical section. To ensure correctness the protocol does not permit tasks with possible dependencies to execute in overlapping periods of time. 15

**node copying** Node copying is a technique for creating new versions of a persistent data structure. 70

**non-blocking algorithm** A non-blocking algorithm ensures that operations competing for a shared resource never have their progress indefinitely postponed by mutual exclusion. 186

**non-recoverable** A transaction schedule in which a transaction may commit before a transaction that wrote a variable that it has read is called non-recoverable. Non-recoverability is an isolation pathology of transactional systems that leads to inconsistent results. 130

**non-repeatable read** A non-repeatable read is an access to a shared variable that can be modified by another transaction. 129

**obstruction-free** A non-blocking algorithm is obstruction-free if it guarantees that when an action is tried repeatedly and eventually encounters no interference from other actions it will complete successfully but it does not guarantee that such a situation will occur. 187

**orchestrate** Work that is automatically assigned to a particular processor outside the control of the application is described as being orchestrated. 37

**parallel execution** Parallel execution refers to the simultaneous execution of tasks on different hardware processors. 19

**partially persistent** A data structure is said to be partially persistent if all past versions can be accessed but only the most recent can be modified. 135

**path** A path is a set of vertices within an Immutable Data Structure that link the root to at least one leaf. A function can modify an Immutable Data Structure by creating a new path using the path copying technique. 65

**path copying** Path copying is a technique for creating new versions of an Immutable Data Structure. 68

**persistent data structure** A data structure is a called a persistent data structure if it permits access to past versions and it is called an ephemeral data structure otherwise. 135

**phantom read** A phantom read is an inconsistent access to shared state. 129

**priority inversion** Priority inversion is a progress pathology of transactional systems that occurs when a long running operation is preempted by an operation of brief duration. 187

**progress pathology** A concurrent application that guarantees that all of its constituent tasks complete in a finite period of time offers a progress guarantee, whereas an application that does not can suffer from a progress pathology. 184

**pure function** A function is said to be pure if its evaluation does not cause any observable effects other than the production of its result. 42

**purely functional data structure** In a functional programming language immutable values are maintained in purely functional data structures. 78

**referentially transparent** An expression is said to be referentially transparent if it can be replaced with its value without changing the behaviour of the program containing it. 42

**root** The top most vertex in a tree is called the root node. The root of an Immutable Data Structure is a mutable memory location that maintains the address of the root node. 65

**semi-persistent** "A semi-persistent data structure permits access only to those past versions that are ancestors of the most recent version [CF08]". 144

**sentinel** "A sentinel leaf is a specifically designated leaf used with a tree data structure as the path terminator. A sentinel leaf does not hold a reference to any data managed by the data structure [Wir85]". In the context of serial execution a sentinel is a programming convenience. However, in a concurrent execution environment a sentinel leaf can be used to make the distinction between an empty data structure and one that has yet to be created. 81

**serialisable** A transaction schedule in which all transactions appear to execute in isolation is said to be serialisable. A serialisable transaction schedule in which the order of conflicting operations matches the order in which the transactions commit is said to be strict. 130

**speculative execution** The effects of a task that is executed speculatively are tentative and may be discarded, in which case they cannot be observed by other tasks executing in the system. A transaction that is executed speculatively makes tentative changes to objects. If it completes without encountering a synchronisation conflict then it can commit, in which case the tentative changes become permanent otherwise it aborts, in which case the tentative changes are discarded. 16

**SQL** Structured Query Language. 29

**STL** Standard Template Library. 83

**strong isolation** Strong isolation is a guarantee relating to all actions in a system. Strongly isolated transactions are isolated from both other transactions and concurrent non-transactional accesses [DS09]. 125

**tardy reader** A tardy reader is a function that accesses a past version of a persistent data structure which was the most recent version at the moment the function started its execution. 138

**Transactional Data Structure** A data structure is a called a Transactional Data Structure if it permits access to past versions, although not all accesses are successful. 138

**validate function** A validate function ensures that conflicting operations conform to a concurrency control protocol. 176

