RC23089 (W0401-136) January 29, 2004
Computer Science

IBM Research Report

ABA Prevention Using Single-Word Instructions

Maged M. Michael
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

ABA Prevention Using Single-Word Instructions

Maged M. Michael
IBM Thomas J. Watson Research Center
P. O. Box 218, Yorktown Heights, NY 10598, USA
magedm@us.ibm.com

Abstract

The ABA problem is a fundamental problem that affects almost all lock-free algorithms. The atomic primitives LL/SC/VL (Load-Linked, Store-Conditional, Validate) offer a convenient way for algorithm designers to reason about lock-free algorithms, without concern for the ABA problem. However, for practical architectural reasons, no processor architecture supports the ideal semantics of LL/VL/SC.

It is relatively easy to implement LL/SC/VL—and prevent the ABA problem—using double-word atomic instructions. However, most current mainstream 64-bit architectures support only single-word atomic instructions. The best known constructions of LL/SC/VL using single-word instructions entail substantial space overhead when applied to a large number of memory locations, and require knowing the maximum number of threads in advance.

This report presents simple lock-free constructions using only practical single-word instructions for implementing ideal LL/SC/VL, and hence preventing the ABA problem, with reasonable space overhead. These constructions can also be used to implement arbitrary width atomic operations.

1 Introduction

A shared object is *lock-free* (also called nonblocking) if it guarantees that whenever a thread executes some finite number of steps towards an operation on the object, *some* thread (possibly a different one) must have made progress towards completing an operation on the object, during the execution of these steps. Thus, unlike conventional lock-based objects, lock-free objects are immune to deadlock when faced with thread failures, and offer robust performance even when faced with arbitrary thread delays.

A subtle problem associated with most lock-free algorithms is the ABA problem. It was first reported in association with the introduction of the CAS (Compare-and-Swap) instruction on the IBM System 370 [3]. CAS takes three arguments: the address of a memory location, an expected value, and a new value. If and only if the memory location holds the expected value, the new value is written to it, atomically. A Boolean return value indicates whether the write occurred.

```
// Shared variables
Top:*NodeType;  // Initially null

Push(node:*NodeType) {
    do {
1:      t ← Top;
2:      node^.Next ← t;
3:  } until CAS(&Top,t,node);
}

Pop() : *NodeType {
    do {
4:      t ← Top;
5:      if t = null return null;
6:      next ← t^.Next;
7:  } until CAS(&Top,t,next); }
8:  return t;
}
```

Figure 1: ABA-prone lock-free LIFO list algorithm.

The ABA problem occurs when a thread reads a value A from a shared location, and then other threads change the location to a different value, say B, and then back to A again. Later, when the original thread checks the location, e.g., using read or CAS, the comparison succeeds, and the thread erroneously proceeds under the assumption that the location has not changed since the thread read it earlier. As a result, the thread may corrupt the object or return a wrong result.

Figure 1 shows a variant of the IBM lock-free LIFO free list algorithm [3] stripped of ABA prevention code. Consider a list that contains three nodes A, B, and C. Thread X reads the value A from the shared variable *Top* in line 4 and then proceeds to line 6 and reads the value B from A^.Next, and then gets delayed. Then, a thread Y, pops the node A from the list, then pops the node B, and finally pushes A again. Thus, leaving the variable *Top* with the value A and the list containing two nodes A and

```
Top:⟨*NodeType,integer⟩;  // Initially ⟨null,0⟩
Pop() : *NodeType {
      do {
 4:     ⟨t,tag⟩ ← Top;
 5:     if t = null return null;
 6:     next ← t^.Next;
 7:   } until CASdbl(&Top,⟨t,tag⟩,⟨next,tag+1⟩);
 8:   return t;
}
```

Figure 2: Lock-free Pop using an ABA-prevention tag.

```
Pop() : *NodeType {
     do {
4:     t ← LL(&Top);
5:     if t = null return null;
6:     next ← t^.Next;
7:   } until SC(&Top,next);
8:   return t;
}
```

Figure 3: Lock-free Pop using LL/SC.

C. When X resumes execution, it proceeds to line 7 and the CAS instruction succeeds, thus setting *Top* to B and consequently corrupting the list since B is no longer part of the list and possibly corrupting other structures that may contain B. The intention of the algorithm designer is for X's CAS in line 7 to fail in such a case.

