# Contents

## 6.4 Distribution and Scheduling

The benefits of concurrent execution come at the cost of distributing and scheduling work and detecting any conflicts. In a Transactional Memory system the scheduler is regarded as a component of the transaction manager. This section describes how the scheduling problem can be reduced to one of load-balancing concurrent execution. A two-level scheduler intended for a parallel workload can be utilised to load-balance concurrent execution.

The overhead associated with distributing parallel work on a Chip Multi-Processor is high and for many workloads the overhead of distribution exceeds the benefit of parallel execution. The overhead associated with distributing and scheduling concurrent Memory Transactions is significantly higher than that associated with distributing a similar amount of parallel work because of the additional effort required to ensure correct concurrent execution.

The main contribution of this section is observation that, once isolation and progress pathologies have been eliminated, the problem of scheduling Memory Transactions is similar to that of distributing parallel work. This section focuses on using an existing two-level scheduler to schedule Memory Transactions.

### 6.4.1 Scheduling

Transactional Memory systems implement transaction scheduling strategies that do not make a distinction among the transaction management tasks of concurrency control, contention management and load-balancing.

Transactional Memory systems may try to improve the efficiency of concurrency control by scheduling transactions to avoid conflicts and reduce the overhead of wasted work. These benefits should be balanced against the scalability restrictions of centralised concurrency control.

Transactional Memory systems that make weak progress guarantees may schedule transactions to avoid progress pathologies. The benefits of guaranteed progress should be balanced against the scalability restrictions of centralised contention management.

A Transactional Memory system should execute a workload that is known not to contain conflicting tasks without incurring the overhead of concurrency control.

The problem of scheduling parallel work on a Chip Multi-Processor is solved

by using a two-level scheduler.

Section 6.4.3 describes the scheduling of parallel work on a Chip Multi-Processor.

The overhead of scheduling work on a parallel system imposes a lower limit on the size of chunks of work that are worthwhile scheduling and the additional overhead of concurrency control raises this limit further.

Section 6.4.5 describes how these limits influence the design of a transactional system.

An access function of an Immutable Data Structure is responsible for concurrency control, which alleviates the need for centralised concurrency control, and it also guarantees progress, which alleviates the need for centralised contention management. The only transaction management task that requires centralisation is scheduling.

A solution to the scheduling problem should isolate and simplify the scheduling component of transaction management and make it compatible with mechanisms for distributing parallel work.

### 6.4.2 Load-balance

The task of scheduling transactions can be simplified to the point that it is similar to that of load-balancing parallel work. This proposal satisfies the requirements because it isolates the scheduling task and provides a mechanism for scheduling tasks which are known not to conflict.

The overheads associated with scheduling concurrent work to reduce conflicts are difficult to justify through increased speed-up because scheduling around conflicts requires a centralised transaction manager and this restricts scalability.

The overheads associated with scheduling concurrent work to ensure progress are difficult to justify through increased speed-up because scheduling transactions to ensure progress requires a centralised view of contention management and this restricts scalability.

When these requirements are removed the problem of scheduling is reduced to one of load-balancing. If it is known that there are no dependencies between access functions then a parallel work scheduler can schedule them to be executed in parallel without the overhead of concurrency control.

The execution of an access function may be regarded as a Memory Transaction. The access functions of an Immutable Data Structure implement a distributed transaction manager internally. When a conflict is detected the transaction manager schedules the transaction for re-execution by adding it to the work-list of the scheduler.

The validate and meld functions are used to implement concurrency control in a Canonical Binary Tree. These functions can be wrapped by the functions which implement an ADT so a function acting on an Immutable Data Structure can be regarded as a chunk of work that can be scheduled by a two-level scheduler. If the validate function fails then the version of the Immutable Data Structure created by the function is discarded and the function may be re-tried. Re-try is implemented by placing the chunk of work back on the work-list.

### 6.4.3 Scheduling parallel work

The science of High Performance Computing focuses on the parallel execution of programs on supercomputers. Its main application is in the simulation of physical systems which evolve over time. Dowd provides a general introduction to High Performance Computing [Dow93]. Kumar describes how schedules for executing parallel work can be determined statically, by the analysis of parallel algorithms [KGGK94]. Parallel algorithms focus on orchestrating the execution of discrete units of work which can be performed in parallel.

The scheduling of parallel work on a Chip Multi-Processor is different from orchestrating parallel work on a supercomputer. Chip Multi-Processors generally have a lower number of processors than Supercomputers and each processor shares a common cache and a common path to main memory. The effects of caching mean that the tasks scheduled on separate execution units affect each other in ways that are difficult to predict. Mattson describes common parallel application design patterns, which are very different from those of High Performance Computing [MSM04].

The problem of scheduling parallel work on a Chip Multi-Processor cannot be addressed by static analysis of algorithms alone, so parallel work should be marshalled and load-balanced by a dynamic scheduler. A two-level scheduler implements a dynamic scheduling algorithm for parallel work. Two-level schedulers are designed to permit parallel workloads, such as the simulation of physical systems, to be efficiently executed by a Chip Multi-Processor.

Blumofe introduces CILK which is a two-level scheduling system for parallel workloads [BJK+96]. CILK implements a run-time scheduler which frees the programmer from static scheduling considerations. The programmer specifies chunks of work which can be performed in parallel by describing them using the CILK programming language. The chunks are assigned to processors by the high-level scheduler. The low-level scheduler marshalls the chunks to be performed by a particular processor.