**wait-free** A non-blocking algorithm is wait-free if it guarantees that every action completes eventually. "A concurrent object implementation is wait-free if each method call finishes in a finite number of steps [HS08]". 187

**weak isolation** A weakly isolated transactional system is one in which successful transactional execution may be affected by other transactions or by non-transactional execution taking place concurrently. Our definition differs from the one used by some authors who refer to a weakly isolated Transactional Memory system as one in which successful transactional execution may be affected by non-transactional execution but not by transactional execution taking place concurrently [DS09]. 125

# 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.

[AF92] Juan Allemany and Ed Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, pages 125–134. ACM Press, August 1992.

[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.

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

[AJK+09] Mohammad Ansari, Kim Jarvis, Christos Kotselidis, Mikel Luján, Chris Kirkham, and Ian Watson. Profiling transactional memory applications. In *PDP '09: Proceedings of the 17th Euromicro International Conference on Parallel, Distributed, and Network-based Processing*. IEEE Computer Society Press, February 2009.

[ALK+09] Mohammad Ansari, Mikel Luján, Christos Kotselidis, Kim Jarvis, Chris C. Kirkham, and Ian Watson. Steal-on-abort: Improving

transactional memory performance through dynamic transaction reordering. In *High Performance Embedded Architectures and Compilers, Fourth International Conference*, pages 4–18, 2009.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA, 1967. ACM.

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[AN95] Arne Andersson and Stefan Nilsson. Efficient implementation of suffix trees. *Softw. Pract. Exper.*, 25:129–141, February 1995.

[And93] Arne Andersson. Balanced search trees made simple. In *WADS '93: Proceedings of the Third Workshop on Algorithms and Data Structures*, pages 60–71, London, UK, 1993. Springer-Verlag.

[And09] Mark Anderson. Sun can kill rock, but not its memory tech. *IEEE Spectr.*, June 2009.

[Arm07] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[ART08] Adl-Tabatabai Ali-Reza and Xinmin Tian. The intel software transactional memory compiler. `http://software.intel.com/file/8097`, November 2008.

[Bag01] Phil Bagwell. *Ideal Hash Trees*. PhD thesis, Department of Computer Science, Ecole Polytechnique Federale de Lausanne, 2001.

[BBB+06] David Bacon, Joshua Bloch, Jeff Bogda, Cliff Click, Paul Haahr, Doug Lea, Tom May, Jan-Willem Maessen, Jeremy Manson, John D. Mitchell, Kelvin Nilsen, Bill Pugh, and Emin Gun Sirer. The "double-checked locking is broken" declaration. `http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html`, 2006.

[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.

[BGH+06] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[BH01] Cuneyt F. Bazlamacci and Khalil S. Hindi. Minimum-weight spanning tree algorithms a survey and empirical study. *Computers & Operations Research*, 28(8):767 – 785, 2001.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BJK+96] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *J. Parallel Distrib. Comput.*, 37(1):55–69, 1996.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, February 1972.

[BMT+07] Woongki Baek, Chi Cao Minh, Martin Trautmann, Christos Kozyrakis, and Kunle Olukotun. The openTM Transactional Application Programming Interface. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 376–387, Washington, DC, USA, 2007. IEEE Computer Society.

[BMV+07] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of the 34rd Annual International Symposium on Computer Architecture. International Symposium on Computer Architecture*, pages 81–91, 2007.

[CBM+08] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software Transactional Memory: Why is it only a research toy? *Communications of the ACM*, 51(11):40–46, November 2008.

[CCE+09] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. Rock: A high-performance Sparc CMT processor. *IEEE Micro*, 29(2):6–16, 2009.

[CCG08] Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 304–315, New York, NY, USA, 2008. ACM.

[CF08] Sylvain Conchon and Jean-Christophe Filliâtre. Semi-persistent data structures. In *Proceedings of the Theory and practice of software, 17th European conference on Programming languages and systems*, ESOP'08/ETAPS'08, pages 322–336, Berlin, Heidelberg, 2008. Springer-Verlag.