If it can be guaranteed that the CAS in line 7 cannot succeed if the value of *Top* has changed since the current thread's execution of line 4, then the ABA problem becomes impossible. The earliest and simplest solution to this problem is the IBM tag methodology [3]. A tag (update counter) is associated with the pointer *Top* (i.e., the target of the ABA-prone CAS in line 7), such that when the pointer *Top* is changed the tag is also incremented atomically. By using double-width CAS, both the pointer and the tag can be checked and updated atomically. Treiber [6] pointed out that this need only be applied to the Pop routine. Figure 2 shows the Pop routine augmented with the IBM ABA-prevention tag mechanism. The tag is assumed to have enough bits to make full wraparound practically impossible between a thread's execution of lines 4 and 7.

One problem with the IBM tag methodology is that—unless the data occupies only a small part of a word—it requires double-width versions of atomic instructions. Most current mainstream 64-bit processor architectures do not support double-width instructions. Our goal in this report is to present ABA-safe constructions that use only single-word (i.e., pointer-sized) instructions.

Tightly related to the ABA problem is the set of instructions LL/SC/VL (Load-Linked, Store-Conditional, Validate). 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. 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.

If ABA-prone validation conditions and CAS instructions are replaced by VL and SC, and if the original read of the target variable is replaced by LL, then the ABA problem becomes impossible. Figure 3 shows an ABA-safe version of the Pop routine using LL and SC. By definition, the SC in line 7 must fail if *Top* has been written since the current thread executed line 4.

However, 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 allows nesting or interleaving of LL/SC pairs, and most prohibit any memory access between LL and SC. Accordingly, the restricted semantics of LL/SC supported in practice offer no or little help with preventing the ABA problem.

The ideal semantics of LL/SC/VL offer a convenient and simple framework for designing lock-free algorithms. This report presents constructions using practical pointer-sized instructions for implementing ideal LL/SC/VL.

Incidentally, the constructions presented in this report can also implement atomic instructions of arbitrary width using only single-word instructions.

In Section 2 we discuss ABA prevention and LL/SC/VL constructions under automatic garbage collection. In Section 3, we present the constructions for LL/SC/VL in the absence of support for automatic garbage collection.

2 Under Garbage Collection

A common misconception is that automatic garbage collection (GC) inherently makes the ABA problem impossible. While this is true for many lock-free algorithms, it is not always the case. GC guarantees that as long as a thread holds a reference to a dynamic node that node will not be reclaimed.

Figure 4 shows a lock-free stack algorithm, where GC indeed prevents the ABA problem. It is impossible for a scenario similar to that mentioned in Section 1 to happen. GC makes it impossible for thread Y to allocate node A after it has been popped, since at that time thread X is still holding a reference to A. GC guarantees that as long as some thread (in this case X) holds a reference to a

```
Push(data:DataType) {
    node ← NewNode();
    node^.Data ← data;
    do {
1:      t ← Top;
2:      node^.Next ← t;
3:  } until CAS(&Top,t,node);
}

Pop() : DataType {
    do {
4:      t ← Top;
5:      if t = null return EMPTY;
6:      next ← t^.Next;
7:  } until CAS(&Top,t,next);
8:  return t^.Data;
}
```

Figure 4: ABA-safe lock-free LIFO stack under GC.

node (in this case A), that node cannot be collected as garbage and made available for new allocations..

On the other hand, consider a program that moves dynamic nodes back and forth between two lists using the algorithms in Figurereffig-stack. The ABA problem is possible in such a case even with perfect GC. Thus, contrary to common belief, the ABA problem *can happen* under GC. However, we note that indeed GC can always be used to prevent the ABA absolutely. That is, any lock-free algorithm can be made ABA-safe under GC.

Figure 5 shows implementations of arbitrary-sized variables that support LL/SC/VL as well as read and write using only single-word read, write and CAS. The main idea is using a level of indirection similar in form to that used in Herlihy's universal methodology [1]. Implemented shared variables are represented by a pointer to a block of the implemented variable's size. Whenever the implemented variable is written, the address of a new block with the new value replaces the prior pointer. GC guarantees that the old block will not be recycled while any thread holds a reference to it, hence preventing the ABA problem.

