A Speculative Execution Approach to Provide Semantically Aware Contention Management for Concurrent Systems

By Craig Sharp

A thesis submitted for the degree of

PhilosophiæDoctor (PhD)

School of Computing Science

Newcastle University

October 2013

Abstract

Most modern platforms offer ample potention for parallel execution of concurrent programs yet concurrency control is required to exploit parallelism while maintaining program correctness. Pessimistic concurrency control featuring blocking synchronization and mutual exclusion, has given way to transactional memory, which allows the composition of concurrent code in a manner more intuitive for the application programmer. An important component in any transactional memory technique however is the policy for resolving conflicts on shared data, commonly referred to as the contention management policy.

In this thesis, a Universal Construction is described which provides contention management for software transactional memory. The technique differs from existing approaches given that multiple execution paths are explored speculatively and in parallel. In the resolution of conflicts by state space exploration, we demonstrate that both concurrent conflicts and semantic conflicts can be solved, promoting multi-threaded program progression.

We define a model of computation called Many Systems, which defines the execution of concurrent threads as a state space management problem. An implementation is then presented based on concepts from the model, and we extend the implementation to incorporate nested transactions. Results are provided which compare the performance of our approach with an established contention management policy, under varying degrees of concurrent and semantic conflicts. Finally, we provide performance results from a number of search strategies, when nested transactions are introduced.

Acknowledgements

I would like to extend my immense gratitude to my supervisor, Doctor Graham Morgan. His guidance and support has been invaluable throughout my time at Newcastle. This thesis would not be possible without the encouragement and experience that Graham has provided.

I would like to thank my colleagues and numerous teaching staff at Newcastle, who have helped in various ways during my PhD studies. Thanks in particular go to Yousef Abusnagh, whose friendship has been particularly valuable. I am indebted to my family and friends, both home and abroad. Special thanks in particular to my Mother and Father, whose unconditional support has been a source of great strength.

Most importantly of all, I would like to express my utmost thanks and appreciation to my loving Wife, Wenyan and my Daughter, Rebecca. Without both of you, none of this would have been possible.

Contents

1 Introduction **1**

1.1 Parallel Computing 1

1.1.1 Classifications of Parallelism 2

1.1.2 The Limitations of Parallelism 4

1.2 Multi-Threading 5

1.2.1 Concurrency Control 6

1.2.2 Mutual Exclusion 7

1.2.3 Transactional Memory 9

1.2.4 Contention Management 10

1.3 Thesis Contribution 11

1.4 Publications 12

1.5 Thesis Outline 12

2 Background and Related Work **13**

2.1 Foundations and Universality 14

2.1.1 Concurrent Objects 16

2.2 Pessimistic Approaches 18

2.2.1 Locking 18

2.2.2 Two-Phase Locking 20

2.2.3 Time Stamps 21

2.3 Speculative Approaches 24

2.3.1 Thread-Level Speculation 24

2.3.2 Speculative Synchronization 25

2.3.3 Speculative Concurrency Control 26

2.4 Optimistic Approaches 27

2.4.1 Transactions 28
2.4.2 Hardware Transactional Memory 31
2.4.3 Software Transactional Memory 33
2.4.4 Coordinating Transactions 35
2.4.5 STM Contention Management 38
2.5 Related Work 43
2.5.1 Serialising Contention Management 43
2.5.2 *Shrink* and Predictive Scheduling 45
2.5.3 *TLSTM* 46
2.5.4 Universal Constructions 48
2.6 Summary and Thesis Contribution 51
2.6.1 Contribution 52

3 The Many Systems Model 55
3.1 Overview 55
3.2 Model Components 59
3.2.1 Events 59
3.2.2 Systems and Processes 60
3.2.3 Expansion 61
3.3 Solution Space 62
3.4 Waiting 63
3.5 Example 65
3.6 A Universal Construction 68
3.6.1 System Processes 68
3.6.2 Universal Construction Processes 69
3.6.3 Proofs 74
3.7 Properties 77
3.7.1 Containment 77
3.7.2 Isolation 77
3.7.3 Liveness 77
3.7.4 Scalability 78
3.7.5 Composable Correctness Criteria 78
3.8 Summary 79

4 Implementation **80**
4.1 Basic Contention Management 82
4.1.1 Overview 82
4.1.2 Preliminaries 84
4.1.3 Registration Phase 87
4.1.4 Speculation Phase 87
4.1.5 Commit Phase 95
4.1.6 Validation 100
4.2 Managing Nested Transactions 101
4.2.1 Speculative Nesting 101
4.2.2 Overview 105
4.2.3 Data Structures 106
4.2.4 Child Session Management 110
4.3 Nested Search Strategies 115
4.3.1 Back-Tracking Search 116
4.3.2 Pseudo Threads 116
4.4 Summary 120

5 Results and Analysis **122**
5.1 Environment 122
5.2 Benchmarked Results 123
5.2.1 Transaction Throughput 124
5.2.2 Average Transaction Execution Time (ATET) 125
5.3 Nested Transaction Results 127
5.3.1 Nested Search Strategies 128
5.3.2 Nested Throughput 129
5.3.3 Nested ATET 131
5.3.4 Registered Versus Commit Rate 132
5.4 Summary 135

6 Conclusion **137**
6.1 Thesis Summary 137
6.2 Main Contributions 137
6.3 Future Work 139

7 Appendix **142**

7.1 Processes 142

7.2 Special Events 143

7.3 Functions 143

References **146**

List of Figures

1.1 A Race Condition 8

2.1 A Non-Serializable Schedule 21
2.2 Two Phase Locking 22
2.3 Speculative Concurrency Control 27
2.4 Transaction Contention 29
2.5 Cache Coherence 32
2.6 Transaction Coordination 36
2.7 Contention Managers 41
2.8 A Universal Construction 49

3.1 The Dining Philosophers 57
3.2 Expansion and Compression 73

4.1 Phases of Contention Management 83
4.2 Serialising Aborted Transactions 84
4.3 Child Sessions 106
4.4 Nested Transaction Execution 107
4.5 The Table Mask Structure 109

5.1 Transaction Throughput 126
5.2 Transaction Timing (in Ticks) 127
5.3 Nesting Throughput Results 130
5.4 Nested Average Transaction Execution Time 133
5.5 Registered Versus Committed 134

List of Tables

1.1 Flynn's Taxonomy 2

2.1 Concurrent Computability Table 15

5.1 Environmental Parameters 123

List of Algorithms

1 The CallTx Function 85
2 Session Registration 88
3 Atomic Object Ownership and Consistency 91
4 The Permutation Functions 92
5 The Greedy Algorithm 93
6 Session Synchronization 95
7 The Contest Algorithm 98
8 Updating the UC Log 100
9 Nested Transactions 103
10 The Table Mask Get Algorithm 110
11 The New Execute Algorithm 111
12 The Nested-Execute Algorithm 112
13 Commencing Nested Execution 113
14 Ending Nested Execution 114
15 The New Permutation Commit Algorithm 115
16 The Back-Tracking Algorithm 117
17 The Pseudo Thread Functions 118

Chapter 1

Introduction

1.1 Parallel Computing

In the field of Computer Science and Software Engineering, Parallel Computing covers numerous techniques and offers several advantages over Sequential Computing, specifically:

Speed – Multiple processing elements can compute the solution to certain problems in less time than a single processor. Given a computer program designed to solve a series and a number of processors, the parallel solving of those tasks can provide gains in speed. As more processors are added to the computation, we hope that the time required to reach a solution decreases;

Problem Solving – Some problems are highly parallel in nature but require excessive time to compute in a sequential algorithm. In theory they can be solved more easily if the computation is performed on a parallel system;

Fault Tolerance – Parallelism offers safety in numbers. Having multiple processors working on the same problem can provide a measure of fault tolerance of the application. Methods of replication have been applied in hard real-time systems using parallel computations.

Processing frequency scaling has meant that sequential programming has provided an easier alternative to the more difficult task of parallel programming. As

processors frequency increased, so too increased the speed with which sequential programs could be executed. Around the beginning of the 21st century, however, limitations on frequency scaling have placed renewed emphasis on parallel computing as the only way in which the maximum performance can be obtained from multi-processor platforms. As of writing, computing platforms with increasing numbers of processor cores are appearing on the market and nearly all modern processors incorporate parallel execution at multiple stages of their design.

The fundamental difficulties of Parallel Programming have yet to be addressed in a comprehensive manner and it is common to find that many applications make inefficient use of the parallel resources at their disposal. Computer programs that are constructed to exploit parallelism must address numerous challenges particular to parallel programming. In general, the parallel programmer must first identify tasks that can be executed in parallel and those tasks must be distributed among available processing resources. Finally, once the computation is completed the tasks must be synchronized to present the user with the final solution.

1.1.1 Classifications of Parallelism

It is useful to identify the patterns of parallelism that exist in computing. Flynn's Taxonomy presents a classification of parallel and sequential systems. Flynn described any system in terms of four classifications: Single Data, Single Instruction (SDSI); Single Data, Multiple Instruction (SDMI); Multiple Data, Single Instruction (MDSI) and Multiple Data, Multiple Instruction (MDMI). Table 1.1 shows the relationship between instructions and data in the context of his parallel taxonomy.

	Single Instruction	**Multiple Instruction**
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Table 1.1: Flynn's Taxonomy

Different approaches to hardware and software design exemplify Flynn's Taxonomy:

Bit-Level Parallelism – Parallelism begins at the microprocessor level. Beginning in the 1970s, microprocessor design incorporated increasing word sizes, thus increasing the amount of data the processor could manipulate per processor cycle. Beginning with 4-bit processors, word size gradually doubled until, at the time of writing, 64-bit word sizes are common;

Instruction-Level Parallelism – Following on from the gains in Bit-Level Parallelism, Instruction Level Parallelism further enhances the parallel performance of the processor. Modern microprocessors feature instruction pipelines. At each stage of the pipeline the processor applies an action to the executing program's data (e.g. instruction decode, memory access, execute). The introduction of multiple pipelines allowed instructions to be executed in parallel by the processor (per processor clock-tick);

Data Parallelism – Programs usually feature areas of code wherein the data can be computed in parallel; data parallelism is concerned with identifying and executing these areas. Loop parallelism and matrix calculations are two examples which typically feature heavy data parallelism when a series of loop iterations can be performed in any order. (This model of parallelism is sometimes classified as SIMD);

Task Parallelism – The rise of multi-threading to accomplish several tasks in parallel. Modern multi-process operating systems for example, perform multiple tasks in parallel (e.g. executing applications, performing services, initiating network connections). Under Flynn's terminology, Task Parallelism can also be referred to as MIMD.

The Taxonomies of Parallelism have been combined and applied comprehensively in the design of modern computing platforms. For example, modern hardware architectures feature multi-core processors with regular registers of 64-bit word size and floating point (SIMD) registers with a capacity of 128 bits. Operating System tasks can be executed in parallel while modern compilers can also influence processor operation to speculatively executed instructions out of program order.

1.1.2 The Limitations of Parallelism

The goal of Parallel Computing is to provide the user with an abstraction which hides the complexities of parallelism, while making the most efficient use of the parallel processing resources of the target platform. Ideally, increasing parallelism should provide increasing gains in programmer productivity coupled with increasing performance as long as more parallel resources are made available.

An optimal rate of speed-up of a sequential program would ideally halve the time required for the computation whenever the number of processors assigned to the task is doubled. In general this performance gain has not been possible to achieve. Typically one finds that speed up is achieved at an almost linear rate for small numbers of processors. As the number of processors increase, the rate of speed-up diminishes until the addition of another processor provides no gain in speed at best, and significant degradation in performance at worst.

Even if potential parallelism can be identified in a given program, and parallel tasks of execution can be constructed, two issues must still be tackled:

- The *granularity* of the task, specifically the cost required to complete the work, should be greater than the cost of distributing and retrieving the results work.
- The degree of inter-processor coordination required to complete the work should be sufficiently small so that excessive time is not wasted performing thread communication (i.e. the *locality* of the task).

Amdahl's Law During the 1960s, Gene Amdahl presented a basic equation to analyse the potential parallel speed-up of an algorithm. By dividing any algorithm into a parallel and a sequential component, Amdahl proposed that the time spent computing the sequential component would place an upper limit on the speed-up that could be attained in the parallel component. Equation (1.1) expresses Amdahl's law with the speed-up (S) that can be expected given the fraction of time spent executing the sequential component (t).

$$S = 1/t \tag{1.1}$$

Gustafson's Law Gustafson's law addresses the implied limitations of Amdahl's law. Specifically, Gustafson disputed the impact of the serial component of execution. Gustafson argued that the serial component of a program is not static and could be diminished by increases in power and resources. Gustafson's law is shown in equation (1.2). With P processors, the speed-up S which can be attained is shown.

$$S(P) = P - t(P - 1) \tag{1.2}$$

In conclusion, while Gustafson offers hope that producing scalable parallel programming techniques is feasible, the dynamism, complexity and uniqueness of computer programs presents an inherent difficulty that hinders the accomplishment of this goal.

1.2 Multi-Threading

Multi-threading provides programmers with an execution model which allows the creation of multiple independent processes (namely Threads), within a single operating system process. At the time of writing, multi-threading in tandem with shared memory, is arguably the most widely used model of concurrent programming.

Before the advent of multi-processor platforms, multi-threading provided a programming abstraction more aesthetically intuitive and natural (than a purely sequential execution) with respect to some programming tasks. Once platforms incorporated multiple processors, however, true parallelism could be exploited by multi-threaded programs (with sufficient support from the Operating System). Multi-threading thereafter provided the ability to exploit the parallelism of the host platform to increase the speed and responsiveness of program execution.

Multi-threading, while seemingly intuitive as a model of execution, has enormous implications for program design:

- Determinism is lost when multi-threading is introduced because threads can access shared data in different orderings from one execution of a program to the next. The loss of determinism has widespread implications for the ability of programmers to predict the outcome of concurrent programs;

- Non-deterministic execution via multi-threading can introduce inconsistency in shared data, complicating the programmers debugging efforts.

When reasoning about program execution, sequential programming has the benefit of *determinism*, such that multiple executions of the same sequential program, with the same inputs, yield the same results. When errors arise in program execution, the predictable nature of sequential programming has the benefit that those errors can often be reproduced, isolated and (hopefully) corrected. Essentially, this predictability is lost to a greater extent in concurrent programming and Concurrency Control is required to reintroduce determinism where predictable execution is necessary (while otherwise allowing the exploitation of parallelism).

1.2.1 Concurrency Control

Unlike sequential programming, in a multi-threaded program, execution is composed of non-deterministic inter-leavings of sequential threads of execution. Concurrency control essentially enforces determinism at critical sections of shared memory access, removing the interleaving of thread execution. Unfortunately, understanding the extent to which determinism should be enforced is a major obstacle for most programmers. If concurrency control is applied too restrictively, then the parallelism afforded by the host platform cannot be exploited. Unfortunately, increasing the number of threads in an application exacerbates the difficulty of determining the correct level of determinism.

There are a number of approaches to implementing concurrency control, but in the field of shared memory computing, the two most prominent are *Pessimistic* and *Optimistic* concurrency control:

Pessimistic – Approaches to concurrency control where inconsistencies caused by non-determinism are prevented, typically by blocking synchronization, before they can take place;

Optimistic – Concurrency control which typically features speculative execution, where inconsistencies caused by non-determinism are detected and undone after they have taken place.

Blocking synchronisation via *mutual exclusion* has been a common approach in implementing pessimistic methods of concurrency control. At the time of writing, *transactional memory* is a popular optimistic technique. In practice, however, there exist so many applications where concurrency control is needed, and therefore no single approach works best in every situation. Whether optimistic or pessimistic methods are used, however, the main goals of any concurrency control mechanism include:

Correctness – The concurrency control technique should not infringe the logical correctness of any program to which it is applied (i.e. based on the equivalence of a multi-threaded execution with a sequential execution).

Efficiency – The concurrency control technique should not be an undue burden on the execution platform. Resources are at the concurrency control mechanism's disposal should be used to maintain efficiency (i.e. to minimise the enforcement of deterministic execution).

Pessimistic concurrency control was arguably most prominent when computing resources were relatively scarce with respect to 21st century standards (in terms of memory capacity, for instance). Under such conditions, pessimistic methods offered a justifiably conservative approach. In distributed database programming, however, where resources tended to be greater, optimistic methods were favoured. Transactions were first developed for distributed database software; user requests were executed concurrently, undone and retried if a concurrency error was detected. Given the high latency of networked requests, the costs of executing transactions rather than mutual exclusion were not considered excessive. As computing resources have expanded universally, optimistic methods now feature in many areas where pessimistic methods were once used.

1.2.2 Mutual Exclusion

When programs involve multi-tasking, care must be taken to ensure tasks accessing the same shared data do not inadvertently introduce concurrency errors into the program. Figure 1.1 illustrates the typical concurrency error known as a race

condition, where both threads ($T1$ and $T2$), read the same memory location (x) and increment the value held therein. Because the act of incrementing a memory location consists of multiple machine instructions, threads $T1$ and $T2$ may find that their instructions are arbitrarily interleaved to produce an outcome that is not expected.

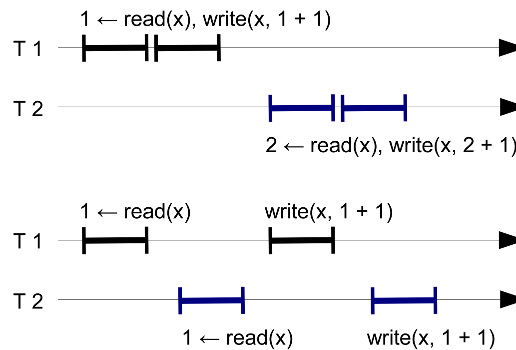

Figure 1.1: A Race Condition *In the top scenario, threads T1 and T2 execute their statements without interference, and the value of memory location x holds the correct value 2. In the bottom scenario, however, their instruction are interleaved and the final value is set to 1.*

Most errors that are generated in concurrent programs are fundamentally examples of race conditions, yet identifying and correcting race conditions becomes extremely difficult in large software applications with many interacting components. Mutual exclusion has been a standard approach to dealing with race conditions for many years, especially where memory access is restricted. The aim of mutual exclusion is to limit access to only one concurrent task at a time in a *critical region*. Mutually exclusive critical regions provide two principle benefits:

- Determinism is restored during the execution of the critical section as only a single thread can access the shared data;
- By providing predictable modification of shared data, Mutual exclusion can provide coordination between the actions of threads.

The restoration of determinism maintains consistency of shared data, while coordination promotes efficiency with respect to the expedient execution of the

program. Programming constructs such as *mutexes* (or *semaphores*) may be enhanced to provide user-defined conditions which allow more sophisticated, and complex coordination of thread execution. Care must be taken when using any blocking synchronization technique, however. Three adverse effects can be caused by improper use of a critical region, namely:

- The potential parallelism is restricted if the critical region is too long as only a single thread can make progress.
- Deadlock can occur if two or more threads wait for each other to finish some action.
- Resource starvation can occur if some threads continually monopolise a critical section, blocking access for others.
- Livelock can result if threads find themselves perpetually responding to each other's actions, instead of progressing with their own execution.

Deadlock in particular is a catastrophic state for any program, given that no further progress can be made. Deadlock can occur in even seemingly trivial programs. For example, thread 1 may gain exclusive access to a variable a while it requires variable b. Concurrently there exists some thread which has already acquired exclusive access to b but now requires access to a. In this situation, neither thread can ever progress with their executions. Where blocking synchronisation is used, care must be taken to avoid deadlock, or methods must be available to detect and then abort the occurrence of a deadlock.

1.2.3 Transactional Memory

Concurrency control can be implemented optimistically in multi-threaded programs. If potential race conditions arise from the shared memory accesses of threads, their effects are detected and discarded. Transactional memory is a popular example of such an approach, where threads contain their shared memory accesses within the execution of transactions. The transactional model generally requires that changes to shared data are made speculatively by the executing

thread and if no concurrency conflict is detected, then the changes are made permanent (by committing the transaction). When conflicts are detected, then some transaction must abort and restart.

To maintain consistency of shared data, transactions must be: (i) *atomic*, such that either all the execution steps of the transaction occur or none at all; (ii) *consistent* so that the logic of the application is not infringed and (iii) *isolated* so that changes to shared data cannot be observed by any other thread until the transaction is committed.

The main benefit of transactional memory in comparison to the use of*critical regions*, is that deadlock can be eliminated from the program because the various threads of execution have the property of *obstruction freedom*. Obstruction freedom states that in the absence of activity by other threads, some thread can always make progress. In addition, the execution of transactions can be composed, and so transactional memory offers an approach that is general purpose and scalable in terms of providing concurrency control for complex applications.

1.2.4 Contention Management

While transactional memory addresses many of the problems associated with pessimistic approaches (e.g. deadlock due to blocking synchronization), thread starvation may arise when the demand for shared memory access is high. In systems programming (the domain of transactional memory) multi-threaded access for shared data tends to be more excessive than in distributed database applications. Contention management is therefore a crucial component of transactional memory, particularly in the field of software transactional memory.

Contention management is typically described in terms of a contention management policy (CMP). A number of early CMP implementations used various criteria to determine a back-off time, and a conflicting thread would typically abort its transactions and sleep for the duration of the back-off time. More recent policies attempt to reorganise the thread schedules to reduce the likelihood of a future conflict from arising. The aim of most policies, however, is to enhance the throughput of transaction execution by minimising the likelihood of inter-thread conflict.

1.3 Thesis Contribution

The Thesis addresses scientific and engineering problems associated with concurrency control in general purpose computational systems. Focus is placed on deriving software solutions for practical deployment of concurrency control on multi-core hardware architectures. The Thesis presents work suitable for current commercial hardware and future hardware where core numbers are expected to rise significantly.

The Thesis provides the following contributions:

- A fundamental rephrasing of the concurrency control problem from that of primarily managing conflict that inhibits correctness to one of searching for optimal execution patterns in parallel at run-time. We show, theoretically, that searching execution spaces can be achieved in a wait-free manner while a general purpose technique can be implemented to provide real-time concurrent execution in a scalable, lock-free manner.

- For the first time, a solution that tackles semantic issues within the application to determine the validity of the concurrency control step. Ordering of accesses can increase throughput and semantic correctness. For example, if two accesses (pop and push) on a shared empty list are ordered pop first then, semantically, there will be a failure to retrieve. However, if the push were ordered before the pop then this would be correct semantically. A search of possible future states of execution has a higher likelihood of discovering this. As our search is independent of execution or data structure, this element of the solution is general purpose in nature.

- An embodiment of our theories within a software engineered solution that is benchmarked against standard experiments to demonstrate effectiveness.

For the sake of deriving a practical realisation of our approach we bound our state exploration in a programmer-defined manner popularised within transactional systems (begin/commit/abort). As such, contention management in Software Transactional Memory (STM) coupled with rescheduling aborted transactions, is the closest neighbour to our engineered solution. Therefore, it is with these approaches that we compare our solution.

1.4 Publications

The following official publications represent contributions that the author has produced/participated in, during the creation of this thesis:

Hugh: A Semantically Aware Universal Construction for Transactional Memory Systems Sharp C, Morgan G. 19th International Conference on Parallel and Distributed Computing (Euro-Par) 2013.

Volatility Management of High Frequency Trading Environments Brook M, Sharp C, Blewitt W, Ushaw G, Morgan G. 15th IEEE Conference on Business Informatics 2013.

Semantically Aware Contention Management for Distributed Applications Brook M, Sharp C, Morgan G. Distributed Applications and Interopable Systems 2013 (Pages 1-14).

Liana: A Framework that Utilizes Causality to Schedule Contention Management across Networked Systems. Abushnagh Y, Brook M, Sharp C, Ushaw G, Morgan G. On the Move to Meaningful Internet Systems: OTM 2012 (Pages 871-878).

1.5 Thesis Outline

The Thesis is organised into 6 chapters. In Chapter 2, background technologies and related work are provided to contextualise the contribution of the thesis. Chapter 3 provides a description of the Many Systems model, and Chapter 4 describes an implementation of a Many Systems contention manager. Chapter 5 provides results from a number of experiments designed to evaluate the performance of the contention manager, and Chapter 6 concludes the thesis and discusses possibilities for future work.

Chapter 2

Background and Related Work

In this chapter a selection of developments in the area of concurrency control is described, which relate to the contribution of the thesis. We introduce the chapter with a description of synchronization primitives and the problem of consensus, followed by an overview of the various progress conditions that are a feature of concurrent programming. These provide the 'building blocks' for all the higher level lock-free/wait-free data structures that are ubiquitous with concurrent programming. The remaining background material of this chapter describes a (broadly) chronological evolution from early approaches to more recent developments in concurrency control, which might be considered 'state of the art'. Three approaches in particular are covered, namely:

Pessimistic Approaches – Blocking synchronization is introduced, specifically in the application of locking data to provide conservative progression of concurrent programs.

Speculative Approaches – Parallel processing redundancy is exploited to speed up program execution with speculative execution in both sequential and concurrent programs.

Optimistic Approaches – Where speculation is applied in the execution of transactions; threads make modifications to shared data optimistically and abort their transactions if conflicts occur.

The Related Work section focuses on strategies which provide Contention Management in Software Transactional Memory. A number of recent techniques are described to put the work of this thesis into context. Finally, in Section 2.6 we summarise the contribution of this thesis in the context of related work.

2.1 Foundations and Universality

Many synchronization primitives (or atomic instructions) have been devised and some of these have found their way into the instruction sets of modern processors. *Atomic operations*, providing linearizable load and store operations, are available in many different specifications with respect to how many readers and writers are supported. Read-Modify-Write (RMW) operations are available on many platforms with operations such as *fetch-and-add* and *test-and-set*. And yet more complex operations like *compare-and-swap* or *load-linked/store-conditional* are included in the instruction set of most modern processor architectures.

Herlihy proved that an important aspect of these synchronization primitives is that they are not equally useful when it comes to solving a range of synchronization problems [1]. Instead, one may assess the power of a synchronization primitive by the degree to which 'higher level' concurrent objects can be supported by a particular primitive. Crucial to the evaluation of synchronization primitives is the problem of *consensus*. *Consensus* has far reaching consequences for the design of any system where multiple participants are involved who need to reach a shared agreement in finite time. The essence of any *consensus* protocol, regardless of how many threads of execution are involved, is the fulfilment of the following requirements:

Agreement − All participating threads must *decide* on the same result.

Integrity − Each participating thread *decides* at most one value, and whichever value is *decided* must have been proposed by some participating thread.

Termination − All participating threads *decide* on some value.

Validity − The agreed result must have been proposed by a thread participating in the *consensus* protocol.

Herlihy provided a *consensus* hierarchy (shown in Table 2.1) to illustrate the power of a particular synchronisation primitive with respect to solving *consensus*. Any particular synchronisation primitive corresponds to a maximum number of threads for which that primitive can solve *consensus*, thus each primitive has a *consensus* number. For example, *fetch-and-add* has a *consensus* number of 2 and therefore can only solve *consensus* for up to 2 participating threads.

The limitations of *consensus* numbers are important for the design of *Wait-Free* concurrent objects. The Wait Free property is described in Section 2.1.1. Specifically, when using a synchronisation primitive such as *fetch-and-add*, it is not possible to build a consistent wait-free object with more than 2 threads.

Consensus No.	**Object**
1	*atomic registers*
2	*test-and-set, swap, fetch-and-add, queue, stack*
...	...
$2n-2$	*n register assignment*
...	...
∞	*memory-to-memory move, compare and swap,* *load-linked/store conditional*

Table 2.1: Concurrent Computability Table provides the universality hierarchy of synchronization operations

Observe in Table 2.1 that some synchronization primitives have an infinite *consensus* number, and can be used to solve *consensus* for any number of threads. One such primitive is *compare-and-swap*, which requires three arguments: typically a memory location (m), a compare value (c) and a value to swap (s). *Compare-and-swap* operates by comparing the contents of m with c; if they are equal, the contents of m are replaced with s. If the swap is performed, then the overwriting of the contents with s takes place in one “atomic” operation. Overwriting m can be made on the safe assumption that the *compare-and-swap* operation read the most up-to-date contents of m.

With good knowledge of the limitations of various synchronization primitives, in terms of their *consensus* numbers, a concurrent programmer can avoid the

wasted effort of solving what are essentially impossible *consensus* problems using atomic operations of lower *consensus* numbers. The applicability of the limits of *consensus* has far reaching consequences, given that almost any synchronization technique can be implemented using these atomic primitives.

2.1.1 Concurrent Objects

One may describe concurrency control from the perspective of *concurrent objects*, where a *concurrent object* refers to any data structure or entity (a register, a container etc.), which provides some equivalent behaviour to a sequential object. For example, if a *concurrent object* is a stack data structure, then equivalent behaviour would require the ability to invoke *push* and *pop* operations on the concurrent stack. We may reason about *concurrent objects* by their *correctness* and *progress* properties:

Correctness – Threads should interact with the object in a manner which does not introduce inconsistency of the object. One may identify correctness by examination of the history of thread interaction with a concurrent object.

Progress – With respect to delays that are introduced, in order to maintain correctness when threads interact with the object. In a *blocking* implementation, a single thread can delay other threads. In a *non blocking* implementation, a single thread cannot delay other threads.

Correctness and Memory Consistency In order to determine whether a *concurrent object* exhibits correctness, it is first necessary to define a criteria for evaluating the behaviour of a *concurrent object*; behaviour that can be considered from the effects of the methods executed upon it. We assume that multiple threads may invoke these methods in parallel, and consider the execution of a method as a 'blackbox', only paying attention to the method invocation, and the response that is returned. Under this criteria, there are three levels of consistency:

Quiescent Consistency – Is the weakest consistency level and suited for scenarios where the greatest freedom is permitted in the interleaving of thread execution. The only requirement is that method calls separated by a period

of inactivity (i.e. quiescence) should respect their real time order, although for method calls which overlap, we make no assumptions;

Sequential Consistency – Requires that an interleaved method execution by multiple threads produce a state that is equivalent to a sequential schedule containing only non-interleaved method execution.

Linearizability – requires that method calls on *concurrent objects* appear to take place atomically. Linearizability judges actions on a *concurrent object* from the program as a whole and as such allows composition.

Progress When reasoning about the progress property, we are interested in the liveliness of the interactions with the object. Three degrees of liveness in particular are available:

Obstruction Freedom – is the weakest liveliness condition and requires that any thread, executed in isolation from obstructing threads, can complete its operation within a bounded number of steps.

Lock Free – characterises a technique where no thread is blocked and waiting indefinitely for the execution of another thread. *Deadlock* can not occur in a lock-free concurrency control method (although *Livelock* remains a possibility).

Wait Free – where each function called by a thread of execution, finishes in a finite number of steps. Each execution step by a thread brings progress to the system as a whole and *livelock* is not possible. Wait-free presents a somewhat stronger guarantee than lock-free but is typically more difficult to implement.

Interestingly, in many situations, lock-free solutions have been judged more efficient than wait-free counterparts, because of the extra work involved in any wait-free mechanism. Although the wait-free solution appears superior to the lock-free solution, in real systems the wait-free approach is often abandoned in favour of a lock-free one.

2.2 Pessimistic Approaches

Pessimistic concurrency control techniques are numerous but they may all be categorized by the activity of blocking (i.e. one thread of execution may interrupt the activity of other threads when contention for shared data access arises). The term *pessimistic* is used because the approach assumes that the worst case will occur with respect to concurrent interference of shared data. Hence the characteristic of a *pessimistic* approach is to take whatever steps are necessary to avoid such interference from arising.

2.2.1 Locking

Early concurrent programmers devised locking constructs to prevent race conditions, and yet locking still remains a widely used technique to date. Essentially, locking data requires that threads sacrifice access to shared data on occasion, so that the locked data can be accessed or modified in a deterministic manner. A trade-off ensues where the gain from maintaining correctness offsets the gains in parallel speed, with threads requiring a longer average period of time to complete their data accesses.

