Reuse, don't Recycle: Transforming Lock-free Algorithms that Throw Away Descriptors

Maya Arbel-Raviv and Trevor Brown

Technion, Computer Science Department, Haifa, Israel
mayaarl@cs.technion.ac.il, me@tbrown.pro

─── **Abstract** ───

In many lock-free algorithms, threads help one another, and each operation creates a *descriptor* that describes how other threads should help it. Allocating and reclaiming descriptors introduces significant space and time overhead. We introduce the first *descriptor* abstract data type (ADT), which captures the usage of descriptors by lock-free algorithms. We then develop a *weak descriptor* ADT which has weaker semantics, but can be implemented significantly more efficiently. We show how a large class of lock-free algorithms can be transformed to use weak descriptors, and demonstrate our technique by transforming several algorithms, including the leading k-compare-and-swap (k-CAS) algorithm. The original k-CAS algorithm allocates at least $k+1$ new descriptors *per k-CAS*. In contrast, our implementation allocates two descriptors *per process*, and each process simply reuses its two descriptors. Experiments on a variety of workloads show significant performance improvements over implementations that reclaim descriptors, and reductions of up to three orders of magnitude in peak memory usage.

1 Introduction

Many concurrent data structures use locks, but locks have downsides, such as susceptibility to convoying, deadlock and priority inversion. Lock-free data structures avoid these downsides, and can be quite efficient. They guarantee that some process will always makes progress, even if some processes halt unexpectedly. This guarantee is typically achieved with *helping*, which allows a process to harness any time that it would otherwise spend waiting for another operation to complete. Specifically, whenever a process p is prevented from making progress by another operation, it attempts to perform some (or all) of the work of the other operation, on behalf of the process that started it. This way, even if the other process has crashed, its operation can be completed, so that it no longer blocks p.

In simple lock-free data structures (e.g., [30, 16, 25, 28]), a process can determine how to help an operation that blocks it by inspecting a small part of the data structure. In more complex lock-free data structures [14, 19, 29, 10], processes publish *descriptors* for their operations, and helpers look at these descriptors to determine how to help. A descriptor typically encodes a sequence of steps that a process should follow in order to complete the operation that created it.

Since lock-free algorithms cannot use mutual exclusion, many helpers can simultaneously help an operation, potentially long after the operation has terminated. Thus, to avoid situations where helpers read inconsistent data in a descriptor and corrupt the data structure, each descriptor must remain consistent and accessible until no helper will ever access it again. This leads to *wasteful algorithms* which allocate a new descriptor for each operation.

In this work, we introduce two simple abstract data types (ADTs) that capture the way descriptors are used by wasteful algorithms (in Section 2). The *immutable descriptor* ADT provides two operations, *CreateNew* and *ReadField*, which respectively create and initialize a new descriptor, and read one of its fields. The *mutable descriptor* ADT extends the immutable descriptor ADT by adding two operations: *WriteField* and *CASField*. These allow a helper to modify fields of the descriptor (e.g., to indicate that the operation has been partially or fully completed). We also give examples of wasteful algorithms whose usage of descriptors is captured by these ADTs.

The natural way to implement the immutable and mutable descriptor ADTs is to have *CreateNew* allocate memory and initialize it, and to have *ReadField*, *WriteField* and *CASField* perform a read, write and CAS, respectively. Every implementation of one of these ADTs must eventually reclaim the descriptors it allocates. Otherwise, the algorithm would eventually exhaust memory. We briefly explain why reclaiming descriptors is expensive.

In order to safely free a descriptor, a process must know that the descriptor is no longer *reachable*. This means no other process can reach the descriptor by following pointers in shared memory *or* in its private memory. State of the art lock-free memory reclamation algorithms such as hazard pointers [26] and DEBRA+ [7] can determine when no process has a pointer in its *private* memory to a given object, but they typically require the underlying algorithm to identify a time t after which the object is no longer reachable from *shared* memory. In an algorithm where each operation removes all pointers to its descriptor from shared memory, t is when O completes. However, in some algorithms (e.g., [11]), pointers to descriptors are "lazily" cleaned up by subsequent operations, so t may be difficult to identify. The overhead of reclaiming descriptors comes both from identifying t, and from actually running a lock-free memory reclamation algorithm.

