

IBM Research Report

Practical Lock-Free and Wait-Free LL/SC/VL Implementations Using 64-Bit CAS

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

Practical Lock-Free and Wait-Free LL/SC/VL Implementations Using 64-Bit CAS

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

May 2004

Abstract

The ABA problem is a fundamental problem for lock-free and wait-free algorithms. The ideal semantics of the atomic primitives LL/SC/VL (Load-Linked, Store-Conditional, Validate) are inherently immune to that problem. However, for practical architectural reasons, no processor architecture supports the ideal semantics of LL/VL/SC. Current mainstream architectures support either CAS (Compare-and-Swap) or LL/SC with restricted semantics, which are susceptible to the ABA problem. Furthermore, most current mainstream 64-bit architectures do not support atomic instructions on more than 64-bit memory blocks, thus making LL/SC/VL implementations that require support for the atomic manipulation of wider memory blocks impractical. The recent 64-bit wait-free implementations of 64-bit LL/SC/VL variables by Jayanti and Petrovic entail space overhead per LL/SC/VL variable that is proportional to the number of threads that may operate on it. In programs with a large number of LL/SC/VL variables, the space overhead can be unacceptable in practice. The implementations also require advance knowledge of the maximum number of threads that may operate on the LL/SC/VL variables, which is not always possible to predict without making conservative estimates that further increase the space overhead beyond practical limits. In this paper, we present practical lock-free and wait-free implementations of arbitrary-sized LL/SC/VL that require only 64-bit CAS in 64-bit programs. The implementations require only a constant space overhead per LL/SC/VL variable (one word for the lock-free implementation and four words for the wait-free implementation), and linear space per participating thread that is amortized over all the LL/SC/VL variables in the program, and may be subsumed by the independent memory reclamation needs of the program. The implementations do not require advance knowledge of the maximum number of participating threads. In the wait-free implementation, LL and VL take constant time, and SC takes constant amortized expected time. In the lock-free implementation, VL takes constant time and SC takes constant amortized expected time, and in the absence of interfering successful SC operations, LL takes constant time. In both implementations—regardless of contention—concurrent LL, VL, and unsuccessful SC operations do not interfere with each other, and SC and VL are wait-free. Using the work performance measure, the amortized expected complexity of any set of LL/SC/VL operations using either of our implementations is the same as that assuming hypothetical hardware support for ideal LL/SC/VL.

A preliminary version of this work appears in [9].

1 Introduction

A shared object is *lock-free* [1, 3] if whenever a thread executes some finite number of steps toward an operation on the object, some thread must have made progress towards completing an operation on the object, during the execution of these steps, regardless of thread speeds, scheduling or arbitrary termination. A lock-free shared object is also *wait-free* [2, 6] if progress is also guaranteed per operation. Thus, unlike conventional lock-based objects, lock-free objects are immune to deadlock, regardless of thread scheduling policies and arbitrary thread termination, in addition to performance advantages such as tolerance to preemption.

A subtle problem that affects the design of almost every lock-free algorithm is the ABA problem. If not prevented, it can cause the corruption of lock-free objects as well as unrelated objects that happen to reuse dynamic memory removed from affected lock-free objects, and it can cause the program to crash or return incorrect results. The ABA problem was first reported in the documentation of the Compare-and-Swap (CAS) instruction and its use for implementing lock-free freelists on the IBM System 370 [5]. CAS takes three arguments: the address of a memory location, an expected value, and a new value. If the memory location is found to hold the expected value, the new value is written to it, atomically. A Boolean return value indicates whether the write occurred. If it returns true, it is said to succeed, otherwise, it is said to fail.

The ABA problem occurs when a thread reads some value A from a shared variable, and then other threads write to the variable some value B , and then A again. Later, when the original thread checks if the variable holds the value A , using read or CAS, the comparison succeeds, while the intention of the algorithm designer is for such a comparison to fail in this case, and to succeed only if the variable has not been written after the initial read. However, the semantics of read and CAS prevent them from distinguishing the two cases.

The semantics of the theoretical instructions LL/SC/VL (Load-Linked, Store-Conditional, Validate) make them inherently immune to the ABA problem. 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. If the location was not written 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 the memory location was not written since the current thread last read it using LL. If SC or VL returns true, it is said to succeed, otherwise, it is said to fail.

For practical architectural reasons, none of the architectures that support LL/SC (PowerPC, MIPS, Alpha) support VL or the ideal semantics of LL/SC as defined above. The nesting or interleaving of LL/SC pairs is not allowed, and the time between LL and SC must be short. On MIPS and Alpha, memory accesses are not allowed between LL and SC. These architectures also infrequently allow SC to fail spuriously (i.e., even when the target location was not written since the last LL). The restricted semantics of LL/SC supported in practice offer little or no help with preventing the ABA problem, and in most lock-free algorithms restricted LL/SC are used just to simulate CAS.

Until recently, implementations of LL/SC/VL variables required atomic operations on both the implemented variable and an additional tag field. As most 32-bit architectures support 64-bit—as well as 32-bit—atomic instructions, these mechanisms are feasible to varying degrees in 32-bit applications running on 32-bit as well as 64-bit architectures. However, most current mainstream 64-bit architectures do not support atomic instructions on more than 64-bit blocks, thus it is no longer possible to pack a large tag with pointer-size values in 64-bit applications.

Jayanti and Petrovic [8] present wait-free implementations of 64-bit LL/SC/VL using 64-bit CAS. However, these implementations require space overhead per LL/SC/VL variable that is proportional to N , where N is the maximum number of threads that may operate on the LL/SC/VL variable. The implementations also require the use of N -sized arrays, which are problematic to implement without advance knowledge of the value of N or a conservative estimate of it. These requirements limit the practicality of these implementations to special cases where the maximum number of LL/SC/VL variables and the maximum number of threads in the program that may operate on these variables are known in advance to be small, and the number of threads can be bounded in advance to a small number.

To be generally practical, a LL/SC/VL implementation must (1) use only 64-bit CAS, (2) perform well under various levels of contention, (3) be capable of adapting dynamically to the number of participating threads, and (4) have practically acceptable space overhead, in particular it must not require more than a small constant overhead per LL/SC/VL variable.

