Contents

6.1 Progress and Contention Management

Concurrent applications suffer from the progress pathologies of blocking, live-lock and priority inversion. Progress pathologies can be alleviated by a contention manager. However, centralised contention management restricts the scalability of a concurrent system. This thesis proposes that a concurrent application should make strong progress guarantees to alleviate the need for contention management.

A concurrent application that guarantees that all of its constituent tasks complete in a finite period of time offers a progress guarantee, whereas an application that does not can suffer from a progress pathology. A useful concurrent application should make a strong guarantee of progress but it is difficult to write concurrent applications that guarantee progress.

The main contribution of this section is the observation that concurrent applications that make strong progress guarantees alleviate the need for contention management. This section focuses on guaranteeing that functions acting on an Immutable Data Structure eventually complete.

6.1.1 Blocking

A program that executes on a Transactional Memory system either blocks or guarantees obstruction-free progress. Obstruction-freedom is the guarantee that if a transaction is repeatedly re-tried and eventually encounters no interference from other transactions, it will complete. Obstruction-freedom does not guarantee that all of the transactions that constitute a concurrent program eventually complete. Programs that execute on a Transactional Memory system offer weak progress guarantees and are therefore prone to progress pathologies.

Section 6.1.3 discusses progress guarantees and progress pathologies.

Transactional Memory systems can implement a contention manager to alleviate progress pathologies and ensure the progress of the transactions executing in the concurrent system. The contention manager has an overview of the concurrent processing and can intervene to ensure that the application eventually completes. However, contention management is a necessarily centralised task so it restricts the scalability of the concurrent system.

6.1.2 Guaranteed Progress

An application should guarantee lock-free progress to alleviate the need for a concurrent system to implement contention management. Centralised contention management is a fundamental barrier to the scalability of a concurrent system, whereas the difficulty of creating applications that guarantee progress is a problem that can be overcome. This thesis focuses on making it easier to write concurrent applications that guarantee progress.

Rajwar describes how concurrent applications based on the Time Stamp Ordering concurrency control protocol can be made lock-free [Raj02]. A lock-free concurrent application guarantees system-wide progress but permits individual operations to postpone indefinitely. A lock-free application is prone to the pathology of live-lock and priority inversion. However, it will be argued that the use of Time Stamp Ordering as the concurrency control protocol reduces the likelihood of either pathology occurring so a concurrent application that guarantees lock-free progress does not require contention management.

Live-lock can occur in a transactional system that uses Time Stamp Ordering. In practice, continual live-lock is unlikely because the unique monotonically increasing time stamp assigned to each transaction acts as a priority causing a single transaction to succeed eventually in any conflict between transactions.

Priority inversion can occur in lock-free applications that implement the Time Stamp Ordering concurrency control protocol. In practice, priority inversion can be addressed by ensuring that all transactions execute for similar durations. Long running transactions do not occur when transactions are implemented at the granularity of accesses to a data structure.

Bernstein explains in detail why database systems that implement Time Stamp Ordering do not require contention management [BHG87].

6.1.3 The Dining Philosophers

The dining philosophers problem can be used as an illustration of progress guarantees and progress pathologies. Five philosophers are sitting round a table dining on bowls of rice. Five chopsticks are placed between the bowls. Each philosopher sits in front of a bowl and can only reach the chopstick to his immediate left or right. A philosopher must have a pair of chopsticks in order to eat. The action of the philosophers is determined by a dining algorithm.

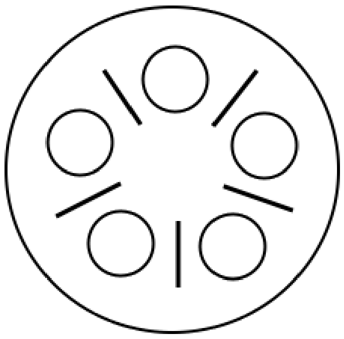

Figure 6.1: **The dining philosophers** each have a rice bowl but there are insufficient chopsticks for them to all eat at once.

Figure 6.1 illustrates the arrangement of bowls and chopsticks.

