Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects

Maged M. Michael

July 2, 2024

1 Introduction

A shared object is **lock-free** (also called nonblocking) if it guarantees that whenever a thread executes some finite number of steps toward an operation on the object, some thread (possibly a different one) must have made progress toward completing an operation on the object, during the execution of these steps.

The core idea is to associate a number (typically one or two) of single-writer multireader shared pointers, called **hazard pointers**, with each thread that intends to access lock-free dynamic objects. A hazard pointer either has a null value or points to a node that may be accessed later by that thread without further validation that the reference to the node is still valid. Each hazard pointer can be written only by its owner thread, but can be read by other threads.

The methodology requires lock-free algorithms to guarantee that no thread can access a dynamic node at a time when it is possibly removed from the object, unless at least one of the thread's associated hazard pointers has been pointing to that node continuously, from a time when the node was guaranteed to be reachable from the object's roots. The methodology prevents the freeing of any retired node continuously pointed to by one or more hazard pointers of one or more threads from a point prior to its removal.

Whenever a thread retires a node, it keeps the node in a private list. After accumulating some number R of retired nodes, the thread scans the hazard pointers of other threads for matches for the addresses of the accumulated nodes. If a retired node is not matched by any of the hazard pointers, then it is safe for this node to be reclaimed. Otherwise, the thread keeps the node until its next scan of the hazard pointers.

By organizing a private list of snapshots of nonnull hazard pointers in a hash table that can be searched in constant expected time, and if the value of R is set such that $R = H + \Omega(H)$, where H is the total number of hazard pointers, then the methodology is guaranteed in every scan of the hazard pointers to identify $\Theta(R)$ nodes as eligible for arbitrary reuse, in $O(R)$ expected time. Thus, the expected amortized time complexity of processing each retired node until it is eligible for reuse is constant.

2 Rreliminaries

2.1 The Model

Informally, in this model, a set of threads communicate through primitive memory access operations on a set of shared memory locations. Threads run at arbitrary speeds and are subject to arbitrary delays. A thread makes no assumptions about the speed or status of any other thread.

A shared object occupies a set of shared memory locations. An object is an instance of an implementation of an abstract object type, that defines the semantics of allowable operations on the object.

2.2 Atomic Primitives

In addition to atomic reads and writes, primitive operations on shared memory locations may include stronger atomic primitives such as compare-and-swap (CAS) and the pair load-linked/store-conditional (LL/SC).

LL takes one argument: the address of a memory location, and returns its contents. SC takes two arguments: the address of a memory location and a new value. Only if no other thread has written the memory location since the current thread last read it using LL, the new value is written to the memory location, atomically. A Boolean return value indicates whether the write occurred. An associated instruction, Validate (VL), takes one argument: the address of a memory location, and returns a Boolean value that indicates whether any other thread has written the memory location since the current thread last read it using LL.

For practical architectural reasons, none of the architectures that support LL/SC (Alpha, MIPS, PowerPC) support VL or the ideal semantics of LL/SC as defined above. None allow nesting or interleaving of LL/SC pairs, and most prohibit any memory access between LL and SC. Also, all such architectures, occasionally—but not infinitely often—allow SC to fail spuriously; i.e., return false even when the memory location was not written by other threads since it was last read by the current thread using LL.

2.3 The ABA problem

3 The Methodology

Observation 3.1. In the vast majority of algorithms for lock-free dynamic objects, a thread holds only a small number of references that may later be used without further validation for accessing the contents of dynamic nodes, or as targets or expected values of ABA-prone atomic comparison operations.

The core idea of the new methodology is associating a number of single-writer multireader shared pointers, called **hazard pointers**, with each thread that may operate on the associated objects. The number of hazard pointers per thread depends on the algorithms for associated objects and may vary among threads depending on the types of objects they intend to access. Typically, this number is one or two. For simplicity of presentation, we assume that each thread has the same number K of hazard pointers.

3.1 The Algorithm

```
// Hazard pointer record
struct HPRecType {
    NodeType *HP[K];
    HPRecType *Next;
};
// The header of the `HPRec` list
HPRecType *HeadHPRec;
// Per-thread private variables
listType rlist; // initially empty
int rcount;     // initially 0
```

Listing 1: Types and structures

4 Problems

1. 2.3

5 References

References