In this paper, we present lock-free and wait-free implementations of arbitrary-sized LL/SC/VL variables using 64-bit CAS. The implementations require only constant space overhead per LL/SC/VL variable (one word for the lock-free implementation and four words for the wait-free implementation), and linear space per participating thread that is amortized over all the LL/SC/VL variables in the program, and may be subsumed by the independent memory

reclamation needs of the program. The implementations do not require advance knowledge of the maximum number of participating threads.

In the wait-free implementation, LL and VL take constant time, and SC takes constant amortized expected time. In the lock-free implementation, VL takes constant time and SC takes constant amortized expected time, and in the absence of interfering successful SC operations, LL takes constant time. In both implementations—regardless of contention—concurrent LL, VL, and unsuccessful SC operations do not interfere with each other, and SC and VL are wait-free. Using the work performance measure, the amortized expected complexity of any set of LL/SC/VL operations using either of our implementations is the same as that assuming hypothetical hardware support for ideal LL/SC/VL.

The rest of this paper is organized as follows. In Section 2, we discuss memory reclamation and other related issues. In Section 3, we present the lock-free implementation, and in Section 4, we present the wait-free implementation. We discuss the complexity of the implementations in Section 5, and conclude with Section 6.

2 Preliminaries

Memory Reclamation

The memory reclamation problem is the problem of allowing dynamic blocks removed from lock-free objects to be freed, while guaranteeing that threads operating on these objects do not access the memory of blocks freed by concurrently running threads. The term *free* here is used in a broad sense, including reusing the block, dividing or coalescing its memory, or unmapping its memory.

In this paper, we make use of the hazard pointer memory reclamation method [10], as it is portable across operating systems and architectures and uses only 64-bit instructions in 64-bit programs. Briefly, the method uses single-writer multireader pointers called hazard pointers. When a thread sets one of its hazard pointer to the address of a block, it in effect announces to other threads that if they happen to remove that block after the setting of the hazard pointer, then they must not free it as long as the hazard pointer continues to point to it. So, when a thread removes a block from a participating lock-free object—and before it can free the block—it has to scan the list of hazard pointers and check if any of them points to the block. Only if no match is found then the block is determined to be safe to free.

There are many possible ways for scanning the list of hazard pointers. In a preferred implementation [10] using amortization, only constant amortized expected time is needed for processing each removed block until it is determined to be safe to free. A thread scans the hazard pointers after accumulating $H + \Theta(H)$ removed blocks, where H is the number of hazard pointers in the program. Then, the thread reads the H hazard pointers (nonatomically) and organizes the non-NULL values read from them in an efficient private search structure such as a hash table (with constant expected lookup time). Then, for each of the blocks that it has accumulated it searches the hash table for matching values. As described in [10], the procedure is wait-free, it takes $O(H)$ expected time, and it is guaranteed to identify $\Theta(H)$ blocks as safe to free. Most importantly, the hazard pointer method uses only pointer-size instructions and therefore it is practical for 64-bit applications.

Threads can join the hazard pointer method dynamically, retire dynamically, and also acquire and release hazard pointers dynamically. Neither the number of participating threads N nor the number of hazard pointers H needs to be known in advance. We discuss the tradeoffs and possibilities regarding the space complexity of the hazard pointer method in the course of discussing the space complexity of our LL/SC/VL implementations in Section 5.

Definitions

A thread p is said to hold an *active reservation* for LL/SC/VL variable \mathcal{O} at time t , if p has performed LL(\mathcal{O}) at time $t_0 < t$ and it is possible for p to perform SC(\mathcal{O}, v) or VL(\mathcal{O}) at some time $t_1 \geq t$ without performing another LL(\mathcal{O}) during the interval $[t, t_1]$. Accordingly, a thread can hold at most one active reservation for the same LL/SC/VL variable at the same time.

In this paper, we assume that programs that use LL/SC/VL are well-formed. That is, a thread p performs SC(\mathcal{O}) or VL(\mathcal{O}) only when it is holding an active reservation for \mathcal{O} .

We define K as the highest number of active reservations that p needs to hold concurrently. Typically, K is a small constant. For example, in optimized LL/SC/VL-based versions of known lock-free algorithms, $K=1$ for FIFO queues [12], LIFO lists [5], Jayanti’s f -arrays [7], and Herlihy’s universal methodologies [3], and $K=2$ for linked lists and hash tables [10, 13].

3 Lock-Free LL/SC/VL Implementation

In the lock-free implementation (shown in Figure 1) the LL/SC/VL variable \mathcal{O} is represented by a pointer X . The current value of \mathcal{O} is always held in the dynamic block currently pointed to by X . Whenever \mathcal{O} is written, a new block holding the new value replaces the old block pointed to by X .

The subscript i in the names of some functions and variables is used to distinguish among the reservations that may be held concurrently by the same thread.

The basic idea of the implementation is for $\text{LL}_{p,i}(\mathcal{O})$ to read the value in the current block pointed to by X and use a hazard pointer to protect the block from being reused prematurely, i.e., as long as p holds the reservation for \mathcal{O} . Subsequently, if $\text{SC}_{p,i}(\mathcal{O},v)$ or $\text{VL}_{p,i}(\mathcal{O})$ find X pointing to the same block, then it must be the case that \mathcal{O} was not written since $\text{LL}_{p,i}(\mathcal{O})$ was performed.

$\text{LL}_{p,i}(\mathcal{O})$: The implementation of $\text{LL}_{p,i}(\mathcal{O})$ proceeds as follows. In line 2, thread p reads a pointer value from X into a persistent private variable $\text{exp}_{p,i}$, with the intention of reading \mathcal{O} 's value from $*\text{exp}_{p,i}$ (as in line 5). However, p cannot just proceed to read $*\text{exp}_{p,i}$, as it is possible that after line 2, another thread has removed $\text{exp}_{p,i}$ (by performing a successful SC on \mathcal{O}) and then freed it before p manages to read it.