Hoare reformulated a five computer synchronisation problem, originally posed by Dijkstra, as the dining philosophers problem [Hoa83]. Krishnaprasad describes how a number of synchronisation strategies can be expressed in terms of dining algorithms [Kri03].

The following discussion considers progress guarantees made by dining algorithms and the progress pathologies they are prone to.

Deadlock is a circular wait condition that occurs when each of the philosophers reaches for a second chopstick but finds that their neighbour has already taken it. The philosophers involved in the deadlock will starve because eating requires a pair of chopsticks.

Mutual exclusion is a convention that relies on blocking the progress of concurrent processes to prevent simultaneous execution. Deadlock can be prevented in a system in which mutual exclusion is enforced by serialising access to a single entity. To prevent deadlock a dining algorithm introduces a single napkin with the rule that only the philosopher in possession of the napkin can eat. The lack of a napkin blocks the other philosophers from eating. The napkin is placed on the table and all the philosophers try to possess it but only one is successful. When the successful philosopher has finished eating he places the napkin back on the table. A single philosopher can dominate the napkin causing the others to starve.

A non-blocking algorithm ensures that operations competing for a shared resource never have their progress indefinitely postponed by mutual exclusion. A non-blocking algorithm guarantees that a philosopher will not starve as a result of mutual exclusion.

A non-blocking algorithm is obstruction-free if it guarantees that when an action is tried repeatedly and eventually encounters no interference from other actions it will complete successfully but it does not guarantee that such a situation will occur. An obstruction-free dining algorithm guarantees that a philosopher will be able to eat when the other philosophers are not attempting to eat.

Obstruction-free algorithms can suffer from the pathology of live-lock. Live-lock occurs when two or more competing operations cause each other to restart, preventing any of them making progress. A dining algorithm can live-lock when each of the philosophers reaches for both chopsticks simultaneously but withdraws when he observe his neighbour behaving likewise. All of the philosophers involved in the live-lock will starve.

Obstruction-free algorithms can also suffer from the pathology of priority inversion. Priority inversion occurs when a long running operation is preempted by an operation of brief duration. A dining algorithm in which a philosopher procrastinates when he has an opportunity to eat can suffer from priority inversion. The procrastinating philosopher might starve because he is continually interrupted by requests from other philosophers which prevent him from eating.

A non-blocking algorithm is lock-free if it guarantees that at least one action eventually completes. A dining algorithm is lock-free if it guarantees that at least one of the philosophers eventually eats. Lock-free algorithms can suffer from live-lock and priority inversion but these pathologies do not prevent all operations from making progress. In practice, live-lock and priority inversion are less likely to occur in an algorithm that guarantees lock-freedom than one that only guarantees obstruction-freedom.

A non-blocking algorithm is wait-free if it guarantees that eventually every action completes. A wait-free dining algorithm guarantees that all of the philosophers eventually get to eat. Wait-free algorithms are not prone to the pathologies of live-lock and priority inversion. All concurrent algorithms can be converted into implementations that are wait-free but the overheads of the conversion are prohibitive [Her88] [FHS04].

At a philosophy conference philosophers have a choice of tables at which different dining algorithms are used. The wait-free table is the best because all of the philosophers are guaranteed to eat eventually and the blocking table is the worst because all the philosophers may starve because of deadlock. A lock-free table is preferable to an obstruction-free table because at the lock-free table at least one philosopher does not starve. Wait-freedom is the strongest progress guarantee and the other guarantees are progressively weaker.

6.1.4 Previous work

Applications executing on Transactional Memory systems suffer from the progress pathologies of live-lock and dead-lock. In weakly isolated systems these pathologies can occur in combination with the isolation pathology of cascading aborts. Bobba categorises Transactional Memory pathologies and describes them in detail [BMV+07].

Many Software Transactional Memory implementations block at some point in their execution. Blocking Software Transactional Memory systems are easier to design than non-blocking systems. A blocking Software Transactional Memory hides blocking from the application programmer by implementing it internally [DDS06] [HPST06] [SATH+06].