[CGE08] Dave Cunningham, Khilan Gudka, and Susan Eisenbach. Keep off the grass: Locking the right path for atomicity. In *CC '08: Proc. International Conference on Compiler Construction*, pages 276–290, March 2008.

[Cha00] Bernard Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *J. ACM*, 47(6):1028–1047, 2000.

[CMCKO08] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[Col86] Richard Cole. Searching and storing similar lists. *J. Algorithms*, 7(2):202–220, 1986.

[CRS05] Joao Cachopo and Antonio Rito-Silva. Versioned boxes as the basis for memory transactions. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, October 2005.

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

[DDS06] O. Shalev D. Dice and N. Shavit. Transactional locking ii. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.

[DGJe09] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson (eds.). *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. American Mathematical Society, 2009. DIMACS Series in Discrete Mathematics and Theoretical Computer Science.

[DHM+06] Anthony Discolo, Tim Harris, Simon Marlow, Simon Jones, and Satnam Singh. Lock free data structures using STM in Haskell. In *Eighth International Symposium on Functional and Logic Programming*, April 2006.

[DL09] Jack Dongarra and Alexey L. Lastovetsky. *High Performance Heterogeneous Computing*. Wiley-Interscience, New York, NY, USA, 2009.

[DLMN09] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 157–168. ACM, March 2009.

[Dow93] Kevin Dowd. *High performance computing*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1993.

[DS09] Luke Dalessandro and Michael L. Scott. Strong isolation is a weak idea. In *TRANSACT '09: 4th Workshop on Transactional Computing*, February 2009.

[DSST86] J R Driscoll, N Sarnak, D D Sleator, and R E Tarjan. Making data structures persistent. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 109–121, New York, NY, USA, 1986. ACM.

[DT92] Dorit Dor and Michael Tarsi. Graph decomposition is npc - a complete proof of holyer's conjecture. In *STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 252–263, New York, NY, USA, 1992. ACM.

[Duf10] Joe Duffy. A (brief) retrospective on tranasactional memory. `http://www.bluebytesoftware.com`, January 2010.

[Enn06] Robert Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, January 2006.

[FH07] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25, May 2007.

[FHS04] Faith Fich, Danny Hendler, and Nir Shavit. On the inherent weakness of conditional synchronization primitives. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, pages 80–87. ACM Press, 2004.

[FK03] Amos Fiat and Haim Kaplan. Making data structures confluently persistent. *J. Algorithms*, 48(1):16–58, 2003.

[GBB+06] Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, David Holmes, and Tim Peierls. *Java Concurrency in Practice*. Addison-Wesley Longman, Amsterdam, 2006.

[GK08] Rachid Guerraoui and Michał Kapałka. On obstruction-free transactions. In *SPAA '08: Proc. twentieth annual symposium on Parallelism in algorithms and architectures*, pages 304–313, June 2008.

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[Gra02] Steve Graves. In-memory database systems. *Linux J.*, 2002:10–, September 2002.

[GS78] Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *FOCS*, pages 8–21. IEEE, 1978.

[GT09] Michael T. Goodrich and Roberto Tamassia. *Algorithm Design: Foundations, Analysis and Internet Examples*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2009.

[GZU+09] Vladimir Gajinov, Ferad Zyulkyarov, Osman S. Unsal, Adrian Cristal, Eduard Ayguade, Tim Harris, and Mateo Valero. Quaketm: parallelizing a complex sequential application using transactional memory. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 126–135, New York, NY, USA, 2009. ACM.

[HCW+04] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Programming with transactional coherence and consistency (tcc). In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 1–13. ACM Press, October 2004.

[Her88] Maurice P. Herlihy. Impossibility and universality results for wait-free synchronization. In *PODC '88: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 276–290, New York, NY, USA, 1988. ACM.

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

[Hic11] Rich Hickey. Clojure concurrency (video). `http://blip.tv/file/812787`, January 2011.

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

[HLMS03] Maurice Herlihy, Victor Luchangco, Mark Moir, and William Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proc. 22nd ACM Symposium on Principles of Distributed Computing*, pages 92–101, July 2003.