The CILK scheduler implements a scheduling policy called work stealing. The low-level scheduler maintains a queue of chunks to be executed. It removes a chunk of work from the front of the queue and executes it. When the queue is exhausted the low-level scheduler steals chunks from the back of a queue belonging to another thread. This makes the scheduling task scalable, because the centralised high-level scheduler is only involved in the initial assignment of the chunks to the queues of each processor.

Intel's Threading Building Blocks product [Int09] is a parallel programming solution for Chip Multi-Processors. Threading Building Blocks applications are written in the C++ programming language and the Threading Building Blocks product is implemented as a library which is linked with the application. The product includes a two-level work stealing scheduler which dynamically schedules chunks of work provided to it on a work-list. This scheduler is similar to that provided by CILK. However, Threading Building Blocks frees the programmer from having to learn a new programming language in order to make use of a two-level scheduler. Reinders provides an accessible introduction to the features of the Threading Building Blocks product [Rei07].

### 6.4.4 Previous work

Ansari proposes a scheduling technique called Dynamic Transactional Reordering [ALK+09]. This technique reduces wasted work by re-trying conflicting transactions serially so that they do not repeatedly conflict. It also attempts to avoid both isolation and progress pathologies by implementing a transaction aware work stealing scheduler.

Ansari proposes a scheduling technique based on using information obtained by profiling transactional applications [AJK+09]. Profiling information can be used as input to a scheduler which anticipates conflicting transactions and schedules them to execute serially. Ansari notes that the speed-up obtained by reducing

wasted work does not always overcome the scheduling overheads.

The high overhead associated with the distribution and scheduling of parallel work can be contrasted with the low overhead of scheduling Memory Transactions assumed in the Transactional Memory literature. Warg describes how the overhead of thread creation prevents fine grained speculative execution on a Chip Multi-Processor from being worthwhile [WS01]. However, some studies of speculative execution assume that the time required to create and schedule a thread is lower than the access time to the first level cache. Quiñones describes an infrastructure for speculative execution which assumes a thread creation time of ten clock cycles [QnMS+05].

### 6.4.5 Transaction granularity

The overhead of scheduling concurrent work places a lower bound on the granularity of transactions that are worthwhile scheduling. Transaction granularity influences many aspects of Transactional Memory system design. Assumptions about transaction granularity influence the style of transactional programming a system permits. For example, at a fine level of transaction granularity it is possible for a compiler to analyse the instructions within a Memory Transaction, whereas at a coarser level of granularity the compiler is less able to reason about the execution.

At a fine level of transaction granularity the amount of speculative state produced by a Memory Transaction is small and the probability that transactions conflict is small, so the amount of work wasted when a conflict is detected is small and the likelihood of work being wasted is low. However, at a coarse level of transaction granularity large amounts of speculative state are produced and the probability of conflict is high, so the amount of work wasted when a conflict is detected is large and the likelihood of work being wasted is high.

To make use of a two-level scheduler the programmer divides an application into chunks of work that are large enough to be worthwhile scheduling. If the chunks are too small then the overheads associated with scheduling each chunk can outweigh the benefits of executing it in parallel with other chunks. If the chunks are too large then the scheduler may not be able to load-balance the work evenly among processors. In practice, it is often difficult to divide an application into suitably sized chunks because it is the expected execution time that determines chunk size. The execution time of the chunks is typically dominated by

the latency of cache misses, which are difficult to predict.

The overhead of scheduling concurrent work is high. Threading Building Blocks requires that a chunk of work should contain at least 10,000 instructions [Int09]. The documentation does not define an instruction in this context but assuming that an instruction completes each cycle, a chunk of work should have an elapsed execution time of at least 10,000 clock cycles to be worthwhile scheduling. In practice, it is difficult to divide an application into transactions which take at least 10,000 clock cycles to execute.

The number of clock cycles required to perform a data structure access is normally dominated by the latency of cache misses. Jacob found that the latency of a single cache miss is around 200 clock cycles and that the latency of consecutive cache misses to dis-contiguous locations is considerably longer [Jac09]. A function acting on a large memory resident data structure may require thousands of clock cycles to execute so functions that access a data structure are potentially worthwhile scheduling for concurrent execution.

# Bibliography


[AJK+09] Mohammad Ansari, Kim Jarvis, Christos Kotselidis, Mikel Luján, Chris Kirkham, and Ian Watson. Profiling transactional memory applications. In *PDP '09: Proceedings of the 17th Euromicro International Conference on Parallel, Distributed, and Network-based Processing*. IEEE Computer Society Press, February 2009.

[ALK+09] Mohammad Ansari, Mikel Luján, Christos Kotselidis, Kim Jarvis, Chris C. Kirkham, and Ian Watson. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In *High Performance Embedded Architectures and Compilers, Fourth International Conference*, pages 4–18, 2009.

[BJK+96] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *J. Parallel Distrib. Comput.*, 37(1):55–69, 1996.

[Dow93] Kevin Dowd. *High performance computing*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1993.

[Int09] Intel. Intel Threading Building Blocks: Programming for Current and Future Multicore Platforms. *IEEE/ACM International Symposium on Code Generation and Optimization*, July 2009.

[Jac09] Bruce L. Jacob. *The Memory System: You Can't Avoid It, You Can't Ignore It, You Can't Fake It*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[KGGK94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing: design and analysis of algorithms*.

Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[MSM04] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, 2004.

[QnMS+05] Carlos García Quiñones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 269–279, New York, NY, USA, 2005. ACM.

[Rei07] James Reinders. *Intel Threading Building Blocks - Outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.

[WS01] Fredrik Warg and Per Stenström. Limits on speculative module-level parallelism in imperative and object-oriented programs on cmp platforms. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 221–230, Washington, DC, USA, 2001. IEEE Computer Society.