There are many approaches available with respect to lock implementations but when a thread of execution cannot gain access to the lock, two fundamental approaches exist:

Spinning – if the thread does not expect the lock to be held for a long duration of time, then it can repeatedly poll the lock until it becomes available, commonly referred to as *spinlocks* or *busy waiting*;

Blocking – if the thread expects the lock to be held for a long period of time, then the waiting thread can be suspended so that another thread can gain access to the scheduler via an operating system context switch. As context switching tends to be expensive, this only makes sense if the period of waiting is expected to be long.

Most operating systems have access to operations which facilitate the implementation of locks (e.g. non-interruptible critical regions) and locking has been

used extensively for many years in a range of application domains, particularly in operating system programming. As a result, locks are frequently encountered in 'legacy code' and there exist numerous software implementations of locking constructs, widely available for programmer use. In addition, locking applications are supported by extensive documentation.

When confronted with sophisticated multi-threaded programs, however, programmers who use locks encounter great difficulty given that locks cannot be composed. For example, while it is possibility to apply mutual exclusion to prevent the occurrence of race conditions on a single data structure, a thread cannot guarantee that acquiring multiple locks will not introduce a deadlock. Devising an algorithm that avoids deadlock tends to introduce a non-transferable bespoke solution, suitable only for a particular piece of software. In addition, a great deal of time is typically required to prove that complex locking applications do not introduce deadlocks into the program.

In addition to the difficulty of implementing sophisticated locking protocols, locking also raises the issue of efficiency. Given the pessimistic nature of blocking thread execution with locks, application progress is often hindered even in situations where concurrent access to shared data would not have caused inconsistencies. Pessimistic approaches assume that errors will always arise if threads are not inhibited, and hence locking can result in execution bottlenecks, especially as the number of threads increases.

Read-Write Locks To reduce potential bottlenecks and increase the level of concurrency possible when locking is used, different types of locks are available which grant different levels of access rights to shared data. Most simply, locks can be distinguished between *read* locks and *write* locks. Typically, a lock will only allow exclusive access to a resource, but if there are a number of threads which only wish to *read* from the resource, then it is inefficient to prohibit these *read only* threads from accessing the resource in parallel (since *read only* threads provide the guarantee that they will not modify the resource).

Read and write locks allow a thread to specify whether they intend to access a resource for reading or writing. Whenever a thread wishes to write/modify the resource, it must acquire a write lock which behaves like a typical lock, granting

exclusive access to the writer. Whenever one or more threads wish to read the resource, they may gain access to the critical region in parallel, thus improving the level of concurrency available to the threads. As an example of the error prone nature of locks, however, even this simple mechanism introduces the possibility of catastrophic errors:

- Given that a writer can only lock a resource if there are no readers, there is the possibility that the writer will 'starve' if reading of the resource is prolific. In such situations, the writer cannot gain access to the resource because there are too many readers creating a perpetual stream of requests for the resource.

- Queuing writers for access to a shared resource, and prioritizing queued writers can address the problem of writer starvation. However, such a solution reduces the very parallelism that the read/write lock was supposed to alleviate. This is because readers and writers now have to perform coordinated queuing operations to respect the dynamic priorities assigned to writers.

2.2.2 Two-Phase Locking

Locking alone can produce non-serializable schedules when multiple locks are used by multiple threads during an execution schedule. With two threads, a serializable schedule contains read/write instructions which can be reordered, so that the reordered schedule is equivalent to a serial schedule (where one thread's instructions completely preceded the other). However, if a thread reads or writes to a data item which another thread has written to, those instructions are causally linked and cannot be reordered without violating the semantics of the thread's read/write instructions. An example is provided in Figure 2.1 which shows an interaction between two threads which produces an execution which is not serializable. Note that:

- Thread 1 could not have preceded Thread 2 because item B was written to by Thread 2 and subsequently read by Thread 1.

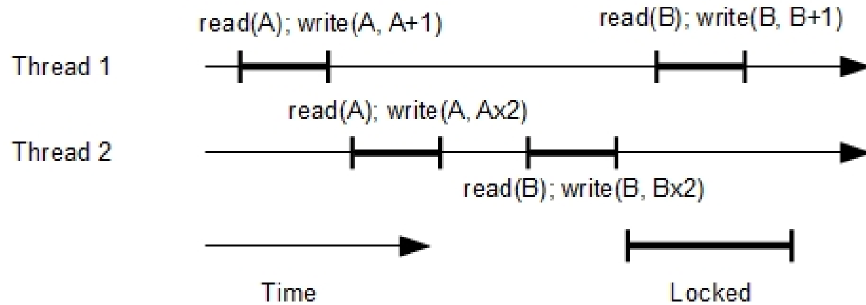

Figure 2.1: A Non-Serializable Schedule *The read/write instructions of both threads cannot be reordered to produce an equivalent schedule where either Thread 1 or 2 executed strictly in serial.*

- Thread 2 could not have preceded Thread 1 because item A was written to by Thread 1 and subsequently read by Thread 2.

It is not possible to reorder the instructions in a manner that either thread executed strictly in serial. Consider the implications of this non-serializable schedule if a constraint exists on data items A and B which states that they must always hold equivalent values; in this instance the constraint would be violated even though both threads had executed statements which respected the constraint.

Two-Phase locking can overcome this problem without resorting to locking data items unnecessarily (and thus reducing the potential concurrency of the application). Any thread or process utilizing a Two-Phase locking scheme performs an *acquisition* phase and a *release* phase (see Figure 2.2). During the *acquisition* phase, a thread attempts to acquire the locks it needs to guarantee exclusive access to the shared objects it seeks to access/modify. Once such a thread has acquired the necessary locks, it makes the desired modifications and releases the previously acquired locks. Under a Two-Phase locking scheme the constraint on data items A and B is not violated because Threads 1 and 2 must first lock both data items before performing any changes.

2.2.3 Time Stamps

An alternative method to locking, particularly in some database applications, is to enforce concurrency control using timestamps. Where concurrent access to shared

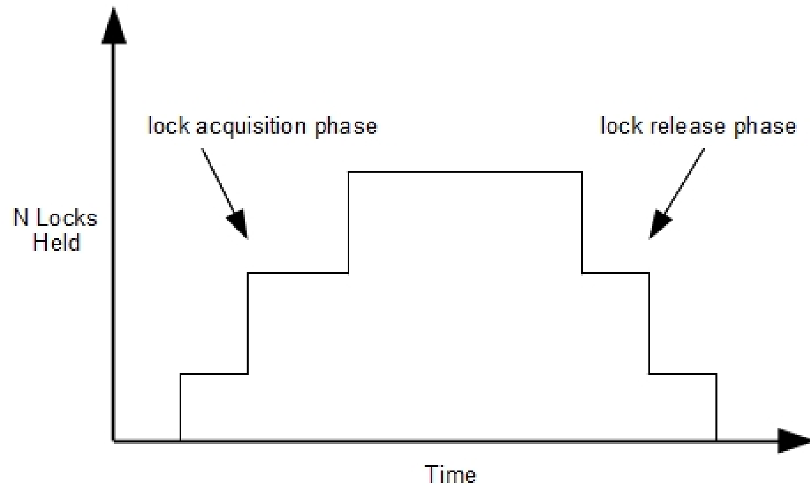

Figure 2.2: Two Phase Locking *The time/n-locks graph shows a single thread or process gradually acquiring locks, executing on the shared lock-protected data before gradually releasing the acquired locks.*

objects exist, timestamps are essentially a value granted to each participating thread, such that:

1. The timestamps generated should be unique to each participating thread;

2. The timestamps must allow orderings to be identified between multiple concurrent actions.

For example, if one considers a database which holds multiple shared objects, then the reads and writes of each thread can be identified with a timestamp to dictate in what ordering those read and writes should occur to maintain a sequential history of access on the shared objects. Because the asynchronous nature of thread communication means that accesses can be received outside of timestamp order, the timestamp allows such *out-of-order* accesses to be detected and aborted.

Aborting a thread's shared access requests when a timestamp violates the sequential ordering can have a negative effect on performance when there are a large number of requests that must be aborted. Improved performance can be achieved, however, with the help of a buffer [2]. Rather than requiring the server

to deal with requests 'as they arrive', they can instead join a buffer for a specified duration. Then, by examination of the buffered requests (and depending to some extent on the behaviour of the application), an ordering of access requests can be selected which reduces the number of accesses that have to be aborted. The difficulty in applying this technique effectively, however, is knowing a priori: (i) the duration of time that requests should be buffered and (ii) how many requests should be stored.

Various methods of generating timestamps have been implemented. For example, timestamps can be generated from the system clock on the host platform or monotonically increasing integer values (sometimes called a logical clock) can be used. Other timestamps schemes apply combinations of system clocks and logical clocks. For instance, timestamps can be used to maintain serializable executions over distributed systems. In a distributed system, each site can be given a unique ID, and a logical clock is appended to the site ID to form the timestamp. With this method, timestamps remain unique to a particular thread running on a particular site.

Compared to locking, timestamping is particularly well suited to the task of maintaining causality over geographically distant hosts, due to the high latency involved with inter-host communication. The following issues, however, are significant when implementing concurrency control with timestamps:

Resolution – When clocks are used to generate timestamps, the granularity of the timing mechanism used must be sufficiently precise to ensure that time-stamps generated at very close intervals possess unique values.

Locking – Whether a real-time clock or logical clock is used, some concurrency control mechanism is still required to ensure threads receive unique timestamps (atomically incrementing a counter for instance).

Bounds – Memory is finite, and hence any representation of a timestamp in memory, has a maximum number of values that can be represented. When the numeric value of a timestamp exceeds the capacity of the memory, care must be taken to ensure errors do not result.

2.3 Speculative Approaches

Near the end of the 20th Century, as PC architectures begin to feature increasing parallel processing capabilities, demand also increased for techniques which could exploit the newly available parallel redundancy. Advancements in processor design also witnessed the addition of new logical units to modern processors, which could buffer operations for increased execution speed. In this section we present an overview of approaches using speculation about future execution and exploit parallel processing redundancy offered by the hardware to improve the execution time of multi-threaded programs.

2.3.1 Thread-Level Speculation

Developments in automating parallelization have enhanced the role of the compiler in utilising the parallel processing capabilities of the host platform. A common application of such speculative techniques involves the parallelization of *loop* constructs in sequential programs. Rather than having a thread execute each iteration of a loop sequentially, iterations can be assigned to multiple threads and executed in parallel. These compiler-generated threads alleviate the application programmer from the burdens of thread management while permitting the compiler to tailor the degree of thread creation to the resource availability of the host platform. OpenMP [3], and more recently Threading Building Blocks [4], are two examples were such techniques are implemented.

Primary obstacles inherent to such approaches are the complexity of program control flow and the unpredictability of memory access within critical regions. These two factors mean that it is very difficult for any compiler to statically determine whether the threads it wishes to create will act independently, or will interfere with each other's updates (thus introducing race conditions). Thread-Level Speculation (TLS) is an area of research that attempts to mitigate this problem. Generally, TLS incorporates the following features:

- Speculative threads are created on-the-fly by the compiler/run-time system with the aim of speeding up sections of program code (e.g. *for loops*);

- Inconsistencies between the read and writes of speculative threads are detected at run-time and resolved by the TLS technique. Thread execution is managed so that inconsistent execution is discarded and error free execution is maintained;
- Regardless of how thread execution is carried out by the TLS technique, the user observes execution which reflects the code as programmed.

A popular method of detecting inconsistencies requires that the TLS approach utilise the cache protocols of the host platform. Hence many TLS solutions require modifications to be made at the hardware level, specifically with regard to the manipulation of data held in the Cache Hierarchy. Steffan et al [5], for example, proposed modifications to the Write-Back Invalidation-Based Cache Coherence Protocol of the host platform. Once implemented at the hardware level, the application programmer remains unburdened by inconsistencies arising from speculative thread execution. Unfortunately, integrating TLS at the hardware level is more costly and less flexible than a purely software solution.

2.3.2 Speculative Synchronization

Speculative Syncronization [6] was developed by Martinez and Torrellas to provide Thread-Level Speculation (TLS) to explicitly parallel applications. The constructs enhanced with speculative adaptation consisted of memory barriers, locks and flags. These comprised the typical synchronization primitives used consistently with pessimistic concurrency control.

Before Speculative Synchronization, TLS allowed speculative threads to be created from sequential sections of code, which would be executed in parallel with 'safe code' (speculative threads could venture into unsafe regions of code rather than be held up by pessimistic locking constructs, blocking progress). In theory, redundancy in the form of parallel processing resources and memory provided by the host platform could be utilized more effectively to speed up program execution. Speculative Synchronization proposed a similar approach with explicitly parallel sections of code. For instance, where TLS could be applied

to perform parallelization of a *for loop*, Speculative Synchronization could work with program threads and processes.

The contribution of Martinez and Torrellas comprised of proposed hardware and software additions to support speculative execution, namely:

- A Speculative Synchronization Unit (SSU) that in theory would be added to the cache hierarchy of each processor. The SSU would contain space for a cache line holding the data of a variable under speculation while the processor could execute instructions speculatively. If necessary, execution could be rolled-back to the state held in the SSU.
- A set of library primitives which allow speculative execution to be applied to synchronization constructs. These consisted of primitives provided to acquire/release locks and signal an access conflict or a cache overflow.

2.3.3 Speculative Concurrency Control

Speculative Concurrency Control (SCC) was developed by Bestavros [7] and later extended by Haubert et al [8], as a class of Concurrency Control Algorithms designed for application in the area of real-time database management. The novel contribution of SCC was the design of a technique where redundant computations would be executed to cover possible alternative schedules of concurrent activity when a conflict was detected that may invalidate the consistency of the host database.

Rather than increasing concurrent throughput of transactions, SCC focused on the issue of transactions not being able to meet real-time deadlines due to the need to rollback once interference with shared data had been detected. With the SCC technique, consistency conflicts could be detected as soon as they occur (as with a typical pessimistic approach) but before the conflicting transaction has validated. This consistency conflict would then generate a shadow thread re-executing the conflicting transaction.

Figure 2.3 illustrates the idea behind the SCC approach. Transaction T1 and T2 are executing. A conflict is generated when T2 reads data item A after T1 has written to it. Once the conflict is detected, another thread is created which

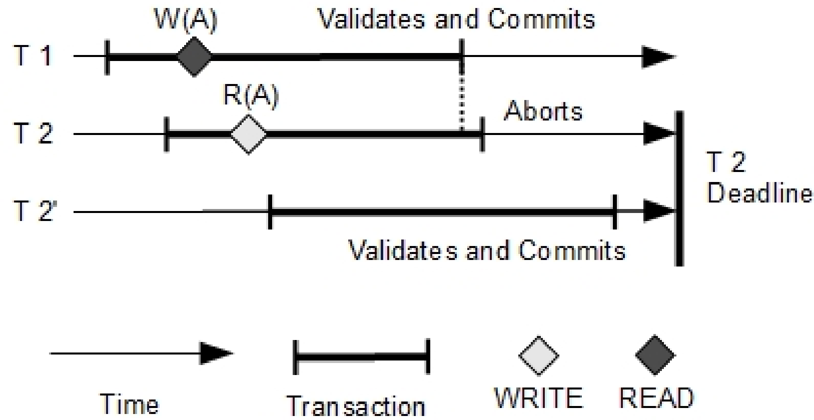

Figure 2.3: Speculative Concurrency Control *A conflict on Data Item A is detected between T1 and T2. This causes a new Speculative Thread T2' to be created which re-executes T2. If T2 has to abort because of T1's access of Data Item A, then hopefully, T2' will still have time to commit before T2's deadline.*

re-executes the transaction T2 (labeled T2'). If T2 validates before T1, and thus commits successfully then the shadow thread T2' is destroyed; if on the other hand T1 validates and commits first then T2' continues executing. Observe that in Figure 2.3, T2' can now validate and commit its transaction before its deadline.

Bestavros provides the framework of an algorithm where the degree of shadow thread creation can be varied; the algorithms presented in their literature comprise:

- A Basic SCC Algorithm which generates potentially many shadow threads and requires the most available redundancy;
- A Two-Shadow SCC Algorithm which generates at most 2 shadow threads where available redundancy is scarce.

2.4 Optimistic Approaches

Issues with pessimistic concurrency control arise in the following areas:

- Locking tends to reduce the performance of multi-threaded applications because blocking threads during synchronisation inhibits their progress. In

addition, blocking requires the operating system save the state of one thread and load another (i.e. context switch).

- Blocking operations may introduce a deadlock (especially if several locks are combined incorrectly), where no thread can make further progress with catastrophic consequences for the application;
- Locking as a general solution is not composable (as mentioned in Section 2.2.1), such that two or more sections of code which implements lock-based critical sections, cannot easily be combined and still guarantee that errors will not be introduced to the system. Consequently, locking solutions on a system-wide scale tend to be ad-hoc and constructed by the system programmer, thus making them more difficult to understand, replicate and debug.

An alternative to the pessimistic approach is to provide a mechanism which allows threads to make changes to shared data first, and then to detect any possible inconsistencies afterwards that arise following concurrent interference. If no such interference has taken place, an 'optimistic' approach allows the modifying thread to carry on with its execution. Should interference occur, this can be detected and changes are undone. The aborting thread may then attempt to repeat its modifications. Because blocking is avoided, the possibility of a deadlock can be eliminated.

2.4.1 Transactions

Concurrency control on general purpose multi-processor platforms can be implemented in a manner which follows the same principles applied in database applications. In database systems, many concurrent clients submit modifications to the state of the database in the form of transactions. Transactions contain the changes a client wishes to make to the state of the database. On each attempt to modify the state of the database, either all changes in the transaction are applied successfully, or no changes are made. A database manager resolves conflicts between the transaction requests received, and the clients are alleviated of the complexity of concurrent programming. To illustrate the process, Figure 2.4

shows a potential time-line with three threads executing updates to the state of a shared database using transactions.

The key to understanding how database systems can manage concurrency (while maintaining predictable and reproducible state progression) lies in the application of the ACID properties:

Atomicity – an action by a transaction must either take effect in its entirety or not take affect at all;

Consistency – the behaviour of transactions should be consistent with the constraints of the data being modified;

Isolation – transactions modify data in isolation from other transactions, hence no transaction should witness the effects of another transaction until the latter has completed successfully;

Durability – the effects of transactions, once committed must be durable and persist.

Primarily, the ACID properties are useful in that they provide a framework which makes it easier to reason about the state of a Database system by constraining design within the confines of intuitive behaviour. If all transaction managers observe the ACID properties, this can provide generality to transactional systems which is lacking in ad-hoc pessimistic approaches.

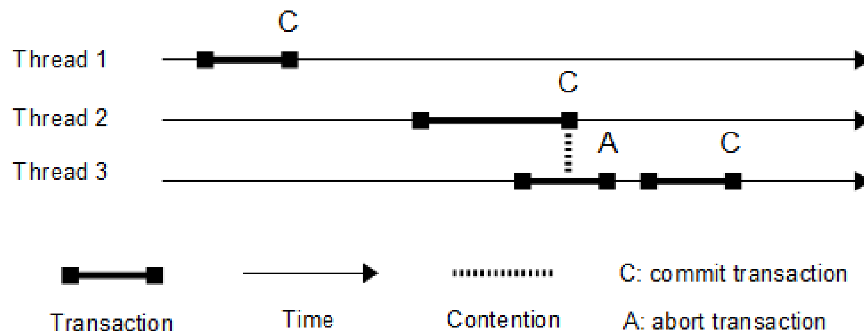

Figure 2.4: Transaction Contention *Thread 1 commits its transaction without interference. Thread 3 on the other hand experiences interference with Thread 2 and must abort its transaction, before retrying and finally committing.*

The properties of atomicity and isolation are especially useful for application developers because they allow multiple operations on shared data to be contained within a single atomic block. As such, operations on shared data can be composed, greatly reducing the difficulty for constructing complex software which requires interacting concurrent access to shared memory.

Inspired by the success of concurrent consistency in distributed databases, parallel programming communities have entertained transactions as an increasingly popular means to implement concurrency control on general purpose platforms. While database systems are concerned with applying changes to disk, however, general purpose platforms have concentrated on the state of (volatile) memory. At the time of writing, research into transactional memory is extensive, yet most techniques fall into three areas of application:

Software Transactional Memory (STM) – generally provides implementations of non-durable transactions (the ACI properties) for threads accessing shared data. The added flexibility that software provides makes STMs a good vehicle for experimentation on aspects of transactional memory design.

Hardware Transactional Memory (HTM) – typically concerned with modifications to cache protocols and architectural design which provide the functionality of transactional memory semantics. HTM design typically excels over STM design when execution speed is an issue. An additional benefit of the HTM approach is that the mechanics of transactional memory are completely oblivious to the user.

Distributed Transactional Memory (DTM) – concerned with providing transactional memory services across geographically distinct hosts using message passing protocols (as opposed to Shared Memory). Recent work by Gramoli et al [9] has shown that a DTM approach can also be suitable for a 'many core platform' where the need to ensure starvation freedom is essential.

In practice, these three areas tend to overlap considerably, especially with respect to HTM and STM (often referred to as Hybrid Transactional Memory).

Unlike locking, transactional memory can provide solutions to the problems of deadlock and concurrent composition. However, transactional memory must address their own issues, namely:

High Contention – One of the main drawbacks of transactional memory comes from the wasted work that is produced when aborted transactions must rollback and retry their execution. In a transactional memory system, many tentative changes may be attempted, especially if contention for shared resources is high and conflicts are frequent.

Starvation – Care must be taken to ensure that one or more transactions do not find themselves perpetually rolling-back. For instance, if there exists a long transaction and many short-lived transactions, the long transaction may find that it can never commit because on each attempt, a short transaction modifies the shared state and invalidates the long transaction.

Addressing the problems of high contention and starvation requires a Contention Management Policy (CMP). Various CMPs are discussed in Section 2.4.5.

2.4.2 Hardware Transactional Memory

Interest in transactional memory is not restricted to software applications. Hardware Transactional Memory (HTM) is another area where the transaction mechanism can be implemented. Augmentation and modification of hardware architectures provides the focus for supporting transactions in HTM. In order to support transactions in hardware (*i*) a thread must be able to execute instructions in isolation, (*ii*) there must be a mechanism to detect conflicts in the consistency of the data (*iii*) and there must be a way for the thread to undo changes or commit results. Several mechanisms featured in hardware can provide these requirements specifically in the areas of Memory Consistency Models, Cache Coherence Protocols and Speculative Execution techniques.

The techniques used by HTM existed before the emergence of HTM as a concurrency control technique, having been previously employed to manage the consistency of data held within processor and memory caches. For example,

Cache Coherence Protocols detect erroneous data that has occurred due to the presence of multiple versions of data held in separate caches. Speculative Execution supports modern processors by allowing them to execution instructions out of program order and roll-back execution to a previous state when inconsistencies arise. Finally, Memory Consistency Models enable the processor to detect and avert consistency errors which may be introduced by executing instructions out of program order.

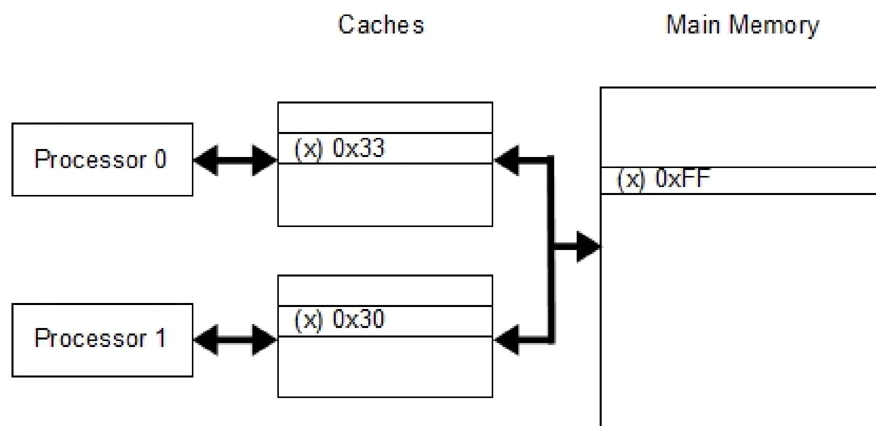

Figure 2.5: Cache Coherence *data item x currently holds three values as multiple versions of the same data location exist in both caches and within main memory. Cache Coherence must ensure that a single value is seen by all threads.*

Figure 2.5 illustrates a scenario where a data item (labelled x) is present in three locations, namely: in two cache structures and in main memory. The Cache Coherence Protocol ensures that data items can be located and updated such that every thread in the system sees the same sequence of modifications to those data items. For example, the history of modifications to data item x from the perspective of all threads can either be $0xff$, $0x33$, $0x30$ or $0xff$, $0x30$, $0x33$, but not both. Current Cache Coherence Protocol technology can achieve consistency either by broadcasting consistency information via a *snooping* type protocol, or by maintaining a directory of which data is currently in a consistent state.

By utilizing the Cache Coherence Protocol a transactional memory system can emulate the concept of an atomic action within the confines of the cache.

The cache allows a particular thread to effectively execute a number of program instructions in isolation from other threads. With the support of a Memory Consistency Model, the Cache Coherence Protocol can be adapted so that when thread executes 'atomic instructions', conflicts can be detected. The Speculative Execution component of the processor then allows a thread to roll-back its execution to its previously 'consistent' state.

2.4.3 Software Transactional Memory

Software Transactional Memory (STM) covers a collection of techniques implemented in software with the aim of providing non-durable transactions (atomicity, consistency and isolation) to manage concurrency control at the application level. Given the flexibility of software over hardware, and the relative ease with which experimentation can be conducted in software solutions, STMs are easier to integrate with programming language support and provide a good method of producing prototype solutions. The overheads associated with STM approaches tends to be much greater than that of HTM, however, and much research has been conducted with the goal of minimizing these overheads.

Numerous STM techniques exist at the time of writing, yet whichever STM is implemented, each technique can often be categorised with respect to the treatment of shared data. Approaches to shared data can be considered under the following three categories: (i) the granularity of shared data; (ii) the overhead of updating shared data and (iii) the synchronization mechanism used to regulate access to shared data.

If we consider the first, the granularity of shared data, STMs can be further divided into two approaches:

(Atomic) Object Based STMs – (such as the DSTM2 benchmark suite for instance [10]), represent shared data as objects from which concurrent data structures can be composed. Object Based implementations have the benefit that they can be integrated with Object Orientated languages relatively easily;

Word Based STMs – (such as TinySTM [11]) access and modify shared data at the granularity of memory-words, and as such are considered somewhat 'lower-level' than their object based counterparts (TinySTM for instance, provides functions for reading and writing memory words within transactional blocks of code).

STM implementations vary with regard to the overheads required for shared data access, with two modes of operation available:

Deferred Update – Threads which operate in STM implementations using the *deferred update* model, modify copies of shared objects during transaction execution. Copies are stored in caches private to a particular thread, and often separated into *read sets* and *write sets.* When the thread reaches the end of its transaction, it must replace the objects it has modified with the contents of its read/write sets.

Direct Update – STM software which uses a *direct-update* model can reduce the overhead of creating per-thread copies of atomic objects by restricting ownership of an object to a single thread during its transaction (although a single copy of shared objects are still required even in a *direct-update* scheme). As a thread executes its transaction, it attempts to acquire ownership of each shared object it wishes to modify; if another thread owns an object already then a contention management policy is consulted and either the owning thread or the acquiring thread must abort.

We may also examine STM implementations with respect to the particular synchronization technique employed when accessing shared data. With some important exceptions (i.e. [12, 13]), most STM implementations can be categorised by the following designs:

Obstruction Free – in an obstruction free STM design, threads are able to make 'progress' with their transactions (whether they commit or abort), if isolation from the activity of other threads is maintained. The obstruction free library of DSTM2 for instance [10], grants ownership of shared objects by the successful execution of the *Compare-And-Swap* synchronization primitive.

Lock Based – in a lock-based STM design short critical sections guarded by conventional locks are used to allow ownership of shared objects. To avoid the possibility of a deadlock, threads also use time-outs limiting the maximum number of attempts to acquire a lock before aborting their transactions.

Each approach to shared data access has its own implications and benefits. With respect to *direct* versus *deferred* updates, while the latter requires more memory for the read/write sets, *deferred* update is easier to implement than a *direct* update approach. For example, 'cache bouncing' is a phenomenon where excessive overhead is caused by moving the contents of cache lines around a multi-core cache hierarchy. A *direct* update scheme is more suspectible to cache bouncing, and so most *direct* update schemes support *visible* and *invisible* reads.

A prominent feature of all STMs is the Contention Management Policy (CMP). When there are a low number of threads, and transaction execution is sparse, access conflicts between transactions are rare and the thread execution is not impaired by the need to abort and retry transactions. Conversely, when the level of contention increases and conflicts are frequent, a CMP is indispensable to ensure that all threads can commit their transactions as expediently as possible.

2.4.4 Coordinating Transactions

Transactional memory semantics alone do not consider the effects of ordering transaction execution. Rather, much existing research has focused on the production of transactional schedules which are serializable. When transactions were originally implemented in distributed database applications, transaction ordering was less of a concern because transactions tend to be independent operations. In transactional memory, however, transactions tend to be tightly coupled and coordination becomes more prominent an issue. (For instance, *producer* and *consumer* transactions may be modifying a shared buffer; a *producer* transaction must precede a *consumer* transaction if the buffer is empty.)

In this section two mechanisms are covered, which provide the application programmer with the ability to coordinate transactions, namely: conditional primitives and transactional nesting.

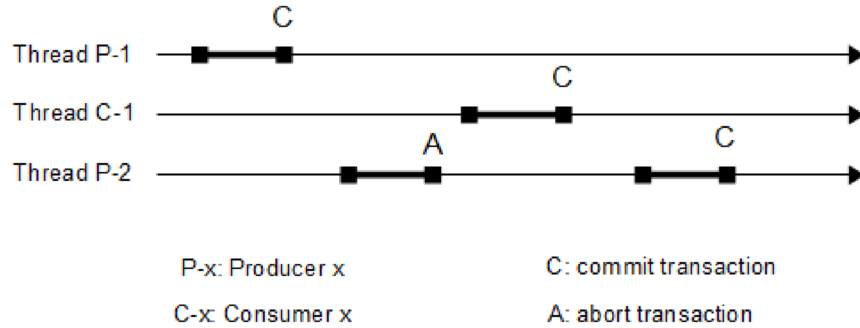

Figure 2.6: Transaction Coordination *Thread P-2 aborts its transaction explicitly in this scenario because the shared buffer has reached capacity after Thread P-1's transaction. Once Thread C-1 removes an item from the buffer, Thread P-2 can retry and commit.*

Conditional Primitives A number of primitives have been developed to provide the programmer who values transaction coordination with greater flexibility. These include the ability to explicitly abort a transaction and the *retry* and *orElse* statements first described by Harris et al [14]:

1. Programming Languages which support exception handling allow a transaction to be terminated prematurely and explicitly by the programmer. A transaction may throw an exception from within itself, to be caught back in the application. Some implementations of transactional memory also provide the keyword *abort* so that a transaction can be aborted and rolled-back explicitly.

2. Harris et al [14] first introduced the *retry* statement into transactional semantics. If a transaction reaches a *retry* statement, the transaction is allowed to abort for whatever arbitrary reason the programmer specifies, and another attempt is then made to execute the transaction.

3. The *orElse* statement (also courtesy of Harris et al [14]) provides conditional coordination between two transactions. Using *orElse*, an expression like (T_1 *orElse* T_2) means that if transaction T_1 commits then T_2 will not be executed, but if T_1 aborts then T_2 is executed. Now if T_2 commits, the

orElse statement is completed, but if T_2 also aborts, the *orElse* statement is re-executed.

With coordination primitives, sophisticated orderings of transaction execution can be expressed at the cost of requiring application programmers to explicitly program transaction coordination themselves. Because these decisions tend to be specific to each particular application, this may be a necessity. However, sophisticated coordination between parallel components will cause difficulty for application programmers, raising the risk of introducing consistency errors. (For example, use of the *retry* or *orElse* may result in a live-lock).

Nested Transactions Transactions that execute wholly within the execution of another (parent) transaction are called nested transactions. By accommodating nested transactions, transactional memory can provide the following benefits:

Composition – Analogous to the use of functions in most programming languages, allowing one transaction to be executed from inside another transaction is desirable because it allows transactions to be composed, just as an atomic block allows concurrent statements to be composed.