So, in line 3, p sets the hazard pointer $\text{hp}_{p,i}$ to $\text{exp}_{p,i}$ in order to prevent $\text{exp}_{p,i}$ from being freed before line 5 if it happens to be removed by another thread's SC before then. However, it is possible that $\text{exp}_{p,i}$ has already been removed before p set the hazard pointer in line 3. Therefore, p needs to check X again (in line 4). If $X \neq \text{exp}_{p,i}$, then it is possible that $\text{exp}_{p,i}$ was removed before p set $\text{hp}_{p,i}$ in line 3, and hence $\text{exp}_{p,i}$ might have been—or will be—freed before line 5. As it is not safe to proceed to line 5 in this case, p starts over again from line 2.

If in line 4, p finds $X = \text{exp}_{p,i}$, then it is safe for p to read $*\text{exp}_{p,i}$ in line 5. This is true whether or not X has changed between line 2 and line 4. What matters is that at line 4, $X = \text{exp}_{p,i}$ (and hence $\text{exp}_{p,i}$ is not removed or free at that point) and $\text{hp}_{p,i} = \text{exp}_{p,i}$ already from line 3. Therefore, according to the hazard pointer method, from that point (line 4), if $\text{exp}_{p,i}$ is removed by another thread, it will not be freed as long as $\text{hp}_{p,i}$ continues to point to it, which is true as long as the current reservation is alive (i.e., at least until after line 5).

$\text{LL}_{p,i}(\mathcal{O})$ is linearized [4] at line 4. At that point $X = \text{exp}_{p,i}$ and accordingly $\mathcal{O} = *\text{exp}_{p,i}$. By the hazard pointer method, the value of $*\text{exp}_{p,i}$ remains unchanged between lines 4 and 5. Therefore, $\text{LL}_{p,i}(\mathcal{O})$ returns the value of \mathcal{O} at the time of execution of line 4. Also, as described later, subsequent SC and VL operations—for the same reservation initiated by $\text{LL}_{p,i}(\mathcal{O})$ —will succeed if and only if \mathcal{O} has not been written since line 4 of $\text{LL}_{p,i}(\mathcal{O})$.

As long as the reservation initiated by $\text{LL}_{p,i}(\mathcal{O})$ is alive, p must keep $\text{hp}_{p,i}$ and $\text{exp}_{p,i}$ unchanged. Once p reaches a point where it will not be possible to issue $\text{SC}_{p,i}(\mathcal{O})$ or $\text{VL}_{p,i}(\mathcal{O})$ corresponding to the current reservation, then p can simply reuse $\text{hp}_{p,i}$ and $\text{exp}_{p,i}$ for a new reservation, or even p can use $\text{hp}_{p,i}$ for non-LL/SC/VL-related memory reclamation purposes.

$\text{SC}_{p,i}(\mathcal{O},v)$: The implementation of $\text{SC}_{p,i}(\mathcal{O},v)$ proceeds as follows. In lines 1 and 2, p allocates a safe block and sets it to the new value v . According to the hazard pointer method, a block b is safe if there is no thread q with any reservation j —for any LL/SC/VL variable not just \mathcal{O} —with $\text{hp}_{q,j} = b$. That is, if this SC succeeds, then the new pointer value of X —i.e., b —will be guaranteed to be different from all the $\text{exp}_{q,j}$ pointer values associated with all live reservations at the time.

In line 3, the CAS succeeds if and only if $X = \text{exp}_{p,i}$. When SC is issued it is guaranteed that $\text{hp}_{p,i} = \text{exp}_{p,i}$ and that they were unchanged since the linearization of $\text{LL}_{p,i}(\mathcal{O})$ that initiated the reservation. By the hazard pointer method, if $\text{exp}_{p,i}$ was removed after $\text{LL}_{p,i}(\mathcal{O})$, then no other thread could have subsequently allocated $\text{exp}_{p,i}$ as a safe block during the lifetime of the reservation. Therefore, the CAS succeeds if and only if, \mathcal{O} was not written since $\text{LL}_{p,i}(\mathcal{O})$. So, $\text{SC}_{p,i}(\mathcal{O},v)$ is linearized at line 3.

If $\text{SC}_{p,i}(\mathcal{O},v)$ succeeds (i.e., the CAS in line 3 succeeds), the old block removed from X (i.e., $\text{exp}_{p,i}$) cannot be reused immediately and its value should not be changed until it is determined to be safe to free by going through the hazard pointer method by calling `RetireNode` (defined in [10]). If $\text{SC}_{p,i}(\mathcal{O},v)$ fails, then the block b can be reused immediately safely, as it was already safe in line 1 and it has not been observed by other threads since then.

The functions `GetSafeBlock` and `KeepSafeBlock` can be replaced with `malloc` and `free`, respectively, which can be implemented completely in user-level in an efficient lock-free manner [11]. However, our LL/SC/VL implementations have the feature that as long as N (the number of participating threads) is stable (which is an implicit requirement in Jayanti and Petrovic's implementations [8]), the number of blocks needed per thread remains stable, and so these functions can be completely private. Each participating thread maintains two private lists of safe and not-safe-yet blocks with combined maximum size equal to the batch size needed for the memory reclamation method. `GetSafeBlock` pops a block from the (private) safe list. `KeepSafeBlock` pushes a block into that list. When the thread

Types

blocktype = valuetype = arbitrary-sized value

Shared variables representing each LL/SC/VL variable \mathcal{O}

X: pointer to blocktype

initially $(X = b \neq \text{NULL}) \wedge (*b = \mathcal{O}'\text{s initial value})$

Per-thread shared variables (for thread p)

for $i \in \{0, \dots, K-1\}$ $\text{hp}_{p,i}$: hazard pointer

Private persistent variables (for thread p)

for $i \in \{0, \dots, K-1\}$ $\text{exp}_{p,i}$: pointer to blocktype

$\text{LL}_{p,i}(\mathcal{O})$: valuetype

```

1: repeat
2:    $\text{exp}_{p,i} := X$ 
3:    $\text{hp}_{p,i} := \text{exp}_{p,i}$ 
4: until  $(X = \text{exp}_{p,i})$ 
5: return  $*\text{exp}_{p,i}$ 

```

$\text{SC}_{p,i}(\mathcal{O}, v)$: boolean

```

1:  $\bar{b} := \text{GetSafeBlock}()$ 
2:  $*b := v$ 
3:  $\text{ret} := \text{CAS}(X, \text{exp}_{p,i}, b)$ 
4: if ( $\text{ret}$ )
5:    $\text{RetireNode}(\text{exp}_{p,i})$ 
6: else
7:    $\text{KeepSafeBlock}(b)$ 
8: return  $\text{ret}$ 

```

$\text{VL}_{p,i}(\mathcal{O})$: boolean

```

1: return  $(X = \text{exp}_{p,i})$ 

```

Figure 1: Lock-free implementation of LL/SC/VL using pointer-size CAS.

calls `RetireNode` and processes the blocks in the (also private) not-safe-yet list, it pushes the blocks identified to be safe into the safe list.

