Lock-Free Data Structures with Hazard Pointers

Andrei Alexandrescu Maged Michael

October 16, 2004

I am very honored to introduce Maged Michael as a coauthor of this Generic⟨Programming⟩ installment. Maged is a leading authority in lock-free algorithms, and has found incredibly ingenious solutions to hard problems [3]. We'll be presenting one such exciting solution in the article you're now reading, having Andrei fulfill the modest role of kibitz-in-charge. The algorithm described henceforth manipulates limited resources to achieve goals in a "so cool it's indistinguishable from magic" way.

So if Generic⟨Programming⟩ were a soap opera, this episode is going to shoo the villain away. To remind who the villain was, let's review the previous episode [1]. (By the way, call it author lock-in, but the rest of this article assumes you have given at least a in-the-subway read of the previous installment of Generic⟨Programming⟩, and that you are somewhat acquainted with the problem we're addressing.)

So, the past article, after introducing the concept of lock-free programming, set out to implement lock-free "Write-Rarely-Read-Many" maps. These structures are often encountered in the form of global tables, factory objects, caches, etc. The problem was memory reclamation: given that things are lock-free, how can memory be freed? As discussed, this is a tough problem because lock-free implies unrestricted opportunity for a thread to operate on any object, at any time. To address that problem, the past article tried to:

- *Use reference counting with the map.* That approach was doomed to failure because it needed to update the pointer and the reference count (sitting at a different location) at the same time (atomically). In spite of a small flurry of papers using the elusive DCAS (Double Compare-And-Swap) instruction that does exactly that, the concept never caught on because we can do a lot without it and it is not powerful enough for implementing arbitrary transactions efficiently. See [2] for a discussion of the usefulness of DCAS.

- *Wait and delete.* Once a thread figures out a pointer is to be **`delete`**d, it can wait for a "sufficiently long time" and then **`delete`** it. The problem is deciding how long to wait. This solution sounds an awful lot like "allocate a large enough buffer"—and we all know how well that works.

- *Keep a reference count next to the pointer.* That solution uses the less-demanding, reasonable-to-implement CAS2 primitive, which is able to atomically compare-and-swap two adjacent words in memory. Most 32-bit machines have it, but not a lot of 64-bit machines (on the latter, however, there are tricks you can play with the bits inside the pointer).

That last approach turned out to work, but it also turned our shiny "Write-Rarely-Read-Many" map into a much less glorious "Write-Rarely-Read-Many-But-Not-Too-Many" map. This is because writers need to wait for a quantum of time when there are *absolutely zero* readers. As long as at least one reader starts before all other readers finish using the map, the writer threads wait powerlessly—and that ain't lock-free anymore.

Detective Bullet entered his debtor's office, sat on a chair, lit his pipe, and uttered calmly with a tone that could have frozen the boiling coffee in the pot: "I am not leaving until I have my money back."

A possible approach using reference counting is to separate the reference counter from the pointer. Now writers will not be blocked by readers anymore, but they can free replaced maps only after they observe the reference counter with a zero value [5]. Also, using this approach we only need single word CAS, which makes the technique portable to architectures that don't support CAS2. However, there is still a problem with that approach. We didn't eliminate the wait, we just deferred it—and gave opportunity for more damage in terms of memory consumption.

Detective Bullet coyly opened his debtor's office door and asked hesitantly: "Do you have my $19,990 yet? No? Oh, no problem. Here is my last $10. I'll drop by some other time to see if you got my $20,000."

There is no bound on the number of replaced maps that are kept by the writers without being reclaimed waiting—possibly forever—for the reference count to go down to zero. That is, the delay of even one reader can prevent unbounded amount of memory from being freed, and the longer that reader is delayed, the worse it gets.

What's really needed is some mechanism for readers to tell writers to not reclaim replaced maps from under them, but without allowing the readers to force the writers to hang on to unbounded number of replaced maps. There is a solution [3] that is not only lock-free, but actually wait-free. (To recap the definitions in our previous installment: lock-free means that progress is guaranteed for *some* thread in the system; the stronger wait-free means that progress is guaranteed for *all* threads.) Moreover, this method asks for no special operations—no DCAS, no CAS2, only the trusty CAS. Interested? Read on.

1 Hazard Pointers

To bring up the code again, we have a template `WRRMMap` that holds a pointer to some classic single-threaded `Map` object (think `std::map`) and provides a multithreaded lock-free access interface to it:

```
template <class K, class V>
class WRRMMap {
  Map<K, V> * pMap_;
  ...
};
```

Whenever the `WRRMMap` needs to be updated, the thread wanting to do so creates an entire new copy of the map pointed to by `pMap_`, replaces `pMap_` with that new copy, and then disposes of the old `pMap_`. We agreed that that's not an inefficient thing to do because `WRRMMap` is read often and updated only rarely. The nagging problem was, how can we dispose of `pMap_` properly, given that there could be other threads reading through it at any time?