Coordination – Nested transactions provide a degree of transaction coordination which is absent from the general purpose transactional semantics of commit or abort.

Efficiency – Under some approaches to nested transactions, rolling-back the effects of a nested transaction does not automatically abort its parent transaction. This is beneficial if the nested transaction can retry and commit because the parent transaction does not have to undo its changes and execute repeatedly.

Although conceptually intuitive, in practice the management of nested transactions requires a substantial degree of complexity to implement, and a number of approaches which accommodate nested transactions have thus far been developed:

Flattened Transactions – Flattened transactions are the easiest to implement; the nested transactions are simply incorporated into the parent transaction and the result is executed as a single transaction. However, they offer only 'syntactic sugar' to the user, and the parent transaction aborts along with any child transactions.

Closed Transactions – If a child transaction aborts then this does not cause its parent transaction to abort. If a child transaction commits then its changes are immediately observable to the parent transaction, but not to any other transaction. The behaviour of closed transactions and flattened transactions is indistinguishable when the transactions commit.

Open Transactions – When an open transaction commits, its changes are immediately observable to any other transaction in the system. This is true regardless of the state of the parent transaction and whether the parent transaction commits or aborts. Thus open transactions allow nested sections of transactional code to execute unrelated tasks (such as memory management), which the application programmer does not want to be undone.

A number of techniques have been developed recently to allow greater parallelism when executing nested transactions. For example, both HParSTM [15] and [16], describe schemes to allow the parallel execution of nested transactions.

It should be noted that while not all applications will necessarily benefit from or require the use of nested transactions, those that do tend to feature in the domain of transactional memory (as opposed to database transactions). Where processes or threads are executing transactions in memory, there tends to be a greater need for coordination around accesses of shared data. Object orientated programming is no exception to this pattern, as shared data tends to be grouped logically into shared objects.

2.4.5 STM Contention Management

A prominent source of inefficiency in transactional memory comes from the cost of transactions having to abort and rollback their actions because of concurrent

interference. As more transactions execute, or transactions execute for longer periods of time, the contention for shared resources increases, causing a greater frequency of aborted transactions and reducing the overall progress of the application.

To mitigate the degree of wasted time caused by roll-backs, much research has been developed by the concurrency community, principally in STM, to provide contention management and reduce the occurrences of aborted transactions. Approaches to contention management at the time of writing can be categorised into the following three groups:

Exception-Based – The programmer dictates how an aborting transaction should be treated. Exception handling provides the mechanism for such approaches in programming languages that support the 'throwing' and 'catching' of exceptions.

Wait-Based – Contention management typically resolves conflicts between two transactions by causing one transaction to back-off and wait for a period of time before retrying. The criteria for deciding the outcome of the conflict may vary depending on the contention management policy used.

Serializing – Contention management typically resolves conflicts between conflicting transactions by rescheduling the execution of the aborted transaction (usually to execute after the 'winning' transaction). Unlike wait-based approaches, serializing contention management often requires control over the allocation of transactions to threads so that conflicting transactions can be scheduled to execute by a single thread (thus avoiding the conflict through serial execution).

With the exception-based approach, the main burden still falls on the application programmer to handle aborts and therefore this requires more work from the programmer. The benefit for the programmer comes from the increased flexibility. When one considers real time deadlines for instance, a programmer may require application specific actions to take place when a transaction has aborted and failed to achieve a specific deadline (or perhaps the transaction must be abandoned completely).

Wait-based contention management, incorporates some of the earliest approaches to contention management in STMs. Numerous policies exist which resolve contention by emphasizing various properties of conflicting transactions. At the time of writing, the most prominent are:

Polite – When any two transactions conflict, the CMP decides which should abort and the aborting transaction waits for an increasing back-off time before re-executing. Theoretically, waiting should allow time for the failed transaction to avoid a conflict on its next attempt.

Karma – The number of shared objects accessed by a transaction is maintained and the CMP uses this number to assign a priority to conflicting transactions. Should two transactions conflict, the CMP aborts the transaction with the lower priority. The goal of the CMP is to abort those transactions which will incur the least cost of rolling back. The difference between the competing priority values is used to compute the number of times an aborting transaction will retry before aborting the transaction with the higher priority.

Polka – This CMP is a combination of the *Polite* and *Karma* policies. The *Polka* CMP uses an exponential back-off with aborted transactions.

Timestamp – The CMP will abort transactions based on the time when they began executing. Should two transactions conflict, the newest transaction is aborted.

Heber et al [17] identify a flaw in time-based contention management. Specifically, if two transactions conflict and the aborting transaction re-executes too soon, it will again have to abort because the conflict will arise again before the winning transaction completes. Even if the conflict is avoided when the aborting transaction re-executes, it is possible that the aborting transaction had to wait an excessive amount of time before it could finally commit. Heber argues that serializing contention management avoids this drawback, endemic in time-based approaches. This is because the aborting transaction can resume execution immediately after the conflicting transaction terminates (presumabley resulting in

better performance). In Figure 2.7 three scenarios are depicted which illustrate the limitations of the time-based policies, and how these can be resolved via a serializing approach.

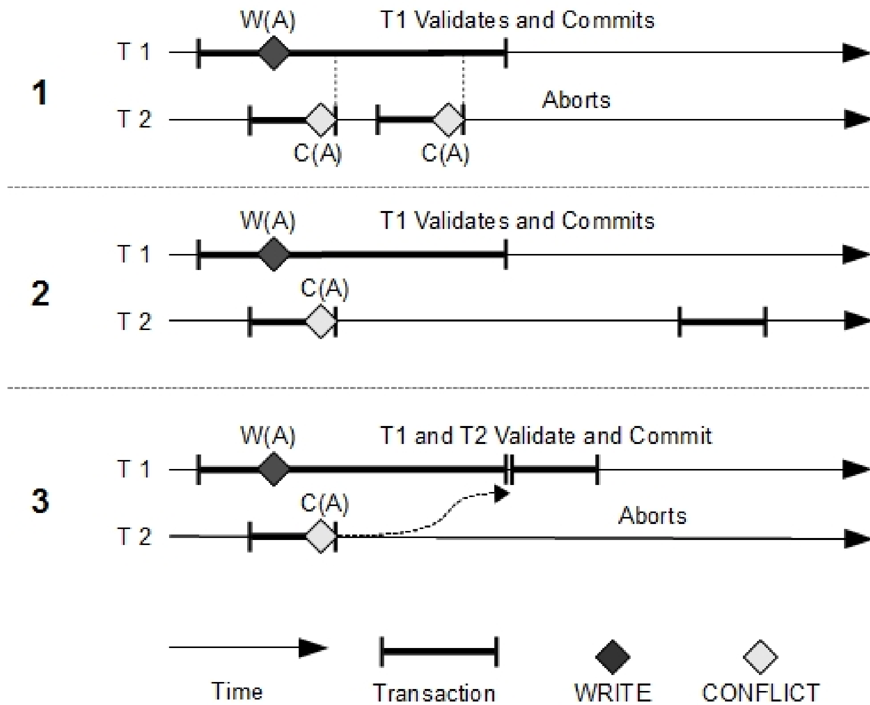

Figure 2.7: Contention Managers *Three scenarios are shown to demonstrate a flaw with time-based contention management (1 and 2) and how this can be addressed using a serializing approach (3). In scenario 1, two threads (T1 and T2) have conflicted and a time-based CM causes T2 to abort and retry, however the time period is too short and so T2 aborts again because T1 is still executing. In scenario 2 the same conflict has occurred but this time the CM makes T2 wait too long before re-executing. In scenario 3, a serializing approach is used which moves the transaction of T2 to a queue belonging to T1. Now once T1 has completed its transaction, it executes the transaction of T2. Meanwhile, T2 is now free to execute other transactions.*

Serializing contention management is a relatively new technique, but numerous approaches exist at the time of writing. A common theme with serializing contention managers is to resolve a conflict between two transactions by rescheduling the aborted transaction such that it executes shortly after the successful transac-

tion. For instance, CAR-STM [18] serializes transaction execution by supplying each thread with a transaction queue. When a transaction aborts, the contention manager reschedules the aborted transaction by moving it to the queue of the successful transaction. Steal on Abort [19] implements a similar idea complete with transaction queues, but adds extra functionality by allowing threads to steal the aborted transactions of other threads to balance their workloads.

Yoo and Lee introduced a contention manager called ATS [20] which demonstrated improvements in performance with workloads exhibiting little parallelism. ATS allowed the CM to adapt to varying levels of contention amongst threads executing transactions. Their approach required each thread to monitor the level of contention (called the *contention intensity*), and if this breached a certain threshold, transactions are serialized into a single queue such that they can commit serially and subsequently reduce the level of contention.

Heber et al [17] followed up on the work of adaptive serializing contention management by analyzing how different workload characteristics are hindered or enhanced by serializing transactions, and provided algorithms which could allow the degree of serialization to be modified dynamically (algorithms were more sophisticated than the approach provided by Yoo and Lee).

One issue which serialisation raises is the cost of relocating transactions to be executed by another processor. To outperform a wait-based technique, one must assume that a smaller overhead is required to move a transaction to execute on another processor than the overheads of the time-based approaches. It would be interesting to see if this assumption would hold on a many core platform.

In summary, early approaches to STM design allowed for greater flexibility by leaving the resolution of conflicting transactions in the hands of the user. The difficulty of resolving concurrent conflicts prompted innovations in time-based contention management. The design of contention management policies allowed various techniques to be implemented and compared. Finally, the inherent inefficiency of time-based techniques led to developments in serializing contention management. Serialising contention management policies control and manipulate the allocation of both transactions to threads and threads to processors.

2.5 Related Work

In this section several implementations are described which address contention management in STM design. To evaluate these approaches in the context of this thesis, the following criteria is used:

Environment – What requirements are made of the executing platform and can the technique exploit the parallel processing resources of the host?

Conflicts – What kind of conflicts does the technique address and is there some facility to explore the coordination of transactions? For example, how does the technique handle nested transactions?

Adaptation – Does the technique provide some facility to adapt its behaviour in response to changes in workload?

In this thesis, we are interested in how present techniques perform in an increasingly parallel environment. *Environment* and *adaptability* are significant with respect to how the technique may be affected by future developments in parallel architectures. We are also interested in addressing a wider range of conflicts, that the application developer might consider important for program progression. Hence the flexibility of the approach when dealing with various types of conflicts is considered.

2.5.1 Serialising Contention Management

Some approaches to contention management involve rescheduling transactions to avoid repeated conflicts. Such techniques can be described as serializing contention managers. Ansari et al [19] produced one such approach, called *Steal on Abort*, where each thread maintains several work queues containing transaction requests from the application. When a conflict between executing transactions occurs, the thread of the committing transaction 'steals' the aborted transaction from its current work queue and places it in its own. By rescheduling the aborting transaction to the queue of the successful transaction, the likelihood of a repeat conflict is reduced given that the aborting transaction will re-execute after the

successful transaction (essentially, the two transactions have been compulsorily serialized).

Environment – *Steal on Abort* requires concurrent data structures (e.g. work queues) to hold transactions. Threads must be divided into application threads (which issue transactions) and transactional threads (which execute transactions). Each transactional thread is then allocated a work queue such that only thread i can execute a transaction on queue i. The *Steal on Abort* technique requires that transactional threads supply the results of each transaction asynchronously, to the issuing application thread.

Conflicts – As with many contention managers, *Steal on Abort* addresses concurrency conflicts. Ansari et al assert that a benefit of their system is that no application specific knowledge is required. As such, transaction coordination and nested transactions are not covered. Steal on Abort also provides a number of 'stealing strategies'. For example, Steal-Tail attempts to separate conflicting transactions which have executed close together, while Steal-Head aims to reduce the occurrence of cache misses.

Adaptation – *Steal on Abort* implements 'Work Stealing'; when transactional threads run out of transactions, they may steal transactions from the work queue of another transactional thread. This allows the contention manager to adapt by balancing the workloads. In addition, the number of transactional threads can be increased under high workloads to produce a better distribution of work.

A potential issue with *Steal on Abort* is that transaction coordination and nesting are not addressed. For example, if a transaction cannot commit because of a coordination (e.g. semantic) conflict then reordering the transaction does not guarantee that the semantic condition has been addressed when the transaction re-executes (even if the transaction commits, if the semantic condition remains unfulfilled then the transaction will likely need to re-execute in future). With long/nested transactions, this issue could greatly hinder the progression of the program, as the aborted transaction may have to be rescheduled numerous times.

How *Steal on Abort* would fair on a highly parallel system is interesting because of the synchrony required in distributing work among the transactional threads. For example, substantially increasing the number of transactional threads may itself cause much contention (which requires synchronization) given that the degree of work-stealing will increase. One would also expect that where there is a lot of work stealing occurring, the benefits of serialization would be reduced; transactions will be moved to queues where they will execute concurrently once again and a conflict may be repeated.

2.5.2 *Shrink* and Predictive Scheduling

Techniques such as *Steal On Abort*, which reorder and serialize conflicting transactions, attempt to resolve conflicts once they have arisen. An alternative is to try and prevent conflicts ever occurring by rescheduling transactions that are likely to conflict before they execute. Dragojevic et al provide one such approach called *Shrink* [21]. The technique used by *Shrink* involves *prediction* and *serialization*, specifically:

- *Shrink* generates prospective read and write sets for each transaction before it executes. The likelihood of a conflict is estimated by comparing the contents of the prospective read/write sets with the data being modified by any currently executing transactions. Bloom Filters [22] are used to determine the likelihood of a conflict quickly.
- If a conflict is deemed likely to occur then threads try to acquire a global lock. Once acquired, the lock is held by a thread until its own transaction commits or aborts. Using a global lock in effect allows a transaction to execute in serial isolation.

Dragojevic contends that serialization of transactions should only occur when the contention among shared access is sufficiently high to justify the overhead of rescheduling transactions. The measure of contention is referred to as the *serialization affinity* and this is used to decide whether *Shrink*'s serialization mechanism (i.e. the global lock) should be engaged.

Environment – As noted, the approach used by *Shrink* for serialization comprises a global lock rather than requiring one thread move its transaction to the work queue of another thread.

Conflicts – *Shrink* addresses concurrent conflicts and proposes a scheme that records previous memory accesses, to help predict future memory accesses with the support of a probabilistic construct called a Bloom Filter.

Adaptation – A key contribution of *Shrink* is the computation of a *serialisation affinity*, which allows the contention management policy to decide at runtime whether serialisation should be applied.

The *Shrink* mechanism as described seems to suggest that the accuracy of predictions depends to some extent on the number of items in the read/write sets. The predictive power of *Shrink* should be optimum in the case of short/repetitive transactions. Long and nested transactions are likely to introduce unpredictable data accesses, given that different nested transactions may execute depending on different states encountered by parent transactions. As a result, *Shrink* may perform poorly with such transactions.

2.5.3 *TLSTM*

TLSTM is an adaptation to the SwissTM software transactional memory manager [23], provided by Barreto et al [24]. The aim of TLSTM is to combine apsects of both Thread Level Speculation (TLS) and Transactional Memory (TM). TLSTM divides thread execution into speculative tasks of execution (which may run in parallel). As with existing TLS techniques, speculative tasks are provided automatically by the compiler. Tasks may, however, incorporate both sequential code and concurrent transactional code; specifically, statements executed within atomic blocks.

It is anticipated that this task based model of speculation will provide better performance on architectures featuring an abundance of parallel processing resources. Barreto et al provide benchmarked results which demonstrate increased transactional throughput, achieving up to 48% speed up on SwissTM alone.

Environment – SwissTM is a word-based, lock-based TM. Extending SwissTM with TLS retains these features. Rather than submitting transactions to be executed by a thread-pool, transactions are executed directly through programmer macros. However, it is assumed that the compiler can decompose the execution of the thread into tasks. Tasks are then executed in parallel among available processing resources (although the mechanism for distributing tasks is beyond the scope of the paper).

Conflicts – The programming model proposed in TLSTM must provide opacity of user transactions (i.e. correctness). This includes resolving both concurrency conflicts (inter-thread conflicts) and speculative conflicts (intra-thread conflicts). Intra-thread conflicts consist of Write-After-Read (WAR) or Write-After-Write (WAW) conflicts, arising from the execution of task out of program order. Hence, accesses made by tasks belonging to the same thread must behave as if they were executed sequentially. For simplicity, transactions are assumed to be flat, although the authors suggest that nested transactions can be incorporated in the model.

Adaptability – The degree to which thread execution may be decomposed into parallel tasks (*speculative depth*) can be configured by the application. The authors cite a number of approaches for deciding the value of the *speculative depth*, specific to TLS approaches, including [25] and [26].

One assumes that the criteria for deciding the granularity to which threads as decomposed should be informed by both the parallel processing resources available and the size of the transactions that will be split among tasks. Decomposing a task on a single processor architecture would seem to be bad for performance. Very short, read-only transactions would also seem unlikely candidates for decomposition given that the overhead of decomposition and synchronisation (in order to commit each task), would almost certainly outweigh the gains in throughput.

The TLSTM approach is exciting from the perspective of highly parallel architectures and programming scenarios featuring long, complex transactions. The TLSTM model raises the exciting possibility of executing tasks speculatively ahead of time as dictated by parallel processing availability rather than strict

program order. In this respect, TLSTM may be particularly beneficial when one considers transaction coordination in order to reduce the possibility of semantic conflicts. For example, if the TLSTM system could anticipate semantic conflicts then the process of dissecting threads into tasks could be exploited to find a task execution which reduces the possibility of coordination/semantic conflicts.

2.5.4 Universal Constructions

An important contribution with respect to building wait-free algorithms was provided by Herlihy [1] who described a hierarchy of consensus numbers relating to various concurrency primitives (atomic registers, fetch-and-increment, compare-and-swap, etc.). Herlihy then defined the concept of a Universal Construction [27], which is fundamentally a procedure to transform any deterministic sequential object into a wait-free linearizable object. Such an object can be accessed and updated concurrently by any number of threads.

Figure 2.8 illustrates the general idea behind the creation of a wait-free linearizable object via a Universal Construction. The approach is similar in concept to a wait-free mechanism, which allows multiple threads to append items to a shared list. In this case the shared list is simply a series of instructions or invocations of methods on some data structure. When executed in an identical sequence, the Construction ensures that the data structure will hold the same state from the perspective of every thread in the system.

To ensure that every accessing thread has a consistent view of the object, an n-thread consensus protocol is used. The protocol ensures that each thread perceives the same order of method invocations that have been added to the shared list. To ensure that the Universal Construction is wait-free (i.e. to ensure that each accessing thread can append to the list in a finite number of steps), threads help each other to resolve failed attempts to add their invocations to the list.

Recently, there have been a number of approaches which seek to extend the Universal Construction to the execution of transactions. For instance, Wamhoff and Fetzer described a Universal Construction that could deal with non-terminating transactions by restricting transaction execution times [28]. Around the same

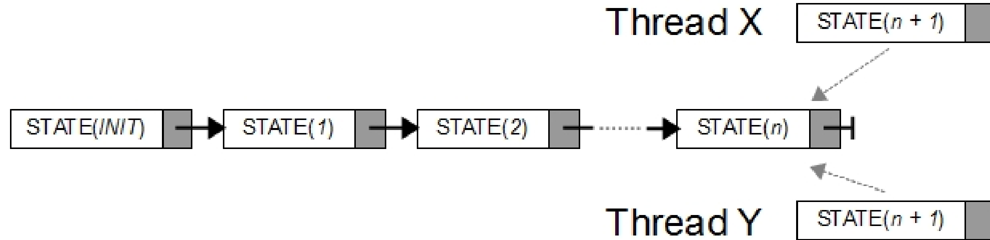

Figure 2.8: A Universal Construction *Thread X and Y attempt to append their respective method invocations to the end of a shared log which is shown in the form of a linked list of nodes containing states. The log maintains a progression of invocations, starting with the initial state of the object, effectively providing instructions on how to 'build' the most current state of the object.*

time, Chuong et al described a Universal Construction [29] for creating shared objects that is 'transaction friendly', such that failed attempts to modify the shared object can be undone like a transaction.

At the time of writing, one of the most recent publications in this area is provided by Crain et al [30]. Crain argues that STM systems do not sufficiently free the programmer from the management of synchronisation related issues and that transactions should appear to execute exactly once from the application layer point of view (there should be no observed abort or commit). The aim of Crain's Universal Construction is to ensure that every transaction eventually commits without the application being aware of aborted transactions.

Crain describes a concurrent environment comprised of m processors where each processor is responsible for a subset of processes, such that:

- The aim of a processor is to ensure the progress of each process for which it bears responsibility.
- When a processor encounters a transaction that aborts, it seeks help from another processor to commit that transaction.

The approach considers only atomic registers, which are referred to as shared objects. A subset of these objects are called transactional objects (*t-objects*) which are modifiable within the execution of a transaction. Crain et al hold

the state of the *t-objects* in a list, in a structure similar to the list of invocations described in Herlihy's original Universal Construction. Each element in the list identifies a committed transaction and the state of all accessed *t-objects* are recorded, prior to and after the successful transaction execution.

Transactions possess timestamps, which are issued by a logical clock. The logical clock is incremented via *fetch-and-increment* operations. An array is used to log the progress of each processor (so that multiple processes can help one another to retry aborted transactions). In addition, transactions may be shared between processors and speculatively executed, although the sharing mechanism is intended to allow progress of individual processes by reducing the possibility of concurrent conflicts.

Environment – Crain's approach assumes that there is a fixed number of processors and that each processor assumes responsibility for executing a subset of transactions. Similarly with *Steal on Abort* [19], Crain assumes that processors can help one another by appropriating the transactions of another processor when a transaction is aborted (therefore, control of thread scheduling is expected).

Conflicts – Crain's Universal Construction does not address conflicts in detail, transaction coordination or nested transactions. Crain assumes that transactions abort because of concurrency conflicts between shared data and all transactions are expected to terminate successfully at some point in time. Eliminating the observation of abort/retry is entirely premised on this, and it is assumed that rescheduling and re-executing any transaction will allow it to commit.

Adaptation – Much like *Steal on Abort*, Crain's approach relies on a pool of transactional threads, to which transactions are allocated. Transactional threads can execute the transactions of other transactional threads on the occurrence of a transaction aborting. This provides the Universal Construction some measure of adaptation in the presence of concurrent conflicts.

2.6 Summary and Thesis Contribution

In this chapter a number of approaches were explored for the provision of concurrency control. A description of synchronization primitives and their the relevance when solving the problem of consensus was introduced. Pessimistic approaches were then discussed, principally in the application to locking. A brief evolution of techniques were mentioned, which use speculative operations and exploit redundancy in the processing capacity of the host platform. Speculation was followed by optimistic methods, focusing particularly on transactions. Both Hardware and Software Transactional Memory (STM) were discussed, with greater emphasis on STM. Transaction coordination (e.g. primitives and nesting) and contention management policies were described in the context of STM. Finally, in the Related Work section, a number of recent approaches to contention management for STM were described.

In summary, lessons and key points consisted of:

- The benefit of locking is that it can be simple and intuitive for trivial problems, and locking features heavily in a great deal of legacy code (typically in the area of operating systems), where high performance is considered a priority.
- Unfortunately, solutions which use locking cannot easily be composed without the risk of introducing errors into the application, especially with respect to the possibility of introducing a deadlock or livelock. Furthermore, even where composition is achieved, the solution tends to be somewhat bespoke in nature and cannot be applied generally.
- As computing resources have increased in terms of both memory and parallel processing power, optimistic techniques have become more attractive as a means of solving the complexities inherent in locking. In particular, transactional memory offers application programmers an intuitive interface which allows the composition of concurrent code in a generalised way and without the risk of introducing deadlocks.

- Unfortunately, transactional memory does not perform well when contention for shared resources is high and coordination of access is particularly important (the number of aborted transactions increases and application progression suffers). Consequentially, there has been much focus on techniques which can provide effective contention management policies to mitigate the degradation that occurs as a result of high contention.

Addressing the implications of interference and rollback, numerous contention management policies (CMP) have been proposed, which at the time of writing, tend to fall into three categories:

1. Exception handling, where the programmer is responsible for handling contention management explicitly.
2. Wait-based CMPs, where aborted transactions are delayed or alter priority, to allow other transactions to complete.
3. Serialising CMPs, which reschedule the execution of transactions to minimise the possibility of interference.

Generally speaking, the development of wait-based CMPs incurs a loss of flexibility for the application developer present in the exception handling approach. The benefit is that he/she is no longer burdened with the need to explicitly handle aborted transactions. Serialising CMPs attempt to improve on the performance of wait-based CMPs. The work in serialising CMPs has renewed interest in Universal Constructions, with several approaches recently published, which provide the construction of a set consisting of transaction executing threads.

2.6.1 Contribution

If we consider the role of contention management as simply to provide the greatest degree of throughput by reducing the number of read/write (concurrent) conflicts then we do not have to concern ourselves with the coordination of transaction execution. Both the wait-based and serialising contention managers seem to begin from this assumption. This does not guarantee the best progression of

the program as a whole, however. For example, multiple threads which access a shared data structure often need to coordinate their accesses to ensure the data structure is not empty, or filled to capacity before they conduct insertion or deletion operations on that structure. If accesses to the data structure are made within the execution of transactions, then the repeated execution of transactions (because the state of the data structure is such that the thread cannot progress) will incur needless future conflicts.

Although a contention manager may increase the throughput of transactions which successfully commit, one may lose sight of the fundamental reason for writing multi-threaded applications to begin with; namely, to provide a means to execute programs more expediently. If it is the case that in systems programming (where STM is implemented), the coordination of multi-threaded execution is paramount for the progression of the program as a whole, then semantic issues should be an integral feature of a concurrency control solution. With this consideration in mind, we shall provide the following contributions:

1. Speculative techniques have shown that parallel processing redundancy can be exploited by the execution of speculative threads; speculation has the potential to discover more efficient paths of execution within multi-threaded programs. We begin with this principle and tackle the progression of multi-threaded programs with a speculative exploration of state space. Therefore, the first contribution of this thesis is a model of computation, called *Many Systems*, where a state space of concurrent execution is generated and managed within finite resources.

2. Transactional memory is an attractive approach to concurrency control, given its ability to compose concurrent execution and the highly speculative nature of transaction execution. Therefore we provide an implementation of a Universal Construction to provide contention management for STM, based on the concepts of the Many Systems model. Unlike the approaches cited in the Related Work Section, the Universal Construction shall be implemented such that both concurrent conflicts and semantic conflicts can be resolved to promote the progression of multi-threaded programs.

3. Given that transaction coordination and exploration is our emphasis, unlike the techniques in the Related Work, we extend the implementation to allow for the execution of nested transactions. Nested transaction execution is conducted in a manner which is novel, yet scalable with respect to the platform resources and thus beneficial to the progression of nested transactions.

4. Finally, we provide performance results which demonstrate the applicability of our approach in comparison with an existing contention management policy using established benchmarks. We show that: (i) without the presence of semantic conflicts, our approach performs as well as an existing policy; (ii) when semantic conflicts are introduced, our approach outperforms the existing policy (in terms of transaction throughput) and (iii) in the presence of semantic conflicts, techniques designed to reduce conflicts (data structures such as hash tables) are of little benefit.

To support the main contributions we endeavour to fulfil the following objectives:

Versatility – In describing the Many Systems model, we identify properties and suggest benefits which are not restricted to transactional memory, but may extend to concurrency control in general.

Efficiency – We use wait-free and lock-free algorithms wherever possible, while taking account of issues which are detrimental to performance on today's architectures (cache-bouncing, context switching, etc.).

General Purpose – For the scope of this thesis, we implement our prototype in software only and make no scheduling demands, bespoke to particular operating systems. Hence we provide an implementation that is general purpose and an interface consistent with existing transactional memory libraries (unlike *Steal on Abort* or Crain's Universal Construction for example, our implementation does not require a thread pool where transactions are allocated to threads to be executed asynchronously with application threads).

Chapter 3

The Many Systems Model

In this chapter we present the principles of a model we call *Many Systems* (*Many Systems* is a reference to the Many Worlds Implementation of Quantum Physics by Everett et al [31] which has served as an inspiration for the model). To encourage clarity, the model is presented in an abstract process language called Communicating Sequential Processes [32] (hereafter CSP). To help those not acquainted with CSP syntax, the model uses a minimum of CSP features. In addition, an appendix is provided, explaining selected constructs and terminology in greater detail.

We begin the model by defining the main components, which are used to generate a state space of concurrent activity. The *Dining Philosophers* problem [33] is used as a running example throughout the chapter to show how the model may be applied to an established concurrency control problem. We then reason about the state space, and a new Universal Construction is proposed, based on Many Systems principles. Finally, the chapter closes with several properties of the Many Systems model.

3.1 Overview

In the *Dining Philosophers* problem there are forks interspersed equally amongst a number of seated philosophers. The number of philosophers is equal to the number of forks, and each philosopher is required to hold two forks in order to eat. Sometimes a philosopher may think for a while before attempting to eat,

but as there are not enough forks for all philosophers to eat simultaneously, a philosopher must occasionally wait for a fork to become available. With respect to concurrency control, this simple scenario describes the fundamentals of several important problems, which may occur in a multi-threaded program with limited shared resources, specifically:

- If philosophers proceed without due care, deadlock will ensue if all philosophers acquire a single fork at the same time.
- In addition, the scenario may lead to 'starvation' if some philosophers continuously utilize forks while other philosophers persistently cannot acquire forks.
- Starvation may lead to a livelock if philosophers continuously utilize forks at equal time intervals, thus preventing some philosophers from ever making progress (i.e. dining).

Concurrency control is required to ensure that those events which philosophers observe (e.g. picking up and putting down forks) are scheduled according to a ordering which allows all philosophers to eat. In essence, reads and writes that represent access to shared state must be ordered appropriately. In the *Dining Philosophers* example, reads assess the availability of a fork and writes change a fork's availability. If P is used to denote a dining philosopher then the ordering of their reads and writes (to reflect an ability to eat) would be:

$$READ_{P,FORK1}\ (\textit{available}); WRITE_{P,FORK1}\ (\textit{pick up});$$
$$READ_{P,FORK2}\ (\textit{available}); WRITE_{P,FORK2}\ (\textit{pick up});$$
$$WRITE_{P,FORK1}\ (\textit{put down}); WRITE_{P,FORK2}\ (\textit{put down});$$

With multiple philosophers, (and hence multiple read/write operations), the priority of concurrency control is to interleave such orderings as to maximise the progress for each philosopher while averting the possibility of deadlock, livelock and starvation.