$\text{VL}_{p,i}(\mathcal{O})$: The implementation of $\text{VL}_{p,i}(\mathcal{O}, v)$ simply checks if X is equal to $\text{exp}_{p,i}$. As argued regarding $\text{SC}_{p,i}(\mathcal{O}, v)$, this is true if and only if \mathcal{O} has not been written since the linearization of $\text{LL}_{p,i}(\mathcal{O})$. $\text{VL}_{p,i}(\mathcal{O})$ is linearized at line 1.

4 Wait-Free LL/SC/VL Implementation

In the LL/SC/VL implementation in Section 3, $\text{LL}_{p,i}(\mathcal{O})$ is not wait-free because thread p is always trying to capture the value of \mathcal{O} from a specific block $\text{exp}_{p,i}$. However, by the time p reads the pointer value to $\text{exp}_{p,i}$ from X and is about to protect it using the hazard pointer $\text{hp}_{p,i}$, the block $\text{exp}_{p,i}$ might have already been removed and possibly freed, and p is forced to start over.

Unlike hazard pointers which prevent specific blocks from being freed, we introduce the notion of a *trap* which can capture *some block* (not a specific one) that satisfies certain criteria. So, in $\text{LL}_{p,i}(\mathcal{O})$, we use a trap to guarantee that in a constant number of steps, *some block* holding some value of \mathcal{O} will be guaranteed not to be freed until p reads a value of \mathcal{O} from it and supply the return value of LL.

To be linearizable, LL needs to return a value that was held by \mathcal{O} between LL's invocation and its response [4]. Therefore, a trap must avoid capturing a block that holds an old value of \mathcal{O} that was overwritten before LL's invocation. For that purpose, we maintain a sequence number for each LL/SC/VL variable. The sequence number is incremented after every successful SC. When a thread sets a trap for a variable, it also specifies the minimum acceptable sequence number.

4.1 Trap Functionality

Our wait-free LL/SC/VL implementation uses the following interface with the trap mechanism: $\text{SetTrap}_p(\mathcal{O}, seq)$, $\text{GetCapturedBlock}_p()$, $\text{ReleaseTrap}_p()$, and $\text{ScanTraps}_p(b)$. Between every two calls to SetTrap_p there must be a call to ReleaseTrap_p . Thread p 's trap is said to be active after a call to SetTrap_p and before the corresponding call to ReleaseTrap_p . $\text{GetCapturedBlock}_p$ is called only when p 's trap is active. A block b is passed as argument of ScanTraps only after b has been removed from a LL/SC/VL variable. Blocks passed to ScanTraps are only freed (i.e., determined to be safe) by ScanTraps . The trap mechanism offers the following guarantees:

- After thread p calls $\text{SetTrap}_p(\mathcal{O}, seq)$ and before the following call to $\text{ReleaseTrap}_p()$, if a call to $\text{GetCapturedBlock}_p()$ returns b then either $b = \text{NULL}$ or b hold the n^{th} value of \mathcal{O} where n is some number greater than or equal to seq .

- If thread p calls $\text{SetTrap}_p(\mathcal{O}, seq)$ at time t , and if block b is removed from \mathcal{O} (before or after t) $\wedge b$ holds the n^{th} value of $\mathcal{O} \wedge n \geq seq \wedge b$ is passed to ScanTraps_q by any thread q after t , then b will not be freed before one or both of the following takes place:
 - p calls $\text{ReleaseTrap}_p()$.
 - The trap captures a block $b' \neq b$. That is, a call to $\text{GetCapturedBlock}_p()$ before the next call to $\text{ReleaseTrap}_p()$ will return $b' \neq \text{NULL}$.

First we present the wait-free LL/SC/VL functions assuming this trap functionality, then we describe the trap implementation in detail.

4.2 LL/SC/VL Functions

Figure 2 shows the structures and functions of the wait-free LL/SC/VL implementation. We start with the sequence number. An additional variable Seq is used per LL/SC/VL variable \mathcal{O} . Between every two consecutive SC operations on \mathcal{O} , Seq must be incremented exactly once. Therefore, if $\text{Seq} = n$, then there must have been either n or $n+1$ successful SC operations performed on \mathcal{O} so far.

Two additional fields are added to the block structure. The field Var points to the LL/SC/VL variable, and the field Seq holds a sequence number. If the n^{th} successful SC on \mathcal{O} sets X to b , then it must be the case that at that point $b \rightarrow \text{Seq} = n$ and $b \rightarrow \text{Var} = \mathcal{O}$. The purpose of these two fields is to enable a trap to capture only a block that holds a value for a specific LL/SC/VL variable with a sequence number higher than or equal to some sequence number.

$\text{LL}_{p,i}(\mathcal{O})$: Lines 1–4 of $\text{LL}_{p,i}(\mathcal{O})$ are similar to the lock-free implementation in Figure 1. In the absence of intervening successful SC operations by other threads between p 's execution of line 1 and line 3, $\text{LL}_{p,i}(\mathcal{O})$ returns at line 4. In such a case $\text{LL}_{p,i}(\mathcal{O})$ is linearized at line 3. If intervening successful SC operations are detected in line 3, p sets a trap before it tries to read X again, to ensure completion in constant time even if more intervening successful SC operations occur.