Additionally, in some applications, such as embedded systems, it is important to have a small, predictable number of descriptors in the system. In such cases, one must use memory reclamation algorithms that aggressively reclaim memory to minimize the number of objects that are waiting to be reclaimed at any point in time. Such algorithms incur high overhead. For example, hazard pointers can be used to maintain a small memory footprint, but a process must perform costly memory fences *every* time it tries to access a new descriptor.

To circumvent the aforementioned problems, we introduce a *weak descriptor* ADT (in Section 3) that has slightly *weaker semantics* than the mutable descriptor ADT, but can be implemented *without memory reclamation*. The crucial difference is that each time a process invokes *CreateNew* to create a new descriptor, it *invalidates* all of its previous descriptors. An invocation of *ReadField* on an invalid descriptor *fails* and returns a special value ⊥. Invocations of *WriteField* and *CASField* on invalid descriptors have no effect. We believe the weak descriptor ADT can be useful in designing new lock-free algorithms, since an invocation of *ReadField* that returns ⊥ can be used to inform a helper that it no longer needs to continue helping (making further accesses to the descriptor unnecessary).

We also identify a class of lock-free algorithms that use the descriptor ADT, and which can be *transformed* to use the weak descriptor ADT (in Section 3.2). At a high level, these are algorithms in which (1) each operation creates a descriptor and invokes a *Help* function on it, and (2) *ReadField*, *WriteField* and *CASField* operations occur only inside invocations of *Help*. Intuitively, the fact that these operations occur only in *Help* makes it easy to determine how the transformed algorithm should proceed when it performs an invalid operation: the operation being helped must have already terminated, so it no longer needs help. We prove correctness for our transformation, and demonstrate its use by transforming a wasteful implementation of a double-compare-single-swap (DCSS) primitive [17].

We then present an extension to our weak descriptor ADT, and show how algorithms that perform *ReadField* operations *outside* of *Help* can be transformed to use this extension (in Section 4). We prove correctness for the transformation, and demonstrate its use by transforming wasteful implementations of a k-compare-and-swap (k-CAS) primitive [17] and the LLX and SCX primitives of Brown et al. [11]. These primitives can be used to implement a wide variety of advanced lock-free data structures. For example, LLX and SCX have been used to implement lists, chromatic trees, relaxed AVL trees, relaxed (a, b)-trees, relaxed b-slack trees and weak AVL trees [10, 8, 18].

We use mostly known techniques to produce an efficient, provably correct implementation of our extended weak descriptor ADT (in Section 5). The high level idea is to (1) store a sequence number in each descriptor, (2) replace pointers to descriptors with *tagged sequence numbers*, which contain a process name and a sequence number, and (3) increment the sequence number in a descriptor each time it is reused.

With this implementation, the transformed algorithms for k-CAS, and LLX and SCX, have some desirable properties. In the original k-CAS algorithm, *each operation attempt* allocates at least $k+1$ new descriptors. In contrast, the transformed algorithm allocates only two descriptors *per process, once, at the beginning of the execution*, and these descriptors are reused. Similarly, in the original algorithm for LLX and SCX, each SCX operation creates a new descriptor, but the transformed algorithm allocates only one descriptor per process, at the beginning of the execution. This entirely eliminates dynamic allocation *and* memory reclamation for descriptors (significantly reducing overhead), and results in an extremely small descriptor footprint.

We present extensive experiments on a 64-thread AMD system and a 48-thread Intel system (in Section 6). Our results show that transformed implementations always perform at least as well as their wasteful counterparts, and *significantly* outperform them in some workloads. In a k-CAS microbenchmark, our implementation outperformed wasteful implementations using fast distributed epoch-based reclamation [7], hazard pointers [26] and read-copy-update (RCU) [13] by up to 2.3x, 3.3x and 5.0x, respectively. In a microbenchmark using a binary search tree (BST) implemented with LLX and SCX, our transformed implementation is up to 57% faster than the next best wasteful implementation.

