Scalable Automatic Synchronization Combining Optimism with Pessimism

September 3, 2014

Abstract

We present an *automatic* approach for parallelizing sequential data structures in a way that is both safe and scalable. While there exist pessimistic transformations that make code thread-safe by adding (either global or fine-grained) locks, this approach is limited in its performance due to synchronization bottlenecks, for example, locking the root in a tree data structure. In this paper, we improve the performance and scalability of such synthesized code by reducing bottlenecks. Specifically, we present an automatic approach to eliminate many of the locking steps, relying instead on optimistic partial traversals of the data structure. We realize our approach for tree data structures, using the domination locking technique. The resulting code scales well, significantly outperforms pessimistic approaches, and achieves performance close to those of custom-tailored concurrent data structures. Our work thus shows the promise that automated approaches bear for overcoming the difficulty involved in manually hand crafting concurrent data structures.

1 Introduction

2 Preliminaries

2.1 Shared Memory Model

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.

2.2 Locking Protocols

Get some text-book definitions of locking protocols.

In order for our algorithm to work, the initial locking protocol must have one of the following properties:

- The protocols allows early lock release.
- The protocols requires that all locks are acquired before the first store step.

2.3 Correctness Conditions

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.

3 ...

We address the problem of introducing optimism to a pessimistic locking protocol. Given an implementation using a pessimistic locking protocol, performing load and store steps on locked objects only, we add optimistic synchronization. The idea of optimistic synchronization is to load shared variables without locks and start using the locking protocol only when the operation reaches a store step. Our algorithm shows that it is redundant to acquire locks if they are freed before any change is made.

This optimistic scheme separates the operation to three *phases*, an optimistic *read phase*, a pessimistic *read-write phase* and a *validation phase* that connect them.

3.1 Detailed Algorithm

Each object maintains a counter, incremented every time the object is locked. This counter is used to validate the correctness of the optimistic read phase.

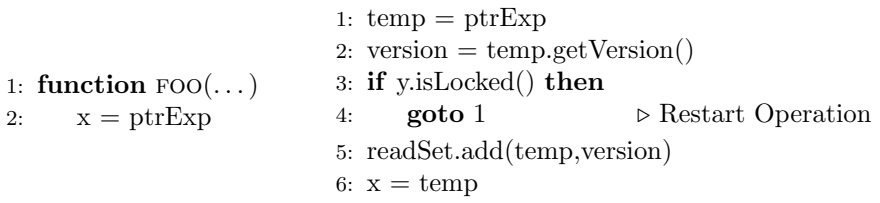

Figure 1: Read phase transformation, the code in line 2 is replaced with the code on the right.

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 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 use of timeout does not guarantee that the operation reads a consistent snapshot of the memory, thus, ...

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. During the read_set validation, each reference saved in the read_set is checked to be unlocked and that the current version matches the version saved in the read_set.

Read-Write Phase This phase enforces the locking protocol while maintaining the local versions, i.e., the local version of an object is incremented every time it is locked. Once the read-write phase begins, the operation is guaranteed to to finish without restarts.

4 Algorithm's Correctness

We will prove that if the original locking protocol is conflict-serializable then our algorithm is conflict-serializable.

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

```
1: function FOO(...)
   ...
                                                  ▷ begin read_set validation
2:     for all (ref,version)∈ readSet do
3:         if ref.isLocked() and ref.lockedBy!= self then
4:             goto 1                                  ▷ Restart Operation
5:         if version!= ref.getVersion() then
6:             goto 1                                  ▷ Restart Operation
   ...
```

Figure 2: read_set validation

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 $\pi_{LP} = \pi_{lp1}, \pi_1, \ldots, \pi_{lpi}, \pi_i$ where π_{lpi} is a p_i-only execution of original locking protocol 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 the original locking protocol, taking place at the point just before the read_set validation starts.

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

Proof Proof by induction on $p_1, p_2, \ldots, p_n$. Base case is immediate.

Let $\pi' = \pi_{lp1}, \pi_1, \ldots, \pi_{lpk-1}, \pi_{k-1}$ be the feasible construction so far, and let p_k be the next operation to be added.

Assume by contradiction that $\pi' \cdot \pi_{lpk} \cdot \pi_k$ cannot be constructed, thus, some object v that p_k locks in π_{lpk} 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 to a version bigger than the local version read by p_k, contradicting the successful validation of p_k. ■

Lemma 4.2 *π_{LP} 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 the original locking of p_k after $\pi' = \pi_{lp1}, \pi_1, \ldots, \pi_{lpk-1}, \pi_{k-1}$ is conflict-equivalent to the original read phase. The read-write phase remains unchanged, maintaining conflict-equivalence to π. ■

Figure 3: Throughput of unbalanced data structures.

Figure 4: Throughput of balanced data structures.

5 Evaluation

In order to apply our approach to a data structure, a "black-box" pessimistic locking is required. One such approach was presented in [4], automatically applying domination locking protocol to forest based data structures. We applied domination locking followed by our optimistic transformation to two tree based data structures, a simple unbalanced binary search tree and a treap (randomized binary search tree) [1].

Setup We compared the performance of our automatic implementations, the automatic binary search tree (Auto-Tree) and the automatic treap (Auto-Treap), to the following custom tailored approaches:

- LO-Tree- The locked-based unbalanced tree of Drachsler at al.[3].
- LO-AVL- The locked-based relaxed balanced AVL tree of Drachsler et al.[3].
- OPT-Tree- The locked based relaxed balanced AVL tree of Bronson et al.[2].
- Skip-List- The non-blocking skip-list by Doug Lea included in the the Java standard library.

We also compared our algorithms to previous automatic approaches, mainly global locking (Lock-Tree, Lock-Treap) and domination locking (Domination-Tree, Domination-Treap).

We ran our experiments on a ...

We evaluated the performance on a variety of workloads, each workload is defined by the percentage of read-only operations (GET queries) and the remaining operations are divided equally between insert and delete operations. Our workloads include heavy read-only workloads (100%,70% GET operations), medium read-only workload (50% GET operations) and update only workload (0% GET operations).

We used two key ranges $[0, 2 \cdot 10^5]$ and $[0, 2 \cdot 10^6]$, for each range, the tree was pre-filled until the tree size was within 5% of half the key range.

We ran five seconds trials measuring the total throughput (number of operations per second) of all threads. During the trial, each thread continuously executed randomly chosen operations according to the workload distribution using uniformly random keys from the key range.

We ran every trial 7 times and calculated the average, with a warm-up phase before the actual trials. We eliminated two ..., reporting the average of the remaining 5 experiments.

Results Figure 3 reports the throughput of unbalanced data structures and Figure 4 reports the throughput of the balanced data structures.

6 Related Work

References

[1] Cecilia R. Aragon and Raimund Seidel. Randomized search trees. In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 540–545, 1989.

[2] Nathan Grasso Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*, pages 257–268, 2010.

[3] Dana Drachsler, Martin T. Vechev, and Eran Yahav. Practical concurrent binary search trees via logical ordering. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 343–356, 2014.

[4] 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.