Featherweight Transactions: Decoupling Threads and Atomic Blocks

Virendra J. Marathe
University of Rochester
vmarathe@cs.rochester.edu

Tim Harris James R. Larus
Microsoft Research
tharris@microsoft.com larus@microsoft.com

Categories and Subject Descriptors [*D.1.3 Concurrent Programming*]: Programming Abstractions

General Terms Algorithms, Languages

Keywords atomicity, transactional memory, data parallel programming, work groups

1. Introduction

Transactional memory is a powerful programming abstraction that enables a programmer to turn a complex, composite collection of statements into an atomic operation. Previous work usually expresses this abstraction as an atomic block, which offers mutual exclusion for code running on threads [3]. The implicit connection between transaction memory and threads has the unfortunate effect of limiting the use of transactions because threads are expensive to create and use in most systems.

This paper introduces an alternative form of transactional memory that supports much finer grain transactions. Featherweight transactions are atomic operations that execute to completion (commit, abort, or retry). They mesh very nicely with a data parallel programming style, in which each data parallel operation executes in a featherweight transaction. Section 2 describes our implementation.

Executing each data parallel operation in a separate memory transaction introduces asynchrony and composable synchronization into data parallel programming. An operation can use the `retry` construct to abort its computation and wait for values to change. When the values change, the operation re-executes. A collection of asynchronously executing operations can be treated profitably as a single data parallel operation.

To illustrate this, we built a highly-parallel implementation of the Chaff [4] SAT solver. Section 3 describes the parallel version of Chaff constructed using our new techniques. Chaff is typical of an important group of applications such as theorem provers and constraint optimization systems. These applications naturally exhibit large degrees of data-level parallelism that is difficult to exploit using existing data parallel paradigms.

2. Atomic Work Items

Memory transactions provide a concise mechanism for mutual exclusion and have been extended to offer condition synchronization through the `retry` language construct [1]. However, while investigating Chaff we realized that existing abstractions are not appropriate to express the parallelization of applications that contain a large amount of fine-grain data parallelism, but in which concurrent transactions interact in non-trivial ways. The main difficulty is correctly coordinating the transactions. The `retry` construct is a good starting point that offers a mechanism to convey information between concurrent transactions; however, a programmer must explicitly and carefully tie different concurrent computations (transactions) together to coherently get the application's desired behavior. We believe that the abstractions proposed in this paper significantly simplify this difficult aspect of implementing some important parallel applications.

Transactions are usually tied to a thread, a rather heavy-weight parallel programming abstraction. An executing transaction typically *owns* the host thread. No other transaction (with the exception of nested transactions) can be executed by the thread simultaneously because the owner transaction uses the host thread's runtime stack. Applications that might be best expressed as a large number (potentially in the order of millions) of fine-grain transactions are simply infeasible in this environment.

A key insight is that *completed* (committed, aborted, or retried) transactions do not require a runtime stack. They require a stack when they execute, but when terminated their stack frames can be reused for another transaction. To eliminate the stack frames preceding a transaction, we restrict the programming model by requiring that an entire method body be enclosed in a transaction. Transactions can now be multiplexed onto a thread provided each is guaranteed to run to completion on the thread.

A transaction performs a unit of work atomically. In data parallel programming, each data parallel operation can be executed in a transaction. We define a new abstraction called an *atomic work item* (henceforth called "work item") that associates a transaction with a particular data item in a data parallel aggregate. In our implementation, a work item is an instance of class `WorkItem` and is instantiated by passing a function *delegate* (an abstraction for type-safe function pointers) and the data as the `WorkItem`'s constructor parameters. The work item's data object is essentially the sole parameter to its delegate function. The runtime system executes a work item by invoking its delegate function and passing its data object as a parameter.

In Chaff, it is necessary to repeatedly executed work items when they are successful. To permit repeated execution of work items, we introduce the notion of *daemon* workers, which repeatedly execute work items after they commit (a work item is re-executed when it aborts due to data conflicts or blocks because of retry). In our implementation, a daemon is an instance of class `TxnWorker`. The programmer must suspend a daemon using a special `TxnWorker.Suspend()` method call. This prevents subsequent execution of the suspended daemon's work items.

PPoPP'07 March 14–17, 2007, San Jose, California, USA.
ACM 978-1-59593-602-8/07/0003.

Another useful abstraction is a group of work items, which we call a *work group* (referred to as `TxnGrps` in our implementation). Work groups correspond to aggregates in data parallel programming. Work groups provide a programmer with useful collective operations, such as starting execution of all work items in a group, waiting for all members of a group reach a quiescent state (using say the `TxnGrp.WaitForAll()` method), suspend all work items in a specific group, perform group level joins, splits and reductions, etc. We believe that there is great potential for far richer semantics of work groups, which we leave for future work.