In line 5, p reads Seq into a local variable seq . At that point, there must have been so far either seq or $seq+1$ successful SC operations performed on \mathcal{O} . In line 6, p sets a trap for \mathcal{O} with a sequence number seq . The trap guarantees that from that point until the release of the trap, either the trap has captured a block that contains a value of \mathcal{O} with a sequence number greater than or equal to seq , or no block that contains a value of \mathcal{O} with a sequence number greater than or equal to seq has been freed.

Now p proceeds to read X again into $exp_{p,i}$ in line 7, and in line 8 it sets $hp_{p,i}$ to $exp_{p,i}$. At this point, it must be the case that $exp_{p,i} \rightarrow \text{Seq} \geq seq$.

If the trap set in line 6 does not capture a block before the setting of the hazard pointer in line 8, then $exp_{p,i}$ could not have been freed after line 7 and will not be freed as long as $hp_{p,i}$ continues to point to it. Therefore, if p finds in line 9 that no block has been captured yet by the trap, then it must be safe to read $exp_{p,i} \rightarrow \text{Value}$ in line 11. In such a case, $\text{LL}_{p,i}(\mathcal{O})$ is linearized at line 7. This is correct for the following reasons:

- At line 7, $X = exp_{p,i}$ and $exp_{p,i} \rightarrow \text{Value}$ remains unchanged after line 7 (throughout the lifetime of the reservation). Therefore, $\text{LL}_{p,i}(\mathcal{O})$ returns the value of \mathcal{O} at the time of p 's execution of line 7.
- By the guarantee of the memory reclamation method, $exp_{p,i}$ will not be freed after line 7 until the end of the reservation. Therefore, subsequent calls to SC and VL for the same reservation are guaranteed to succeed—by finding $X = exp_{p,i}$ —if and only if \mathcal{O} has not been written since p 's execution of line 7.

If p finds in line 9 that the trap set in line 6 has indeed captured some block b (may or may not be the same as $exp_{p,i}$), then it might be possible that $exp_{p,i}$ has already been removed before setting the hazard pointer in line 8. Therefore, it may not be safe to access $exp_{p,i}$. However, as long as the trap is not released, it is safe to access b . So, p proceeds to read $b \rightarrow \text{Value}$ in line 13.

In this case (i.e., a block b was captured before line 9), $\text{LL}_{p,i}(\mathcal{O})$ is linearized just before the SC that removed b from X , which is guaranteed to have occurred between p 's execution of line 1 and line 9. If $b \rightarrow \text{Seq} \geq seq+1$ then b must have been removed from X after p 's execution of line 5. If $b \rightarrow \text{Seq} = seq$ then b must have been removed from X after p 's execution of line 1.

In line 14 $exp_{p,i}$ is set to NULL in order to guarantee the failure of subsequent $\text{SC}_{p,i}(\mathcal{O}, v)$ and $\text{VL}_{p,i}(\mathcal{O})$ operations for the same reservation, as they should. At this point we are already certain that \mathcal{O} has been written after the linearization point of $\text{LL}_{p,i}(\mathcal{O})$, as only removed blocks get trapped.

Types

valuetype = arbitrary-sized value

seqnumtype = 64-bit unsigned integer

blocktype = record Value: valuetype; Seq: seqnumtype; Var: pointer to LL/SC/VL variable end

Shared variables representing each LL/SC/VL variable \mathcal{O}

X: pointer to blocktype

Seq: seqnumtype

initially $(X = b \neq \text{NULL}) \wedge (b \rightarrow \text{Value} = \mathcal{O}'\text{s initial value}) \wedge (b \rightarrow \text{Seq} = \text{Seq} = 0) \wedge (b \rightarrow \text{Var} = \mathcal{O})$

Per-thread shared variables (for thread p)

for $i \in \{0, \dots, K-1\}$ $hp_{p,i}$: hazard pointer

Private persistent variables (for thread p)

for $i \in \{0, \dots, K-1\}$ $exp_{p,i}$: pointer to blocktype

<pre> <u>LL_{p,i}(\mathcal{O}) : valuetype</u> 1: $exp_{p,i} := X$ 2: $hp_{p,i} := exp_{p,i}$ 3: if $(X = exp_{p,i})$ 4: return $exp_{p,i} \rightarrow \text{Value}$ 5: seq := Seq 6: SetTrap_p(\mathcal{O}, seq) 7: $exp_{p,i} := X$ 8: $hp_{p,i} := exp_{p,i}$ 9: $b := \text{GetCapturedBlock}_p()$ 10: if $(b = \text{NULL})$ 11: $v := exp_{p,i} \rightarrow \text{Value}$ 12: else 13: $v := b \rightarrow \text{Value}$ 14: $exp_{p,i} := \text{NULL}$ 15: ReleaseTrap_p() 16: return v </pre>	<pre> <u>SC_{p,i}(\mathcal{O}, v) : boolean</u> 1: if $(exp_{p,i} = \text{NULL})$ return FALSE 2: $b := \text{GetSafeBlock}()$ 3: $b \rightarrow \text{Value} := v$ 4: $b \rightarrow \text{Var} := \mathcal{O}$ 5: $seq := exp_{p,i} \rightarrow \text{Seq}$ 6: $b \rightarrow \text{Seq} := seq + 1$ 7: if $(\text{Seq} < seq)$ 8: CAS($\text{Seq}, seq - 1, seq$) 9: $ret := \text{CAS}(X, exp_{p,i}, b)$ 10: if $(ret = \text{TRUE})$ 11: CAS($\text{Seq}, seq, seq + 1$) 12: ScanTraps_p($exp_{p,i}$) 13: else 14: KeepSafeBlock(b) 15: return ret </pre>
<pre> <u>VL_{p,i}(\mathcal{O}) : boolean</u> 1: return $(X = exp_{p,i})$ </pre>	<pre> 1: return $(X = exp_{p,i})$ </pre>

Figure 2: Wait-free implementation of LL/SC/VL using 64-bit CAS.

After reading a valid value either in line 11 or line 13, it is safe to release the trap in line 15. Therefore, each participating thread needs only one trap, even if it may hold multiple reservations concurrently.

$SC_{p,i}(\mathcal{O}, v)$: In line 1, p checks if $LL_{p,i}(\mathcal{O})$ has already determined that this SC will certainly fail (by setting $exp_{p,i}$ to NULL). If so, $SC_{p,i}(\mathcal{O}, v)$ returns FALSE and is linearized immediately after its invocation.