Using the *Dining Philosophers* example we present a sketch of the 'Many Systems' approach in a scenario involving only two philosophers. This does not

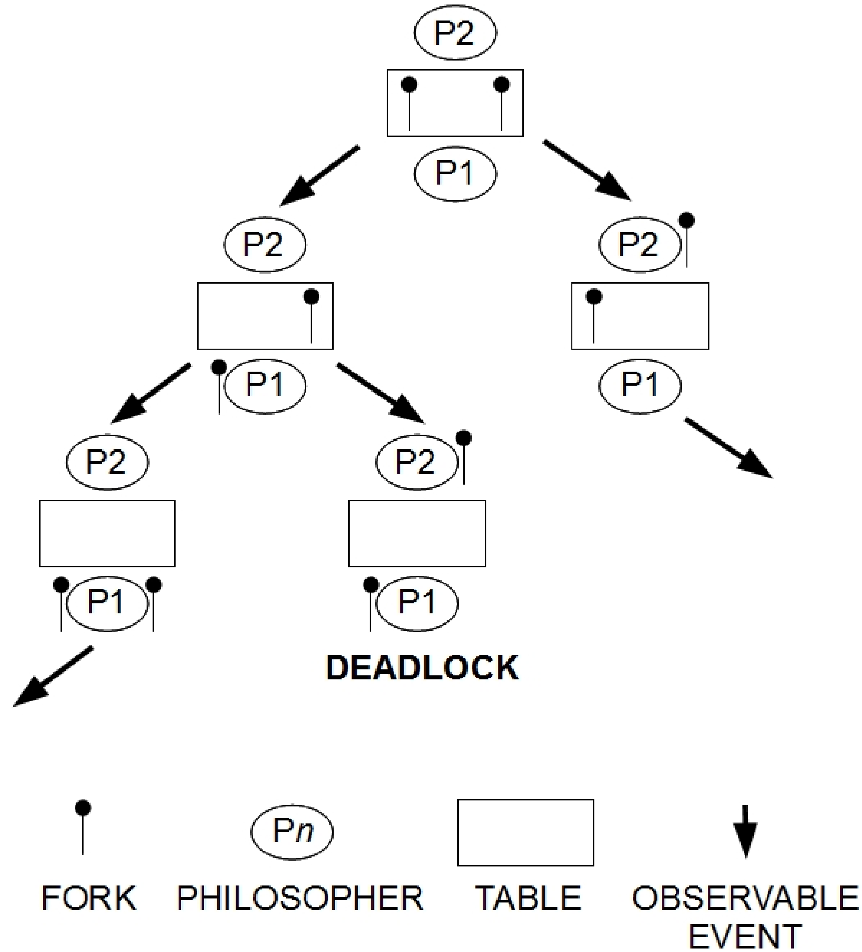

Figure 3.1: The Dining Philosophers Tree of Execution

give a full description or complete understanding, but it does allow the idea behind the Many Systems model to be grasped in the first instance. We label our two philosophers P_1 and P_2. Their instructions remain unchanged, namely: *pick up fork to the left, pick up fork to the right, eat, put down fork to the left, put down fork to the right, think, repeat.*

The scenario begins as both P_1 and P_2 attempt to pick up their first forks (F_1 and F_2 respectively). P_1 picks up F_1 followed by F_2 while P_2 attempts to pick up F_2. At this point three additional philosophers (P_{2A}, P_{2B}, P_{2C}) are created to reflect the three different causally consistent realities that can now take place

1. P_{2A} gains F_2. In this reality, P_2 picked up F_2 before P_1 picked up F_1;

2. P_{2B} gains F_2. In this reality, P_2 picked up F_2 before P_1 picked up F_2;
3. P_{2C} does not gain F_2. In this reality, P_2 tried to pick up F_2 after P_1 picked up F_1 and F_2.

P_{2C} now dies as its fork (F_2) was not available and so P_{2C} could not make progress. This brings to an end the reality of P_{2C} and the resources maintaining the reality of P_{2C} are 'released' (however, P_{2C} can pass in the knowledge that somewhere a doppelganger lives on).

Carrying on with the scenario, P_1 now places F_1 on the table, which due to causality, cannot happen before P_1 has picked up F_1 and F_2. A new future is created (let's simply label it 4) where P_{1A} places F_1 on table. This is the only possible future at this point because:

- P_1 can't place F_1 down in the future of P_{2A} because P_1 has yet to pick up F_1;
- P_1 can't place F_1 down in the future of P_{2B} because P_1 has yet to pick up F_2;
- P_1 can't put F_1 down in the future of P_{2C} because that reality ceased to exist.

The creation of reality 4 allows P_2 to revisit its attempt to pick up F_2. P_2 will not be successful, however, until the creation of a new reality after P_1 eventually places F_2 back on the table. It is important to understand that realities will terminate if advancement cannot be made within them (without philosophers waiting for forks). As a consequence, deadlock cannot occur at any stage in the scenario.

At first thought one may assume that efficiency, in terms of space, would render this a pointless avenue of research. However, considering the example described one may start to speculate about the possible benefits:

Reduced contention – A reality is essentially write once read many (as a new write creates a future reality or state). A similar model of execution has long been understood to promote concurrency and reduce contention in multi-version distributed databases [34].

Composability and Correctness Merge – As long as at least one partial ordering exists that produce results that satisfy the correctness criteria it doesn't matter how many failed results there are. In addition, more than one criteria of 'correct execution' can be identified in the state space. For instance, in addition to paths of execution free of concurrent conflicts (race conditions, deadlocks, etc) paths will also exist which are free of logical/semantic conflicts. For example, there will be paths where the ordering of execution is such that all processes can complete their execution in the least number of steps.

In summary, speculation is an already established approach in concurrency control, and a degree of speculative execution is seen as both necessity and beneficial. In the Many Systems model, we speculatively replicate not only memory (as in the case of transactions), but also thread execution.

3.2 Model Components

3.2.1 Events

Events form the foundation of the Many Systems model and represent actions by processes. Events fall into two categories, specifically *hidden* and *observable*. Hidden events represent those actions a process may carry out which are unimportant from the perspective of other processes (e.g. reading, changing local variables, etc.). An observable event, on the other hand, represents a change in shared state that is significant to more than one process (e.g. writing to a shared variable, committing a transaction, etc.).

Definition 3.2.1 *The set of observable events of a process P_x is the set of events produced by the intersection of the alphabet of P_x with the union of the alphabets of all processes excluding P_x:*

$$\text{Let } O_x = \alpha P_x \cap \bigcup_{i \in P \setminus \{x\}} \alpha P_i$$

3.2.2 Systems and Processes

Processes in the Many Systems model are referred to as System Processes.

Definition 3.2.2 *A System Process (P_x) is defined as a deterministic process with an initial state s, followed by a finite number of hidden events H, culminating in an observable event O. An observable event will lead to either another System Process, the* Stop *process or the* Skip *process:*

$$H^n O_i N_i = P_{x,s} \rightarrow (|_{i>0} O_i) \rightarrow (P_{x,s+1} \mid \textsc{Stop} \mid \textsc{Skip})$$

Stop and Skip (as defined in [32]) represent abnormal and normal termination of a process respectively. The actions of a dining philosopher can be represented by System Processes:

$$\textsc{Philo}_i \setminus \{i.sits\ down, eat, i.puts\ down\ fork.i, i.puts\ down\ fork.(i \oplus 1)\}$$
$$\textsc{Philo}_{i,s} = \ (i.picks\ up\ fork.i \rightarrow \textsc{Philo}_{i,s+1})$$
$$\textsc{Philo}_{i,s+1} = (i.picks\ up\ fork.(i \oplus 1) \rightarrow \textsc{Philo}_{i,s+2})$$
$$\textsc{Philo}_{i,s+2} = (i.gets\ up \rightarrow \textsc{Skip})$$
$$\textsc{Philo}_i = (\textsc{Philo}_{i,s}; \textsc{Philo}_{i,s+1}; \textsc{Philo}_{i,s+2})$$

The hiding operator (\) lists the hidden events, in this case all events except those that pick up a fork and those that signal the philosopher 'getting up'. The subscript i identifies the owning System Process, and the subscript s identifies the System Processes' state. System Processes separated by a semi-colon denote sequentially executing processes (i.e. the execution of $\textsc{Philo}_{i,s}$ precedes the execution of $\textsc{Philo}_{i,s+1}$).

Definition 3.2.3 *A System (represented by τ) is a state of a multi-process execution from which zero or more new states may arise. New states are defined as ordered sets of observable events. The notation $\tau(i)$ shall refer to the ith observable event possible from τ.*

3.2.3 Expansion

We assume the existence of a Root System (τ_0) from which initial computation occurs.

Definition 3.2.4 *The Root System (e.g. τ_0) represents an initial state of a multi-process execution. τ_0 may expand by giving rise to $size(\tau_0)$ child systems, representing $size(\tau_0)$ possible future states of τ_0.*

The *size*() operation returns the number of observable events possible from a given System. To model the creation of child Systems from parent Systems, we define a transition function trn.

Definition 3.2.5 *A new child System may be created by calling the trn operation, which requires an existing System τ, and an observable event that is a member of τ ($\tau(i)$):*

$$trn(\tau, \tau(i)) = (\tau \parallel i : (\tau \setminus \tau(i) \parallel \tau(i+1)))$$

The *trn* operation creates a copy of the initial System τ ($i : \tau$). The copy is labelled with a prefix (i) to distinguish between events of the initial System and the copy. Calling the *trn* operation is equivalent to first creating a clone of τ, and then applying the ith observable event in τ ($\tau(i)$) to the clone ($i : \tau$). Note that the initial System remains unchanged after the application of the *trn* operation.

Definition 3.2.6 *Multiple child Systems are possible from a Single system by applying trn for each element of τ. An expansion of τ ensues where $n = size(\tau_0)$:*

$$trn(\tau, \tau(0)) = (\tau \parallel 0 : \tau)$$
$$trn(\tau, \tau(0)); trn(\tau, \tau(1)) = (\tau \parallel 0 : \tau \parallel 1 : \tau)$$
$$trn(\tau, \tau(0)); trn(\tau, \tau(1)); trn(\tau, \tau(n)) = (\tau \parallel 0 : \tau \parallel 1 : \tau \parallel n : \tau)$$

Many Systems is recursive as the *trn* function may be further applied to each child System. We term the collection of Systems and their child Systems a Supersystem, denoted as Θ.

Definition 3.2.7 Θ *is in the form of an n-ary directed tree structure with each node representing a System and each arc representing the transition of an observable event.*

Theorem 3.2.1 Θ *maintains causality.*

Proof 3.2.1 *As* Θ *is an n-ary directed tree (Definition 3.2.7) and it may only contain future states created from past states (Definition 3.2.4) then a child System is caused by a parent System. As each child System is created by no more than one System Process proceeding sequentially (Definition 3.2.6) then causality is guaranteed.*

3.3 Solution Space

The primary purpose of the model is to construct a parallel execution that attains a correct observable state if one exists. The following Lemma therefore concern the occurrence of correct and erroneous paths of execution in Θ.

Lemma 3.3.1 *The observation of a semantically correct state (in the context of the application execution) in* Θ *is reachable via a path of prior states.*

Proof 3.3.1 *We may state that given a* τ_n *that is free from error, and there exists at least one child system of* τ_n *also free from error, there must be some error-free progression from parent to child. Assuming that a system exists in* Θ *that is correct with respect to an observable state then we may extrapolate that there exists a corresponding path in* Θ *that is error-free that may reach such a state.*

Lemma 3.3.2 *The observation of a semantically erroneous state in* Θ *is reachable via a path of prior states.*

Proof 3.3.2 *In the same manner we reasoned about correctness, we can state that given a* τ_n *that is not free from error then at least one child system of* τ_n *will not be free from error. Assuming that a system exists in* Θ *that is in error with respect to an observable state then we can extrapolate that there exists a corresponding path in* Θ *that contains errors that may reach such a state.*

Correctness and failure may only be realized via observation of Θ. A state of say τ_n, may be considered erroneous or error-free only after an observation of failure or correctness in subsequent child systems of τ_n. Consequently, the Many System model does not require the program to determine if any given System is erroneous or error-free before or during its creation. However, with the possibility of searching all partial orderings of execution, a solution will be forthcoming.

3.4 Waiting

In our unconstrained model, many systems exist in parallel at every observable state of execution. Therefore, there is no requirement for System Processes in distinct systems to 'wait' for each other. The definition of Wait-Free synchronization is not sufficient for describing waiting properties in our approach so we define the notion of No-Wait synchronization:

Definition 3.4.1 *No-Wait synchronization describes a parallel execution within which a logical representation of a System Process can always carry out its shared access requests in the same number of steps as an equivalent sequential implementation.*

Considering our definition of No-Wait we can state the following:

Lemma 3.4.1 *No-Wait synchronization guarantees a System Process can always determine their next action in one execution step.*

Proof 3.4.1 *Consider a System, labelled τ_{curr}, containing a System Process P_w. Suppose P_w periodically queries τ_{curr} for the occurrence of an observable event e and let us label those queries $q\{1, 2...n\}$ to denote n queries. If q_1 does not detect e in τ_{curr}, then it follows that no q_i in $q\{2...n\}$ will detect e in τ_{curr} because the occurrence of e would create a future System, say τ_e, wherein e would be observed. We can rule out queries $q\{1, 2...n\}$ causing e, given that querying is not an observable event.*

Note that τ_e (wherein e may be observed) must contain P_w at a state before q_1 was executed given that querying is not observable and only observable events are carried forward into future Systems. When P_w executes q_1 in τ_e it will detect e

and so $q\{2...n\}$ are superfluous. Hence no System Process need query more than once the occurrence of an observable event.

No-Wait has implications for how deadlock is handled:

Lemma 3.4.2 *Deadlock is eradicated by No-Wait synchronization.*

Proof 3.4.2 *By definition of Lemma 3.4.1, any System Process can determine its next action in one execution step (given that a System which does not permit an observable event to occur at time t, will never permit that event). Hence, no System Process P_x should wait for another System Process P_y to release a shared resource (as signalled by an observable event). By Definition 3.2.2, if P_x cannot participate in an observable event with P_y then it must terminate via the* STOP *or* SKIP *processes. If ever P_x and P_y have acquired mutually required resources, then both System Processes will terminate rather than wait on one another to release those resources.*

In the *Dining Philosophers* scenario, for example, deadlock is encountered if any philosopher waits indefinitely for a fork to transit to the state where it can be used. In the Many Systems model, however, no philosopher should ever wait for a fork to become available. If a philosopher finds a fork unavailable it must terminate (with the assumption that there must be some alternative philosopher, somewhere in Θ, that has found the fork ready and can acquire it). Execution paths in Θ where all philosophers would have waited indefinitely for one another to release forks instead manifest as Systems where no observable event can be generated. In such 'dead-end' Systems, no child Systems can be created and so no further resources are required. Alternate paths of execution in Θ may proceed unhindered by such Systems.

Considering Lemma 3.4.1 and 3.4.2 together we may deduce:

Corollary 3.4.1 *Many Systems concurrency control provides No-Wait semantics and is deadlock free.*

3.5 Example

We now demonstrate our model via an example based on the *Dining Philosophers* problem (described in Section 2.5.2 of Hoare's CSP book [32]). Our example requires the reader understand the original CSP example given by Hoare to understand how our model deviates. For brevity, our example only considers two dining philosophers and two forks.

We begin by creating System Processes to model the philosophers and their forks; however, the implications of No-Wait synchronization means that the model of philosopher behaviour must be altered:

$$\begin{aligned} \textit{let}\ \textsc{Philo}_i &= (\textsc{Philo}_{i,s}; \textsc{Philo}_{i,s+1})\ \textit{where} \\ \textsc{Philo}_{i,s} &= (\ \textit{i.picks up fork.i} \rightarrow \textsc{Philo}_{i,s+1} \\ &\quad \mid \textit{i.missing fork.i} \rightarrow \textsc{Skip}) \\ \textsc{Philo}_{i,s+1} &= (\ \textit{i.picks up fork}(i \oplus 1) \rightarrow \textsc{Philo}_{i,s} \\ &\quad \mid \textit{i.missing fork}(i \oplus 1) \rightarrow \textsc{Skip}) \end{aligned}$$

An extra event has been added to the specification of the Philo system process (*missing fork*). If a System Process cannot obtain a fork, without waiting, it may now execute the *missing fork* event and transit to the Skip process (thus terminating its future execution). This effectively requires blocking behaviour (such as waiting for a fork) be replaced with non-blocking behaviour. Note that in the original *Dining Philosophers* scenario, erroneous execution would ensue if a philosopher terminated abruptly when no fork was available as the original scenario relies on philosophers waiting until a fork is ready. Conversely, in the Many Systems scenario, such abrupt termination shall avert erroneous execution. As it is not important from the perspective of other System Processes, the *missing fork* event is hidden:

$$\begin{aligned} &\textsc{Philo}_x \backslash \{\textit{i.missing fork.i}, \textit{etc}\} \\ &\textsc{Philo}_i = (\textit{i.picks up fork.i} \rightarrow (\textit{i.picks up fork}(i \oplus 1) \rightarrow \textsc{Philo}_i \mid \textsc{Skip}) \\ &\qquad\qquad \mid \textsc{Skip}) \end{aligned}$$

System Processes are now required to model the behaviour of forks:

$$\begin{aligned}
\textsc{ForkUnused}_i &= (\ \textit{i.picks up fork.i} \rightarrow \textsc{ForkUsed}_i \\
&\quad \mid (i \ominus 1)\textit{.picks up fork.i} \rightarrow \textsc{ForkUsed}_i) \\
\textsc{ForkUsed}_i &= (\ \textit{i.puts down fork.i} \rightarrow \textsc{ForkUnused}_i) \\
&\quad \mid (i \ominus 1)\textit{.puts down fork.i} \rightarrow \textsc{ForkUnused}_i)
\end{aligned}$$

With both philosophers and forks defined, we now need to describe how these components are combined to create Systems. We use the same terminology as Hoare and identify a System τ as a COLLEGE of philosophers for this purpose. A COLLEGE is equivalent to our notion of a System (τ). A COLLEGE with two System Processes (dining philosophers) is as follows:

$$\begin{aligned}
\mathrm{P}_0 &= (\textsc{Philo}_0 \parallel \textsc{ForkUnused}_0) \\
&= (\textit{0.picks up fork.0} \rightarrow (\textsc{Philo}_0 \parallel \textsc{ForkUsed}_0)) \\
\mathrm{P}_1 &= (\textsc{Philo}_1 \parallel \textsc{ForkUnused}_1) \\
&= (\textit{1.picks up fork.1} \rightarrow (\textsc{Philo}_1 \parallel \textsc{ForkUsed}_1)) \\
\textsc{College}_0 &= (\mathrm{P}_0 \parallel \mathrm{P}_1)
\end{aligned}$$

The Root System is identified as $\textsc{College}_0$ with the subscript 0. We can now produce two potential successor states from the initial state. Specifically, by applying the *trn* function to $\textsc{College}_0$ we may generate two child Systems from $\textsc{College}_0$:

$$\begin{aligned}
&(\textit{available events}: \ \textit{0.picks up fork.0}, \textit{1.picks up fork.1}) \\
&trn_0(\textsc{College}_0, P_{0,0}) = (\textsc{College}_0 \parallel 1 : \textsc{College}) \\
&trn_1(\textsc{College}_0, P_{1,0}) = (\textsc{College}_0 \parallel 2 : \textsc{College}) \\
&\textit{where}: \\
&1 : \textsc{College} = \ (\textsc{Philo}_0 \parallel \textsc{ForkUsed}_0 \parallel \textsc{Philo}_1 \parallel \textsc{ForkUnused}_1) \\
&2 : \textsc{College} = \ (\textsc{Philo}_0 \parallel \textsc{ForkUnused}_0 \parallel \textsc{Philo}_1 \parallel \textsc{ForkUsed}_1)
\end{aligned}$$

(1 : College) represents the state of College_0 after philosopher 0 has picked up his first fork and (2 : College) represents the state of College_0 after philosopher 1 has picked up his first fork. Applying *trn* to (1 : College) produces the following:

$$
\begin{aligned}
&(\textit{available events}: \ \textit{0.picks up fork.1}, \textit{1.picks up fork.1}) \\
&trn_0(1 : \text{College}, P_{0,1}) = (1 : \text{College} \parallel 3 : \text{College}) \\
&trn_0(1 : \text{College}, P_{1,0}) = (1 : \text{College} \parallel 4 : \text{College}) \\
&where : \\
&3 : \text{College} = \ (\text{Philo}_0 \parallel \text{ForkUsed}_0 \parallel \text{Philo}_1 \parallel \text{ForkUsed}_1) \\
&4 : \text{College} = \ (\text{Philo}_0 \parallel \text{ForkUsed}_0 \parallel \text{Philo}_1, \parallel \text{ForkUsed}_1)
\end{aligned}
$$

(4 : College) is effectively in a state of deadlock given that any new System created via the *trn* function will results in a Philo process ending due to the next fork being unavailable. However, given the existence of (2 : College) and (3 : College), progress may still be made.

The number of Systems that are generated in this trivial example are $(1 + 2(2) + 2)$, representing: the Root System (1); two sequential orderings (2(2)) and two pre-deadlocked Systems (2). As each philosopher is composed from 2 System Processes ($\text{Philo}_{i,s}$ *and* $\text{Philo}_{i,s+1}$), the value 2(2) represents two executions of the philosopher processes. The worst case rate of expansion may be expressed by the *multinomial coefficient* equation:

$$(\sum_{i=1}^{n} q_i)!/(q_1!q_2!.....q_n!)$$

where $(\sum_{i=1}^{n} q_i)$ corresponds to the sum of System Processes of all processes and each q_i corresponds to the number of System Processes produced from the i-th process. A non-terminating recursive definition of this example increases to an infinite number of states. In the following section, however, we provide a practical application of the Many Systems model where finite resources are considered. Specifically, we show how: (i) Θ can be constructed and managed

with limited resources; and (ii) the results of execution can be provided to a concurrent application in a lock-free manner.

3.6 A Universal Construction

We now demonstrate how a Supersystem can be managed as a Universal Construction (hereafter UC) of system processes. A UC is typically described as a class that can transform a sequential object into either a Wait-Free or Lock-Free linearizable object, allowing concurrent access by any number of threads [27]. Although the objects under consideration may range from simple containers (such as stacks or queues) to a collection of transactions on shared data [28, 29, 30], the purpose of the UC is to provide each participating thread with the same view of the object.

We begin our description of a Many Systems UC by defining the System Processes which essentially conduct application logic. Processes which create and maintain Θ are described next, followed by a set of proofs which show that UC execution is Lock-Free.

3.6.1 System Processes

System processes will be modelled as transactions. System processes may generate two observable events: the *txdone* and *txabort* events capture the behaviour of transactions for the purpose of our model (see Equation 3.1).

$$\textsc{Txn}_i = (txdone \rightarrow \textsc{Skip} \mid txabort \rightarrow \textsc{Skip}) \tag{3.1}$$

With System Processes comprised of transactions, expansion of the Supersystem produces permutations of transaction execution. As we limit the states observable within Θ to the execution of transactions by multiple processes, the initial state (τ_0) represents a point in program execution from which each process may start a new transaction.

3.6.2 Universal Construction Processes

The relationship between the Universal Construction (UC) and the Concurrent Environment (*env*) is defined in Equation 3.2:

$$\begin{aligned} env \;&=\; uc : (\text{MS}\textsc{ystem}_0 \parallel \text{W}\textsc{kr}\text{P}\textsc{ool}) \mathbin{/\!\!/} (|||_{p \leq m}\; p : \text{P}) \\ where\; &\text{W}\textsc{kr}\text{P}\textsc{ool} = (\|_{w \leq n}\; w : \text{W}\textsc{kr}) \end{aligned} \tag{3.2}$$

The Concurrent Environment is comprised of the UC (labelled *uc*) and application processes (p : P). The application processes rely on the UC to provide Concurrency Control via the execution of transactions. We want the UC to act as the shared communication medium for application processes, therefore we specify this relationship with the CSP subordination operator ($/\!\!/$). When two or more processes are interleaved ($|||_{p \leq m}\; p$: P), then they may only communicate via a subordinate process (the UC in this case).

The UC itself is comprised of a Managed Root System ($\text{MS}\textsc{ystem}_0$) containing the initial state of the Concurrent Environment, and a pool of n concurrent worker processes ($\|_{w \leq n}\; w$: WKR), which are collectively referred to as a WKR-POOL. Workers are responsible for updating the state of the UC. By grouping workers into a worker pool, the degree to which processing resources can be controlled by specifying the size of the worker pool. Let us now define the Managed System and the Worker processes in turn.

Managed Systems The Root System for the Universal Construction is defined in Equation 3.3. The Root System is an aggregate of a system (τ) and a manager process (MGR) which we refer to as a Managed System (MSYSTEM):

$$\begin{aligned} \text{MS}\textsc{ystem}_0 &= ((cns := 0; val := 0; \tau) \parallel \text{M}\textsc{gr}) \\ where\; \text{M}\textsc{gr} &= (start \rightarrow \text{T}\textsc{mr}_{t,t}; \text{U}\textsc{pdate}; \text{M}\textsc{gr}) \end{aligned} \tag{3.3}$$

The System τ represents the initial state of the UC and maintains two variables (*cns* and *val*):

- (cns) records the number of child Systems created from this System and is initially zero;
- (val) records the 'value' of the System and is considered application specific. The significance of val to the operation of the UC is that it will be used to determine which System will be chosen over others (when deciding on the next state of the UC).

The manager process (Mgr) coordinates the activities of the worker processes, and updates the Root System to reflect changes in state generated by the workers. The manager first signals the worker processes to begin working (*start*), and then the manager commences a timer process (Tmr) to ensure that worker processes terminate expansion after a set amount of execution time. Once the timer expires (by signalling the *timeout* event), the manager performs the Update process defined in Equation 3.4.

$$\textsc{Update} = (\textsc{Max}_{0,n}(\tau, m); assign(\tau, m); sched(\tau); \textsc{Skip}) \quad (3.4)$$

The Max process (see Appendix, Equation 7.2), evaluates the value (val) of each child system of τ and records the identity of the 'winning system' in m. Then the *assign* function replaces the state of τ with the new state in m and the manager reschedules the events of τ by invoking the *sched* function. The *sched* function reorders the events of τ to prevent the possibility that an event in τ is never explored by a worker process (i.e. to prevent the starvation of any transaction when the number of workers is less than the number of transactions that can be executed from τ). Both the *assign* and *sched* functions are described in the Appendix (see Table 7.4).

Worker Processes (Wkr) carry out the task of executing transactions and generating new systems (see Equation 3.5). Each worker has access to a history stack (*hist*) to keep a record of systems the worker has created. Workers perform expansion (DoExp) and compression (DoCmp) of the Supersystem. Importantly, the worker processes are restricted to the degree of time granted by the Tmr process of the Manager:

$$\begin{aligned}\textsc{Wkr} = \; & (hist : \textsc{Stack}) /\!\!/ (start \rightarrow \textsc{Iter}; \textsc{Wkr}) \\ & \quad where\ \textsc{Iter} = (expand \rightarrow \textsc{DoExp}; \textsc{Iter}) \\ & \qquad\qquad\qquad\quad \mid timeout \rightarrow \textsc{DoCmp}; \textsc{Wkr})\end{aligned} \tag{3.5}$$

Expansion The expansion process (DoExp) is defined in Equation 3.6 and is performed by each Worker. The expansion process is comprised of the DFSearch and Aqr processes.

$$\begin{aligned}\textsc{DoExp} = \; & (\textrm{DF}\textsc{Search}(Root) \parallel \textsc{Aqr}) \\ \textsc{Aqr} = \; & (searched.head(A) \rightarrow \\ & \quad if\ (x := ncas(A.cns, A.cns + 1) < obs(A))\ then \\ & \qquad hist.push(\{A, x\}) \rightarrow \\ & \qquad trn(A, A(x)) \rightarrow \\ & \qquad valuate(child(A, x)) \rightarrow \textsc{Skip} \\ & \quad else\ fail \rightarrow \textsc{Skip})\end{aligned} \tag{3.6}$$

- The DFSearch process performs depth-first search of the Supersystem tree (see Appendix, equation 7.3). The 'root' system (from where any search begins) is either τ if hist is empty, or *child(X, i)* if the head of *hist* equals $\{X, i\}$. The search halts when the worker can acquire a system in the Supersystem tree or when all Systems have been explored.
- The Aqr process defines how workers may uniquely 'acquire' observable events within Systems. The creation of new (child) systems by worker processes is then performed with the *trn* function. Each system may produce 0 or more child systems depending on the number of observable events (obs(A)) that are possible from a given system. Workers compete for each child system by using the *ncas* function, to gain unique ownership of an observable event. If a worker processes acquires an event, then the *trn* function (see Definition 3.2.5) is executed to create a new child system

$(trn(A, A(x)))$ and the child system is assigned a value using the *valuate* function (see Table 7.4 in the Appendix for definitions of the *obs*, *ncas* and *valuate* functions).

Once a worker process has acquired and expanded a system, it performs the depth-first search procedure again. Subsequent searches commence from the last child system the worker created (i.e. *hist.head(X)*) so that any new systems created by a worker are restricted to a single path of the Supersystem tree. New systems are created until no further child systems are possible or the *timeout* event has occurred.

Compression Once a worker process has finished expanding, it performs a tree compression algorithm (DoCmp) to remove systems from the Supersystem until there remains only the root node and its child nodes (see Equation 3.7).

$$\begin{aligned} \textsc{DoCmp} &= (if\ !(hist.empty)\ do \\ &\qquad \textsc{Cprs}(hist.head, 0) \rightarrow hist.pop \rightarrow \textsc{DoCmp}) \\ &\quad else\ \textsc{Skip}) \\ \textsc{Cprs}(\{Root, i\}, m) &= (if\ !(leafnode(child(Root, i)))\ then \\ &\qquad \textsc{Max}_{0,n}(child(Root, i), m) \rightarrow \\ &\qquad contract(Root, i, m) \rightarrow \textsc{Skip} \\ &\quad else\ \textsc{Skip}) \end{aligned} \tag{3.7}$$

Each worker performs the compression process for each system in its history stack (*hist*). Each member of the history stack identifies a child system which the worker created during expansion, and so once all workers have performed compression for each member of their history stacks, the only systems which remain will be the root node and its immediate child nodes.

The Cprs process accepts a root node *Root* and the i-th child of *Root*. If the i-th child is not already a leaf-node, then the grand-child of *Root* with the maximum value is located using the Max process. The new i-th child of *Root* is then set to m by invoking the *contract* function.

Once compression has been completed for every worker process, the manager process can complete its Max process and, by extension, its Update process. A new round of expansion can then commence by the manager processes updating the state of τ and signalling the *start* event. Figure 3.2 shows the processes of expanding and contracting the Supersystem.

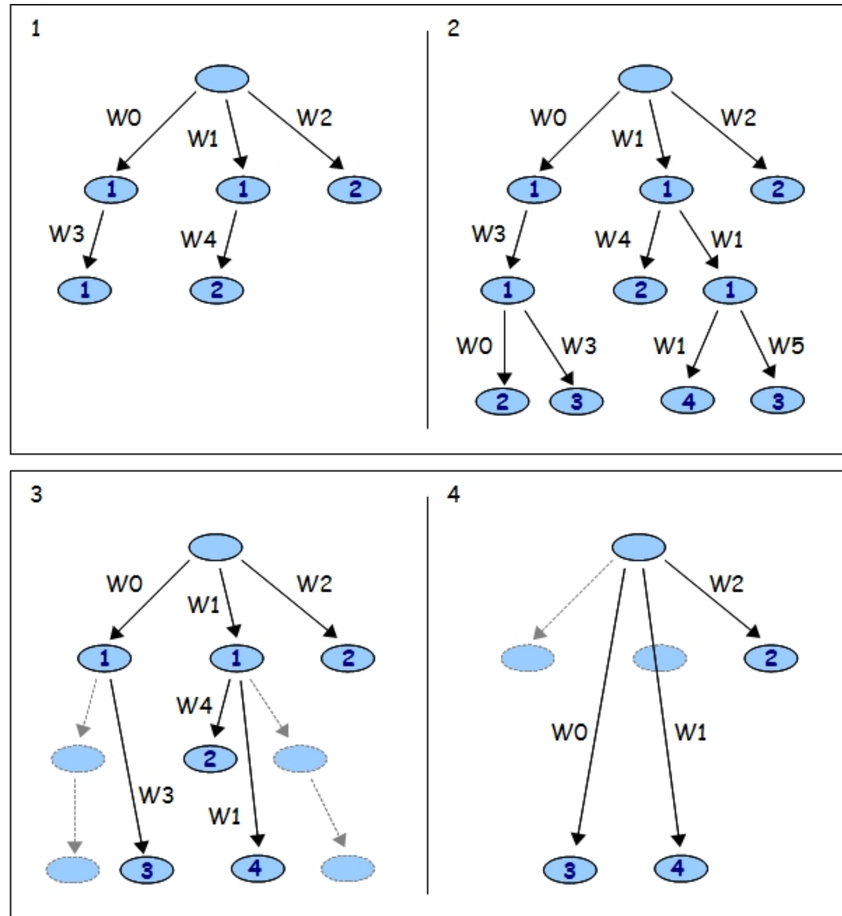

Figure 3.2: Expansion and Compression *Images 1 and 2 show expansion of the Supersystem by Worker Processes. Images 3 and 4 show the compression of the Supersystem until it has a depth of 1. Hypothetical values for each system (the nodes) are shown. Shaded nodes with dashed arcs denote branches of the Supersystem that have been removed during the compression process.*

3.6.3 Proofs

Before describing the proofs, we must first define some properties of the systems in the Supersystem (i.e. the nodes of the tree) which will aid in the process of proving the lock-free property of the Universal Construction. We state that, at any time, $\forall$ *System* $S \in \Theta$, either:

1. $leaf(S) = true$ where $childset(S) = \emptyset$ (i.e. S has no child Systems);
2. $parent(S) = true$ where $\forall$ *System* $X \in childset(S) \bullet leaf(X)$;
3. $gparent(S) = true$ where $\forall$ *System* $X \in childset(S) \bullet leaf(X) \vee parent(X)$;
4. $ancestor(S) = true$ where neither $leaf(X)$, $parent(X)$ nor $gparent(X)$ hold $\forall$ *System* $X \in childset(S)$.