A key problem was the semantics of exception handling in work groups. In prior work, exceptions reaching boundaries of an atomic block aborts the work done within the block and is rethrown to the enclosing context. In the context of atomic work items, an exception generated by a work item is considered to be an exception generated by the enclosing group. Thus, when a work item throws an exception, the entire group's activity is suspended and the exception is percolated to the thread that waits for the group to reach a quiescent state. Notice that multiple work items may simultaneously generate exceptions in a group. All but one exception is suppressed. It may be valuable to permit dispatch of multiple exceptions from a work group, but we leave that design for future work.

There are several other important operations on these abstractions that are useful for the underlying runtime system as well as for user programmers, but we will not discuss them due to space restrictions. We have implemented our abstractions in the Bartok STM system [2].

3. Parallelizing ZChaff

Since the boolean satisfiability problem (SAT) is NP-complete, there is no efficient solution to this problem. Existing solvers rely on heuristics to make literal assignment decisions. However, most, if not all, SAT solvers rely on the standard *boolean constraint propagation (BCP)* algorithm to propagate implied literal assignments once an explicit literal assignment (suggested by the decision heuristic) is made. It is also widely known that BCP is the most time consuming (roughly about 80% of the execution time) operation in any SAT solver. We focus on this BCP component of ZChaff (a C# implementation of Chaff) in our parallelization.

Our implementation of ZChaff processes formulas in the CNF SAT form. In the sequential version, whenever an explicit literal assignment is made, say l, it is posted in a global implication queue. The BCP algorithm thereafter gets the implication queue's first literal entry and looks up the clauses containing the *negation* of that literal ($\neg l$ in our example). Since l is assigned the value `true` its negation $\neg l$ is `false`. ZChaff then determines if any clause containing $\neg l$ contains a *single* unassigned literal and if all other literals have the value `false`. If so, the unassigned literal is implied to be `true` and is in turn posted in the implication queue. After processing all clauses corresponding to $\neg l$, the algorithm checks if a new implication queue entry was added and processes it in a similar fashion.

A coarse-grain method of parallelizing ZChaff is to fork off two threads at the point at which an explicit literal assignment is made; one thread takes the literal and the other takes its negation. In existing implementations, this approach has led to performance improvements that vary widely based on the input formula. An alternate, fine-grain parallelization approach focuses on the BCP component of SAT solvers wherein "computational units" are dedicated to process distinct sets of clauses in the SAT formula. An explicit literal assignment triggers activity in these computational units that collectively perform the BCP task.

Fine-grain parallelization has definite benefits, provided the concurrency achieved is sufficient to offset the co-ordination cost involved. The BCP component is highly parallel (thousands or millions of clauses), but the computations are fine grained. Directly expressing them may lead to an unmanagably large number of threads (computational units). In addition, writing such an application is difficult because of the difficulty of explicitly controlling co-ordination among these computational units. We believe that our atomic work item abstractions mitigate these difficulties.

Using our abstraction, parallelizing ZChaff is simple: dedicate a distinct daemon worker for each clause in the formula. The computation for each work item starts by reading the variables in its clause. If there exists a literal assignment that may lead to an implied literal assignment, make that literal assignment and commit. If there is no such literal assignment, then simply retry (and wait for the clause's variables to change). A co-ordinator thread manages explicit literal assignments in the formula. After making the literal assignment, the main thread waits for completion of BCP activity by calling `WaitForAll()` on the work group.

If a clause evaluates to `false` during BCP, an exception is raised by the relevant work item, which in turn suspends execution of the entire work group. The `WaitForAll()` method called by the main thread returns this exception. On receiving an exception, the main thread generates a *conflict clause*, adds it to the existing list of clauses, rolls back literal assignments up to a point at which there is no conflicting literal assignment, and resumes with more explicit literal assignments. Conflict clauses are of great value in pruning large search spaces in SAT solvers. The input formula is designated unsatisfiable if rollback happens all the way beyond the very first explicit literal assignment made by the main thread. If the algorithm cannot make any new explicit literal assignments and there are no outstanding conflicts, the formula is designated satisfiable.

We are implementing our runtime system and parallel ZChaff.

References

[1] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable Memory Transactions. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60, 2005.

[2] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing Memory Transactions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 14–25, 2006.

[3] J. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2007.

[4] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings on the 38th Design and Automation Conference*, pages 530–535, 2001.