Automatic Specialized Synchronization for Dynamic Data Structures

August 24, 2014

1 Introduction

2 Preliminaries

A *module* defines a set of *abstract data types* and a set of *operations* that may be invoked by clients of the module. Each operation is invoked with possible parameters and returns with a response. The invocation is a local step of a thread, followed by a execution of a sequence of atomic steps. Atomic step is either a computation on local variables or a primitive operation on shared variable. Read and write to shared memory are denoted load and store. The parameters and local variables of an operation are private to the invocation of the operation (thread local). There are no static or global variables shared by different invocations of the operations.

A *configuration* is an instantaneous representation of the system, including the state of the shared memory and the local variables. In the *initial configuration* all variables hold an initial value. An *execution* is an alternating sequence of configurations and steps, $C_0, s_1, \ldots, s_i, C_i, \ldots$, where C_0 is the initial configuration, and each configuration C_i is the result of executing step s_i on configuration C_{i-1}.

An execution is *non-interleaved* (abbreviated NI-execution) if primitive operation of different operations are not interleaved, i.e., for every pair of operations invocations $p_i \neq p_j$ either all primitive operations of p_i come before any primitive operation of p_j, or vice versa.

Given an execution, we say that two primitive operations *conflict* if (i) they are executed by two different threads, (ii) they access some common global variable or a heap allocated object (iii) at least one of the conflicting instruction is a write.

Executions π_1 and π_2 are *conflict-equivalent* if they include the same primitive operations and they agree on the order between conflicting operations. An execution is *conflict-serializable* if it is conflict-equivalent with a non-interleaved execution.

2.1 Domination Locking

Domination Locking Protocol is a conflict serializable locking protocol presented in [1]. It requires a fixed total order $\leq$ on all heap objects. An execution satisfies the the Domination Locking protocol, with respect to $\leq$ if it satisfies the following conditions:

1. A thread t can access a field of an object u only if u is currently locked by t.
2. ...
3. ...
4. ...

3 Automatic Optimistic Synchronization

We address the problem of adding an optimistic synchronization to a module. The idea of optimistic synchronization is to read shared variables without locks and start using a locking protocol only when the operation reaches a store step. In order to implement such synchronization scheme, we divide the operation to three *phases*:

Read Phase Stats at the beginning of the operation and ends at any point before the first store operation. During this phase the operation maintains a read_set, containing references to objects loaded and any other information needed to later ensure that the read saw a consistent snapshot of the data.

Validation Phase This phase connects the read phase with the read-write phase. It has two requirements: (i) lock local variables of the operation and (ii) ensure that the values read during the read phase are consistent, i.e. as if the values were read while executing the locking protocol. To avoid deadlocks, the locks are acquired using a try_lock operation, if the try_lock fails, the operation restarts from the beginning. Next, the read_set is validated, if the validation fails the operation restarts from the beginning.

Read-Write Phase This phase enforces domination locking protocol as described in [1]. Once the read-write phase begins, the operation is guaranteed to to finish without restarts.

3.1 Global Version Algorithm

The module maintains a counter denoted *global version*. In high level, the global version is used to identify the order of write operations and freshness of objects.

In the beginning of the read phase, the operation atomically reads the global counter and saves the returned value in a read_version variable. After each load step, the operation checks that the version field of the object read is not larger than the value of read_version. If the checks fails, the operation restarts from the beginning (with a new read_version value). Since the version of the object is not changed atomically with the write to the object, the object is also checked to be unlocked. In this implementation the read_set contains only references to all objects read by the operation.

The validation phase have an additional requirement, to acquire a unique write_version. To ensure that versions of objects are not decremented, acquiring the write_version needs to be done atomically with the validation of the read_set. One possibility is holding a lock on the global version during the validation, an optimistic approach is to read the global version before validation and incrementing it using a CAS operation after the validation. In the read_set validation each object's version is compared with the read_version, and is checked to be unlocked.