To prove that the Universal Construction is lock-free, we need first to prove that the Max process is lock-free because this is the only process where the progression of one process is dependent on another. Specifically, we must show that the Max process will always complete within a finite number of operations, and to demonstrate this we present five Lemmas which show that the Supersystem tree will always reduce to a form consisting only of leaf-nodes and parent-nodes (i.e. *leaf(S)* or *parent(S)*). The Lemma 3.6.6 shows that the Max process will always complete within a finite number of operations, and finally we prove that the Universal Construction is lock-free.

Lemma 3.6.1 $\forall$ Wkr $w \in$ WkrPool, *if the history stack of* w *contains* m *elements and* $m > 1$, *then element* m *must be a descendant system of element* $m - 1$.

Proof 3.6.1 *The first element* $\{X, i\}$ *pushed onto any history stack is such that* X *equals the root system and any further elements pushed onto the history stack are determined by first calling the* DFSearch *process. By definition of the* DFSearch *process, the search always begins with the head of the history stack, therefore any system* Y *that is acquired, must be a descendant of the head of the history stack.*

Lemma 3.6.2 *If $\forall$ System $S \in T \bullet leaf(S)$ then every call to* MAX(T) *will complete in a single invocation (where $T \in \Theta$).*

Proof 3.6.2 *In this case T contains a single system and as* MAX *iterates for every child node, by definition of the* MAX *process,* MAX(T) *will complete in a single invocation.*

Lemma 3.6.3 *If $\forall$ System $S \in T \bullet parent(S)$ then* MAX(T) *will complete in at most n iterations (where $n = size(childset(T))$ and $T \in \Theta$).*

Proof 3.6.3 *As* MAX *iterates for every child node, by definition of the* MAX *process, for n child nodes* MAX(T) *will complete in n iterations.*

Lemma 3.6.4 *For any system X, if gparent(X) is true, then after a finite number of operations, parent(X) must become true.*

Proof 3.6.4 *If gparent(X) is true, then by definition of gparent, $\forall\, Y \in childset(X)$, either $parent(Y)$ or $leaf(Y)$ is true. Furthermore, there must be at least 1 worker process whose history stack contains $\{Y, i\}$ as the head element. When these workers call* DOCMP*, it will terminate in a single invocation because by definition of* DOCMP*, calling* CPRS$(\{Y, i\}, m)$ *terminates in a single invocation when $parent(Y)$ or $leaf(Y)$ is true. By definition of* DOCMP*, the history stack of these workers will be popped, and by Lemma 3.6.1, there must be one or more workers whose new head of the history stack is $\{X, i\}$.*

When CPRS$(\{X, i\}, m)$ *is called, by definition of* CPRS*,* MAX *is called with a $parent(Y)$ which will terminate in at most n steps (by Lemma 3.6.3) and the contract function will be invoked on system X. After a finite number of $contract(X)$ invocations, $parent(X)$ must be true.*

Lemma 3.6.5 *For any system X, if ancestor(X) is true then after a finite number of operations, gparent(X) must become true.*

Proof 3.6.5 *We refer to A as the set of all systems in Θ where $ancestor(S)$ is true. There must be a subset of A, let's say A', where each member $X \in A'$ is the direct parent of a System S, such that $gparent(S)$ is true. By Lemma 3.6.4 we have shown that for all S where $gparent(S)$ is true, after a finite number*

of operations, $parent(S)$ must become true. Therefore, after a finite number of operations, each member of A' must become the parent of a parent system, or in other words, $gparent(X)$ must become true. Furthermore, each ancestor system which is a parent of X, now belongs to the set A'. After a finite number of operations, A' must contain the root system at which point every ancestor system will become a grand-parent system.

Lemma 3.6.6 *The* MAX *process is lock-free.*

Proof 3.6.6 *Lemma 3.6.6 shows that any ancestor system must become a grand-parent system after a finite number of operations and Lemma 3.6.4 shows that grand-parent systems must become parent systems after a finite number of operations. As every System in Θ must satisfy either $parent(S)$ or $leaf(S)$ after a finite number of operations, by Lemma 3.6.3 and Lemma 3.6.2, the* MAX *process must terminate in a finite number of operations.*

Lemma 3.6.7 *The Universal Construction is lock-free.*

Proof 3.6.7 *To show that the Universal Construction is lock-free we need to prove that the* WKR *and* MGR *processes are lock-free. The* WKR *process consists of the* DOEXP *and* DOCMP *processes. For the* DOEXP *process to be lock-free we need to show that the* DFSEARCH *and* AQR *processes are lock-free. Firstly, assuming the number of application processes is at most n and the number of worker processes is m, the number of possible child states from any system is at most min(n,m), (with n decreasing with the creation of every child system). Hence the* DFSEARCH *must terminate after a finite number of operations. At each system during the search, the* AQR *process is called by the worker using the ncas function to acquire an observable event. The ncas function must terminate after at most min(n,m) calls, i.e the number of possible child states from any system. Finally, to show that the* DOCMP *process is lock-free requires first proving that the* CMPRS *process is lock-free. By definition of the* CMPRS *process,* CMPRS *is lock-free given Lemma 3.6.6, which proves that the* MAX *process is lock-free.*

The MGR *process consists of the* TMR *and* UPDATE *processes. By definition, the* TMR *process will always terminate after t iterations. Assuming that the assign and sched functions are lock free, it is trivial to show that* UPDATE *process is lock-free by Lemma 3.6.6, which proves that the only remaining process used by the* UPDATE *process (namely* MAX*), is lock-free.*

3.7 Properties

We now consider the appropriateness of Many Systems Concurrency Control via a number of properties that may be inferred from the Model as presented.

3.7.1 Containment

Containment describes the ability to deterministically restrain computation complexity during execution. This is a property that traditionally is present in all correct programs and is lost only in erroneous programs (e.g., memory leaks). In the Many Systems model, the size of the execution environment (the size of expansion) is unknown beforehand and can only be predicted for the worst case. Therefore the Universal Construction of Section 3.6 allows expansion to be controlled and contained deterministically. Specifically, the Universal Construction can regulate expansion by controlling the size of the worker pool and the value of the timer used by the MGR process. Ultimately, at each stage of expansion, the number of child Systems is limited by the minimum of the number of worker processes m and the number of possible events from a system state n (i.e. *min(m,n)*).

3.7.2 Isolation

Isolation describes an interaction with shared state that is equivalent to an interaction without interference. This is the most important property concurrency control seeks to satisfy. The Many Systems model has this property because Θ is a directed tree. It is impossible for Systems on different paths of Θ to interact (see Definition 3.2.7) while System Processes within a single System may only produce observable events to shared state in new Systems.

3.7.3 Liveness

Liveness is a property that fundamentally describes the usefulness of a concurrent system. More specifically, liveness in a system x indicates that one or more processes in x will eventually progress x as requested by a programmer. The no-wait and deadlock free properties of our model (Corollary 3.4.1) always guarantee

theoretical liveness in that all processors will eventually execute all their steps in a causality preserving manner (Theorem 3.2.1). Due to the nature of the model (rather than any synchronization construct utilized by the programmer), non-deadlocked executions can progress in Θ (Lemma 3.4.1), even in the presence of the logical representations of failed execution paths existing concurrently in Θ (Corollary 3.4.1).

The Lock-Free property of the Universal Construction of Section 3.6 signifies that the state updates generated by the expansion of the Supersystem can be provided to application processes in a bounded number of executions (Lemma 3.6.7). In addition to the general liveness property of the Many Systems Model, concurrently consistent results can be provided to application processes in a timely manner.

3.7.4 Scalability

Scalability in concurrency control can be described as the ability to maintain performance when contention rises on a shared state. In the Universal Construction of Section 3.6, an increase in resources (i.e. worker processes) allows an increase in the exploration of Θ in parallel which, probabilistically, will reduce the time to find a correct system. Conversely, there is sparse opportunity to vary scalability to improve performance (by increasing parallel resources) in existing concurrency control techniques.

3.7.5 Composable Correctness Criteria

Linearisability is the starting point for defining correctness criteria used in many transactional memory approaches. Changing the correctness criteria on a per-thread basis during run-time is impractical in existing solutions. In fact, this makes little sense as determinism is removed from the computation. However, by regarding the solution space as disjoint isolated executions that maintain causality (Theorem 3.2.1), a correctness criteria may be satisfied in some execution within Θ as opposed to a constraint that must be imposed.

An interesting property of the execution in Θ, is that the generated state space may satisfy multiple correctness criteria for the same program execution.

For example, paths of execution may exist which simultaneously exhibit sequential consistency in addition to an ordering of process execution which satisfies priority or timing constraints. Furthermore, the potential for generating multiple future executions, satisfying multiple correctness criteria, allows consideration of a wide range of properties a programmer may define as correctness. For example, when one considers the ordering of concurrent execution, one may identify execution in Θ which improves the overall progression of the concurrent program, where processes rely on the activity of other processes in order to complete their execution. Examples of such activity are common (e.g., a shared buffer may require consumers to follow the execution of producers, or a shared bank account may require withdrawal operations to follow deposit operations).

3.8 Summary

In this chapter we have described a model of computation where concurrent processes participate in the construction of a causally consistent tree structure, which we called a Supersystem. We began by describing the components of the Supersystem such as Observable Events, System Processes and Systems. We then provided proofs about the correctness of execution within the Supersystem and introduced No-Wait Synchronization—a progress condition specific to the Many Systems Model, which allows processes to decide their next execution in a single step. An example is presented using the *Dining Philosophers* problem to clarify the approach for the reader, followed by a Universal Construction, which allows expansion and compression of the Supersystem by the activities of worker processes. Finally, we prove that the Universal Construction is lock-free and close with a number of properties that are a feature of the Model.

Chapter 4

Implementation

In Chapter 3, we described the Many Systems model, where concurrency control is approached from the perspective of state-space management (composed from the execution of multiple processes). In this chapter we put the theories of the Many Systems model into practice and present an implementation of a Universal Construction which provides contention management for Software Transactional Memory (STM). We only explore the state space of transactions that have aborted explicitly and we refer to such conflicts as *semantic conflicts*. The benefit of this design choice is that we can allow a separate Contention Management Policy (CMP) to deal with concurrent (read/write) conflicts until semantic conflicts are encountered.

Our CMP must also handle concurrent interference during the resolution of semantic conflicts, and this is achieved by causing each single thread to execute transactions sequentially. With respect to the model, this means that a single thread explores a unique path in the Supersystem (Θ). The expansion of Θ is expressed by the parallel exploration (by multiple threads) of multiple transaction permutations. The aim of parallel exploration is to discover permutations of transaction execution which promote the logical progression of the concurrent application by resolution of semantic conflicts.

Semantic Conflicts From the programmer's perspective, conflicts fall into two categories: concurrent conflicts and semantic conflicts. A concurrent conflict occurs when the reads and writes of a transaction encounter an inconsistent state

of shared memory and contention management policies combat these types of conflicts. A transaction may execute without interference, however, and still need to re-execute because semantically the application cannot progress (for example, a transaction may need to consume an item from a shared buffer but finds it empty, or a bank account may have insufficient funds to permit a withdrawal).

Typically, a semantic conflict can be dealt with in the application by (i) letting the transaction commit and re-execute in the future, or (ii) by using primitives (*retry*, *orElse* etc) as provided by Harris et al [14], which essentially allow ad-hoc coordination of transaction execution. The issue with the former approach is that needless future conflicts may arise when transactions re-execute. Moreover, the use of primitives places a burden on the application developer that must be addressed with an ad-hoc solution (thus re-introducing a fundamental problem of coordination with pessimistic concurrency control, which Transactional Memory originally sought to address).

Within the scope of this thesis we consider a semantic conflict as simply *the intentional abortion of a transaction by its own thread, which can be avoided by an appropriate coordination of transaction execution.*

For example, the coordination of transaction execution may require the commitment of a depositor transaction before an account possesses enough funds to allow the commitment of a withdrawal transaction. We do not require that the application programmer coordinate such transactions. Instead, that task is delegated to the contention manager.

Shared Data Model As described in Section 2.4.3, STM implementations can be categorized by a number of factors related to how shared data is represented, stored and regulated. Before describing the implementation, we first define the method of shared data access in our approach, specifically:

Object Based – Our implementation is based on the DSTM2 framework provided by Herlihy et al [10], which represents shared data in the form of Atomic Objects. With this model of shared data, threads achieve concurrency by making modifications to shared data-structures composed of Atomic Objects. All Atomic Objects contain methods for shared reading and writing.

Eager Update – During the execution of transactions, threads use the *eager-update* model of access (with visible reads). Every Atomic Object contains a field which identifies the current owner of the object, and whenever threads attempt to modify an atomic-object, they must first attempt to install themselves as the owner of the object.

Obstruction Free – Threads gain ownership of atomic objects with successful calls to the *compare-and-swap* operation (as opposed to a lock-based approach, which uses a short critical section).

4.1 Basic Contention Management

4.1.1 Overview

The concept of the Universal Construction (hereafter UC) was first proposed by Herlihy [27] and essentially allows any sequential data structure to be transformed into a linearizable representation that can be accessed and updated by multiple threads. There are three phases of UC operation: (i) threads prepare and announce a proposed input to add to the UC, (ii) each announcing thread performs consensus to decide which input will be added, and (iii) a log of inputs is updated by the winning thread to reflect its input. We begin with an overview of how we use the UC technique and then provide greater detail in the remainder of this section.

We use the UC technique to provide conflict resolution in the presence of semantic conflicts and therefore our policy will be used by those threads whose transactions have aborted due to a semantic conflict (an explicit abort). Our UC accepts as input a permutation of one or more sequentially executed transactions, and consensus decides which permutation will be added to the log.

When some $thread_a$ encounters a semantic conflict, it inserts its aborted transaction into a global *Transaction Table* (see Figure 4.1, phase 1). The thread then enters a *Speculative Phase*, where it re-executes aborted transactions within the *Transaction Table*. We provide $thread_a$ with a private cache to hold copies of

modified atomic objects, but no transaction is committed. Transactions are executed sequentially to prevent concurrent interference, but transactions may still abort due to semantic conflicts (Figure 4.1, phase 2).

During the *Speculative Phase* of $thread_a$, other threads whose transactions have been aborted, may execute their own speculative transactions in parallel with $thread_a$. Once the *Speculative Phase* ends, each participating thread then enters a *Commit Phase* to decide which single thread's cache of modified atomic objects will be committed using a consensus algorithm. Threads whose transactions are committed return to normal execution, while those that remain aborted commence another *Registration Phase* (Figure 4.1, phase 3).

Figure 4.2 contrasts our approach with a serializing CMP (like [19] for example). Two hypothetical scenarios are shown, both containing a *depositor* and *withdrawer* transaction accessing shared objects. In scenario 1, the CMP reorders transactions to avoid concurrent conflicts. Although the *withdrawer* transaction can commit, it may need to re-execute in future (if deposits must precede withdrawals, for example). In scenario 2, our approach is illustrated where a semantic abort occurs, and each thread re-executes a different permutation of the aborted transactions.

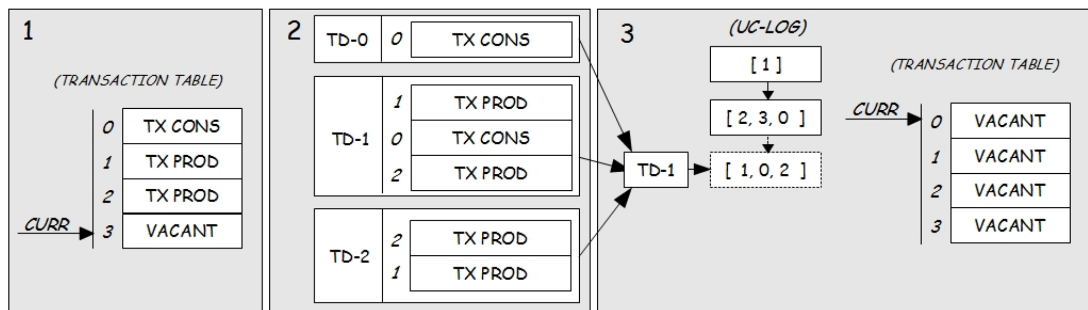

Figure 4.1: Phases of Contention Management *In phase 1, threads add their transactions to the Transaction Table. In phase 2, a thread executes permutations of transactions within the window of the Transaction Table. In phase 3, transactions decide which permutation will be committed, and the result is added to the log of the Universal Construction and the Transaction Table window is advanced.*

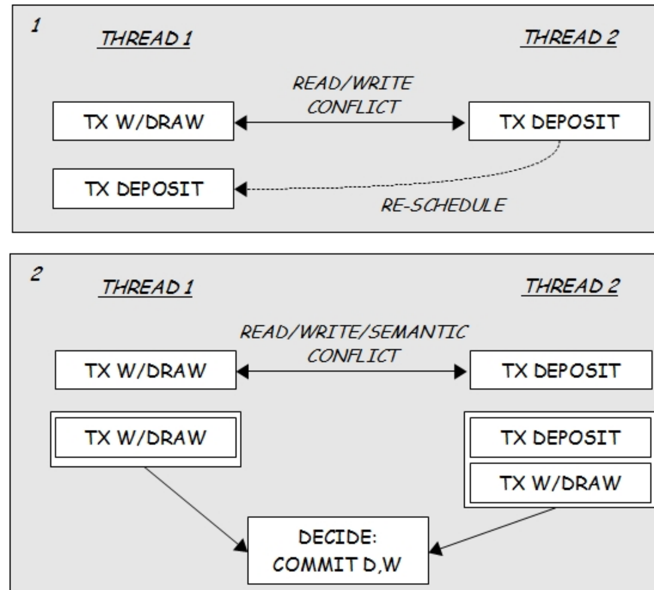

Figure 4.2: Serialising Aborted Transactions *In scenario 1 a read/write conflict has occurred between two transactions called withdraw (w/draw) and deposit. The depositor is aborted and rescheduled to execute after the withdrawer has committed. In scenario 2, Thread 1 aborts the depositor but then also aborts because of a semantic conflict caused by attempting to execute a withdrawal before a deposit. The conflict is resolved by the execution of an ordering which allows both to commit (deposit then withdraw).*

4.1.2 Preliminaries

Whenever a thread has its transaction aborted due to a semantic conflict, it invokes the CallTx function (see Algorithm 1) with the aborted transaction as its argument. The thread then enters a while loop until its transaction has been committed and validated. Within the while loop all the activity of our *UC* takes place, comprising: Registration, Speculation and Commit phases. Before describing each of these activities, we must first define the components of our *UC*:

- A *session* represents a period of execution within our UC and the end of $session_n$ marks the beginning of $session_{n+1}$, therefore no two *sessions* take place concurrently. The *active session* refers to the *session* in which threads are currently executing.

- A *Permutation* is a structure that contains an array of integers and some book-keeping information comprised of integer variables: *commits*, *depth*, *state* and *offset*. Each thread has its own *Permutation* to guide it through its *Speculative Phase*.
- The UC contains a *Log* of past activity as an array of *Permutation* structures. The *Log* has an integer variable *sessionNumber*, initially zero and incremented after each *session* expires (hence *sessionNumber* is equal to the ordinal value of the active *session*).
- The *Transaction Table* holds an array of tuples where a tuple contains a thread's transaction and a Boolean label *occupied*. We refer to each index of the array as a *slot*. The *Transaction Table* has a fixed number of *slots*

Algorithm 1 The CallTx Function

```
1:  function CallTx(txaction)
2:      while true do
3:          if Register(txaction) then
4:              initialise timer
5:              while time remaining do
6:                  call speculation function
7:                  decrement time remaining
8:              end while
9:              call synchronisation function
10:             while ¬(session expired) do
11:                 await session results
12:             end while
13:             if transaction validated then
14:                 reset cache and return
15:             end if
16:         end if
17:         reset cache and handle back-off
18:     end while
19: end function
```

and a *Transaction Table* with n *slots* means that a maximum of n threads can join the *active session*.

- A new *Ticket* is granted to each thread that successfully registers with our UC and is valid for the duration of a single *session*. Each *Ticket* contains a *Permutation*, an integer called *slot*, an integer called *session*, a reference to a cache for the thread's Atomic Object updates, a reference to the *Transaction Table* and a reference to the *Log*.

Data Structures While a *session* is an abstract representation of a time-frame, the *Permutation*, *Log*, *Transaction Table* and *Ticket* are real data structures that support the operation of the *UC*. A *Ticket* is private to a particular thread so we denote this in pseudo-code as $Local_{TICKET}$. Both the cache and *Permutation*, being members of the *Ticket* structure, are denoted $ticket_{CACHE}$ and $ticket_{PERM}$, respectively. Both *Log* and the *Transaction Table*, as members of the *UC*, are denoted: UC_{LOG} and $UC_{TXTABLE}$, respectively. As these can also be referenced via a thread's ticket, however, we will use the syntax $ticket_{LOG}$ and $ticket_{TXTABLE}$ in the pseudo code. Although this double naming my seem superfluous, we shall see in the next section when discussing nested transactions, that this will simplify the pseudo code.

Auxiliary Operations We make use of certain auxiliary operations in the pseudo code, namely CAS GET SET and SWAP. The CAS operation represents *compare-and-swap* which accepts a destination value, an expected value, and a new value respectively; if the destination is equal to the expected value, then the destination is overwritten with the new value and the CAS operation returns true.

Both the GET and SET operations accept a data structure containing an array (e.g. the *Transaction Table*, the *Log* etc) and an integer specifying an index into the array; GET simply retrieves the value at the specified index, and SET accepts a new value which is used to overwrite the value at the specified index. Finally, the SWAP operation accepts a data structure containing an array and two arguments specifying which two members of the array to swap.

4.1.3 Registration Phase

The thread attempts to join the *active session* by calling the REGISTER function (Algorithm 3) with its aborted transaction invocation. The registration algorithm attempts to locate a vacant *slot* in the *Transaction Table* using the synchronization primitive *compare-and-swap* because multiple threads may be attempting to register concurrently (line 8). If the *Transaction Table* has reached maximum capacity then the function returns false and the thread backs off (lines 5-6). Otherwise the thread places its transaction into the *Transaction Table* and sets the occupied flag to true (lines 9-10).

The thread's local *Ticket* is now updated to contain the acquired *slot* number into the *Transaction Table* (line 12), and the value of the *active session* (line 13). In addition, the thread gains a *Permutation* which it will use during its *Speculative Phase* (line 14). The *Permutation*'s integer sequence corresponds to *slots* in the *Transaction Table* and the length of the integer sequence is equal to the length of the *Transaction Table.* The first index of the sequence is equal to the threads acquired index in the *Transaction Table* and the subsequent values of the sequence are comprised of the remaining indices in the *Transaction Table.* For example, if some thread registers and takes the 2*nd* entry into the *Transaction Table* then a valid permutation for that thread would be $\{2, 1, 0, 3\}$ (see Figure 4.1(1)).

Finally, the *Ticket*'s cache is prepared (line 15), references to the *Log* and *Transaction Table* are set (lines 16-17) and the thread's transaction is set to the speculator (line 18). The thread may now proceed to the *Speculative Phase.*

4.1.4 Speculation Phase

Once a thread has registered its aborted transaction, it commences its *Speculative Phase* (for brevity, we shall hereafter refer to these threads as *speculators*). The speculator executes transactions held in the *Transaction Table* with the aim of executing as many transactions to successful completion as possible. While the speculator is executing, new speculators may register (thus causing newly aborted transactions to appear in the *Transaction Table*) and begin their own speculative execution. During the *Speculative Phase*, three conditions related to the execution of speculators must be met, specifically:

Consistency – Exclusivity of atomic objects must be provided to ensure that any speculator's execution of transactions is sequentially consistent and can be committed after the *Speculative Phase*.

Efficiency – Each speculator's execution should explore a unique permutation of transaction execution to reduce the possibility of duplicate speculative exploration.

Termination – All speculators must terminate their *Speculative Phase* and commence their *Commit Phase*.

Maintaining Consistency Speculators do not modify Atomic Objects directly during their *Speculative Phase* because multiple speculators may need to modify

Algorithm 2 Session Registration

1: **function** REGISTER($txaction$)
2: $txtable \leftarrow UC_{TXTABLE}; slot \leftarrow 0$
3: **repeat**
4: $slot \leftarrow txtable_{NEXT}$
5: **if** $slot = txtable_{MAX}$ **then**
6: **return** abort transaction
7: **end if**
8: **until** CAS($txtable_{NEXT}, slot, slot + 1$)
9: SET$(txtable, slot)_{TXACTION} \leftarrow txaction$
10: SET$(txtable, slot)_{OCCUPIED} \leftarrow true$
11: $ticket \leftarrow Local_{TICKET}$
12: $ticket_{SLOT} \leftarrow slot$
13: $ticket_{SESSION} \leftarrow (UC_{LOG})_{CURRENT}$
14: $ticket_{PERM} \leftarrow create\ permutation$
15: $ticket_{CACHE} \leftarrow create\ empty\ cache$
16: $ticket_{LOG} \leftarrow UC_{LOG}$
17: $ticket_{TXTABLE} \leftarrow UC_{TXTABLE}$
18: $Local_{TX} \leftarrow UC_{TX}$
19: **end function**

the same Atomic Objects. Instead, speculators use a private cache to keep copies of any Atomic Objects they use during speculative execution. Essentially, speculators follow the *deferred mode* update model, during their *Speculative Phase*.

To the best of our knowledge, this approach is the first to combine both *direct mode* and *deferred mode* execution in the role of contention management (although SwissTM [23] applies a similar approach where write conflicts are detected when they occur and read conflicts are detected at commit time). It is envisaged that by using *deferred mode*, the possibility of cache bouncing can be significantly reduced (ideally, speculators will execute on different processor cores, so that speculation is performed in parallel to the greatest degree possible).

When Speculator's modify private copies of atomic objects, they must ensure that no active (non-aborted) thread modifies the original object. If the original object is altered then the speculator's execution cannot commit due to the possibility of inconsistent data. Speculators must therefore share exclusive access to any atomic object they 'speculatively update'. To enforce exclusive access, the SERIALIZE function (see Algorithm 3, lines 21-34) is invoked by a Speculator whenever an atomic object is read or written. As speculators cache their updates, exclusivity is not required between speculators.

Active transactions use the *direct update* mode of shared data access which means that an active transaction must install itself as the owner of any object it wishes to modify. In addition to an *owner* field, each atomic object also possesses an integer field denoting its version (*version*) and a reference to a global clock (*clock*). A global transaction called (*spec*) is also provided to denote that an object is currently owned by a speculator. These extra fields enable the possibility of speculator exclusivity. Algorithm 3 shows the pseudo code for gaining ownership of Atomic Objects, in particular:

- The first time an atomic object is accessed by a speculator, it checks whether the object is owned by another speculator (Algorithm 3, line 21). If true, the thread caches a copy of the object and continues its transaction (subsequent accesses modify the copy).
- If the object is not owned by another speculator, the speculator attempts to set ($version = clock + 1$) using *compare-and-swap* (line 24). If the *CAS* call

fails, then another speculator must have already set the *version*. Whichever speculator successfully sets the value of *version* will gain the ownership of the object (line 29).

Before any active thread (executing a non-aborted transaction) tries to install itself as the owner of any atomic object, it first checks whether $version \leq clock$ (Algorithm 3, line 9). If this evaluates to false then the thread knows it must abort because the object is currently being modified by a speculator. Note that by setting the value of *version* before assigning ownership, a speculator eliminates the possibility that another thread can repeatedly prevent the speculator from changing the owner of an atomic object. Once the aborted transactions have been committed, *clock* is atomically incremented so that ($version \leq clock$) is true, and any thread may once again own the object.

Maintaining Efficiency Each speculator uses the Permutation Functions (see Algorithm 4) to access to the *Transaction Table* and modify its *Permutation*. By providing each speculator with a *Permutation* structure (each with an array containing a unique sequence of integers), we ensure each speculator explores a unique permutation of transaction execution, as each integer corresponds to a unique *slot* in the *Transaction Table*.

The TxReady function (lines 1-11) tells the speculator if the next transaction to execute is ready and the NextTx function (lines 12-16) retrieves it from the *Transaction Table*. The PermCommit (lines 17-21) and PermAbort (lines 22-30) functions modify the speculator's *Permutation* on event of the transaction committing and aborting respectively. Each *Permutation* has a *depth* variable which is used to identify the next *slot* into the *Transaction Table* (see lines 2 and 13). The *depth* is increased whenever PermCommit is called (line 19). The permutation's *commits* variable keeps track of how many transactions the thread has committed, increasing on every call of PermCommit (line 18).

When a transaction is aborted and the PermAbort function is called, the *Permutation*'s integer array is modified by a Swap operation (line 28) to ensure that the next time TxReady and NextTx are called, a new *slot* is accessed, and a new transaction is attempted. Each call of PermAbort uses the *depth*

Algorithm 3 Atomic Object Ownership and Consistency

```
1: function OpenWrite(AObj)
2:     me ← Local_TX
3:     if me = UC_TX then
4:         Serialize(AObj)
5:         cache atomic object if not already cached and return
6:     end if
7:     while me_STATE = ACTIVE do
8:         other ← AObj_OWNER
9:         while other_STATE = ACTIVE do
10:            if AObj_VERSION > UC_clock then
11:                Abort(me)
12:            end if
13:            CMResolve(me, other)
14:        end while
15:        if Cas(AObj, other, me) then
16:            return
17:        end if
18:    end while
19:    abort
20: end function

21: function Serialize(AObj)
22:    while AObj_OWNER ≠ UC_TX do
23:        if AObj_VERSION <= UC_CLOCK then
24:            current ← AObj_VERSION
25:            if ¬Cas(AObj_VERSION, current, clock + 1) then
26:                continue
27:            end if
28:        end if
29:        other ← AObj_OWNER
30:        if Cas(AObj_OWNER, other, UC_TX) then
31:            return
32:        end if
33:    end while
34: end function
```

Algorithm 4 The Permutation Functions

1: **function** TXREADY($perm$)
2: $\quad slot \leftarrow$ GET($perm, perm_{DEPTH}$)
3: $\quad ticket \leftarrow Local_{TICKET}$
4: $\quad$ **if** $slot \geq (ticket_{TXTABLE})_{MAX}$ **then**
5: $\qquad$ **return false**
6: $\quad$ **end if**
7: $\quad$ **if** (GET($ticket_{TXTABLE}, slot$))$_{OCCUPIED}$ **then**
8: $\qquad$ **return true**
9: $\quad$ **end if**
10: $\quad$ **return false**
11: **end function**

12: **function** NEXTTX($perm$)
13: $\quad slot \leftarrow$ GET($perm, perm_{DEPTH}$)
14: $\quad ticket \leftarrow Local_{TICKET}$
15: $\quad$ **return** (GET($ticket_{TXTABLE}, slot$))$_{TXACTION}$
16: **end function**

17: **function** PERMCOMMIT($perm$)
18: $\quad perm_{COMMITS} \leftarrow perm_{COMMITS} + 1$
19: $\quad perm_{DEPTH} \leftarrow perm_{DEPTH} + 1$
20: $\quad perm_{OFFSET} \leftarrow 1$
21: **end function**

22: **function** PERMABORT($perm$)
23: $\quad slot \leftarrow perm_{DEPTH} + perm_{OFFSET}$
24: $\quad$ **if** $slot \geq perm_{MAX}$ **then**
25: $\qquad perm_{STATE} \leftarrow expended$
26: $\qquad$ **return**
27: $\quad$ **end if**
28: $\quad$ SWAP($perm, perm_{DEPTH}, slot$)
29: $\quad perm_{OFFSET} \leftarrow perm_{OFFSET} + 1$
30: **end function**

variable plus the *offset* variable, to find the next value to swap (line 23). The *offset* is then incremented (line 29). When there are no more transactions left to access, the value of *depth* plus *offset* exceeds the length of the *Permutation* array, and the *Permutation*'s state is changed to *expended* (lines 24-26).