If $exp_{p,i} \neq \text{NULL}$, then $LL_{p,i}(\mathcal{O})$ must have read its return value from $exp_{p,i} \rightarrow \text{Value}$ and $exp_{p,i}$ has been continuously protected by $hp_{p,i}$. Therefore, the same arguments for the lock-free implementation apply here too, and $SC_{p,i}(\mathcal{O}, v)$ is linearized at the time of applying CAS to X (line 9). However, care must be taken to maintain the invariant that when the $n + 1^{\text{th}}$ successful SC on \mathcal{O} is linearized, $\text{Seq} = n$.

In lines 2 and 3, as in the SC implementation in Section 3, p allocates a safe block b and sets its Value field to the new value v . Also, p sets $b \rightarrow \text{Var}$ to \mathcal{O} in line 4 to allow the trap implementation to associate b with \mathcal{O} .

In lines 5 and 6, p reads the value seq from $exp_{p,i} \rightarrow \text{Seq}$ and then sets $b \rightarrow \text{Seq}$ to $seq + 1$ for the following reason. The current SC can succeed only if it replaces $exp_{p,i}$ in X with b in line 9. Since $exp_{p,i}$ holds the seq^{th} value of \mathcal{O} , then if the current SC succeeds, b will be holding the $(seq + 1)^{\text{th}}$ value of \mathcal{O} . So, p sets $b \rightarrow \text{Seq}$ to $seq + 1$ to allow the trap implementation to associate b with the $(seq + 1)^{\text{th}}$ value of \mathcal{O} .

We now discuss the update of Seq. Lines 7 and 11 are optional optimizations, so we ignore them and focus only on line 8. Note that, Seq is monotonically increasing and always by increments of 1.

Correctness requires that between every two successful SC operations, Seq must be incremented exactly once. That is, if this SC succeeds in line 9, then Seq must be equal to seq at that point. By an induction argument, if this

SC is to succeed then at line 8 either $\text{Seq} = \text{seq}-1$ or $\text{Seq} = \text{seq}$. So, whether the CAS in line 8 succeeds or fails, immediately after line 8 $\text{Seq} = \text{seq}$, if this SC is to succeed.

We then argue that if Seq is incremented by another thread between lines 8 and 9, then this SC must fail. If $\text{Seq} = n > \text{seq}$ at line 9, then some other thread q must have performed CAS on Seq with a new value n , then q must have observed $X = \text{exp}_{q,j}$ with $\text{exp}_{q,j} \rightarrow \text{Seq} = n$, which implies that at least n SC operations on \mathcal{O} have already succeeded before line 9, then this SC cannot be the $(\text{seq}+1)^{\text{th}}$ successful SC on \mathcal{O} , and so this SC must fail.

We now move to line 9. As in the lock-free LL/SC/VL implementation, SC succeeds if and only if the CAS on X succeeds. Line 9 is the linearization point of SC if $\text{exp}_{p,i} \neq \text{NULL}$. If SC succeeds, p —optionally—tries to increment Seq in line 11. It also passes the removed block $\text{exp}_{p,i}$ to ScanTraps so that it will not be freed prematurely. If SC fails, then block b remains safe and can be freed or kept for future use without going through traps or hazard pointers.

$\text{VL}_{p,i}(\mathcal{O})$: As in the lock-free implementation in Section 3, $X = \text{exp}_{p,i}$ if and only if \mathcal{O} has not been written since the linearization of $\text{LL}_{p,i}(\mathcal{O})$. $\text{VL}_{p,i}(\mathcal{O})$ is linearized at line 1.

Note that implementing sequence numbers as 64-bit variables makes wrap-around impossible for all practical purposes. Even if 1,000,000 successful SC operations are performed on \mathcal{O} every second, the 64-bit sequence number would still not wrap around after 584,000 years!

4.3 Trap Implementation

The trap implementation is shown in Figure 3. Each participating thread p owns a trap record trap_p , an additional hazard pointer traphp_p , and a list list_p of removed blocks.

$\text{SetTrap}_p(\mathcal{O}, \text{seq})$: This function starts by setting $\text{trap}_p \rightarrow \text{Var}$ to \mathcal{O} and $\text{trap}_p \rightarrow \text{Seq}$ to seq in lines 1 and 2, in order to indicate that the trap is set for blocks that contain a value of \mathcal{O} with a sequence number not less than seq .

The purpose of $\text{trap}_p \rightarrow \text{Captured}$ is to provide other threads with a location to offer a block that matches the criteria of this trap. When p sets a trap, it cannot just set $\text{trap}_p \rightarrow \text{Captured}$ to NULL , otherwise a slow thread trying to set $\text{trap}_p \rightarrow \text{Captured}$ in relation to an earlier trap might offer the wrong block for the current trap. We need to guarantee that only a thread that intends to offer a block that satisfies the criteria of the current trap succeeds in setting $\text{trap}_p \rightarrow \text{Captured}$.

For that purpose, p uses a unique tag every time it sets a trap. Since arbitrary 64-bit numbers might match pointer values captured by the trap, we use only odd numbers for the tag. By convention and as required by most current processor architectures, the addresses of dynamic blocks must be 8-byte aligned. So, odd tag numbers are guaranteed not to match any block addresses. (NULL must be even.) Thus, each trap can be set 2^{63} times, each with a different tag value. The variable tag_p keeps track of the tag values already used by trap_p . Similar to the sequence number, even if trap_p is used 1,000,000 times per second to perform LL operations where each LL is interfered with by a successful SC, and this continues for 292,000 years, tag_p would still not wrap around. Therefore, wrap-around is not a practical concern in this case either.

In line 3, p sets $\text{trap}_p \rightarrow \text{Captured}$ to tag_p , and in line 4 it also sets traphp_p to the same tag value. We discuss traphp_p in more detail when discussing ScanTraps below.

Setting the trap takes effect only when p sets $\text{trap}_p \rightarrow \text{Active}$ to TRUE in line 5. Finally, in line 6, p prepares a new tag value in tag_p for the next trap.