The only addition to the read-write phase is writing the operation's write_version to every object that is locked.

3.2 Local Version Algorithm

Each objects maintains a counter, incremented every time the object is locked. During the read phase, the read_set maintains both object reference and the local version when it was read. The local versions are incremented during the read-write phase, when the object is locked. Incrementing the version is not atomic with the lock, thus, the object is also checked to be unlocked. The read phase does not validate reads, in order to avoid infinite loops, a timeout is set. If the operation reaches the timeout, a read_set validation takes place, if it fails the operation restarts from the beginning.

The read_set validation first checks that the node is unlocked, (or locked by the current operation), then it checks that the current version is equal to the version saved in the read_set.

During the read-write phase, the operation increment the local version of every node that it locks.

4 Algorithm's Correctness

Let π be an execution of our optimistic automation on a sequential algorithm. We will construct an execution π_{DL} which is an execution following domination locking. We will prove that both executions are conflict-equivalent. Since any execution of domination locking is conflict-serializable, this proves that π is conflict-serializable.

Let $p_1, p_2, \ldots, p_n$ be the operations $\in \pi$ ordered by the order of execution of the first step of a successful read_set validation. (If some operation does not have such point we omit it). Let $\pi_{DL} = \pi_{dl1}, \pi_1, \ldots, \pi_{dli}, \pi_i$ where π_{dli} is a p_i-only execution of domination locking from the root until p_i holds locks only on the local variable locked in the validation phase of $p_i \in \pi$, and π_i is the interval of π starting from the return from the validation of pi until the first step of the successful read_set validation of p_{i+1} that includes only the operations by $\{p_1, \ldots, p_i\}$. In other words, we replace the read-phase and validation phase with an execution of domination locking, taking place at the point just before the read_set validation starts.

Lemma 4.1 *The construction of π_{DL} is feasible.*

Proof Proof by induction on $t_1, t_2, \ldots, t_n$. Base case is immediate.

Let $\pi' = \pi_{dl1}, \pi_1, \ldots, \pi_{dlk-1}, \pi_{k-1}$ be the feasible construction so far, and let t_k be the next operation to be added.

Assume by contradiction that $\pi' \cdot \pi_{dlk} \cdot \pi_k$ cannot be constructed, thus, some object v that p_k locks in π_{dlk} is already locked by $p_j \in \{p_1, p_2, \ldots, p_{k-1}\}$ in the last configuration of π'. If p_j locked v before p_k read v for the first time, then v was locked during the read phase of p_k, in contradiction to p_k reaching its validation. Otherwise, p_j locked v after p_k read v. If v is still locked during the validation of p_k then the validation will fail, contradiction. Alternatively, v was unlocked by p_j before p_k validated v, its version incremented, either to a version bigger than the local version read by p_k (in local version mode), or to a version larger than p_k's read_version (in global version mode), contradicting the successful validation of p_k. ∎

Lemma 4.2 *π_{DL} is conflict-equivalent to π*

Proof Each operation performs a double collect on all the values it reads. The first collect is the read phase and the second is the read_set validation of the validation phase. Since validation was successful, both collect are identical, meaning that the values of the read_set do not change from the return of the last read of the read phase, until the first read of the read_set validation. Therefore, executing domination locking of p_k after $\pi' = \pi_{dl1}, \pi_1, \ldots, \pi_{dlk-1}, \pi_{k-1}$ is conflict-equivalent to the original read phase. The read-write phase remains unchanged, maintaining conflict-equivalence to π. ∎

Lemma 4.3 *In the global version algorithm,* read_set *validation is not required for read-only operations.*

Proof different construction… ∎

References

[1] Guy Golan-Gueta, Nathan Grasso Bronson, Alex Aiken, G. Ramalingam, Mooly Sagiv, and Eran Yahav. Automatic fine-grain locking using shape properties. In *OOPSLA*, pages 225–242, 2011.