Algorithm 5 The Greedy Algorithm

1: **function** GreedyExpand
2: $perm \leftarrow (Local_{TICKET})_{PERM}$
3: **if** $\neg$txReady($perm$) **then**
4: **return**
5: **end if**
6: $txaction \leftarrow$ nextTx($perm$)
7: CheckPoint($(Local_{TICKET})_{CACHE}$)
8: **if** call($txaction$) $=$ *abort* **then**
9: RollBack($(Local_{TICKET})_{CACHE}$)
10: permAbort($perm$)
11: set time remaining to 0 and **return**
12: **end if**
13: permCommit($perm$)
14: save the current best permutation
15: **end function**

Greedy Speculation A Greedy Speculation Algorithm (Algorithm 5) is provided to show how speculators use the Permutation Functions to execute transactions during their *Speculative Phase*. The Greedy Algorithm ends the speculator's *Speculative Phase* as soon as an aborting transaction is encountered by setting the remaining speculation time to zero (line 11).

Observe that the Greedy Algorithm uses two functions called CheckPoint and RollBack. The CheckPoint function creates a checkpoint of the speculator's cache (line 7), and the RollBack function restores the contents of the cache to the previous checkpoint (line 9). A simple (albeit inefficient) way to implement these functions requires the use of a stack (per speculator). Each CheckPoint invocation pushes a copy of the speculator's current cache on the

stack, and each RollBack invocation pops the head of the stack. A speculator can always access the current cache by accessing the head of the stack.

Terminating Speculation To ensure the *Speculative Phase* is bounded with respect to execution time, each speculator initializes a *timer* (a signed integer) at the beginning of its *Speculative Phase* which holds the maximum number of times the Greedy Algorithm will be invoked (see Algorithm 2, lines 4-8). The speculator then decrements *timer* on each iteration of the Greedy Algorithm. When *timer* reaches zero, the thread moves to its *Commit Phase*. The possible conditions under which the value of *timer* reach zero are:

- the first time a transaction aborts;
- all transactions have been executed successfully (hence the state of the speculator's *Permutation* is *expended*);
- the number of times the speculator has invoked the Greedy Algorithm is equal to the initial value of the specuator's *timer*.

The precise initial value used for the *timer* variable is equal to the capacity of the *Transaction Table*. Thus every speculator will advance to its *Commit Phase* in at most n calls to the Greedy Algorithm (where n is equal to the capacity of the *Transaction Table*). Experimentation with the value of *timer* is possible of course. For example, the value of the *timer* may be increased should one desire a longer *Speculative Phase* (during testing, extending the duration of the *Speculative Phase* had no noticable improvement with respect to transactional throughput however).

A suggested additional condition for the termination of the *Speculative Phase* is when the speculator is context switched by the Operating System given that the speculator will likely be unable to execute for a significant duration of time. In programming languages such as *C#* and *Java* for example, this can be detected if the speculator catches a *thread interrupted exception* or equivalent.

4.1.5 Commit Phase

Once a (speculating) thread has entered its *Commit Phase*, if one or more transactions where successfully executed then the thread's cache will contain one or more modified atomic objects. The thread must now determine whether those

Algorithm 6 Session Synchronization

```
1: function SYNCHRONIZE(tree)
2:     ticket ← Local_TICKET
3:     last ← CLOSE(ticket_SLOT, ticket_TXTABLE)
4:     challenge ← make a commit challenge node
5:     challenge_LAST ← last
6:     repeat
7:         other ← get next parent node from tree
8:         if ¬CONTEST(challenge, other) then
9:             return
10:        end if
11:        reset the other commit challenge node
12:    until other is the root node
13:    POSTSESSION(ticket_LOG, challenge)
14:    OPEN(ticket_TXTABLE)
15: end function

16: function CLOSE(slot, table)
17:    if CAS(table_NEXT, slot + 1, table_MAX) then
18:        return true
19:    end if
20:    return false
21: end function

22: function OPEN(table)
23:    set each occupied flag in the table to false
24:    CAS(table_NEXT, table_MAX, 0)
25: end function
```

changes can be committed or whether they should be discarded. By reaching consensus with the other threads of the *active session*, the thread can determine whose cache will be committed. (As each thread's cache contains a potential future state of shared data, if more than one cache is committed then consistency of shared data will be infringed.)

To achieve consensus we use an approach based upon a Combining-Tree method described in Herlihy [35] which aims to provide higher throughput when multiple threads must reach agreement. The idea behind the Combining-Tree approach is to coordinate thread communication using a Binary-Tree. Beginning at the leaf-nodes, threads interact in pairs at each node of the Binary-Tree performing some combining operation. One thread then continues to the parent node while the second waits for the remaining threads to complete the algorithm. Once the root node has been evaluated, a single thread remains in a non-waiting state and the waiting threads discover the result of the combining operation.

Herlihy's Combining-Tree works with a fixed number of threads. For our purposes we require an algorithm that can accommodate a varying number of threads, given that we do not know how many threads have registered and will need to synchronize. The SYNCHRONIZE function defines the process of reaching consensus (see Algorithm 7). Specifically, the SYNCHRONIZE function comprises of three steps:

1. An initial step 'closes' the active session, to limit the number of threads that can take part in the synchronization algorithm.
2. An iterative evaluation is then performed at each node of the Binary Tree where on each iteration, a thread determines if it must wait or proceed towards the root node.
3. A final step (executed by a single thread) terminates the active session and notifies all waiting threads that they may commence their validation.

The SYNCHRONIZE Algorithm also requires access to a Binary Tree structure, with the following properties:

- The Binary Tree contains $n-1$ nodes where n is equal to the number of *slots* within the *Transaction Table*;

- Each node of the tree is called a 'commit-challenge' node and contains: a reference to a thread's *Ticket*, a boolean called *last*, a integer called *state* (the value of which is either *reset*; *writing* or *done*) and an enumerated position (either *left*, *right* or *root*). The *last* and *position* variables are used to dynamically determine the number of threads participating in consensus.

Once the winning cache is found, whichever thread commits the cache is immaterial (for instance, if $thread_A$ commits the cache of $thread_B$ then $thread_B$ has effectively still committed). Commit-challenge nodes therefore contain a reference to a thread's *Ticket*. Given that a thread's *Ticket* holds a reference to a thread's cache, the thread that locates the winning *Ticket* can proceed to commit the contents of the cache held therein.

Having defined the preliminaries, we may now describe the three steps required for synchronisation.

Step One Each synchronizing thread (hereafter we shall refer to these threads as synchronizers) begins by attempting to prevent any new threads from registering with the UC. Each synchronizer calls the CLOSE function (Algorithm 7, line 3) which attempts to set the current variable of the *Transaction Table* to its maximum capacity (line 17). By using *compare-and-swap*, only one synchronizer will successfully close the table and this shall be the the last synchronizer to have registered. The return value of the CLOSE function is used by each synchronizer when they each construct a commit challenge (recall that each commit challenge node contains a flag called last).

Step Two Each synchronizer begins at a leaf node of the binary-tree and ascends until either: (i) the synchronizer determines it must wait or (ii) the root is reached. At each node, synchronizers call the CONTEST function (see Algorithm 8), which compares a synchronizer's current commit-challenge node with its parent node and returns false if the synchronizer must wait. At most one other

synchronizer will also contest the parent node, and so the CONTEST function must resolve which synchronizer is first to access the parent node.

It is possible that in the special case of the last synchronizer to register, only a single synchronizer will visit certain nodes of the tree, and so each synchronizer begins the CONTEST function by determining if an additional synchronizer will visit the parent node (lines 2-5). If the contesting synchronizer was the last to register and its current node's position relative to the parent is to the left, then the synchronizer knows that no additional synchronizer will visit the parent node

Algorithm 7 The Contest Algorithm

1: **function** CONTEST($mine, other$)
2: **if** $mine_{LAST} \wedge (mine_{POS} = left)$ **then**
3: $mine_{POS} \leftarrow other_{POS}$
4: **return true**
5: **end if**
6: **if** CAS($other_{FLAG}, reset, writing$) **then**
7: $other_{TICKET} \leftarrow mine_{TICKET}$
8: $other_{LAST} \leftarrow (other_{LAST} \vee mine_{LAST})$
9: CAS($other_{FLAG}, writing, done$)
10: **return false**
11: **else**
12: **while** $\neg$CAS($other_{FLAG}, done, reset$) **do**
13: **end while**
14: **end if**
15: $mPerm \leftarrow (mine_{TICKET})_{PERM}$
16: $oPerm \leftarrow (other_{TICKET})_{PERM}$
17: **if** $oPerm_{COMMITS} > mPerm_{COMMITS}$ **then**
18: $mine_{TICKET} \leftarrow other_{TICKET}$
19: **end if**
20: $mine_{LAST} \leftarrow (other_{LAST} \vee mine_{LAST})$
21: $mine_{POS} \leftarrow other_{POS}$
22: **return true**
23: **end function**

and so it continues immediately to the next parent node.

We now turn to the case where a node is visited by two synchronizers; let us call them syn_A and syn_B. Each syn_X begins by attempting to lock the node by changing the node's state flag (line 6). If syn_A successfully locks a node for writing that means it must be the first synchronizer to visit this node; so syn_A posts its challenge data into the node, unlocks the node, and the CONTEST function returns false (lines 7-10); syn_A now waits for the synchronization algorithm to complete. If syn_A cannot successfully lock the node, that means another synchronizer, syn_B, must have already visited this node or is in the process of posting its details. In such a case, syn_A waits until the node is unlocked (lines 12-13), and then compares its commit-challenge with the challenge that has already been posted. If syn_A has a better challenge, it resets the node and carries onto the parent node with its own challenge (lines 20-22); otherwise it carries onto the parent node with the posted challenge of syn_B (lines 17-19). Note that the algorithm ensures that the node containing a reference to the cache containing the most committed transactions always ascends towards the root node.

Step Three Whichever synchronizer contests the root node and returns true locates the winning cache of each participant in the Consensus algorithm. The reference to the winning synchronizer's *Ticket* is held at the root node. The remaining synchronizer invokes the POSTSESSION function (see Algorithm 8), to commit the contents of the winning cache and post the winning *Permutation* (held in the winning *Ticket*) into the *Log*. Updating the *Log* acts as a signal to the waiting synchronizers that the *active session* has expired.

The remaining synchronizer can commit the cache by calling the COMMIT function on every Atomic Object within the cache. In order to end the *active session*, the committing synchronizer takes the winning *Permutation* from the *Ticket* of the winning challenge, and copies it onto the end of *Log*. The synchronizer then increments the *sessionNumber* variable of *Log* using an atomic incrementing function, thus ending the *active session*. When the waiting synchronizers see that their own *session* number is no longer the same as the *Log session* number, they know their *session* has expired.

Finally the committing synchronizer calls the OPEN function (Algorithm 7, lines 22-25) to open the *Transaction Table* by setting each occupied entry to false, and then atomically setting the *Transaction Table*'s current variable to zero, thus allowing new threads to once again register with the UC and execute a new session.

Algorithm 8 Updating the UC Log

```
1: function PostSession(log, challenge)
2:     cache ← (challenge_TICKET)_CACHE
3:     for i ← 0, cache_MAX do
4:         AObj ← Get(cache, i)
5:         Commit(AObj)
6:     end for
7:     Set(log, log_NEXT) ← (challenge_TICKET)_PERM
8:     next ← log_NEXT
9:     Cas(log_NEXT, next, next + 1)
10: end function
```

4.1.6 Validation

Each participant of a *session* must validate whether its transaction was committed. Threads perform validation once the *Log* has been updated (via the POST-SESSION function) and the thread's *session* has expired. A validating thread first retrieves the *session* and *slot* variables held in its *Ticket*; these will be used to locate the results of the thread's *session*. The validating thread then accesses the *Log* at the index equal to its *session* variable and retrieves the *Permutation* data held therein. The *Permutation* contains a list of *slot* numbers from those threads whose transactions were successfully committed. In order to validate, the thread reads the *commits* variable of the *Permutation* structure to discover how many *slots* where successful. The thread then searches for its own *slot* number within the *Permutation*, until the number of search iterations exceeds the value of *commits*.

If the validating thread finds its *slot* number in the winning *Permutation* then its transaction has been executed and committed and it may proceed with its own

future execution. If the validating thread's *slot* number is not present, the thread knows its transaction did not take place. The thread must now attempt to retry its transaction by clearing its local cache and attempting to register with a new *session*.

Depending on the length of the winning *Permutation* of transaction execution, zero or more transactions can be committed during a single *session*. Atomicity is still respected, however, because each transaction is executed in its entirety during the *Speculative Phase*.

4.2 Managing Nested Transactions

This section presents additional functionality, which allows exploration of nested transactions during the process of contention management. From the perspective of the application developer, nested transactions facilitate the task of writing complex transactions (just as conventional function syntax facilitates the task of writing complex programs). In particular to this thesis, the benefit of accommodating nested transactions is that a richer set of transaction permutations can be explored during the process of contention management. It is anticipated that a more thorough exploration of transaction permutation will further enhance the ability of our approach to resolve semantic conflicts.

4.2.1 Speculative Nesting

Moss and Hosking [36] provided a reference model for nested transactions, which describes three variations on nested transactions, namely: Flattened, Open and Closed nesting. Conventional Transaction Managers may support one or more of these models (see Section 2.4.4 for an overview). The speculative and exploratory nature of transactions execution within our implementation gives rise to their interleavings which are not covered by the existing nested models, even though such interleavings may be valid according to the criterion of linearisability. Therefore in this section we define a new nesting model called *Speculative Nesting*. This is followed by an example, where *Speculative Nesting* provides benefits over existing models.

The rules of *Speculative Nesting* are as follows:

1. Unlike Flattened, Open or Closed nesting, any transaction (including a nested one) can observe the changes made by any ancestor transaction at the point when the nested transaction was called.

2. As with Open Nesting, *Speculative Nesting* allows the changes of a nested transaction to become visible to any other transactions, once it has committed.

3. Like a flattened transaction, if a nested transaction aborts, then the parent transaction will also abort;

4. Any transaction which has observed changes made by an aborting transaction will also abort. Hence, if the nested transaction (and therefore the parent transaction) aborts then any other transaction which observed the changes made by the aborting transaction will also abort.

Rules 1 and 2 of the *Speculative Nesting* model incur a 'temporary weakening' of the Isolation property, because any speculator can observe changes to shared data before the parent transaction has effectively committed. Rules 3 and 4, however, ensure that the weakening of Isolation does not lead to erroneous executions. When transactions are validated, the ACID properties are maintained because only permutations containing 'atomic blocks' of successful transactions are committed.

Example Let us consider a scenario where *Speculative Nesting* would be advantageous. We shall consider potential permutations of transaction execution that could result from the transactions of Algorithm 9. To simplify matters, let us first assume that the PRODUCER and PRODNEST transactions execute separately, and that the following sequence of transactions were executed:

$$\text{Producer}; \text{Consumer}; \text{ProdNest} \tag{4.1}$$

Observe that this represents a valid linearizable schedule given that (i) a sequential execution could produce the same schedule, and (ii) the semantics of

the shared list are maintained. However, neither of Flattened, Open or Closed nesting models would allow the ProdNest transaction to commit if executed as a nested transaction. This is because the Producer adds an item to a shared list (line 5), which in turn ensures that the conditional statement of the ProdNest transaction evaluates as false (line 10). Given that the statements of Producer and ProdNest will execute atomically within a transaction, no change to the list can be made between these statements by another thread.

A seemingly simple solution to committing ProdNest would be to execute

Algorithm 9 Nested Transactions

```
 1: transaction Producer(item)
 2:     if list_COUNT ≥ 1 then
 3:         abort transaction
 4:     end if
 5:     add(list, item)
 6:     CallTx(ProdNest)
 7:     transaction successful
 8: end transaction

 9: transaction ProdNest
10:     if list_COUNT ≥ 1 then
11:         abort transaction
12:     end if
13:     transaction successful
14: end transaction

15: transaction Consumer
16:     if list_COUNT = 0 then
17:         abort transaction
18:     end if
19:     remove(list)
20:     transaction successful
21: end transaction
```

ProdNest as a separate transaction. Consider how detrimental this would be to performance, however, because ProdNest can only commit if it executes after the Consumer transaction (hence, a very specific schedule of transaction execution is required to allow collaborative semantic success).

The *Speculative Nesting* approach provides greater flexibility. It is more likely that a Consumer will execute between the Producer and ProdNest transactions, as this would constitute a valid permutation which can be explored with *Speculative Nesting*. As long as both the Producer and Consumer threads register in the same *session*, this permutation may be found. One may begin to speculate about possible situations where such expressiveness may be utilised. For instance, the relation of the ProdNest transaction to the Producer transaction is one where the nested transaction is used to provide a signal to the parent (indicating when a Consumer has executed). The benefit of *Speculative Nesting*, therefore, is that such complex coordination can be explored during the process of contention management, removing thus the burden from the application programmer.

Speculative Nesting Limitations Speculative Nesting presents a problem with respect to the property of serializability, where transactions may observe inconsistent states which may in turn generate exceptions. For example, suppose the permutation of transaction(4.1) is executed, and suppose the Producer transaction checks the validity of a shared memory pointer within its transaction. Now assume that the Consumer transaction subsequently nullifies that shared memory pointer. If the ProdNest transaction assumed atomicity since the beginning of the Producer transaction, it might dereference that shared pointer without first checking the reference. Essentially, as with the Open Nesting model, the ProdNest can no longer assume that the code of the Producer transaction has been executed before the ProdNest transaction began.

As discussed by Yang et al [37], the weakening of atomicity (and the dangers associated with this practise) is also a characterisation of Open Nesting. Yang describes the addition of several procedures that must be supplied to a transactional memory system to ensure safe execution. Although not explored within this the-

sis, a necessary avenue for future work is the development of similar mechanisms to ensure safety with Speculative Nesting.

4.2.2 Overview

We begin with an overview, describing the extra functionality required to accommodate nested transactions. We use an example comprising of the PRODUCER and PRODNEST transactions shown in Algorithm 9. Firstly, observe that PRODNEST is a nested transaction within the PRODUCER transaction (line 6). When the EXECUTE function is invoked the thread performs registration (just as it does with the basic implementation described in Section 4.1). During the thread's *Speculative Phase*, it executes the PRODUCER transaction. Once CALLTX is invoked with the PRODNEST transaction, control is delegated to the Nesting Manager (hereafter abbreviated to NM) so that the nested transaction can be executed.

The functions of the NM regulate the resources necessary to accommodate nesting exploration. Threads wishing to explore nested transaction execution must first acquire a resource from the NM. If there are no resources available, the PRODNEST transaction is executed as a *flattened* transaction during its *parent session* (i.e. the nesting is ignored and both transactions are treated as a single transaction). If a resource is granted by the NM, a *child session* begins, and the PRODUCER thread begins a nested *Speculative Phase*, *Commit Phase* and *Validation Phase*. Figure 4.3 shows how *child sessions* begin and end within the confines of a *parent session*, while Figure 4.4 illustrates the three phases of execution with the addition of a *child session*.

In the thread's *child session*, its *Speculative Phase* explores permutations of transaction execution consisting of the PRODNEST transaction and any other transactions that have been added to the *Transaction Table*. Synchronization and Validation within a *child session* allow the thread to identify the optimum permutation of the *child session*. Control then returns to the *parent session*, and the resources of the *child session* are returned to the NM. The shared state modifications made during nested transaction execution, however, are retained in a thread's cache when it returns from a *child session*. If the thread commits

the contents of its cache then the changes will include modifications made in any nested transactions the thread executed.

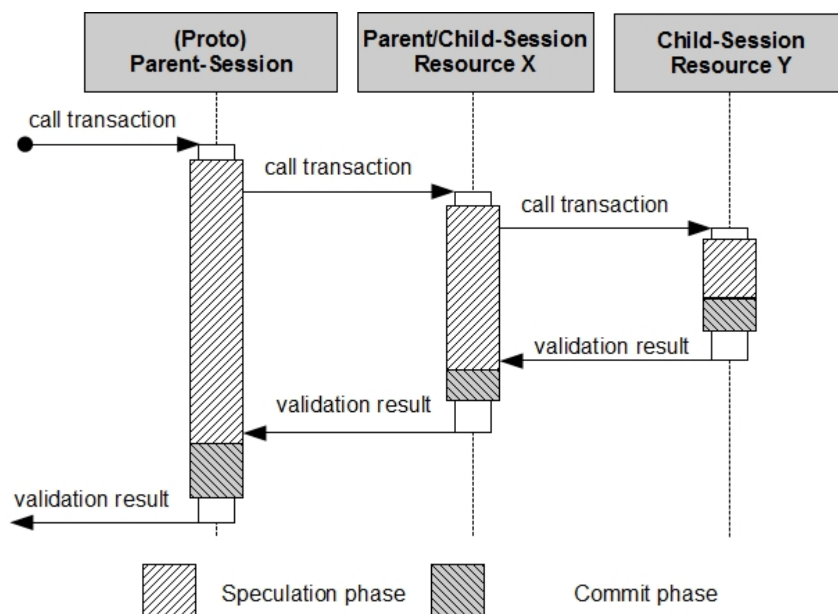

Figure 4.3: Child Sessions *Nested Transactions prompt the creation of child sessions when the* EXECUTE *function is called within a transaction and a resource is subsequently acquired. Permutations of transactions are explored with the nested transactions, and further child sessions may be created if resource availability permits. Once a child session has ended, the result is returned to the parent session.*

4.2.3 Data Structures

Before we describe the process of creating and terminating child sessions, let us first define several data structures which enable transactional nesting:

- *Table Masks* provide proxies for the *Transaction Table* and the *Log* structures, whenever a thread executes within the context of a *child session.*
- The NM possesses a finite collection of *resources.* A single *resource* is defined as a *Table Mask* and *Binary Tree* pair (*Binary Trees* are used to perform Consensus as explained in Section 4.1.5).

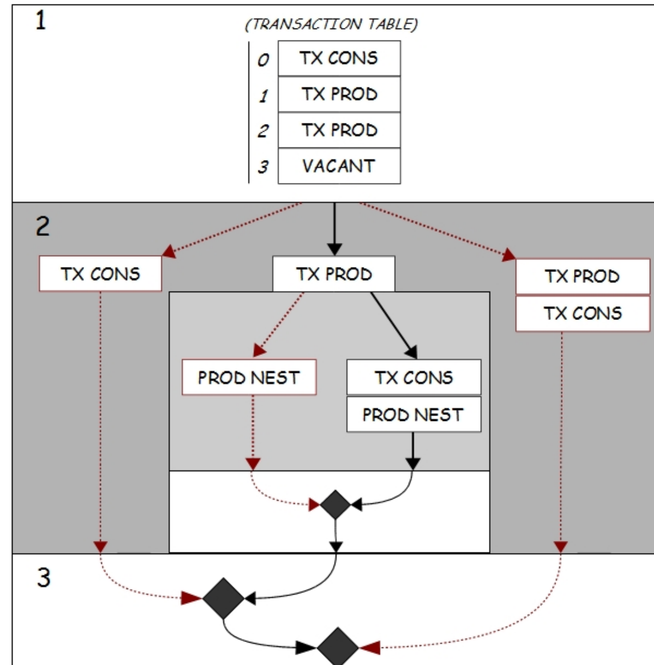

Figure 4.4: Nested Transaction Execution *Phase 1 shows the Transaction Table containing three transactions. In Phase 2 Threads perform speculative execution and in Phase 3, threads reach consensus to determine the optimum execution. A child session is shown, prompted by the execution of the TX PROD transaction, wherein permutations of the PROD-NEST and TX-CONS transactions are explored before the results are synchronized and returned to the parent session. In this hypothetical session, the permutation of committed transactions is shown in black arrows and consists of TX PROD; TX-CONS; PROD-NEST.*

- A *resource stack* is required to regulate access to the *resources*. The *resource stack* is simply a concurrent stack which holds integers. Concurrent stacks allow thread-safe access operations and feature in several programming language libraries (hence we do not describe the implementation of the concurrent stack here).

- A *Ticket* stack is now provided to each thread, rather than a single *Ticket*. Hence CHECKPOINT and ROLLBACK functions are required. CHECKPOINT creates a copy of the head of the stack and replaces the head with the copy. ROLLBACK pops the head of the stack to restore the previous

Ticket.

Resources and Resource Ownership *Resources* are stored in (i) an array of *Table Masks*, and (ii) an array of *Binary Trees*. The size of both arrays is uniform and assumed to be set a priori, depending on the anticipated requirements. The *resource stack* allows threads to concurrently 'pop' an integer when a resource is required, and 'push' an integer when the resource is no longer needed. The value of the 'popped' integer identifies the index into the *Table Mask* and *Binary Tree* arrays.

We say that a thread which successfully 'pops' an integer i is the *owner* of $resource_i$ for the duration of its *child session*. When the same thread pushes the integer i back onto the *resource stack*, it relinquishes ownership of $resource_i$ (thus making $resource_i$ available to other threads). In order to execute a *child session*, a thread must first acquire ownership of a unique *resource*, given that *child sessions* incur demands for memory (without some constraining mechanism, memory could be exhausted by the occurrence of excessive nested transaction calls).

Table Masks A single *Transaction Table* is not sufficient to allow permutations of nested transactions to be explored, because the *Transaction Table* may only hold a single transaction per thread. Therefore, the *Table Mask* structure is provided to enable nested transaction execution. The addition of the *Table Mask* structure fulfills two requirements:

- The *Table Mask* exposes the same interface as the *Transaction Table* and *Log* structures, essentially acting as a proxy for both these structures during nested transaction execution.
- The *Table Mask* stores nested transaction invocations, essentially 'masking' a specific entry in the *Transaction Table* to reflect the context of the *child session.*

This relationship between the *Transaction Table* and *Table Masks* is illustrated in Figure 4.5: Observe that from the perspective of threads X and Y, the

PRODUCER transaction invocation in the *Transaction Table* has been 'masked' by the PRODNEST transaction invocation in the *Table Mask*.

In Section 4.1.4, we described how speculators invoke the Permutation Functions during their *Speculative Phase* to retrieve transactions from the *Transaction Table*. When executing a *child session*, speculators first attempts to retrieve a transactions from the *Table Mask*. Transactions are retrieved via the GET function (see Algorithm 10). If the requested transaction is a 'masked' transaction (PRODNEST for instance) then the transaction is retrieved from the *Table Mask* (line 3). Otherwise, the *Table Mask* retrieves the request from its parent table (line 5). To support this functionality, each *Table Mask* contain a reference to a parent table, which may either be another *Table Mask* or the *Transaction Table*. When new *child sessions* are created from within existing *child sessions* (i.e. when a nested transaction is called from within another nested transaction), a chain of GET invocations may be made, ascending the table hierarchy until the *Transaction Table* is reached.

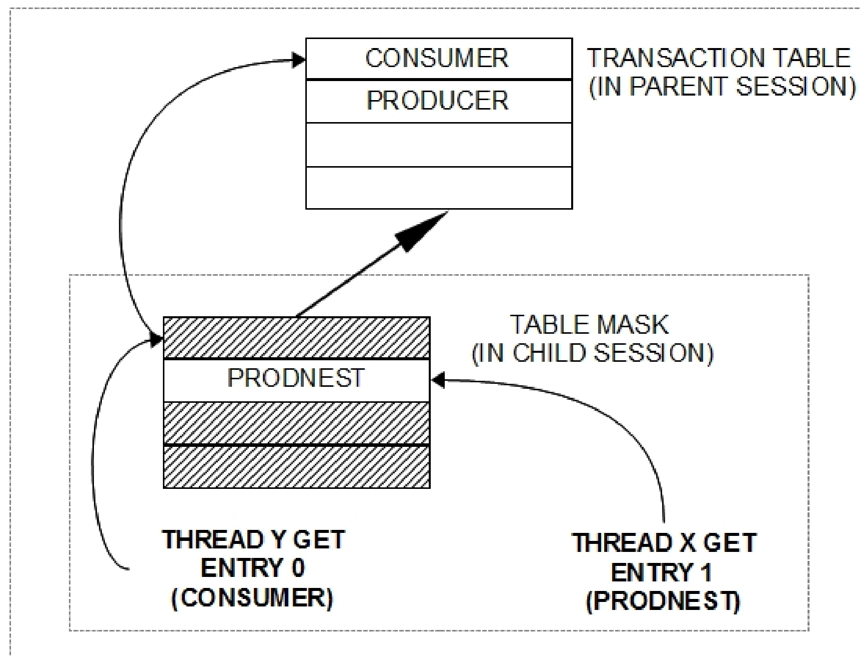

Figure 4.5: The Table Mask Structure

4.2.4 Child Session Management

The execution of a *child session* begins whenever a thread detects that it has called a nested transaction (i.e the argument to the CallTx function is a nested transaction invocation). Each *child session* is then comprised of the following activities:

1. The thread invokes the ExecuteNested function, which contains the logic necessary to execute a *child session.* The ExecuteNested function first attempts to acquire a *resource.*
2. If a *resource* is acquired, the RegisterNested function is invoked to prepare the necessary data structures for the execution of a *child session.*
3. Once registered, the thread executes a nested *Speculative Phase* and *Commit Phase* (in the same manner as described in Section 4.1.4). The ValidateNested function is then called, which terminates the *child session.*

A modification to the CallTx function is required to identify whether the argument supplied to CallTx is a nested or non-nested transaction (see Algorithm 11). All threads possess a private variable to hold their current state. When a thread is first initialised, the variable is assigned the value *ready.* After a thread has called the CallTx function, the variable is set to *registered* (see line 7), and once validation has been performed, the variable is reset to the value *ready* (line 18). If the CallTx function is invoked while the state variable holds the value *registered* then the thread knows that the argument to CallTx is a nested transaction and the ExecuteNested function is invoked (lines 2 and 3).

Algorithm 10 The Table Mask Get Algorithm

1: **function** GET($mask, slot$)
2:     **if** $slot = mask_{SLOT}$ **then**
3:         **return** $mask_{TXACTION}$
4:     **end if**
5:     **return** GET($mask_{PARENT}, slot$)
6: **end function**

Whenever a thread invokes the EXECUTENESTED function (see Algorithm 12), it first attempts to acquire a *resource* (line 2). If there are no *resources* available then the thread simply executes the transaction as a flattened transaction (line 15). Alternatively, if the thread acquires a *resource* then it gains exclusive access to the *Table Mask* and *Binary Tree* referenced by that *resource*. The *Table Mask* is used during the thread's *Speculative Phase* to retrieve transaction invocations. The *Binary Tree* is used during the thread's *Commit Phase* (lines 8-9).

Algorithm 11 The New Execute Algorithm

```
 1: function CallTx(txaction)
 2:     if Local_STATE = registered then
 3:         return executeNested(txaction)
 4:     end if
 5:     while true do
 6:         if Register(txaction) then
 7:             Local_STATE ← registered
 8:             initialise timer
 9:             while time remaining do
10:                 call expansion function
11:                 decrement time remaining
12:             end while
13:             synchronize(TM_bintree)
14:             while ¬(session expired) do
15:                 await session results
16:             end while
17:             if transaction validated then
18:                 Local_STATE ← done
19:                 reset cache and return
20:             end if
21:         end if
22:         reset cache and handle abort
23:     end while
24: end function
```

Nested Registration The REGISTERNESTED function allows a thread to begin its nested *Speculative Phase* once a *resource* has been acquired (see Algorithm 13). Firstly, the thread retrieves the *Table Mask* structure from its acquired *resource* (line 2), and the *Table Mask*'s parent entry is set to the thread's current table (line 3). Next, the thread creates a checkpoint of its *Ticket* structure (line 4) by invoking the CHECKPOINT function. This allows the thread to make changes to its *Ticket* within the *child session*, which can be undone if the *child session* cannot commit. The thread then resets its *slot* and *session* variables before setting up its table references to point to the *Table Mask* (lines 7-10).