The crucial observation in this work is that, in algorithms where descriptors are used only to facilitate helping, a descriptor is no longer needed once its operation has terminated. This allows a process to reuse a descriptor as soon as its operation finishes, instead of allocating a new descriptor for each operation, and waiting considerably longer (and incurring much higher overhead) to reclaim it using standard memory reclamation techniques. The challenge in this work is to characterize the set of algorithms that can benefit from this observation, and to design and prove the correctness of a transformation that takes such algorithms and produces new algorithms that simply reuse a small number of descriptors. As a result of developing this transformation, we also produce significantly faster implementations of k-CAS, and LLX and SCX.

2 Wasteful Algorithms

In this section, we describe two classes of lock-free wasteful algorithms, and give descriptor ADTs that capture their behaviour. First, we consider algorithms with *immutable* descriptors, which are not changed after they are initialized. We then discuss algorithms with *mutable* descriptors, which are modified by helpers.

For the sake of illustration, we start by describing one common way that lock-free wasteful

algorithms are implemented. Consider a lock-free algorithm that implements a set of *high-level* operations. Each high-level operation consists of one or more *attempts*, which either succeed, or fail due to contention. Each high-level operation attempt accesses a set of objects (e.g., individual memory locations or nodes of a tree). Conceptually, a high-level operation attempt locks a subset of these objects and then possibly modifies some of them. These locks are special: instead of providing exclusive access to a *process*, they provide exclusive access to a *high-level operation attempt*. Whenever a high-level operation attempt by a process p is unable to lock an object because it is already locked by another high-level operation attempt O, p first *helps O* to complete, before continuing its own attempt or starting a new one. By helping O complete, p effectively removes the locks that prevent it from making progress. Note that p is able to access objects locked for a different high-level operation attempt (which is not possible in traditional lock-based algorithms), but only for the purpose of helping the other high-level operation attempt complete.

We now discuss how helping is implemented. Each high-level operation or operation attempt allocates a new *descriptor* object, and fills it with information that describes any modifications it will perform. This information will be used by any processes that help the high-level operation attempt. For example, if the lock-free algorithm performs its modifications with a sequence of CAS steps, then the descriptor might contain the addresses, expected values and new values for the CAS steps.

A high-level operation attempt locks each object it would like to access by publishing pointers to its descriptor, typically using CAS. Each pointer may be published in a dedicated field for descriptor pointers, or in a memory location that is also used to store application values. For example, in the BST of Ellen et al., nodes have a separate field for descriptor pointers [14], but in Harris' implementation of multi-word CAS from single-word CAS, high-level operations temporarily replace application values with pointers to descriptors [17].

When a process encounters a pointer *ptr* to a descriptor (for a high-level operation attempt that is not its own), it may decide to help the other high-level operation attempt by invoking a function *Help(ptr)*. Typically, *Help(ptr)* is also invoked by the process that started the high-level operation. That is, the mechanism used to help is the same one used by a process to perform its own high-level operation attempt.

Wasteful algorithms typically assume that, whenever an operation attempt allocates a new descriptor, it uses fresh memory that has never previously been allocated. If this assumption is violated, then an *ABA problem* may occur. Suppose a process p reads an address x and sees A, then performs a CAS to change x from A to C, and interprets the success of the CAS to mean that x contained A at all times between the read and CAS. If another process changes x from A to B and back to A between p's read and CAS, then p's interpretation is invalid, and an ABA problem has occurred. Note that safe memory reclamation algorithms will reclaim a descriptor only if no process has, or can obtain, a pointer to it. Thus, no process can tell whether a descriptor is allocated fresh or reclaimed memory. So, safe memory reclamation will not introduce ABA problems.

2.1   Immutable descriptors

We give a straightforward *immutable descriptor* ADT that captures the way that descriptors are used by the class of wasteful algorithms we just described. A *descriptor* has a set of fields, and each field contains a value. The ADT offers two operations: *CreateNew* and *ReadField*. *CreateNew* takes, as its arguments, a descriptor type and a sequence of values, one for each field of the descriptor. It returns a unique descriptor pointer *des* that has never previously been returned by *CreateNew*. Every descriptor pointer returned by *CreateNew* represents a

new immutable descriptor object. *ReadField* takes, as its arguments, a descriptor pointer *des* and a field *f*, and returns the value of *f* in *des*.

