184.726 Advanced Multiprocessor Programming
Project 10: Wait-Free Linked List

Severin Jäger, 01613004

June 25, 2020

Contents

1 Introduction

The scope of this project was the implementation of the basic version of the wait-free linked list presented in [1]. By exhaustive benchmarking, the advantages, but also the cost, of wait-free lists were examined. As a reference algorithm, the lock-free linked list from [2] was implemented as discussed in the lecture.

2 The Wait-Free List Data Strcuture

Lock-free data structures can relatively easily be constructed from the atomic compare-and-set (CAS) instruction, as the CAS only fails if some other thread has made progress. Achieving wait-freedom requires additional synchronisation between the threads. In the list implementation discussed here, this is done by excessive helping between the threads.

Basically, the list maintains an array of all pending list operations. Whenever a thread initiates an contains, add or delete operation, it publishes an `OperationDescriptor` to this array. The key to wait-freedom is that every thread executing some list operation in the following iterates over this array and helps previous operations. The order of the operations in determined by a `phase`. So it is ensured that all preceeding operations returned before the new operation is conducted. Therefore, maximum response times (or at least maximum numbers of instructions) can be guaraneed. This might be useful for real-time applications, however, it comes with a significant overhead which is discussed in the following.

The wait-free list offers the following operations:

- `contains`
- `add`
- `remove`

It is claimed in [1] that all these operations can be implemented in a wait-free and linearizable manner. All three operations internally use the `search` method, which satisfies the same progress and correctness guarantees.

The authors admit that the the wait-free list performs significantly worse[1] than the lock-free list presented by [2]. However, they came up with some optimizations, which limit the extent of helping. Furthermore, the present a fast-past-slow-path (FPSP) algorithm combining the wait-free and the lock-free approach while maintaining wait-freedom. They claim that this algorithm almost reaches the performance of the lock-free data structure.

3 Implementation Details

The wait-free data structure and the lock-free reference were implemented as C++ classes. The implementation of the lock-free list closely followed the lecture slides. For the lock-free list, the Java implementation from [1] was ported to C++. The most important differences are described in the following.

[1] By a factor of 1.3 to 1.6

One significant advantage of Java for the implementation of concurrent data structures is the presence of the garbage collector, who takes care of freeing memory. However, garbage collection does not provide strict progress guarantees such as lock- or even wait-freedom. In this project memory management was not treated, so the memory leaks. This is unacceptable for practical application, however meaningful benchmarking is still possible with proper experiment design.

Furthermore, the a `VersionedAtomicMarkableReference` is introduced in the Java implementation, which is used to mark nodes as deleted logically and to avoid the ABA problem by the use of pointer versions. In this project the MSB of a pointer is used as flag for logical deletion and the following 15 bits provide a version of the pointer. This is possible because the pointers on the benchmarking system only utilize the lowest 48 bits of the 64 bit pointers.

The Java implementation also provides a `contains` method which is integrated into the helper mechanism. However, [3] proposes a very simple contains method and claims that is wait-free. The authors of [1] mention that wait-freedom can not be guaraneed under presence of unbounded `add` operations. Nontheless, this project focusses on making previously not wait-free operations (`add` and `remove`) wait-free, therefore both implementations use the simple `contains` method from [3].

4 Benchmarking Results

The benchmarks conducted serve two purposes: Firstly, the performance of the wait-free list was evaluated and compared to the lock-free reference in order to reproduce the results of [1]. Additionally, the drawbacks of the wait-free data structure are studied closer, focussing on the CAS misses.

5 Summary and Outlook

References

[1] S. Timnat, A. Braginsky, A. Kogan, and E. Petrank, "Wait-free linked-lists," *ACM SIGPLAN Notices*, vol. 47, pp. 309–310, Sep. 2012.

[2] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," DISC '01, pp. 300–314, 2001.

[3] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*, 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012, ISBN: 9780123973375.