When a thread performs LL, it first reads the pointer *oldptr* to the current data block and then—while guaranteed by GC that the data block will not be reclaimed prematurely—it proceeds to read the data. The threads keeps *oldptr* for future SC and VL operations.

In the SC operation, the thread allocates a new data block and sets it to the new value. Then it performs a CAS on the target variable. The CAS succeeds if and only if the variable has not been written since it was last read using LL by the current thread.

If the CAS fails then it must have found the variable to hold a different pointer value from *oldptr*, then the variable must have been written by one or more Write

```
LL(addr) : [DataType,*DataType] {
  oldptr ← *addr;
  data ← *ptr;
  return [data,oldptr];
}

SC(addr,newval,oldptr) : Boolean {
  newptr ← NewBlock();
  *newptr ← newval;
  return CAS(addr,oldptr,newptr);
}

VL(addr,oldptr) : Boolean {
  return *addr = oldptr;
}

Read(addr) : DataType {
  oldptr ← *addr;
  data ← *oldptr;
  return data;
}

Write(addr,data) {
  newptr ← NewBlock();
  *newptr ← data;
  *addr ← newptr;
}
```

Figure 5: Wait-free constructions of arbitrary sized variables supporting LL/SC/VL using single-word CAS under GC.

and/or SC operations after the current thread read the value *oldptr* from the variable using LL, and thus the SC fails as it should.

If the CAS succeeds then it must have found the variable to hold the value *oldptr*. GC guarantees that no thread could have allocated the data block **oldptr* after the current thread last read the variable using LL, since the current thread has been holding the value *oldptr* during that period and hence preventing GC from reclaiming **oldptr*. Then it must be the case that the variable has not been written since the current thread last read it using LL, and thus the SC succeeds as it should.

The constructions are wait-free. The time overhead is constant, and the space overhead is constant.

3 Without Garbage Collection

This section presents lock-free constructions of arbitrary sized variables that support LL, SC, VL, read, and write using single-word read, write, and CAS for programming environments without support for GC.

The implementations rely on the hazard pointer methodology [5] for the safe reclamation of dynamic

blocks. Each thread owns a number of single-writer multiple-reader pointers called hazard pointers. Whenever a thread needs to announce to other threads that it intends to use a reference to a dynamic block in a hazardous manner without further validation, it sets a hazard pointer to the address of that block. When a thread removes a dynamic block from a lock-free object and before it reclaims its memory (e.g., using `free`), it scans the hazard pointers of the other threads. If it finds no match, then it is safe to free the block. Otherwise, the block is kept until a subsequent scan. For more details see [5]

Figure 6 shows the constructions. As in the constructions under GC, implemented shared variables are represented by a pointer to a block of the implemented variable's size. Whenever the implemented variable is written, a pointer to a new block with the new value replaces the prior pointer. The hazard pointer methodology guarantees that the old block will not be recycled while any thread holds a hazardous reference to it. In this case *holding a hazardous reference* to a block means holding a reference to a block with the intention of using it as a target location or an expected value of an ABA-prone operation.

When a thread performs LL, it first reads the pointer *oldptr* to the current data block and then assigns a hazard pointer to *oldptr*. However, the setting of the hazard pointer may have been too late. By then the block **oldptr* might have been already removed and reclaimed by another thread. Therefore, the current thread must validate that this is not the case and proceeds only if it finds that the reference *oldptr* was valid after setting the hazard pointer. If the hazard pointer was set too late, the thread starts over again. This can happen only if the variable has been written in the meantime. Otherwise if the hazard pointer was confirmed to be indeed protecting **oldptr*, the thread proceeds to read the contents of **oldptr*. The thread keeps the value *oldptr* for use by subsequent SC and VL operations.

As in the SC operation in Figure 5, the thread allocates a new data block and initializes it with the new value. Then it performs a CAS on the target variable. The CAS succeeds if and only if the variable has not been written since it was last read using LL by the current thread. Without support for GC, the thread also needs to explicitly free the new block if the CAS failed, or otherwise pass *oldptr* to the hazard pointer algorithm for matching against the hazard pointers of other threads before being determined to be safe to free.