In wasteful algorithms, whenever a process wants to create a new descriptor, it simply invokes *CreateNew*. Whenever a helper wants to access a descriptor, it invokes *ReadField*.

2.1.1 Progress

If the immutable descriptor ADT is implemented so that *CreateNew* allocates and initializes a new descriptor, and *ReadField* reads and returns a field of a descriptor, then its operations will be *wait-free* (i.e., each operation will terminate after a finite number of its own steps). However, wait-free descriptor operations are not necessary to guarantee lock-freedom for high-level operations that use descriptors. Instead, we simply require descriptor operations to be lock-free. We now explain why this is sufficient to implement lock-free data structures.

Consider a lock-free algorithm that uses a wait-free implementation of the immutable descriptor ADT. Suppose we transform this algorithm by replacing the wait-free implementation of the descriptor ADT with a *lock-free* implementation. We argue that the transformed algorithm remains lock-free. In other words, we show that, if processes take infinitely many steps in the transformed algorithm, then infinitely many high-level operations complete.

In the original algorithm, if processes take infinitely many steps, then infinitely many high-level operations will complete. The only steps we change to obtain the transformed algorithm are invocations of *CreateNew* and *ReadField*, some of which might no longer terminate. Therefore, the only way the transformed algorithm can *fail* to satisfy lock-freedom is if, eventually, all processes take steps only in non-terminating invocations of *CreateNew* and *ReadField*. (Otherwise, processes take infinitely many steps of the original algorithm, so infinitely many high-level operations will succeed.) In this case, only finitely many invocations of *CreateNew* and *ReadField* will terminate. However, since *CreateNew* and *ReadField* are lock-free, infinitely many invocations of *CreateNew* and/or *ReadField* must terminate. Thus, a lock-free implementation of the immutable descriptor ADT is sufficient to implement lock-free algorithms.

2.1.2 Example Algorithm: DCSS

We use the double-compare single-swap (*DCSS*) algorithm of Harris et al. [17] as an example of a lock-free algorithm that fits the preceding description. Its usage of descriptors is easily captured by the immutable descriptor ADT. A $DCSS(a_1, e_1, a_2, e_2, n_2)$ operation does the following *atomically*. It checks whether the values in addresses a_1 and a_2 are equal to a pair of expected values, e_1 and e_2. If so, it stores the value n_2 in a_2 and returns e_2. Otherwise it returns the current value of a_2.

Pseudocode for the *DCSS* algorithm appears in Figure 1. At a high level, *DCSS* creates a descriptor, and then attempts to lock a_2 by using CAS to replace the value in a_2 with a pointer to its descriptor. Since the *DCSS* algorithm replaces values with descriptor pointers, it needs a way to distinguish between values and descriptor pointers (in order to determine when helping is needed). So, it steals a bit from each memory location and uses this bit to *flag* descriptor pointers.

We now give a more detailed description. *DCSS* starts by creating and initializing a new descriptor *des* at line 2. It then flags *des* at line 3. We call the result *fdes* a *flagged pointer*. *DCSS* then attempts to lock a_2 in the loop at lines 4-7. In each iteration, it tries to store its flagged pointer in a_2 using CAS. If the CAS is successful, then the operation attempt invokes *DCSSHelp* to complete the operation (at line 8). Now, suppose the CAS

```
1   DCSS(a₁, e₁, a₂, e₂, n₂) :                     17   type  DCSSdes :
2     des := CreateNew(DCSSdes, a₁, e₁, a₂, e₂, n₂)       {ADDR₁, EXP₁, ADDR₂, EXP₂, NEW₂}
3     fdes := flag(des)
4     loop
5       r := CAS(a₂, e₂, fdes)
6       if  r is flagged then  DCSSHelp(r)        21   DCSSHelp(fdes) :
7       else exit loop                            22     des := unflag(fdes)
8     if  r = e₂ then  DCSSHelp(fdes)             23     a₁ := ReadField(des, ADDR₁)
9     return  r                                   24     a₂ := ReadField(des, ADDR₂)
                                                  25     e₁ := ReadField(des, EXP₁)
11  DCSSRead(addr) :                              26     if  *a₁ = e₁ then
12    loop                                        27       n₂ := ReadField(des, NEW₂)
13      r := *addr                                28       CAS(a₂, fdes, n₂)
14      if  r is flagged then  DCSSHelp(r)        29     else
15      else exit loop                            30       e₂ := ReadField(des, EXP₂)
16    return  r                                   31       CAS(a₂, fdes, e₂)
```