Hazard pointers are a safe, efficient mechanism for threads to advertise to all other threads about their memory usage. Each reader thread owns a single-writer multi-reader shared pointer called *hazard pointer*. When a reader thread assigns the address of a map to its hazard pointer, it is basically announcing to other threads (writers) "I am reading this map. You can replace it if you want, but don't change its contents and certainly keep your **`delete`**ing hands off it."

On their part, writer threads have to check the hazard pointers of the readers before they can **`delete`** any replaced maps. So, if a writer removes a map after a reader (or more) has already set its hazard pointer to the address of that map, then the writer will not **`delete`** that map as long as the hazard pointer remains unchanged.

Whenever a writer thread replaces a map, it keeps the old pointer in a private list. After accumulating some number of removed maps (we'll discuss later how to choose that number), the thread scans the hazard pointers of the readers for matches for the addresses of the accumulated maps. If a removed map is not matched by any of the hazard pointers, then it is safe for this map to be deallocated. Otherwise, the writer thread keeps the node until its next scan of the hazard pointers.

Below are the essential data structures used. The main shared structure is a singly-linked list of hazard pointers (`HPRecType`), pointed to by `pHead_`. Each entry in the list contains the hazard pointer (`pHazard_`), a flag that tells whether the hazard pointer is in use or not (`active_`), and the obligatory pointer to the next node (`pNext_`).

HPRecType offers two primitives: Acquire and Release. HPRecType::Acquire gives a thread a pointer to a HPRecType, call it p. From then on, that thread can set p->pHazard_ and rest assured that all other threads will tread carefully around that pointer. When the thread does not use the hazard pointer anymore, it will call HPRecType::Release(p).

```
// Hazard pointer record
class HPRecType {
  HPRecType * pNext_;
  int active_;
  // Global header of the HP list
  static HPRecType * pHead_;
  // The length of the list
  static int listLen_;
public:
  // Can be used by the thread
  //   that acquired it
  void * pHazard_;

  static HPRecType * Head() {
    return pHead_;
  }

  // Acquires one hazard pointer
  static HPRecType * Acquire() {
    // Try to reuse a retired HP record
    HPRecType * p = pHead_;
    for (; p; p = p->pNext_) {
      if (p->active_ ||
          !CAS(&p->active_, 0, 1))
        continue;
      // Got one!
      return p;
    }
    // Increment the list length
    int oldLen;
    do {
      oldLen = listLen_;
    } while (!CAS(&listLen_, oldLen,
        oldLen + 1));
    // Allocate a new one
    HPRecType * p = new HPRecType;
    p->active_ = 1;
    p->pHazard_ = 0;
    // Push it to the front
    do {
      old = pHead_;
      p->pNext_ = old;
    } while (!CAS(&pHead_, old, p));
    return p;
  }
  // Releases a hazard pointer
  static void Release(HPRecType* p) {
    p->pHazard_ = 0;
    p->active_ = 0;
  }
};
// Per-thread private variable
__per_thread__ vector<Map<K, V> *> rlist;
```

Each thread holds a *retired list* (actually a vector< Map<K,V>*> in our implementation)—a container keeping track of the pointers that this thread finds are not needed anymore and could be **delete**d as soon as no other threads use them. This vector need not be synchronized because it's in per-thread storage; only one thread will ever access it. We gloss over the tedium of allocating thread-local storage by using the magic qualifier __per_thread__.

Given this setup, all a thread needs to do whenever wanting to dispose of pMap_ is call the Retire function below. (Note that, as in the previous Generic⟨Programming⟩ instance, we do not insert memory barriers for the sake of clarity.)

```
template <class K, class V>
class WRRMMap {
  Map<K, V> * pMap_;
  ...
private:
  static void Retire(Map<K, V> * pOld) {
    // put it in the retired list
    rlist.push_back(pOld);
    if (rlist.size() >= R) {
      Scan(HPRecType::Head());
    }
  }
};
```

Nothing up our sleeves! Now, the Scan function will perform a *set difference* between the pointers in

the current thread's retired list, and all hazard pointers for all threads. What does that set difference mean? Let's think of it for a second: it's the set of all old `pMap_` pointers that this thread considers useless, except those that are found among the hazard pointers of all threads. But, hey, these are exactly the goners! By definition of the retired list and that of the hazard pointers, if a pointer is retired and not marked as "hazardous" (i.e. "in use") by any thread, the set intersection of the two sets yields precisely the pointers that can be **delete**d.