When a thread no longer needs a reference, it can simply reuse the corresponding hazard pointer. If an algorithm involves up to K concurrent LL/SC/VL periods per thread, then only K hazard pointers are needed per thread.

If N is the maximum number of threads, M is the number of implemented LL/SC/VL variables, and K is the maximum number of LL/SC/VL period. The static space overhead of the constructions is $O(NK + M)$. The dynamic space overhead is the subject of a tradeoff with the time overhead. The hazard pointer methodology can of-

```
// hp is a pointer to one of the thread's hazard pointers.
LL(addr,hp) : [DataType,*DataType] {
retry:
   oldptr ← *addr;
   *hp ← oldptr;
   if (*addr ≠ oldptr) goto retry;
   data ← *oldptr;
   return [data,oldptr];
}

SC(addr,newval,oldptr) : Boolean {
   newptr ← NewBlock();
   *newptr ← newval;
   if CAS(addr,oldptr,newptr) {
      RetireBlock(oldptr); // defined in [5]
      return true;
   } else {
      FreeBlock(newptr);
      return false;
   }
}

VL(addr,oldptr) : Boolean {
      return *addr = oldptr;
}

Read(addr,hp) : DataType {
retry:
   oldptr ← *addr;
   *hp ← oldptr;
   if (*addr ≠ oldptr) goto retry;
   data ← *oldptr;
   return data;
}

Write(addr,data,hp) {
   newptr ← NewBlock();
   *newptr ← data;
retry:
   oldptr ← *addr;
   *hp ← oldptr;
   if (*addr ≠ oldptr) goto retry;
   if !CAS(addr,oldptr,newptr) goto retry;
   RetireBlock(oldptr);
}
```

Figure 6: Lock-free constructions of arbitrary-sized variables supporting LL/SC/VL using single-word CAS without relying on GC.

fer either constant expected amortized time per block and $O(N^2K)$ worst case dynamic space overhead, or $O(NK)$ time overhead and $O(NK)$ space overhead. Using the former algorithm choice, the construction has a constant expected amortized time overhead and $O(N^2K + M)$ space overhead. The hazard pointer algorithm does not require knowing the maximum number of threads in advance.

The best known single-word LL/SC/VL constructions using single-word CAS by Jayanti and Petrovic [4] is wait-free, has a constant time overhead and $O(NM)$ space overhead. It also requires knowing the value of N in advance. In many cases of dynamic lock-free objects, M is substantially larger than NK, and N may not be known in advance. In such cases, the new constructions present a practical option.

Note that the Pass-the-Buck algorithm [2] for memory reclamation cannot be used for these constructions, as the algorithm itself uses double-width CAS to avoid the ABA problem in its own operations.

4 Discussion

This report presents efficient and practical lock-free of arbitrary-sized LL/SC/VL variables using only practical single-word instructions, without reliance on support for automatic garbage collection.

The main purpose of this report is demonstrate that GC and hazard pointers can *always* be used to prevent the ABA problem completely. While practical and efficient, the constructions in this report are only meant as worst case backup mechanisms. It is common for algorithm designer—including this author—to *think* about algorithms in terms of LL/SC/VL during algorithm development as a matter of convenience. However, it is preferred that lock-free algorithms be *written* in terms of read, ABA-safe CAS, and ABA-safe validation conditions. Writing algorithms in terms of LL/SC/VL in order to prevent the ABA problem is almost always an overkill, as the semantics of LL/SC/VL are often much more restrictive than needed for preventing the ABA problem.

References

[1] M. P. Herlihy. A methodology for implementing highly concurrent objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, Nov. 1993.

[2] M. P. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting dynamic-sized lock-free data structures. In *Proceedings of the Sixteenth International Symposium on Distributed Computing, LNCS volume 2508*, pages 339–353, Oct. 2002.

[3] IBM. *IBM System/370 Extended Architecture, Principles of Operation*, 1983. Publication No. SA22-7085.

[4] P. Jayanti and S. Petrovic. Efficient and practical constructions of LL/SC variables. In *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Distributed Computing*, pages 285–294, July 2003.

[5] M. M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing*, pages 21–30, July 2002. Also IBM T. J. Watson Research Report RC 22317, Jan. 2002.

[6] R. K. Treiber. Systems programing: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, Apr. 1986.