$\text{GetCapturedBlock}_p()$: In this function, p simply checks if any other thread has replaced the (odd) tag value it has put in $\text{trap}_p \rightarrow \text{Captured}$ in SetTrap_p with a block's address (even). If so, GetCapturedBlock returns the address of the captured block, otherwise it returns NULL to indicate that no block has been captured yet by this trap.

$\text{ReleaseTrap}_p()$: In this function, p simply sets $\text{trap}_p \rightarrow \text{Active}$ to FALSE and is ready to reuse the trap if needed. It is essential that $\text{trap}_p \rightarrow \text{Active} = \text{FALSE}$ whenever p writes the other fields of trap_p or traphp_p . Lines 2 and 3 are optional optimizations.

$\text{ScanTraps}_p(b')$: Every successful $\text{SC}_{p,i}$ calls ScanTraps_p in order to start the process of freeing the block $b' = \text{exp}_{p,i}$ that held the previous value before the success of SC. The function can be implemented in many ways. A simple implementation would be to scan the traps upon every call to ScanTraps , but this would take linear time per successful SC. We show an amortized implementation that takes constant amortized expected time per successful SC. In this implementation, p accumulates removed blocks in list_p (line 1). When list_p contains enough removed blocks for amortization ($\Theta(N)$ blocks), p proceeds to scan the trap structures of others participating threads. First, p organizes the addresses of the blocks in list_p in an efficient (private) search structure (hash table) that allows constant expected

Types

tagtype = 64-bit unsigned integer
 traptype = record Active: boolean; Var: pointer to LL/SC/VL variable; Seq: seqnumtype;
 Captured: tagtype or pointer to blocktype end

Per-thread shared variables (for thread p)

trap $_p$: traptype initially trap $_p$ →Active = FALSE
 traphp $_p$: hazard pointer

Private persistent variables (for thread p)

tag $_p$: tagtype initially tag $_p$ = 1

<p><u>SetTrap$_p$(\mathcal{O}, seq)</u></p> <ol style="list-style-type: none"> 1: trap$_p$→Var := \mathcal{O} 2: trap$_p$→Seq := seq 3: trap$_p$→Captured := tag$_p$ 4: traphp$_p$:= tag$_p$ 5: trap$_p$→Active := TRUE 6: tag$_p$:= tag$_p$+2 <p><u>GetCapturedBlock$_p$() : pointer to blocktype</u></p> <ol style="list-style-type: none"> 1: b := trap$_p$→Captured 2: if (b is even) return b else return NULL <p><u>ReleaseTrap$_p$()</u></p> <ol style="list-style-type: none"> 1: trap$_p$→Active := FALSE 2: trap$_p$→Captured := NULL 3: traphp$_p$:= NULL 	<p><u>ScanTraps$_p$(b')</u></p> <ol style="list-style-type: none"> 1: $list_p$.insert(b') 2: if ($list_p$.size() < Threshold) return 3: $list_p$.organize() 4: for all q 5: if ¬trap$_q$→Active skip 6: tag := trap$_q$→Captured 7: if (tag is even) skip 8: var := trap$_q$→Var 9: seq := trap$_q$→Seq 10: b := $list_p$.lookup(var, seq) 11: if (b = NULL) skip 12: if CAS(trap$_q$→Captured, tag, b) 13: CAS(traphp$_q$, tag, b) 14: for all b in $list_p$ 15: RetireNode(b)
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 3: Trap structures and functions.

lookup time. If $list_p$ contains multiple blocks that hold values for the same LL/SC/VL variable, the lookup can just return the block with the highest sequence number.

Now we describe the scanning process. For each participating thread q , p starts by checking trap $_q$ →Active (line 5). If it is FALSE, then it is certain that q does not have an active trap that was set before any of the blocks in $list_p$ were removed. Therefore, p can skip this trap and move on to the next one if any.

If trap $_q$ →Active is TRUE, p proceeds to line 6 and reads trap $_q$ →Captured into the local variable tag . If tag is even, then it is either NULL or actually a block's address and not a tag. If it is NULL, then the trap must have already been released. If it is an actual block, then it must be the case that trap $_q$ has already captured a block. Therefore p need not be concerned about providing a block to trap $_q$ whether or not $list_p$ contains blocks that match the criteria of trap $_q$. So, whether trap $_q$ has been released or has captured a block, p can skip this trap and move on to the next one if any.

If tag is indeed a tag (i.e., odd value), then trap $_q$ has not yet captured a block and so p needs to proceed to lines 8 and 9, to read trap $_q$ →Var and trap $_q$ →Seq. In line 10, p performs a lookup in $list_p$ (in constant expected time) for a block that matches the criteria of trap $_q$. If none are found, then p moves on to the next trap if any.

If p finds a block b in $list_p$ that matches the criteria of trap $_q$, then p tries to install b in trap $_q$ →Captured using CAS in line 12. If the CAS fails then it must be the case that either the trap has been released or some other thread has installed a block in trap $_q$ →Captured, and so p can move on to the next trap if any. If the CAS (in line 12) succeeds, then trap $_q$ has captured b . In this case p needs to ensure that b is not freed before the release of trap $_q$. Therefore it tries to set traphp $_q$ to b in line 13 using CAS.

If the CAS in line 13 fails, then q must have already released the trap. Therefore, neither b nor possibly the other blocks in $list_p$ that matched the criteria in trap $_q$ need to be prevented from being free (at least as far as trap $_q$ is concerned). So, p moves on to the next trap if any.

If the CAS in line 13 succeeds, then the hazard pointer method guarantees that b will not be freed as long as traphp $_q$ remains unchanged, i.e., not before trap $_q$ is released.

After scanning the trap records of all participating threads, p is guaranteed that all the blocks in $list_p$ can be passed on safely to the hazard pointer method (through `RetireNode`) for ultimate determination of when they are safe to be freed. If a block b has been captured by one or more traps then it will not be freed as long as any of these traps has not been released. If a block b has not been captured by any traps, then either it did not match the criteria of any of the traps set before its removal, or it did match the criteria of one or more traps but in each case either the trap has been released or some other block has been captured by the trap. In any case, it is safe to pass b to `RetireNode`.

