Contents

4.3 Entanglement

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

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

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

4.3.1 Fine grained irregular parallelism

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

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

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

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

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

4.3.2 The composition of Immutable Data Structures

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

Expanding on our example, consider a process that removes lower case letters from a queue, which we call the parameter queue, converts them to upper case and places them on another queue, which we call the result queue. The presence of a letter in one or other of the queues, but not both, is an invariant of the algorithm. The queues are said to be in a consistent state when this invariant is true. A consistent state of the algorithm can be check pointed by recording a reference to one of the data structures in the leaf of the other. When a conflict

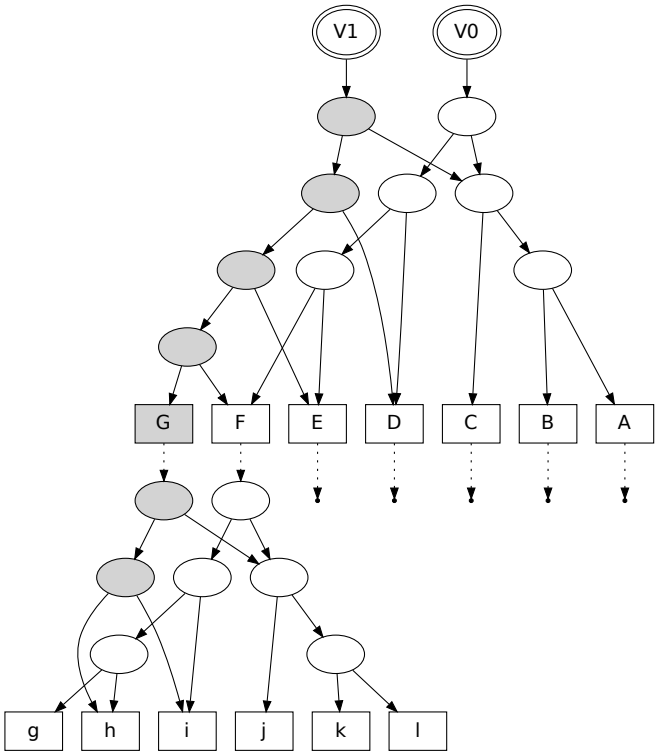

Figure 4.1: **A pair of entangled queues** is created by referencing the root of one queue from the leaf of another. In this example letters are removed from the parameter queue, shown in the lower part of the figure, converted to upper case and inserted into the result queue, shown in the upper part of the figure. In version V0 of the entangled data structure the parameter queue contains the lower case letter g. In version V1 the letter g has been removed from the parameter queue and the upper case letter G has been added to the result queue. The new root of the parameter queue is stored together with the converted letter in the result queue. In this figure prior versions of the parameter queue are hidden for clarity. The links from the leaves to the roots are dotted and the most recently created path is shaded.

is detected the reference in the leaf affected by the conflict indicates the root of the entangled structure at the moment in time prior to the conflict. The data structures can be rolled back to a consistent state by restoring this root. When the root is restored the invariants of both data structures are preserved. To ensure that the roll-back appears instantaneous the root of the entangled queues is modified atomically.

Figure 4.1 illustrates an operation on the two logically separate data structures which have been combined into a single structure by Entanglement.

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

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

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

4.3.3 Entanglement and Persistence

An Immutable Data Structure can be entangled with a past version by adding a reference to the root node of the version it was path copied from. This creates a link between a version of the data structure and a past version. A self-entangled Immutable Data Structure is persistent because past versions can be accessed by a look-up function which follows the links.

Figure 4.2 illustrates an immutable directed min-tree which is entangled in such a way that past versions can be accessed.

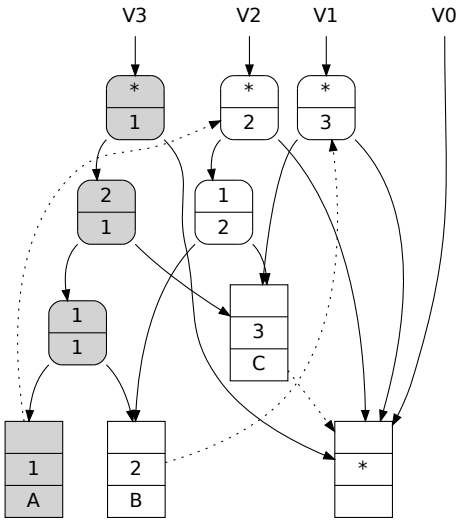

Figure 4.2: **A persistent Directed min-tree** is created by referencing the root of a version from the leaf of the past version from which the path was copied. In this example each leaf is linked to the version of the data structure from which it was created by path copy. The links from the leaves to the roots are dotted and the most recently created path is shaded. The Immutable Data Structure is persistent because past versions are accessible by a look-up function which follows these links.

4.3.4 Previous work

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

Conchon describes how a semi-persistent data structure can be used for check pointing and roll-back [CF08].

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

4.3.5 Low overhead check pointing

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

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

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

Bibliography

[CF08] Sylvain Conchon and Jean-Christophe Filliâtre. Semi-persistent data structures. In *Proceedings of the Theory and practice of software, 17th European conference on Programming languages and systems*, ESOP'08/ETAPS'08, pages 322–336, Berlin, Heidelberg, 2008. Springer-Verlag.

[HK08] Maurice Herlihy and Eric Koskinen. Checkpoints and continuations instead of nested transactions. In *TRANSACT '08: 3rd Workshop on Transactional Computing*, February 2008.

[RW02] Algis Rudys and Dan S. Wallach. Transactional rollback for language-based systems. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 439–448, Washington, DC, USA, 2002. IEEE Computer Society.

[WS08] M. M. Waliullah and Per Stenström. Intermediate checkpointing with conflicting access prediction in transactional memory systems. In *IPDPS, IEEE International Parallel and Distribued Processing Symposium*, pages 1–11, 2008.