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

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[HM97] Pieter Hartel and Henk Muller. *Functional C.* Addison Wesley Longman, April 1997.

[HM08] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *IEEE COMPUTER*, 2008.

[HMPH05] 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.

[Hoa83] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 26:100–106, January 1983.

[HP05] R Hinze and R Paterson. Finger trees: a simple general-purpose data structure. *J. Funct. Prog.*, 16(02):197–217, 2005.

[HP06a] Tim Harris and Simon Peyton Jones. Transactional memory with data invariants. In *TRANSACT '06: 1st Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

[HP06b] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[HPST06] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *Proceedings of the 2006 Conference on Programming language design and implementation*, pages 14–25. ACM Press, June 2006.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[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.

[Int07] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*, August 2007.

[Int09] Intel. Intel Threading Building Blocks: Programming for Current and Future Multicore Platforms. *IEEE/ACM International Symposium on Code Generation and Optimization*, July 2009.

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

[Jac09] Bruce L. Jacob. *The Memory System: You Can't Avoid It, You Can't Ignore It, You Can't Fake It*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[JBR10] Tim Harris Jayaram Bobba, Mark Hill and Ravi Rajwar. The transactional memory bibliography. `http://www.cs.wisc.edu/trans-memory/biblio/index.html`, June 2010.

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

[Jos99] Nicolai M. Josuttis. *The C++ Standard Library: A tutorial and reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[KAJ+07] Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Designing a distributed software transactional memory system. In *ACACES '07: 3rd International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems*, July 2007.

[KAJ+08] Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Distm: A software transactional memory framework for clusters. In *ICPP '08: Proceedings of the 37th IEEE International Conference on Parallel Processing*. IEEE Computer Society Press, September 2008.

[Kap04] Haim Kaplan. Persistent data structures. In *Handbook Of Data Structures And Applications*. Chapman & Hall/CRC, 2004.

[Kar05] Björn Karlsson. *Beyond the C++ Standard Library, an introduction to Boost*. Addison-Wesley Professional, 2005.

[KB09] Seunghwa Kang and David A. Bader. An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs. In *PPOPP*, pages 15–24, 2009.

[KGGK94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing: design and analysis of algorithms*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[KHLW10] Behram Khan, Matthew Horsnell, Mikel Lujan, and Ian Watson. Scalable object-aware hardware transactional memory. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*, EuroPar'10, pages 268–279, Berlin, Heidelberg, 2010. Springer-Verlag.

[Kri03] S. Krishnaprasad. Concurrent/Distributed programming illustrated using the dining philosophers problem. *J. Comput. Small Coll.*, 18:104–110, April 2003.

[LA04] Sean Lie and Krste Asanovic. Hardware support for unbounded transactional memory. Technical report, Masters thesis, MIT, 2004.

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

[Lea06] Douglas Lea. *Concurrent Programming in Java(TM): Design Principles and Patterns (3rd Edition) (Java (Addison-Wesley)).* Addison-Wesley Professional, 2006.

[Lom77] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *SIGPLAN Not.*, 12(3):128–137, 1977.

[Mar08] Petru Marginean. Lock-free Queues. *Dr. Dobb's Journal*, July 2008.

[MBL06] Milo Martin, Colin Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5:17–, July 2006.

[MBM+06] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based Transactional Memory. In *HPCA*, pages 254–265, 2006.

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

[Mor68] Donald R. Morrison. Patricia - practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[Mos99] Graeme E. Moss. Benchmarking purely functional data structures. *Journal of Functional Programming*, 11:525–556, 1999.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. *SIGPLAN Not.*, 40:378–391, January 2005.

[MSM04] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming.* Addison-Wesley Professional, 2004.

[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.

[NMN01] Jaroslav Nešetřil, Eva Milková, and Helena Nešetřilová. Otakar Borůvka on Minimum Spanning Tree Problem, translation of both the 1926 papers. *Discrete Math.*, 233:3–36, April 2001.

[Oka98] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, New York, NY, USA, 1998.