2 The Main Algorithm

Ok, now let's see how to implement the `Scan` algorithm, and what guarantees it can provide. We need to perform a set difference between `rlist` and `pHead_` whenever performing a scan. That operation tantamounts to "for each pointer in the retired list, find it in the hazard set. If it's not, it belongs to the difference, so it can be **delete**d." To optimize that, we can sort the hazard pointers before lookup, and then perform one binary search in it for each retired pointer. Let's take a look at such an implementation of `Scan`:

```
void Scan(HPRecType * head) {
  // Stage 1: Scan hazard pointers list
  // collecting all non-null ptrs
  vector<void*> hp;
  while (head) {
    void * p = head->pHazard_;
    if (p) hp.push_back(p);
    head = head->pNext_;
  }
  // Stage 2: sort the hazard pointers
  sort(hp.begin(), hp.end(),
    less<void*>());
  // Stage 3: Search for'em!
  vector<Map<K, V>*>::iterator i =
    rlist.begin();
  while (i != rlist.end()) {
    if (!binary_search(hp.begin(),
        hp.end(),
        *i) {
      // Aha!
      delete *i;
      if (&*i != &rlist.back()) {
        *i = rlist.back();
      }
      rlist.pop_back();
    } else {
      ++i;
    }
  }
}
```

The last loop does the actual work. It uses a little trick to optimize away shuffling the `rlist` vector: after **delete**ing a removable pointer in `rlist`, it overwrites that pointer with the last element in `rlist`, after which it eliminates that last element. This is allowed because elements in `rlist` needn'd be sorted in any particular order.

We availed ourselves of the C++ standard functions `sort` and `binary_search`. But you can replace the vector with your favorite easily-searchable structure, such as a hashtable. A well-balanced hashtable has constant expected lookup time, and is very easy to construct because it is totally private and you know all the values in it before organizing it.

What performance guarantees can we make about `Scan`? First off, notice that the entire algorithm is wait-free (as advertised in the introduction): there are no loops in which a thread's execution time depends on other threads' behavior.

Second, the average size of `rlist` is the arbitrary value `R` that we chose as our threshold for firing `Scan` (see the `WRRMMap<K, V>::Replace` function above). If we organize the hazard pointers in a hash table (instead of the sorted vector `hp` used above), the expected complexity of the `Scan` algorithm is $O(R)$. Finally, the maximum number of removed maps that are not yet **delete**d is $N \cdot R$, where N is the number of writer threads. A good choice for R is $(1 + k)H$, where H is the number of hazard pointers (`listLen` in our code, and equal to the number of readers in our example) and k is some small positive constant, say $^1/_4$. So R is a number greater than H and proportional to it. In that case, each scan is guaranteed to **delete** $R - H$ nodes (that is $O(R)$ nodes) and—if we use a hash table—takes expected time $O(R)$. So,

the expected amortized time for determining that a node is safe to **delete** is constant.

3 Tying WRRMMap's loose ends

Now, let's stitch the hazard pointers into WRRMap's primitives, namely Lookup and Update. For writers (threads executing Update), all they need to do is call WRRMap<K, V>::Retire in the place where they would normally **delete** pMap_.

```
void Update(const K&k, const V&v){
  Map<K, V> * pNew = 0;
  do {
    Map<K, V> * pOld = pMap_;
    delete pNew;
    pNew = new Map<K, V>(*pOld);
    (*pNew)[k] = v;
  } while (!CAS(&pMap_, pOld, pNew));
  Retire(pOld);
}
```

The readers need first to get a hazard pointer by calling HPRecType::Acquire, then set it to the pMap_ used for searching through. When it is done with a pointer, the thread releases the hazard pointer by calling HPRecType::Release.

```
V Lookup(const K&k){
  HPRecType * pRec = HPRecType::Acquire();
  Map<K, V> * ptr;
  do {
    ptr = pMap_;
    pRec->pHazard_ = ptr;
  } while (pMap_ != ptr);
  // Save Willy
  V result = (*ptr)[k];
  // pRec can be released now
  // because it's not used anymore
  HPRecType::Release(pRec);
  return result;
}
```

But, why does the reader need to recheck pMap_? Consider the following scenario. pMap_ points to the map m. The reader reads &m from pMap_ into ptr, then goes to sleep before it can set its hazard pointer to &m. In the meantime, a writer sneaks from behind, updates pMap_, and retires the map at m. It then checks the hazard pointer of the reader and finds not equal to &m. The writer concludes that it is safe to deallocate m and does so. Now the reader wakes up and sets its hazard pointer to &m. If that was all and the reader goes to dereference ptr, it will read corrupt data or access unmapped memory.