The trap structures and the operations (particularly `ScanTraps`) are similar in many respects to those of the hazard pointer method [10]. Therefore, as it is the case for hazard pointers, threads can participate (i.e., acquire and release trap structures) dynamically, and do not require advance knowledge of the maximum value of N . Also, all of the trap structures can be integrated in the hazard pointer structures. Also, the $O(N)$ blocks per thread that may be accumulated in $list_p$ can be combined with the blocks accumulated by the thread for the sake of amortization in the hazard pointer method.¹

5 Complexity and Practical Implementation Issues

Time and Work Complexity

In the wait-free LL/SC/VL implementation LL, VL, and unsuccessful SC operations take constant time, and successful SC operations take constant amortized expected time. In the lock-free LL/SC/VL implementation, SC and VL are wait-free. Successful SC operations take constant amortized expected time, and VL and unsuccessful SC operations take constant time. LL takes constant time in the absence of intervening successful SC operations.

The conventional performance measure for lock-free algorithms is work. Work is the total number of steps executed by N threads to perform some number r of operations. In our implementations LL, VL, and unsuccessful SC operations do not interfere with each other and all operations take at most constant amortized expected time in the absence of contention. So, the amortized expected work complexity of our implementations is the same as that of a hypothetical hardware implementation of ideal LL/SC/VL. For example, consider r LL operations, r LL/SC pairs, and as many LL/SC operations as needed to have r successful SC operations. Assuming hardware support for ideal LL/SC/VL, the work is $O(r)$, $O(r)$, and $O(r.N)$, respectively. Using either of our LL/SC/VL implementations, the amortized expected work is $O(r)$, $O(r)$, and $O(r.N)$, respectively.

Space Complexity

The space complexity of our LL/SC/VL implementations consists of two components: (1) space overhead per LL/SC/VL variable, and (2) space per participating thread.

Component (1): For both implementations the space overhead per variable is a small constant, one word (variable X) for the lock-free implementation and four words (variables X and Seq, and the Var and Seq fields of the current block) for the wait-free implementation. Component (2): For both implementations, the space per participating thread is $O(N.K)$.

Component (1) is obviously reasonable and constitutes a clear and significant improvement over the space complexity of Jayanti and Petrovic’s implementations [8]. Whether the number of LL/SC/VL variables in a program is large or not is no longer a concern. We now argue that for all practical purposes, component (2) is—or can be made to be—within acceptable practical limits.

If the maximum number of threads (active at the same time) in the program is reasonable (tens or hundreds), then the space is within reasonable limits for 64-bit applications.

If a program may have a large number of threads active concurrently and all may operate on LL/SC/VL-based lock-free objects, it is very unlikely that all these threads are “important” (e.g., high priority threads that operate frequently on lock-free objects). In order to keep the number of participating threads N reasonable, we can use a multi-track organization of hazard pointer structures (including trap structures). Important threads release their hazard pointer structures only before exiting (or when they are no longer important). The hazard pointer structures of important threads are kept in the first track. The other (less important) threads participate in the LL/SC/VL mechanism and hazard pointer method dynamically (still in a wait-free manner [10]) and use the other track(s).

¹The blocks in $list_p$ can be counted with other removed blocks awaiting processing by the hazard pointer method (those in the not-safe-yet list). Blocks in $list_p$ scan traps as well as hazard pointers, while other blocks scan hazard pointers only. The amortized expected time per block identified to be safe is still constant.

These threads acquire hazard pointer structures dynamically before operating on lock-free objects and then release them after they are done with the operation.

Let N_1 be the maximum number of important threads in a program running concurrently. Let n_2 be the number of less important threads that are participating concurrently in the LL/SC/VL mechanism at some time t . Let N_2 be the maximum n_2 . Then N is at most $N_1 + N_2$, which is expected to be much less than the total number of threads if the total is large.

The amortized expected time for the operations of the important threads remains constant, while for the less important threads it is $O(n_2)$ (to acquire hazard pointer structures dynamically). As less important threads are not expected to operate frequently on lock-free objects, then n_2 is expected to be small.

Furthermore, most programs that employ lock-free dynamic objects need to use a memory reclamation method (regardless of employing LL/SC/VL), otherwise the memory of contended blocks removed from lock-free objects cannot be divided, coalesced, unmapped, or reused arbitrarily. Therefore, by intermingling the hazard pointers and the count of accumulated removed blocks of our LL/SC/VL implementations with those independently needed by the program, it is possible that component (2) of the space complexity is already subsumed in part or in full (except for constant space for the trap structure) by the memory reclamation needs of the program. Note that the blocks used by our LL/SC/VL implementations and in general contended blocks that are susceptible to the memory reclamation problem are typically small. It is good programming practice to keep these blocks small and replace large data fields—if any—with pointers to ordinary blocks (to hold these fields) that can be freed without concern for the memory reclamation problem.

6 Conclusion

In this paper, we presented practical lock-free and wait-free implementations of LL/SC/VL that improve on Jayanti and Petrovic’s implementations [8] by limiting the space requirements to practically acceptable levels, and eliminating the need for advance knowledge of the number of threads that may operate on these variables, while still using only 64-bit CAS, and offering low time and work complexity and low latency.

References

- [1] M. B. Greenwald. *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University, Aug. 1999.
- [2] M. P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, Jan. 1991.
- [3] M. P. Herlihy. A methodology for implementing highly concurrent objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, Nov. 1993.
- [4] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.
- [5] *IBM System/370 Extended Architecture, Principles of Operation*, 1983. Publication No. SA22-7085.
- [6] P. Jayanti. Wait-free computing. In *Proceedings of the 9th International Workshop on Distributed Algorithms*, pages 19–50, Sept. 1995.
- [7] P. Jayanti. f -arrays: Implementation and applications. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing*, pages 270–279, July 2002.
- [8] 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.
- [9] M. M. Michael. ABA prevention using single-word instructions. Technical Report RC 23089, IBM T. J. Watson Research Center, Jan. 2004.
- [10] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004. Earlier version in *21st PODC*, pages 21–30, July 2002.
- [11] M. M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, June 2004.
- [12] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275, May 1996.
- [13] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. In *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Distributed Computing*, pages 102–111, July 2003.