[Oka04] Chris Okasaki. Purely functional structures. In *Handbook Of Data Structures And Applications*. Chapman & Hall/CRC, 2004.

[Olu07] Kunle Olukotun. *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*. Morgan and Claypool Publishers, 1st edition, 2007.

[ÖV99] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, Second Edition*. Prentice-Hall, 1999.

[Pap79] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26:631–653, October 1979.

[PCSF08] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. *Version Control with Subversion*. O'Reilly Media, 2 edition, September 2008.

[PDC+98] Allen Parrish, Brandon Dixon, David Cordes, Susan Vrbsky, and John Lusth. Implementing persistent data structures using C++. *Softw. Pract. Exper.*, 28:1559–1579, December 1998.

[Pey07] Simon Peyton Jones. *Beautiful Code: Leading Programmers Explain How They Think*, chapter 24, pages 385–406. O'Reilly Media, Inc., 2007.

[PGF96] S. L. Peyton Jones, A. Gordon, and S. Finne. Concurrent haskell. In *23rd ACM Symposium on Principles of Programming Languages*, pages 295–308, St Petersburg Beach, Florida, January 1996. ACM.

[PLMW08] Frédéric Pluquet, Stefan Langerman, Antoine Marot, and Roel Wuyts. Implementing partial persistence in object-oriented languages. In *ALENEX Algorithm Engineering and Experiments*, pages 37–48, 2008.

[Pri57] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technology Journal*, 36:1389–1401, 1957.

[PW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 71–84, New York, NY, USA, 1993. ACM.

[PW10] Donald E. Porter and Emmett Witchel. Understanding transactional memory performance. In *ISPASS IEEE International Symposium on Performance Analysis of Systems and Software*, pages 97–108, 2010.

[QnMS+05] Carlos García Quiñones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on precomputation slices. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 269–279, New York, NY, USA, 2005. ACM.

[Raj02] Ravi Rajwar. *Speculation-based techniques for transactional lock-free execution of lock-based programs.* PhD thesis, Department of Computer Science, 2002. Supervisor-Goodman, James R.

[Ree79] David P. Reed. Implementing atomic actions on decentralized data. In *Proceedings of the seventh ACM symposium on Operating systems principles*, SOSP '79, pages 163–, New York, NY, USA, 1979. ACM.

[Rei07] James Reinders. *Intel Threading Building Blocks - Outfitting C++ for multi-core processor parallelism.* O'Reilly, 2007.

[RG01] Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proceedings of*

*the 34th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 34, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society.

[RHW09] Christopher Rossbach, Owen Hofmann, and Emmett Witchel. Is transactional memory programming actually easier? In *WDDD '09: Proc. 8th Workshop on Duplicating, Deconstructing, and Debunking*, jun 2009.

[RTD83] Thomas W. Reps, Tim Teitelbaum, and Alan J. Demers. Incremental context-dependent analysis for language-based editors. *ACM Trans. Program. Lang. Syst.*, 5(3):449–477, 1983.

[RW02] Algis Rudys and Dan S. Wallach. Transactional rollback for language-based systems. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 439–448, Washington, DC, USA, 2002. IEEE Computer Society.

[SATH+06] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *Proc. 11th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP '06)*, pages 187–197. ACM, March 2006.

[Sch06] William N. Scherer, III. *Synchronization and concurrency in user-level software systems*. PhD thesis, Department of Computer Science, Rochester, NY, USA, 2006. AAI3204565.

[SCKP07] Jaswanth Sreeram, Romain Cledat, Tushar Kumar, and Santosh Pande. RSTM: A relaxed consistency software transactional memory for multicores. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, page 428. IEEE Computer Society, 2007.

[Sed98] Robert Sedgewick. *Algorithms in C++, parts 1-4: fundamentals, data structure, sorting, searching, third edition*. Addison-Wesley Professional, 1998.

[Sed02] Robert Sedgewick. *Algorithms in C++, part 5: graph algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Shu08] Herb Shutter. Writing Lock-Free Code: A corrected queue. *Dr. Dobb's Journal*, October 2008.