The remaining statements of the REGISTERNESTED function (lines 11-13) are required to support the addition of *Pseudo Threads*, which are explained in Section 4.3.2.

Algorithm 12 The Nested-Execute Algorithm

```
 1: function ExecuteNested(txaction)
 2:     if resource ← pop(NM_CSTACK) then
 3:         RegisterNested(resource, txaction)
 4:         while time remaining do
 5:             call expansion function
 6:             decrement time remaining
 7:         end while
 8:         bintree ← get(NM_TREES, resource)
 9:         synchronize(bintree)
10:         while ¬(session expired) do
11:             await session results
12:         end while
13:         return ValidateNested(resource)
14:     else
15:         return call(txaction)
16:     end if
17: end function
```

Child Session Validation Once a thread has executed a nested *Speculative Phase* and *Commit Phase*, it invokes the ValidateNested function (Algorithm 14). The ValidateNested function begins with the thread invoking the RollBack function to store the head of its *Ticket stack* in a thread-private variable called Head (line 2). Next, the thread relinquishes its *resource* by pushing it back onto the concurrent stack of the NM. The *resource* may now be used by any other threads wishing to conduct nested transactions (line 3).

The thread must now determine whether the speculative changes it made to any atomic objects during its *child session* will be kept or discarded. Retaining those changes depends on whether the thread's child transaction was able to commit during the execution of the *child session*. This can be determined from the *Permutation* saved in the Head variable, specifically:

- If the child transaction was not committed then the thread immediately returns from the ValidateNested function with the value *abort* (line 10).
- Otherwise, the thread's cache of atomic object modifications is updated with the modifications made during the *child session*. The state of the NM

Algorithm 13 Commencing Nested Execution

1: **function** RegisterNested($resource, txaction$)
2: $mask \leftarrow \text{GET}(NM_{MASKS}, resource)$
3: $mask_{PARENT} \leftarrow (Local_{TICKET})_{TXTABLE}$
4: CheckPoint($Local_{TICKET}$)
5: $ticket \leftarrow Local_{TICKET}$
6: $depth \leftarrow (ticket_{PERM})_{DEPTH}$
7: $mask_{SLOT} \leftarrow \text{Get}(ticket_{PERM}, depth)$
8: $ticket_{SLOT} \leftarrow ticket_{SESSION} \leftarrow 0$
9: $ticket_{TXTABLE} \leftarrow ticket_{LOG} \leftarrow mask$
10: $mask_{TXACTION} \leftarrow txaction$
11: $mask_{CACHE} \leftarrow \text{Copy}(ticket_{CACHE})$
12: $mask_{PERM} \leftarrow \text{Copy}(ticket_{PERM})$
13: $\text{Cas}(mask_{NEXT}, 0, 1)$
14: **end function**

is set to *validated*, and the value *commit* is returned from the VALIDATENESTED function (lines 6-8).

The return value of the VALIDATENESTED function will, in turn, determine the status of the parent transaction (and may cause the parent to abort). If a nested transaction aborts then aborting the parent transaction ensures that the *atomicity* property is maintained.

Algorithm 14 Ending Nested Execution

1: **function** VALIDATENESTED($resource$)
2: $Local_{HEAD} \leftarrow$ ROLLBACK($Local_{TICKET}$)
3: PUSH($NM_{CSTACK}, resource$)
4: $new\ cache \leftarrow ((Local_{HEAD})_{TICKET})_{CACHE}$
5: **if** child transaction committed **then**
6: $(Local_{TICKET})_{CACHE} \leftarrow new\ cache$
7: $Local_{NMSTATE} \leftarrow validated$
8: **return** commit
9: **end if**
10: **return** abort
11: **end function**

Once the nested transaction has executed, thread execution returns to the speculative algorithm (i.e. the Greedy Algorithm) in the *parent session* context. If the remainder of the parent transaction is not aborted then the PERMCOMMIT function will be called. The PERMCOMMIT function must now be able to handle cases when a nested transaction has successfully committed, so that the thread's *Permutation* is updated correctly (see Algorithm 15, lines 1-9). Specifically, the PERMCOMMIT function reads the state of the NM, and if this is *validated* then the COMMITNESTED function is invoked (recall that the VALIDATENESTED function sets the status of the NM to *validated* if a nested transaction successfully commits). If this extra functionality was not present then a thread may subsequently execute transactions in its *parent session*, which have already been executed during a previous *child session*.

The COMMITNESTED function is shown in Algorithm 15 (lines 10-17). In the COMMITNESTED function, the thread updates its own *Permutation* structure

with the *Permutation* saved in the HEAD variable. This comprises overwriting the thread's current *Permutation* with the *Permutation* from the previously executed *child session* (line 14). In addition, the *commit count* is adjusted to include the number of transactions committed during the execution of the *child session* (line 12). The final required action is to reset the state of the NM to *ready*, so that further calls of PERMCOMMIT do not erroneously call COMMITNESTED without first executing a new *child session* (line 16).

Algorithm 15 The New Permutation Commit Algorithm

```
 1: function permCommit(perm)
 2:     if Local_NMSTATE = validated then
 3:         CommitNested(perm)
 4:     else
 5:         perm_COMMITS ← perm_COMMITS + 1
 6:         perm_DEPTH ← perm_DEPTH + 1
 7:         perm_OFFSET ← 1
 8:     end if
 9: end function

10: function CommitNested(perm)
11:     newperm ← (Local_HEAD)_PERM
12:     newperm_COMMITS ← newperm_COMMITS + perm_COMMITS
13:     newperm_OFFSET ← 1
14:     perm ← newperm
15:     (Local_TICKET)_CACHE ← (Local_HEAD)_CACHE
16:     Local_NMSTATE ← ready
17: end function
```

4.3 Nested Search Strategies

In this section we describe two search strategies to enhance the exploratory potential during nested transaction execution. In addition to the Greedy Algorithm

described in Section 4.1.4, we also provide a Back-Tracking search algorithm and Pseudo-Threads.

4.3.1 Back-Tracking Search

The Back-Tracking algorithm (Algorithm 16) can 'roll-back' an aborted transaction and continue exploring other permutations of transaction execution. The goal of the Back-Tracking algorithm is to provide a more sophisticated search strategy than the Greedy algorithm at the expense of extra time and memory. The Back-Tracking algorithm requires the use of a stack to retain the results of past permutations, so that returning to previous states of exploration is possible. As such, extra memory is required for the stack.

When the Back-Tracking algorithm commits a transaction, it saves the thread's permutation on the stack before calling PermCommit (lines 20-21). When a transaction aborts and the thread's permutation is expended, the Back-Tracking algorithm pops the head of the stack and sets this to the thread's *Permutation* (lines 11-18). The thread can then explore a new path of execution. It is anticipated that the Back-Tracking algorithm will be more effective than the Greedy algorithm whenever the occurrence of semantic conflicts is particularly frequent.

4.3.2 Pseudo Threads

When a speculator executes within the context of a child session, *Pseudo Threads* may be provided to aid the speculator in exploring transaction permutations for the duration of that *child session.*

Pseudo Thread functionality is comprised of two functions shown in Algorithm 17, namely: PsThreadFtn and RegisterPsThread. Each *Pseudo Thread* monitors a *resource* by repeatedly executing the PsThreadFtn (line 1). Within the PsThreadFtn, *Pseudo Threads* attempt to register with a *Table Mask* by calling the RegisterSThread function (line 3). If successful, the *Pseudo Threads* perform their own *Speculative Phase* and *Commit Phase* in the same manner as a speculator (lines 4-9). *Pseudo Threads* perform no validation, given that once a *child session* has ended, the *Pseudo Thread* takes no further

action with respect to any transactions executed within the *child session*. If the *Pseudo Thread* cannot successfully register with its *Table Mask* then it yields the processor (line 11) before reiterating its execution loop.

Pseudo Thread Registration The REGISTERPSTHREAD function details the process of registering a *Pseudo Thread* (lines 15-31). Furthermore, the REGISTERNESTED function (shown in Algorithm 13) contains a number of statements which support functionality of *Pseudo-Threads*. Firstly, any speculator which

Algorithm 16 The Back-Tracking Algorithm

```
 1: function BTEXPAND(stack)
 2:     perm ← (Local_TICKET)_PERM
 3:     if ¬TXREADY(perm) then
 4:         return
 5:     end if
 6:     txaction ← NEXTTX(perm)
 7:     CHECKPOINT((Local_TICKET)_CACHE)
 8:     if CALL(txaction) = abort then
 9:         ROLLBACK((Local_TICKET)_CACHE)
10:         PERMABORT(perm)
11:         while perm_STATE = expended do
12:             if stack empty then
13:                 set time remaining to 0 and return
14:             end if
15:             ROLLBACK((Local_TICKET)_CACHE)
16:             pop stack; perm ← stack_HEAD
17:             PERMABORT(perm)
18:         end while
19:     else
20:         push perm on stack; perm ← stack_HEAD
21:         PERMCOMMIT(perm)
22:         save the current best permutation
23:     end if
24: end function
```

Algorithm 17 The Pseudo Thread Functions

```
1: function PsThreadFtn(mask, bintree)
2:     while true do
3:         if RegisterPsThread(mask) then
4:             initialise timer
5:             while time remaining do
6:                 call expansion function
7:                 decrement time remaining
8:             end while
9:             Synchronize(bintree)
10:        else
11:            Yield
12:        end if
13:    end while
14: end function

15: function RegisterPsThread(mask)
16:    if mask_NEXT = 0 then
17:        return false
18:    end if
19:    repeat
20:        slot ← mask_NEXT
21:        if slot + (mask_PERM)_DEPTH ≥ mask_MAX then
22:            return false
23:        end if
24:    until CAS(mask_NEXT, slot, slot + 1)
25:    ticket ← Local_TICKET
26:    ticket_SLOT ← slot need convert
27:    ticket_CACHE ← Copy(mask_CACHE)
28:    ticket_PERM ← Copy(mask_PERM)
29:    Swap(ticket_PERM, depth, slot + depth)
30:    return true
31: end function
```

acquires a particular *Table Mask* copies its *Permutation* and cache into memory which is accessible by any *Pseudo Thread* accessing the *Table Mask* (see Algorithm 13, lines 11-12). In addition, each *Table Mask* contains a variable, called NEXT, which is incremented atomically whenever a speculator acquires the *Table Mask* (line 13).

The *Pseudo Thread* begins its registration attempt by reading the NEXT variable of the *Table Mask* (line 16):

- If NEXT equals zero then the *Pseudo Thread* returns the value 'false' to indicate that it cannot register because no speculator is currently using the *Table Mask* (line 17).

- If the NEXT variable exceeds 1, the *Pseudo Thread* attempts to acquire a *slot* in the *Table Mask* using the CAS operation.

Slot acquisition is similar to algorithm which speculators use during their *Registration Phase* (lines 19-23). In the case of *Pseudo Threads*, however, each *Pseudo Thread* first accesses the *Permutation* structure that has been copied to the *Table Mask* during a speculators registration. Then the *Pseudo Thread* adds the DEPTH variable of the *Permutation* structure to the *slot* value when evaluating whether the maximum value MAX has been reached (line 21). This addition is necessary because, when a speculator begins a *child session*, it is possible that in the speculator's *session* history, it has already explored a number of transaction executions. Thus adding the DEPTH value to the value of *slot* prevents the execution of transactions that have already been explored in the *parent session*.

If a *Pseudo Thread* successfully acquires a *slot* number in the *Table Mask* then the *Pseudo Thread* prepares its own *Ticket* structure. This includes setting the cache (line 27) and *Permutation* (line 28) of the *Pseudo Thread*'s *Ticket* to copies of the cache and *Permutation* structures held at the *Table Mask*. These actions ensure that every registering *Pseudo Thread* begin their *Speculative Phase* from the same state as the speculator. A final SWAP operation ensures that the *Pseudo Thread* begins its *Speculative Phase* by executing the transaction that resides in the table entry equal to the slot number obtained (line 29).

4.4 Summary

In this chapter we have described an implementation of a Universal Construction which provides contention management for Software Transactional Memory by resolving conflicting transactions of multiple threads of execution. We began by defining our notion of *semantic conflicts* and described the shared data model used by our implementation to set the context of our approach. We then presented the chapter in three sections. In Section 4.1 we described a basic implementation, and in Section 4.2 we extended the basic implementation to incorporate nested transactions. In Section 4.3 we described some nested searching strategies.

Basic Implementation Registration, speculation and commit phases were described and supported by pseudo code. In this basic implementation section we covered:

- State space management, with the implementation of a *Transaction Table* to limit the number of threads that can update the Universal Construction during a single *session*.
- Modifications to an Atomic Object implementation which allows our contention management policy to operate in parallel with other transaction executing threads without violating Sequential Consistency.
- Permutation functions and a Greedy search algorithm were implemented to promote efficient exploration. The permutation functions enable each thread to execute a unique permutation of transactions.
- We combined the *direct* and *deferred* updates modes of execution, with the suggestion that a *deferred* model was particularly well suited to speculative execution of transactions by multiple threads.
- We provided consensus of speculative execution using a Combining Tree for higher throughput in the presence of many threads. This essentially allowed threads to reach agreement on the next state of the Universal Construction.

Nested Transactions In Section 4.2 the following topics were covered:

- Speculative Nesting was described, which allows the exploration of nested transaction execution, not possible with conventional models of nesting.
- We implemented resource management to provide parallel nesting (via *child sessions*) when resources are available. When resources are expended however, our Contention Manager resorts to a flat nesting to converse memory.
- *Child sessions* were described which allow the parallel speculative execution of nested transactions.

Finally, strategies for nested transaction exploration were introduced in Section 4.3. A Back-tracking search algorithm was presented and functionality was described to support the addition of *Pseudo-Threads*.

Chapter 5

Results and Analysis

In this Chapter the results of performance tests on the implementation of our Contention Manager are presented and discussed. For succinctness, we shall call our implementation *Hugh*[1], both in the text and on the graphs where appropriate. After describing the testing environment, the results are presented in two sections: The first section provides results showing the performance of *Hugh* in comparison with an existing Contention Management Policy, while the second section presents and assesses the performance of our approach with nested transactions. The aim of the first section is to demonstrate our approach in comparison to an existing technique with increasing levels of semantic conflicts. The aim of the second section is to evaluate the effectiveness of a selection of strategies designed to increase the exploratory power of nested transactions in our system.

5.1 Environment

Hugh was implemented on a platform with the hardware and software specifications detailed in Table 5.1. The platform was a Dell 'AlienWare' Desktop PC featuring a capacity for generating eight Hardware Threads. We were interested in assessing our work where concurrent execution and parallelism was a feature of the environment, hence the availability of parallel processing resources afforded

[1]Hugh Everett was the Quantum Physicist who invented the Many Worlds Theory, which inspired the Many Systems Model.

by the platform made a good choice. The Transactional Memory software was executed in Visual Studio 2010 with a C Sharp implementation of the Java DSTM2 benchmark suite [10] (using the obstruction free factory with visible reads) to compare *Hugh* with a conventional approach to Object Based Software Transactional Memory.

Table 5.1: Environmental Parameters

Parameter	**Value**
Processor Spec.	Intel(R) Core(TM) i7-2600 CPU
No. of Cores	8 × 3.40GHz
Cache Size	8 MB Intel(R) Smart Cache
Memory Size	16.0 GB
Operating System	Windows 7 (64 bit)
Language	C Sharp (.NET Framework 4)
IDE	Microsoft Visual Studio 2010 Premium

5.2 Benchmarked Results

In this section we present results from a set of micro-benchmarks. Each experiment is carried out using an increasing number of threads (from 2 to 12) and executed 10 times with the average results provided. The Polka Contention Management Policy [38] has been cited as providing the best performance of wait-based Contention Managers, and so this was used to provide a comparison with our implementation (using the default parameters with respect to back-off time).

Two benchmarks were used to test the performance of our implementation, namely: a linked list and a hash table. In both benchmarks, threads are divided into 'producers' and 'consumers' in equal number. Producers and consumers take a random string value and attempt to *insert* this into the data structure in the case of the producer, or *remove* it in the case of the consumer. The highest frequency of read/write conflicts is expected in the linked list benchmark compared to the hash table which distributes items in an array of linked lists based on hashes generated from each item.

Performance results under increasing levels of semantic conflicts are simulated, specifically:

1. When there are no semantic conflicts (labelled S-L0), then threads only abort transactions if there is a read/write conflict.

2. With S-L1 semantic conflicts, consumer threads explicitly abort their transaction if they attempt to remove a string value which is not already present in the data-structure.

3. With S-L2 semantic conflicts, producers also abort their transactions if they attempt to add a string value to a data-structure which is already present.

The presence of semantic conflicts is designed to simulate the need for threads to coordinate their activities, in order to progress their execution. It is expected that without semantic conflicts, the performance figures with the Polka CMP and *Hugh* while be roughly equal (given that *Hugh* falls back on the Polka technique to address concurrent conflicts). If the results differ then this will reveal the overhead required to implement our approach in scenarios without semantic conflicts.

As semantic conflicts increase, the performance of *Hugh* should degrade less markedly in contrast to the Polka CMP, due to the exploratory element provided by our approach.

5.2.1 Transaction Throughput

Figure 5.1 illustrates the results for transaction throughput. The Y-axis denotes the number of transactions committed per millisecond and X-axis shows the number of threads present. In Graphs A and B, with no semantic conflicts (S-L0) we can see that the performance of both the Polka manager and *Hugh* is roughly equal with both the linked list and the hash table benchmarks. As expected, there is a small increase in throughput for the Polka manager (roughly at most 10-20 extra transactions/millisecond). Both Polka and *Hugh* witness increased throughput when the hash table is used in comparison to the linked list.

Once semantic conflicts are introduced, *Hugh* performs markedly better than Polka under both benchmarks. With S-L1 semantic conflicts, *Hugh* shows a

minimum improvement in throughput over Polka by a factor of approximately 4.3 and 4.5 for the list (Graph C) and hash table (Graph D) respectively. With S-L2 semantic conflicts, *Hugh* shows a minimum improvement by a factor of approximately 40 and 18, for the list (Graph E) and hash tables (Graph F) respectively.

Observe that with the Polka manager, as semantic conflicts are introduced, the type of data structure has less of an impact on mitigating the presence of aborts (as witnessed by a smaller throughput). It seems reasonable to assume, that strategies for mitigating read/write conflicts in transactional memory which rely on more 'concurrent' data-structures, are of little benefit if one takes into account the kinds of semantic conflicts generated in these experiments.

5.2.2 Average Transaction Execution Time (ATET)

In Figure 5.2 the average transaction execution time (ATET) is shown. In each graph, the Y-axis measures the ATET but note that the scale used is logarithmic for greater clarity, and the maximum value is 10^5 ticks for all graphs. Each graph provides the results for a particular contention manager with a particular benchmark, and each bar shows the performance under a different semantic conflict level. The time is measured in elapsed ticks (the fastest unit of time that can be measured on the platform) and denotes the average time spent executing a transaction by all threads.

One would expect that greater throughput generally corresponds to less average time spent executing a transaction (this is not guaranteed, however, as unlike execution time, throughput is also includes time spent outside of transaction execution). Given that *Hugh* resolves both concurrent and semantic conflicts, there should be less time required to execute a transaction when semantic conflicts are introduced, whereas with the Polka manager, transaction time should increase if repeated conflicts cause threads to back off (which involves calling the sleep function).

The performance of the Polka manager is shown in graphs A and C. One may observe that the ATET increases substantially as the level of semantic conflicts is increased. Conversely, the performance of *Hugh* (graphs B and D) does not

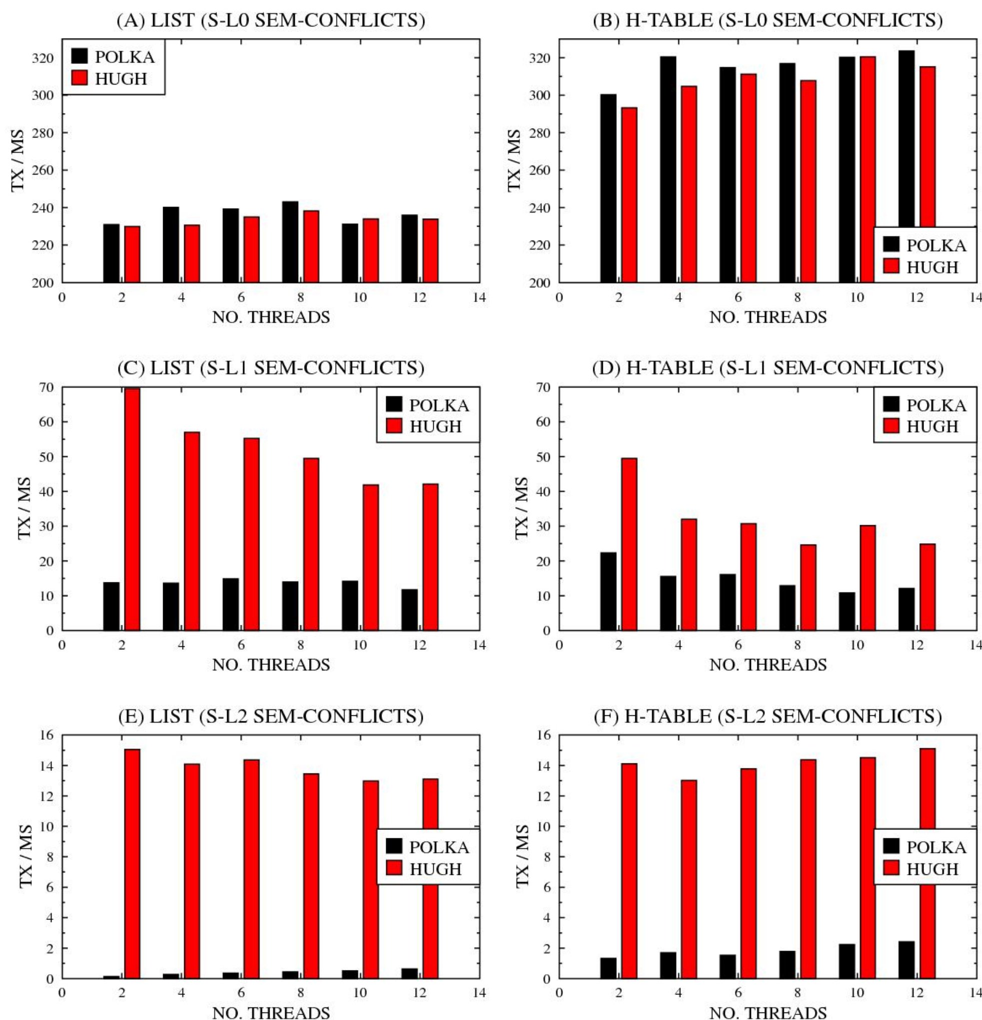

Figure 5.1: Transaction Throughput

exhibit the same degree of increase in ATET as the number of semantic conflicts is increased. This seems to suggest that the overhead of executing our policy does not increase substantially as semantic conflicts increase, unlike the Polka manager (in the case of non-nested transactions at least).

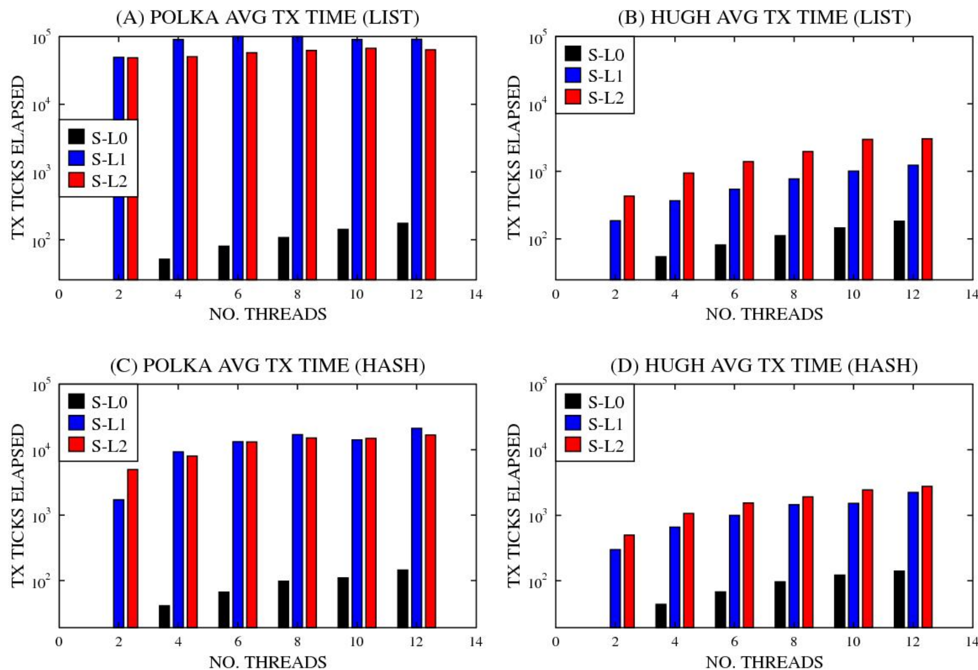

Figure 5.2: Transaction Timing (in Ticks)

5.3 Nested Transaction Results

In this section we evaluate the performance of our system using nested transactions. As with the Benchmark Results of Section 5.2, each experiment is carried out using an increasing number of threads (from 2 to 12) and executed 10 times with the average results provided. In these scenarios, we do not compare our results to Polka CMP for two reasons: the *DSTM2* framework does not support nested transaction execution and substantial changes had to be made in order to support nesting using *Hugh*. In addition, we wish to examine semantic conflict resolution that can only be achieved via our speculative-nesting approach and so we are primarily interested in evaluating a number of search strategies used in our approach.

Only the results for the linked list benchmark are presented for the evaluation of nested transactions. This is because, as noted in Section 5.2, the use of concurrent data structure has little effect on the performance when semantic conflict is a major feature of the tests. Three levels of increasing semantic conflict are

evaluated in the tests that follow, specifically:

1. With no semantic conflicts (labelled S-L0), threads only abort transactions on the occurrence of a read/write conflict;

2. With S-L1 semantic conflicts, consumer threads explicitly abort their transaction if they attempt to remove an item which is not already present in the list.

3. With S-L2 semantic conflicts, producers abort their parent transaction if they attempt to add an item to the list when it is non-empty. In addition, the producer aborts its nested transaction if an item has not been removed from the list by a consumer.

The semantic conflicts simulate concurrent accesses on a shared buffer by multiple threads. When S-L2 semantic conflicts are used, the producer uses its nested transaction as a signal to determine when its item has been consumed from the buffer.

5.3.1 Nested Search Strategies

In addition to varying the levels of semantic conflict, the nested transaction results compare the performance of four search strategies, namely:

Greedy Speculation – With the Greedy Speculation Algorithm (as described in Section 4.1.4 of this thesis), threads proceed to their Commit Phase as soon as a transaction aborts (this is also the strategy used in the results of Section 5.2).

Back-Tracking Speculation – With the Back-Tracking algorithm (as described in Section 4.3.1 of this thesis), threads rollback aborted transactions and explore new permutations as long as they have time remaining.

Pseudo-Threads – Psuedo-Threads (from Section 4.3.2) are used in child sessions to aid exploration. In this scenario Pseudo-Threads and Application Threads both execute the Greedy Algorithm and four Pseudo-Threads are

provided in each test. Note that the choice of four Pseudo-Threads was determined after experimentation with various numbers of Pseudo-Threads, and four was found to produce the best results. Reducing the number of Pseudo-Threads produces performance closer to the Greedy Speculation Algorithm, while increasing produces no clear improvement until too many Pseudo-Threads cause performance degradation. The effectiveness of the Pseudo-Thread approach depends on the degree of parallel processing available on the host platform.

Back-Tracking plus Pseudo-Threads – In this scenario Pseudo-Threads execute the Greedy Algorithm while Application Threads execute the Back-Tracking algorithm.

The additional search strategies require varying levels of overhead. With its use of a stack, the Back-Tracking algorithm is more costly in terms of memory than the Greedy algorithm. The use of Pseudo-Threads requires extra memory and extra processing time to accommodate each additional pseudo-thread. It is expected, however, that as the degree of semantic conflicts is increased, the extra exploratory potential of the Back-Tracking and the Pseudo-Thread approach will provide better performance than the Greedy Algorithm.

5.3.2 Nested Throughput

Figure 5.3 illustrates the results for nested transaction throughput. As with the results from the previous section, the Y-axis denotes the number of transactions committed per millisecond and X-axis shows the number of threads present. Graph A shows the results for S-L0 semantic conflicts and we can see that the as the number of threads increases, the pseudo-thread strategy produces the best performance, while the difference in throughput for Greedy and Back-Tracking is negligible.

In Graph B, S-L1 semantic conflicts are introduced and the general throughput of transactions decreases by a factor of approximately 70-80%. Although the pseudo-thread approach still provides the best throughput the difference is

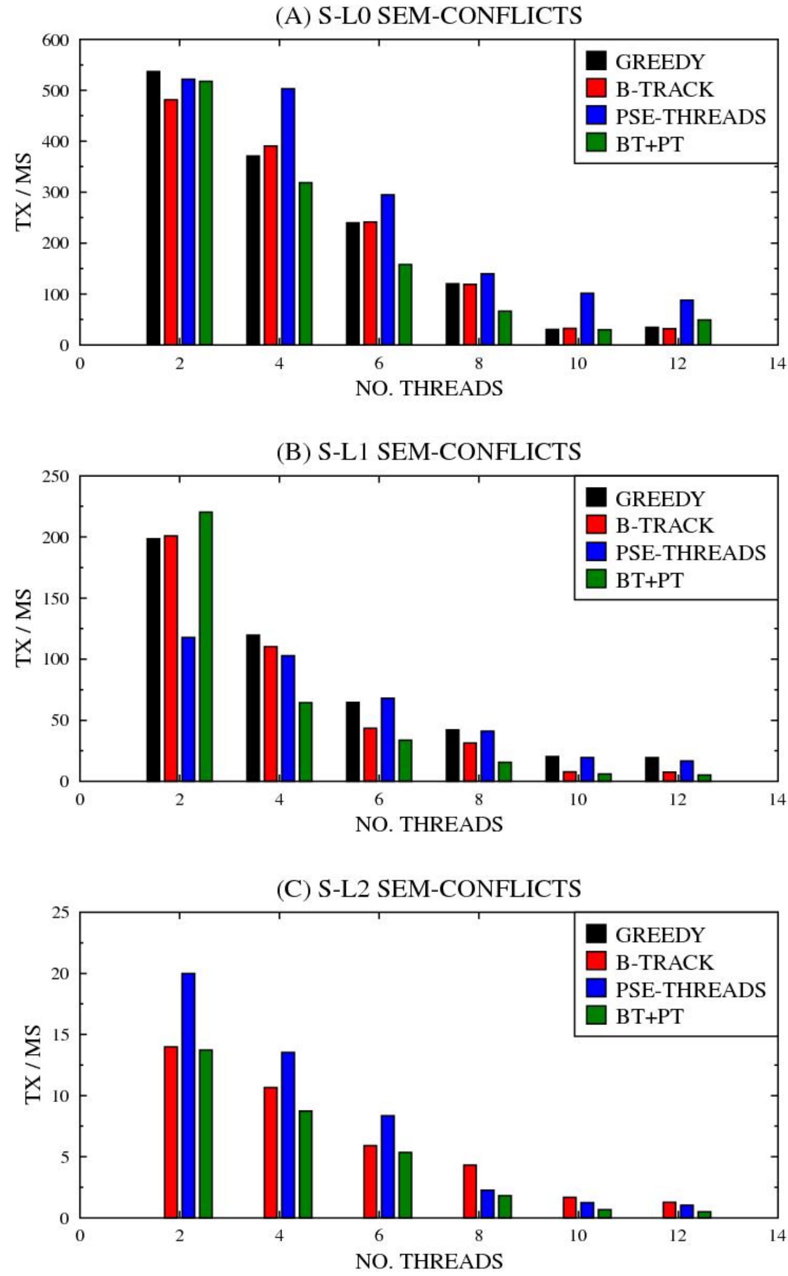

Figure 5.3: Nesting Throughput Results

marginal. At this level of semantic conflicts, the Greedy algorithm still does as well as the Back-Tracking approach.