Several Software Transactional Memory systems guarantee obstruction-freedom [HLMS03] [SCKP07] [CRS05] [TMG+09]. These systems are based on the concept of exclusive but revocable object ownership.

In an obstruction-free Software Transactional Memory system each transaction is associated with a descriptor which indicates whether the transaction is active, committed or aborted. Objects are owned by transactions and have an associated pointer to their owner which is modified by an atomic instruction. When a transaction reads an object it checks the pointer to determine whether another transaction already owns it.

A transaction takes ownership of an object by modifying the pointer so that it references the transaction's descriptor. Once a transaction has taken ownership of all the objects it will access it can commit its changes to those objects. This is done by changing the transaction descriptor from live to committed. This action will atomically commit the changes to all affected objects.

An obstruction-free Software Transactional Memory system can suffer from progress pathologies. Concurrent transactions can prevent each other from owning all of the objects they require, causing live-lock. Short running transactions can prevent long running transactions from obtaining all the objects they require, causing priority inversion.

Exclusive object ownership is a two-phase locking protocol which requires that all the locks that will ever be required by a transaction are acquired before any are released. Bernstein describes the two-phase locking concurrency control protocol in detail [BHG87]. Guerraoui explains that exclusive object ownership cannot provide the stronger guarantee of lock-freedom because systems based on two-phase locking cannot guarantee that at least one transaction will ever complete its operation, while other transactions are active [GK08].

Ennals argues that obstruction-free Software Transactional Memory systems are less scalable than their blocking counterparts [Enn06]. However, we believe that the best approach to overcoming scalability restrictions is to strengthen the progress guarantee, because a concurrent application that does not guarantee that all of its tasks eventually complete can hardly be described as scalable.

Bibliography

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[BMV$^+$07] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of the 34rd Annual International Symposium on Computer Architecture. International Symposium on Computer Architecture*, pages 81–91, 2007.

[CRS05] Joao Cachopo and Antonio Rito-Silva. Versioned boxes as the basis for memory transactions. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, October 2005.

[DDS06] O. Shalev D. Dice and N. Shavit. Transactional locking ii. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.

[Enn06] Robert Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, January 2006.

[FHS04] Faith Fich, Danny Hendler, and Nir Shavit. On the inherent weakness of conditional synchronization primitives. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, pages 80–87. ACM Press, 2004.

[GK08] Rachid Guerraoui and Michał Kapałka. On obstruction-free transactions. In *SPAA '08: Proc. twentieth annual symposium on Parallelism in algorithms and architectures*, pages 304–313, June 2008.

[Her88] Maurice P. Herlihy. Impossibility and universality results for wait-free synchronization. In *PODC '88: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 276–290, New York, NY, USA, 1988. ACM.

[HLMS03] Maurice Herlihy, Victor Luchangco, Mark Moir, and William Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proc. 22nd ACM Symposium on Principles of Distributed Computing*, pages 92–101, July 2003.

[Hoa83] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 26:100–106, January 1983.

[HPST06] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *Proceedings of the 2006 Conference on Programming language design and implementation*, pages 14–25. ACM Press, June 2006.

[Kri03] S. Krishnaprasad. Concurrent/Distributed programming illustrated using the dining philosophers problem. *J. Comput. Small Coll.*, 18:104–110, April 2003.

[Raj02] Ravi Rajwar. *Speculation-based techniques for transactional lock-free execution of lock-based programs.* PhD thesis, Department of Computer Science, 2002. Supervisor-Goodman, James R.

[SATH$^+$06] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *Proc. 11th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP '06)*, pages 187–197. ACM, March 2006.

[SCKP07] Jaswanth Sreeram, Romain Cledat, Tushar Kumar, and Santosh Pande. RSTM: A relaxed consistency software transactional memory for multicores. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, page 428. IEEE Computer Society, 2007.

[TMG$^+$09] Fuad Tabba, Mark Moir, James R. Goodman, Andrew Hay, and Cong Wang. NZTM: Nonblocking zero-indirection transactional memory. In *SPAA '09: Proc. 21st Symposium on Parallelism in Algorithms and Architectures*, August 2009.