[SK86] Abraham Silberschatz and Henry F. Korth. *Database System Concepts, 1st Edition.* McGraw-Hill Book Company, 1986.

[SLL01] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library User Guide and Reference Manual.* Addison-Wesley Professional, December 2001.

[SLS09] William N. Scherer, Doug Lea, and Michael L. Scott. Scalable synchronous queues. *Commun. ACM*, 52:100–111, May 2009.

[SR01] Alexandru Salcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs. *SIGPLAN Not.*, 36:12–23, June 2001.

[SSHT93] Janice M. Stone, Harold S. Stone, Philip Heidelberger, and John Turek. Multiple reservations and the oklahoma update. *IEEE Parallel Distrib. Technol.*, 1(4):58–71, 1993.

[ST86] Neil Sarnak and Robert Endre Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, 1986.

[ST95] Nir Shavit and Dan Touitou. Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213, August 1995.

[Sto06] Jon Stokes. *Inside the Machine: An Illustrated Introduction to Microprocessors and Computer Architecture.* No Starch Press, San Francisco, CA, USA, 2006.

[Swe06] Tim Sweeney. The next mainstream programming language: a game developer's perspective. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 269–269, New York, NY, USA, 2006. ACM.

[Tar85] Tarjan, R. E. Amortized computational complexity. *SIAM J. Alg. and Discr. Meth.*, 6(2):306–318, 1985.

[TMG+09] Fuad Tabba, Mark Moir, James R. Goodman, Andrew Hay, and Cong Wang. NZTM: Nonblocking zero-indirection transactional memory. In *SPAA '09: Proc. 21st Symposium on Parallelism in Algorithms and Architectures*, August 2009.

[Van09] Ashlee Vance. Sun is said to cancel big chip project. *The New York Times*, June 2009.

[VHPN09] Vibhav Vineet, Pawan Harish, Suryakant Patidar, and P. J. Narayanan. Fast minimum spanning tree for large graphs on the gpu. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 167–171, New York, NY, USA, 2009. ACM.

[vRV+09] Vladimir Čakarević, Petar Radojković, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky, and Mateo Valero. Characterizing the resource-sharing levels in the UltraSPARC T2 processor. In *Micro-42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 481–492, New York, NY, USA, 2009. ACM.

[VTG+09] Haris Volos, Andres Jaan Tack, Neelam Goyal, Michael M. Swift, and Adam Welc. xCalls: safe I/O in memory transactions. In *EuroSys*, pages 247–260, 2009.

[WA02] Michael Widenius and Davis Axmark. *MySQL Reference Manual*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 2002.

[Wei93] Mark Allen Weiss. *Data structures and algorithm analysis in C*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1993.

[Wir85] Niklaus Wirth. *Algorithms and data structures*. Prentice Hall, 1985.

[WL83] William Weihl and Barbara Liskov. Specification and implementation of resilient, atomic data types. In *SIGPLAN '83: Proceedings of the 1983 ACM SIGPLAN symposium on Programming language issues in software systems*, pages 53–64, jun 1983.

[WM95] Wm. A. Wulf and Sally A. Mckee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23:20–24, 1995.

[WR08] Michał Wichulski and Jacek Rokicki. Persistent data structures for fast point location. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski, editors, *Parallel Processing and Applied Mathematics*, volume 4967 of *Lecture Notes in Computer Science*, pages 1333–1340. Springer Berlin / Heidelberg, 2008.

[WS01] Fredrik Warg and Per Stenström. Limits on speculative module-level parallelism in imperative and object-oriented programs on cmp platforms. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 221–230, Washington, DC, USA, 2001. IEEE Computer Society.

[WS08] M. M. Waliullah and Per Stenström. Intermediate checkpointing with conflicting access prediction in transactional memory systems. In *IPDPS, IEEE International Parallel and Distribued Processing Symposium*, pages 1–11, 2008.

[Zim81] Hubert Zimmermann. The ISO reference model for open systems interconnection. In *Kommunikation in Verteilten Systemen*, pages 39–57, 1981.