In Graph C with S-L2 semantic conflicts, the Greedy Algorithm can no longer

provide the exploratory power necessary to commit any transactions. This obstacle is expected, because the scenario requires that a consumer transaction executes between a producer's parent transaction and its nested transaction in order for both the producer and consumer to eventually commit. This is a permutation which cannot be discovered using the Greedy algorithm (or any other conventional approach for that matter). Observe that the Back-Tracking and Pseudo-Thread approaches find the necessary permutation and manage to commit their transactions. The pseudo-thread approach produces the best throughput until the number of threads reaches eight, and then the Back-Tracking approach does best (although marginally), possibly due to the fact that the platform can run a maximum of eight hardware threads in parallel.

5.3.3 Nested ATET

In Figure 5.4 the average transaction execution time (ATET) is shown. As in the previous section, in each graph the Y-axis measures the ATET and the X-axis shows the number of threads used. The performance under a different semantic conflict level is shown, and the time is measured in elapsed ticks (the fastest unit of time that can be measured on the platform). As noted in the previous ATET results, a higher throughput for any particular search strategy should generally correspond to a smaller ATET, but other factors may affect the throughput, such as the overhead of managing the Universal Construction.

The performance with S-L0 semantic conflicts is shown in Graph A. The average between the search strategies is fairly uniform until the number of threads is increased to ten. With ten and twelve threads, all strategies except for the Pseudo-thread approach show a marked increase in ATET. An explanation for the increase may be due to the fact that the number of application threads now exceed the maximum eight hardware threads, which the platform allows. The result is that throughput is reduced because of an increase in context switching between thread resources. This is not observed to such an extent for the Pseudo-Thread approach which at first seems counter-intuitive given the extra number of threads being used. However, the Pseudo-Threads demand far fewer processing resources than the application threads, and most of their activity involves yielding

the processor until there is work for a Pseudo-Thread to do. The benefit of the Pseudo-Thread approach is that they help application threads to complete their speculation phases more quickly than with the other techniques, but the ATET of the Pseudo-Threads is not measured.

The performance with S-L1 semantic conflicts is shown in Graph B and we can see that, on average, the ATET has increased by a factor of 70-80%. The smallest ATET can be seen with the Pseudo-Thread and the Greedy approach. In Graph C, S-L2 semantic conflicts are introduced, and the results are shown for the Back-Tracking algorithm and the Pseudo-Threads. Note that the Greedy results are not shown because no transaction could commit using the Greedy algorithm with S-L2 semantic conflicts. The smallest ATET is shown by the Back-Tracking algorithm followed very closely by the Pseudo-Threads, while using both Back-Tracking and Pseudo-Threads was generally worse than using either approach alone.

5.3.4 Registered Versus Commit Rate

The final set of graphs provide a ratio between the average number of transactions registered and the average number of transactions to commit per session (for succinctness, let us refer to this as R/C). For example, when a test is executed with six threads, we would expect the average number of registered transactions n to be in the range of $0 \leq x \leq 6$. If the subsequent number of transactions that commit is close to six then this suggests that the search strategy was effective in finding a permutation of transactions where most could commit successfully.

From these results a number of interesting conclusions can be inferred, namely:

- The degree to which transactions register provides some measure of contention, but in the case of a high number of threads, we can also infer the level of parallelism being afforded by the platform.
- The degree to which transactions commit (such that $R/C \rightarrow 1$) allows one to estimate how effective the particular search strategy was in finding a permutation of transactions where most could commit.

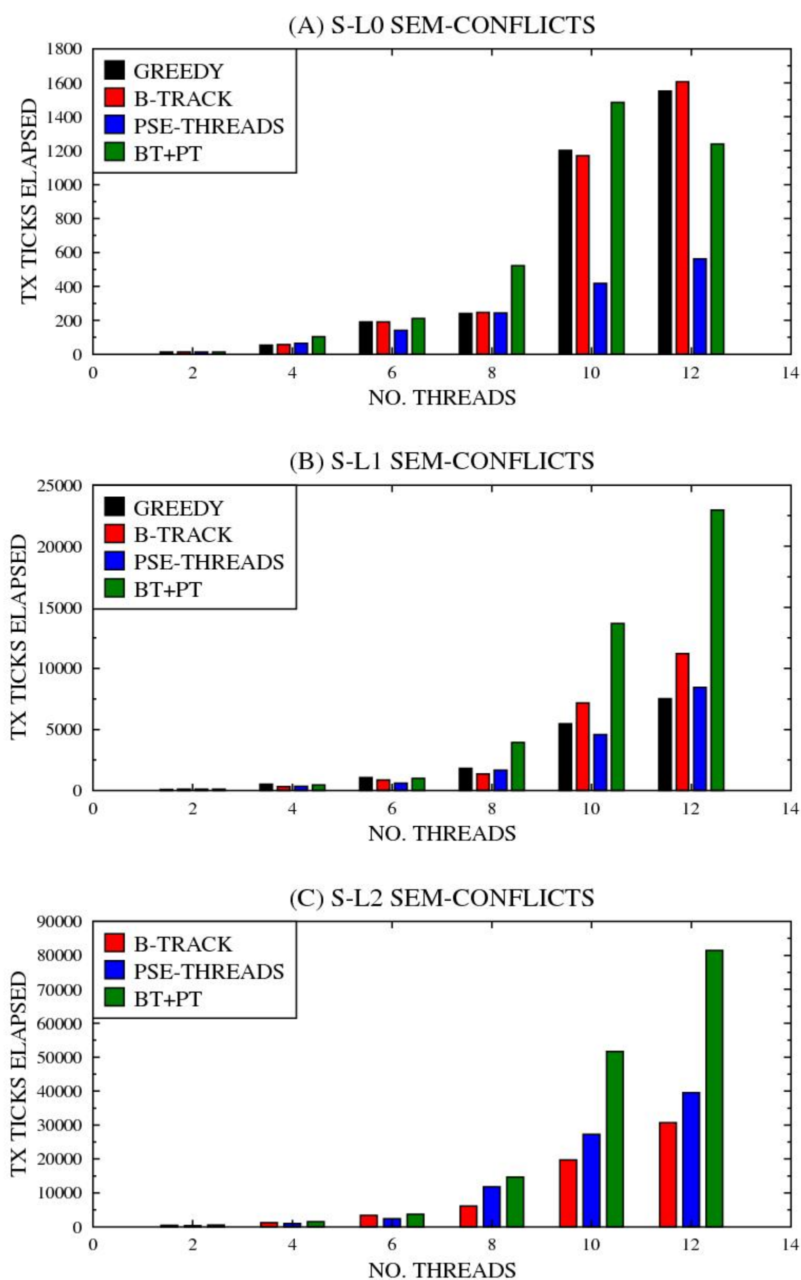

Figure 5.4: Nested Average Transaction Execution Time

The results showing the actual ratios of registered transactions versus committed transactions (R/C) are provided in Figure 5.5. Looking firstly at Graph A, observe that as the number of threads increases, the number of registered trans-

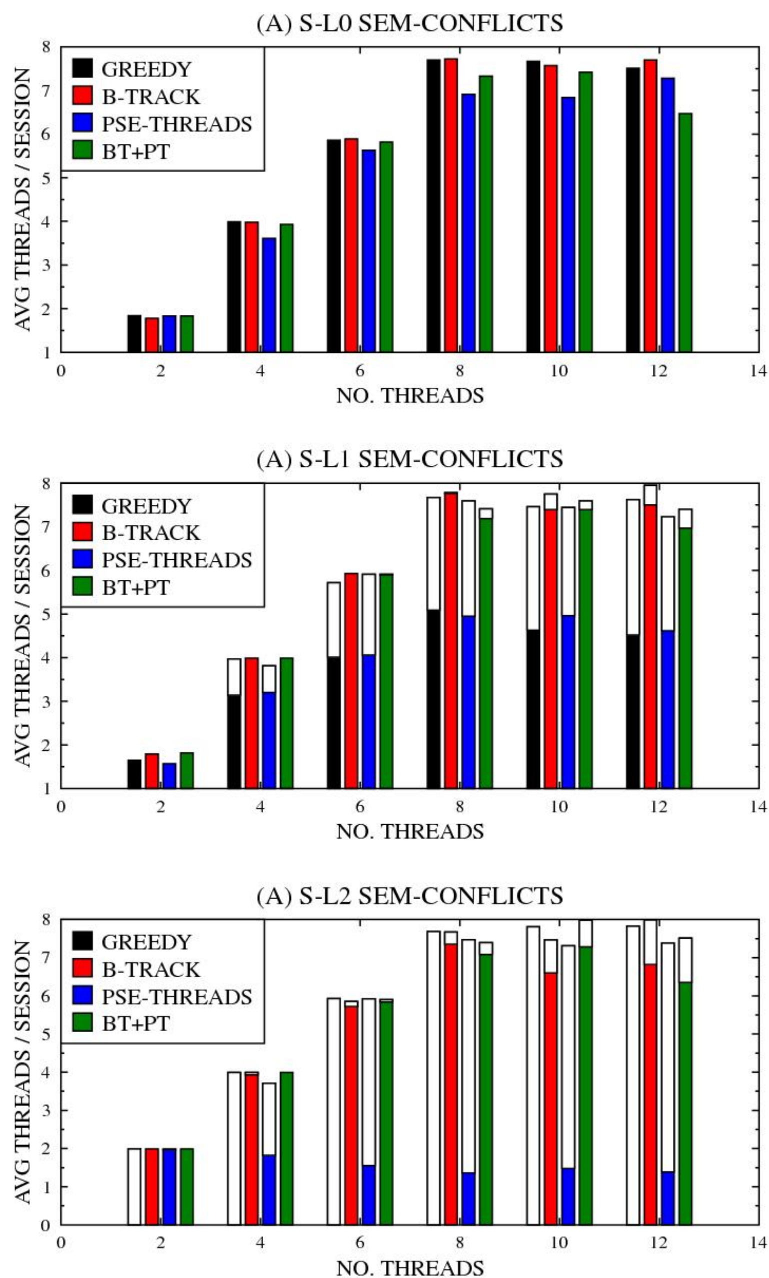

Figure 5.5: Registered Versus Committed

actions also increases (the average never increases above eight, which we would expect given that the maximum level of parallelism afforded by the platform is eight threads). In Graph A, where only concurrent conflicts prevent a transac-

tion from committing, the ratio between R/C is always 1 (every transaction that registers is able to commit).

In Graph B, S-L1 semantic conflicts are introduced, and we can see that the number of transactions registering is roughly the same as in Graph A. This time however, in the case of the Greedy and the Pseudo-Thread strategies, the number of transactions which commit is often less than the number registering. This suggests that the Back-Tracking approach is more effective at finding permutations which allow for a greater commit rate. In Graph C, we can see that the effectiveness of the Pseudo-Thread approach has decreased further while the Back-Tracking approach still provides a higher commit rate (the Greedy approach is not shown because no transactions could commit under S-L2 semantic conflicts).

In summary, the Back-Tracking algorithm appears the most effective for finding permutations of transactions which allow the highest number of commits. However, according to the results provided in Figure 5.3, the Pseudo-Thread approach produced the best throughput. This finding would suggest that locating the best permutation does not always justify the extra time required for the search when throughput is the most important issue.

5.4 Summary

In this chapter a number of experiments were presented using an implementation of the Many Systems approach. In each experiment, we examined the transaction throughput and the average transaction execution time. To begin with, results where compared with an existing Contention Management Policy, to provide a measure of comparison between our approach and an existing technique. This was followed by further tests involving nested transactions, designed to evaluate the exploratory power of our approach. The goal of each test was to examine performance in the face of both concurrent and semantic conflicts. To summarise, the results presented in this chapter suggest:

- In comparison with an existing Contention Management Policy, *Hugh* shows significant improvement in throughput when the presence of semantic con-

flicts is increased. Without semantic conflicts, *Hugh* still provided comparable with Polka performance under both benchmarks.

- While different data structures (linked lists, hash tables) can affect the performance of a conventional system designed to resolve concurrent conflicts, the choice of data-structure has less of an impact when semantic conflicts are taken into account.
- When semantic conflict resolution is combined with nested transactions, transaction throughput deceases, and more elaborate search strategies may be required to allow transactions to commit. A Back-Tracking and Pseudo-Thread approach are presented and evaluated, with the latter having slightly better throughput.

As described in Section 5.1, the environment consisted of a platform with a capacity for eight hardware threads to run in parallel. In further experiments, it would be interesting to observe how the performance alters on platforms which provide more parallelism, especially with respect to the Pseudo-Thread search strategy, which suggests improvements can be obtained with greater parallel resources.

In Section 5.3, results have revealed that throughput has changed only modestly between the Pseudo-Thread and Back-Tracking algorithms. Although the Back-Tracking algorithm is more effective at committing more transactions per session (see Graph 5.5), interestingly, this doesn't correspond to higher throughput in the application (presumably because of the extra time used to search for permutation that allows a small number of extra transactions to commit). As a result of this observation, in future work it may be worthwhile to alter the commit phase algorithms to prefer timely completion over locating the best permutation.

Chapter 6

Conclusion

In this final chapter we provide a summary of the material that has been presented in this thesis, briefly discuss the implications of our work and suggest ideas for future work.

6.1 Thesis Summary

This thesis began with the description of a Many Systems Model in an abstract process language CSP. The model described an approach to Concurrency Control based on state-space exploration. An implementation of our approach in the form of a Universal Construction was described, to provide Contention Management for Software Transactional Memory systems. The implementation is designed to provide conflict resolution which incorporates both concurrent conflict resolution and semantic conflict resolution (where transactions explicitly abort because of some logical condition in the program). We then extended the implementation to deal with nested transactions before providing results which demonstrated the performance of our approach.

6.2 Main Contributions

The main contributions of this thesis can be described in three parts:

1. The Many Systems model of computation was described in Chapter 3, which approached the execution of concurrent threads as a state-space exploration and management problem. We described the concept of a Supersystem, wherein concurrent conflicts can be resolved, and permutations of thread execution explored. In the model we showed how No-Wait Synchronization could be assumed, so that a deadlock is avoided and we described a lock-free Universal Construction to demonstrate how the model could be placed in a practical setting.

2. An implementation of a Universal Construction is provided in Chapter 4, to provide Contention Management for Object-based Software Transactional Memory. The Universal Construction adapted concepts from the Model and resolved contention by generating permutations of conflicted transactions; by doing so the Universal Construction allows the resolution of both concurrent conflicts and semantic conflicts. The implementation was described purely in software and requires no control over the scheduling policy of the underlying Operating System.

3. The Universal Construction was extended to incorporate nested transactions, and we demonstrated how our approach can use them to extend the exploratory element of searching the state-space using speculative nesting. This allowed linearizable schedules to be discovered and committed, which would otherwise be undetectable in an existing nested models (such as Flat, Open or Closed nesting).

Results were provided which compared the performance of our approach with an established Contention Management Policy (namely the Polka CMP), under varying degrees of concurrent and semantic conflicts. We explore transactional throughput using two benchmarks (lists and hash tables) and then provide performance results of a number of search strategies when nested transactions are used. Our approach showed comparable performance with the Polka CMP where no semantic conflicts were present, and improved throughput when semantic conflicts were introduced.

6.3 Future Work

We believe that the treatment of Contention Management, and Concurrency Control in general, as a state space exploration problem offers much in terms of future work. Initially, there exists a great deal of scope for exploring the role of semantic conflicts in more detail. In addition, various optimisations of the existing work are possible, which should boost the throughput of concurrent activity. With respect to short-term developments, we suggest three approaches: (i) reducing nested overhead, (ii) probabilistic analysis of semantic conflicts, and (iii) strategies for increasing throughput (specifically reducing wasted work). We conclude this thesis with an overview of each approach.

Reducing Nested Overhead In Chapter 5, results were presented which showed the transaction throughput when semantic conflicts were introduced with nested transactions under several search strategies. An observation of the results is that the overhead of creating and managing child sessions increased dramatically when semantic conflicts increased (by around 70-80%), whichever search strategy was employed.

A possibility for future work may be to explore ways of reducing child session overhead, and initially, one could consider modifying the algorithms of the *Commit Phase*. For example, the *Commit Phase* as described uses a Combining Tree approach which allows threads to determine the best permutation of n threads. This method can be altered in order to find the best permutation of transaction execution within a fixed number of executions. Essentially this would require transforming the lock-free approach of the Combining Tree into a Wait-free algorithm. Although the optimum execution may no longer be found (i.e. the permutation containing the most number of committable transactions), the results in Figure 5.5 suggest that improved throughput may result, especially if semantic conflicts are particularly sparse.

Conflict Prevention The approach of the work in this thesis is to provide contention management and conflict resolution. More generally, only when concurrent accesses conflict do the mechanisms described in the previous chapters

come into effect. An attractive alternative is to adapt our approach in order to prevent conflicts from occurring in a manner similar to the approach of the *Shrink* system [21] (a brief overview of Shrink is provided in Section 2.5 of this thesis). *Shrink* demonstrated how probabilistic techniques, specifically Bloom Filters, could help order transactions in a schedule to prevent conflicts occurring. Similarly, the approach presented in this thesis could be adapted to provide similar behaviour. In fact, given their deterministic nature, semantic conflicts may be more suspectible to detection by a Bloom Filter than concurrent conflicts.

Adapting our approach to prevent conflicts would involve the following:

1. A probabilistic structure (a Bloom Filter for example) determines the likelihood of a conflict (either a concurrent conflict or a semantic conflict).

2. When a contention threshold is exceeded via the probabilities returned by the Bloom Filter, threads register with the *Transaction Table* and begin speculating as in the implementation.

Increasing Throughput A negative aspect of the approach is that committing a single thread's speculative execution is wasteful if there are many other threads which have executed their own speculation phases. Although multiple transactions can still commit in the approach described, it may be more productive for throughput to: (1) introduce multiple Universal Constructions to allow conflicts on disjoint groups of atomic objects to be resolved in parallel, and (2) use Thread Level Speculation (TLS) to allow per thread permutations transaction execution to execute in parallel.

An implementation featuring multiple Universal Constructions would require three additions to functionality:

1. A process for deciding which Universal Construction each thread shall update (perhaps, by grouping atomic objects according to the probability that conflicts will occur between each other).

2. Logic to detect conflicts between threads updating different Universal Constructions.

3. A mechanism to allow the serialisation of Universal Construction when a conflict is detected.

Introducing TLS should provide better performance with long, nested transactions and could be implemented via the Pseudo Thread technique described in Section 4.3.2. Specifically, per thread permutations of transaction execution could be executed in parallel, and child sessions could commence when a thread first registers its transaction invocation in the *Transaction Table* (in parallel with the parent session of the same thread).

Conflict detection would be required to revert to sequential nested execution when executing a single thread's child session in parallel with its parent session would generate a conflict. Given that during contention management transactions are executed in sequence, only semantic conflicts and TLS conflicts are relevant (e.g. Write after Write (WaW) and Write after Read (WaR)). This is similar to the approach of TLSTM [24], but which must handle the complexity of detecting both concurrent and TLS conflicts. On the other hand, our approach would explore multiple, parallel TLS executions where each thread attempts to execution its own permutation in parallel (aided by Pseudo Threads). As such, it is hoped that exploring multiple TLS executions would increase the probability of reducing both semantic and TLS conflicts.

Chapter 7

Appendix

This appendix is provided as a reference for the CSP terminology used in the Model Chapter. At the time of writing this thesis, the full text ‘Communicating Sequential Processes’ by Tony Hoare, can be accessed for free at:
http://www.usingcsp.com/cspbook.pdf.

7.1 Processes

Notation	**Meaning**
αP	denotes the alphabet of the process P where the alphabet of P is the set of events that P can accept.
$a \rightarrow P$	the event a then P
$(a \rightarrow P \mid b \rightarrow Q)$	a choice between a then P or b then Q
$P \parallel Q$	process P and Q executing in parallel such that P and Q may communicate with each other.
$P \setminus C$	process P with the events of C hidden from specifications.
$P|||Q$	the interleaved execution of process P and Q such that P and Q cannot communicate with each other.
$l : P$	process P assigned the name l. Events generated by l bear the prefix in the form $l.event$.
$P /\!/ Q$	the process P is subordinate to the process Q such that the alphabet of P is a subset of the alphabet of Q.
$(\parallel_{i<n} i : \mathrm{P})$	represented n concurrent processes P where the ith process is prefixed with the label i. Shorthand for $(0 : \mathrm{P} \parallel 1 : \mathrm{P} \parallel n-1 : \mathrm{P})$.

$(\|\|\|_{i<n} i : \mathrm{P})$	represented n interleaved processes P where the *ith* process is prefixed with the label i. Shorthand for $(0 : \mathrm{P}\|\|\|1 : \mathrm{P}\|\|\|n-1 : \mathrm{P})$.

The following symbols refer to special processes defined in this thesis.

Notation	**Meaning**
τ_x	the symbol for a System with the name x where a System is defined as a collection of system processes.
Θ	the symbol for the Supersystem where a Supersystem is defined as a collection of systems aranged as a directed acyclic graph.
τ_0	the symbol for the root System in Θ

7.2 Special Events

Notation	**Meaning**
$l.a$	represents the participation in the event a by a process named l (for example *i.sits down* where $i = l$ and *sits down* $= a$).
$b!e$	on channel b, output the value of e. For example *stack.out*!m represents the output of a message called m, on a channel called *out* by a process named *stack*.
$b?x$	on channel b, input to x. For example, *queue.in*?m represents the input of a message called m, on a channel called *in* belonging to a process named *queue*.
$P \lhd b \rhd Q$	if b is true, then do P, else do Q
$*P$	repeat P.
$b * P$	while b is true, repeat P.
$x := e$	represents the assignment of the value e to the variable x.

7.3 Functions

The following functions are defined in the Model Chapter:

Function	Meaning
$assign(X, Y)$	assigns the state of system X to be Y.
$cas(m, e, u)$	refers to the atomic operation *compare-and-swap*, which requires a variable to modify, (o), an expected value for the variable to modify (e) and an update value (u) which will overwrite m, iff $m = e$. The *cas* function returns true if the modification was successful and false otherwise.
$checkpoint_x(\tau)$	creates a copy of τ prefixed with the label x, so that the state after a call to $checkpoint_x(\tau)$ is equal to $(\tau \parallel x : \tau)$.
$child(\tau_x, i)$	refers to *ith* child system (or transition $trn_x(\tau_x, i)$) from system τ_x.
$contract(P, x, y)$	the *contract* function accepts a parent system and integers x and y. The function is shorthand for $child(P, x) = child(child(P, x), y)$.
$leafnode(X)$	returns true if the number of child systems of system X is zero ($X_{cns} = 0$).
$ncas(m, e, u, n)$	refers to a function that calls the *cas* function at most n times, returning false if *cas* returns false on the *nth* attempt or true otherwise. On each iteration, *ncas* reads the value of m.
$obs(\tau)$	specifies the ordered set of possible observable events in τ which may be executed immediately.
$parent(\tau_x)$	refers to the parent system of τ_x or *nil* if τ_x is the root system (i.e. if $x = 0$).
$sched(X)$	reschedules the observable events of system X.
$\tau(i)$	specifies the *ith* event from the ordered set of possible events in τ. Shorthand for calling $obs(\tau)(i)$.
$trn_x(\tau, \tau(i))$	creates a copy of τ and then executes the event specified by $\tau(i)$. Equivalent to calling $(checkpoint_x(\tau); (x : \tau(i)))$.
$valuate(X)$	assigns some application specific value to system X.

The following support processes are referenced in the Model and are defined here:

Timer Process A simple timer process is defined in Equation 7.1:

$$\textsc{Tmr}_{t,n} = ((t > 0) * (tick \rightarrow \textsc{Tmr}_{t-1,n}); timeout \rightarrow \textsc{Skip}) \tag{7.1}$$

Max Process The MAX process requires a parent system (P) and a value for maximum (m) which will hold the ID of the child node with the highest 'value'. MAX iterates through each child node of (P) from $x = 0$ to n. MAX cannot complete until all child nodes of P are leaf nodes.

$$\begin{aligned}
\textsc{Max}_{x,n}(P, m) = {} & (if\ leafnode(child(P, x))\ then \\
& \quad m := max(m, child(P, x)); \\
& \quad \textsc{Max}_{x+1,n}(P, m)) \\
& else\ \textsc{Max}_{x,n}(P, m)) \\
\textsc{Max}_{n,n}(N, m) = {} & (done \rightarrow \textsc{Skip})
\end{aligned} \tag{7.2}$$

Depth First Search Equation 7.3 provides the definition of a depth-first search algorithm. A stack (labelled *searched*) is used to provide a path of previously explored nodes. Location of the desired node is signalled by the *success* event. At each level of the search tree, the *fail* event signals that the desired node has not been found and the depth-first search algorithm begins searching the next available child node.

$$\begin{aligned}
\textsc{DFSearch}(Root) = {} & (searched : \textsc{Stack} \mathbin{/\!/} (\textsc{Search}(Root))) \\
\textsc{Search}(\mathrm{R}) = {} & searched.push(\mathrm{R}) \rightarrow \\
& \quad (success \rightarrow found \rightarrow \textsc{Skip} \\
& \quad | fail \rightarrow \textsc{Iter}(0, size(\mathrm{R}), \mathrm{R}); \textsc{Skip}) \\
\textsc{Iter}(c, n, N) = {} & (\textsc{Search}(child(\mathrm{N}, c)); \\
& \quad \textsc{Skip} \nleftarrow found \nrightarrow \textsc{Iter}(c+1, n, N)) \\
\textsc{Iter}(n, n, N) = {} & (searched.pop(\mathrm{X}) \rightarrow \textsc{Skip})
\end{aligned} \tag{7.3}$$

References

[1] M.P. Herlihy. **Impossibility and universality results for wait-free synchronization**. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 276–290. ACM, 1988. 14, 48

[2] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, **370**. Addison-wesley New York, 1987. 22

[3] Leonardo Dagum and Ramesh Menon. **OpenMP: an industry standard API for shared-memory programming**. *Computational Science & Engineering, IEEE*, **5**(1):46–55, 1998. 24

[4] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* O'Reilly Media, Incorporated, 2007. 24

[5] J.G. Steffan, C.B. Colohan, A. Zhai, and T.C. Mowry. **A scalable approach to thread-level speculation**. In *ACM SIGARCH Computer Architecture News*, **28**, pages 1–12. ACM, 2000. 25

[6] J.F. Martínez and J. Torrellas. **Speculative synchronization: applying thread-level speculation to explicitly parallel applications**. *ACM SIGOPS Operating Systems Review*, **36**(5):18–29, 2002. 25

[7] A. Bestavros. **Speculative Concurrency Control for Real-Time Databases**. Technical report, Boston University Computer Science Department, 1993. 26

[8] J. Haubert, B. Sadeg, and L. Amanton. **Improving the SCC protocol for real-time transaction concurrency control**. In *Signal Processing and Information Technology, 2003. ISSPIT 2003. Proceedings of the 3rd IEEE International Symposium on*, pages 593–596. IEEE, 2003. 26

[9] V. Gramoli, R. Guerraoui, and V. Trigonakis. **TM2C: a software transactional memory for many-cores**. In *Proceedings of the 7th ACM european conference on Computer Systems, ser. EuroSys*, **12**, pages 351–364, 2012. 30

[10] Maurice Herlihy, Victor Luchangco, and Mark Moir. **A flexible framework for implementing software transactional memory**. In *ACM SIGPLAN Notices*, **41**, pages 253–262. ACM, 2006. 33, 34, 81, 123

[11] T Riegel, P Felber, and C Fetzer. **TinySTM**, 2010. 34

[12] Robert Ennals. **Software transactional memory should not be obstruction-free**. *Intel Research Cambridge Tech Report*, 2006. 34

[13] Keir Fraser and Tim Harris. **Concurrent programming without locks**. *ACM Transactions on Computer Systems (TOCS)*, **25**(2):5, 2007. 34

[14] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. **Composable memory transactions**. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60. ACM, 2005. 36, 81

[15] R. Kumar and K. Vidyasankar. **Hparstm: A hierarchy-based stm protocol for supporting nested parallelism**. In *the 6th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT11)*, 2011. 38

[16] N. Diegues and J. Cachopo. **Exploring parallelism in transactional workloads**. Technical report, Technical Report RT/16/2012, INESC-ID Lisboa, 2012. 38

[17] T. Heber, D. Hendler, and A. Suissa. **On the impact of serializing contention management on STM performance**. *Journal of Parallel and Distributed Computing*, 2012. 40, 42

[18] S. Dolev, D. Hendler, and A. Suissa. **CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory**. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 125–134. ACM, 2008. 42

[19] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. Kirkham, and I. Watson. **Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering**. *High Performance Embedded Architectures and Compilers*, pages 4–18, 2009. 42, 43, 50, 83

[20] R.M. Yoo and H.H.S. Lee. **Adaptive transaction scheduling for transactional memory systems**. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 169–178. ACM, 2008. 42

[21] A. Dragojević, R. Guerraoui, A.V. Singh, and V. Singh. **Preventing versus curing: avoiding conflicts in transactional memories**. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 7–16. ACM, 2009. 45, 140

[22] Burton H Bloom. **Space/time trade-offs in hash coding with allowable errors**. *Communications of the ACM*, **13**(7):422–426, 1970. 45

[23] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. **Stretching transactional memory**. In *ACM Sigplan Notices*, **44**, pages 155–165. ACM, 2009. 46, 89

[24] Joao Barreto, Aleksandar Dragojevic, Paulo Ferreira, Ricardo Filipe, and Rachid Guerraoui. **Unifying thread-level speculation and transactional memory**. In *Middleware 2012*, pages 187–207. Springer, 2012. 46, 141

[25] Hanjun Kim, Arun Raman, Feng Liu, Jae W Lee, and David I August. **Scalable speculative parallelization on commodity clusters**. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 3–14. IEEE Computer Society, 2010. 47

[26] Michael K Chen and Kunle Olukotun. **Exploiting method-level parallelism in single-threaded Java programs**. In *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pages 176–184. IEEE, 1998. 47

[27] M. Herlihy. **Wait-free synchronization**. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **13**(1):124–149, 1991. 48, 68, 82

[28] J.T. Wamhoff and C. Fetzer. **The universal transactional memory construction**. Technical report, Tech Report, 12 pages, University of Dresden (Germany), 2010. 48, 68

[29] P. Chuong, F. Ellen, and V. Ramachandran. **A universal construction for wait-free transaction friendly data structures**. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pages 335–344. ACM, 2010. 49, 68

[30] T. Crain, D. Imbs, and M. Raynal. **Towards a universal construction for transaction-based multiprocess programs**. *Distributed Computing and Networking*, pages 61–75, 2012. 49, 68

[31] Hugh Everett, Bryce Seligman DeWitt, Neill Graham, and Bryce Seligman Dewitt. *The many-worlds interpretation of quantum mechanics*. Princeton University Press, 1973. 55

[32] C.A.R. Hoare. **Communicating sequential processes**. *Communications of the ACM*, **21**(8):666–677, 1978. 55, 60, 65

[33] Edsger W. Dijkstra. **Hierarchical ordering of sequential processes**. *Acta informatica*, **1**(2):115–138, 1971. 55

[34] Philip A Bernstein and Nathan Goodman. **Multiversion concurrency controltheory and algorithms**. *ACM Transactions on Database Systems (TODS)*, **8**(4):465–483, 1983. 58

[35] M. Herlihy and N. Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2008. 96

[36] E. Moss and T. Hosking. **Nested transactional memory: Model and preliminary architecture sketches**, 2005. 101

[37] 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 *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 68–78. ACM, 2007. 104

[38] William N Scherer III and Michael L Scott. **Advanced contention management for dynamic software transactional memory**. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248. ACM, 2005. 123