This is why the reader needs to check pMap_ again. If pMap_ is not equal to &m, then the reader is not really sure that the writer who removed m has seen its hazard pointer set to &m. So it is not safe to go ahead and read from m, and the reader should start over.

If the reader finds pMap_ pointing to m, then it is safe to read from m. Does that mean that pMap_ hasn't changed in the time between the two reads? Not necessarily. m could have been removed from and installed in pMap_ one or more times during that interval, and that doesn't matter. What matters is that at the time of the second read, m is certainly not removed (because pMap_ points to it) and at that point the reader's hazard pointer already points to it. So from that point forward (until the hazard pointer is changed) it is safe for the reader to access m.

Now, both lookup and update are lock-free (admittedly not wait-free though): readers don't block writers, or get in each other's way (unlike reference counting). It's the perfect Write-Rarely-Read-Many map: reads are very fast and don't interfere with one another, and updates are still fast and guaranteed to make global progress.

If we want lookups to be wait-free, we can use the trap technique [4] which is built on top of hazard pointers. In the code above, when a reader sets its hazard pointer to ptr it is basically *trying* to capture a *specific* map *ptr. Using the trap technique, the reader can set a trap that will *definitely* capture *some* valid map, and so the lookup becomes wait-free—as it is the case if we have automatic garbage collection (remember last article). For the gory details of the trap technique, see [4].

4 Generalization

We're pretty much done with a solid map design. There are a few more things worth addressing, that we'd like to mention here to give you a comprehensive overview of the technique. We point you to the paper [3] for full reference.

We figured out what to do to share a map, but what if we need to share many other objects? That's not a problem; first, the algorithm extends naturally to accomodate multiple hazard pointers per thread. But oftentimes, there are very few pointers that a thread needs to protect simultaneously at any given time. Besides, hazard pointers can be "overloaded"—reason for which, by the way, they are untyped (`void*`) in the `HPRecType` structure: a thread can use the same actual hazard pointer on any number of data structures as long as it is operating on them one at a time. Most of the time one or two hazard pointers per thread are enough for the whole program.

Finally, notice that Lookup compulsively calls `HPRecType::Release` as soon as it is done doing *one* lookup. In a performance-minded application, the hazard pointer could be acquired once, used for multiple lookups, and released only later.

5 Conclusion

People have for so long tried to solve the memory deallocation problem with lock-free algorithms, at a point it looked like there is no satisfactory solution. However, with a minimum scaffolding and by maneuvering carefully between thread-private and thread-shared data, it is possible to devise an algorithm that gives strong and satisfactory speed and memory consumption guarantees. Besides, although we used `WRRMMap` as an example throughout, the hazard pointers technique is of course applicable to much more complex data structures. The memory reclamation problem is more important in dynamic structures that can grow and shrink arbitrarily, for example, a program that has thousands of linked lists that may grow to have millions of dynamic nodes and then shrink. That would be where hazard pointers would show their full power.

The worst that a reader thread could ever do is die and leave all of its hazard pointers set, thus forever keeping allocated at most one node for each of its hazard pointers.

Detective Bullet rushed into his debtor's office and figured out at once he won't get his money today. Without missing a beat, he said: "You're on my list, pal. I'll come after you again. Only death can absolve you, and even in that case, you'll never be able to be in debt more than $100. Cheers."

6 Acknowledgments

Will go here.

References

[1] Andrei Alexandrescu. Generic⟨Programming⟩: Lock-Free Data Structures. *C++ Users Journal*, October 2004.

[2] Simon Doherty, David L. Detlefs, Lindsay Grove, Christine H. Flood, Victor Luchangco, Paul A. Martin, Mark Moir, Nir Shavit, and Jr. Guy L. Steele. DCAS is not a silver bullet for non-blocking algorithm design. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 216–224. ACM Press, 2004. ISBN 1-58113-840-7.

[3] Maged M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. In *IEEE Transactions on Parallel and Distributed Systems*, pages 491–504. IEEE, 2004.

[4] Maged M. Michael. Practical Lock-Free and Wait-Free LL/SC/VL Implementations Using 64-Bit CAS. In *Proceedings of the 18th International Conference on Distributed Computing, LNCS volume 3274*, pages 144–158, October 2004.

[5] Hong Tang, Kai Shen, and Tao Yang. Program transformation and runtime support for threaded MPI execution on shared-memory machines. *ACM Transactions on Programming Lan-*

guages and Systems, 22(4):673–700, 2000. URL citeseer.ist.psu.edu/tang99program.html.