Figure 1 Code for the *DCSS* algorithm of Harris et al. [17] using the *immutable descriptor* ADT.

fails. Then, the *DCSS* checks whether its CAS failed because a_2 contained another *DCSS* operation's flagged pointer (at line 6). If so, it invokes *DCSSHelp* to help the other *DCSS* complete, and then retries its CAS. *DCSS* repeatedly performs its CAS (and helping) until the *DCSS* either succeeds, or fails because a_2 did not contain e_2.

DCSSHelp takes a flagged pointer $fdes$ as its argument, and begins by unflagging $fdes$ (to obtain the actual descriptor pointer for the operation). Then, it reads a_1 and checks whether it contains e_1 (at line 26). If so, it uses CAS to change a_2 from $fdes$ to n_2, completing the *DCSS* (at line 28). Otherwise, it uses CAS to change a_2 from $fdes$ to e_2, effectively aborting the *DCSS* (at line 31). Note that this code is executed by the process that created the descriptor, and also possibly by several helpers. Some of these helpers may perform a CAS at line 26 and some may perform a CAS at line 28, but only the first of these CAS steps can succeed.

When a program uses DCSS, some addresses can contain either values or descriptor pointers. So, each read of such an address must be replaced with an invocation of a function called *DCSSRead*. *DCSSRead* takes an address $addr$ as its argument, and begins by reading $addr$ (at line 13). It then checks whether it read a descriptor pointer (at line 14) and, if so, invokes *DCSSHelp* to help that *DCSS* complete. *DCSSRead* repeatedly reads and performs helping until it sees a value, which it returns (at line 16).

2.2   Mutable descriptors

In some more advanced lock-free algorithms, each descriptor also contains information about the *status* of its high-level operation attempt, and this status information is used to coordinate helping efforts between processes. Intuitively, the status information gives helpers some idea of what work has already been done, and what work remains to be done. Helpers use this information to direct their efforts, and update it as they make progress. For example, the state information might simply be a bit that is set (by the process that started the high-level operation, or a helper) once the high-level operation succeeds.

As another example, in an algorithm where high-level operation attempts proceed in several phases, the descriptor might store the current phase, which would be updated by helpers as they successfully complete phases. Observe that, since lock-free algorithms cannot use mutual exclusion, helpers often use CAS to avoid making conflicting changes to status information, which is quite expensive. Updating status information may introduce contention. Even when there is no contention, it adds overhead. Lock-free algorithms typic-

ally try to minimize updates to status information. Moreover, status information is usually simplistic, and is encoded using a small number of bits.

Status information might be represented as a single field in a descriptor, or it might be distributed across several fields. Any fields of a descriptor that contain status information are said to be *mutable*. All other fields are called *immutable*, because they do not change during an operation.

Mutable descriptor ADT. We now extend the immutable descriptor ADT to provide operations for changing (mutable) fields of descriptors. The *mutable descriptor* ADT offers four operations: *CreateNew*, *WriteField*, *CASField* and *ReadField*. The semantics for *CreateNew* and *ReadField* are the same as in the immutable descriptor ADT. *WriteField* takes, as its arguments, a descriptor pointer *des*, a field f and a value v. It stores v in field f of *des*. *CASField* takes, as its arguments, a descriptor pointer *des*, a field f, an expected value *exp* and a new value v. Let v_f be the value of f in *des* just before the *CASField*. If $v_f = exp$, then *CASField* stores v in f. *CASField* returns v_f. As in the immutable descriptor ADT, we require the operations of the mutable descriptor ADT to be lock-free.

2.2.1 Example Algorithm: k-CAS

A k-$CAS(a_1, ..., a_k, e_1, ..., e_k, n_1, ..., n_k)$ operation atomically does the following. First, it checks if each address a_i contains its expected value e_i. If so, it writes a new value n_i to a_i for all i and returns true. Otherwise it returns false.

The k-CAS algorithm of Harris et al. [17] is an example of a lock-free algorithm that has descriptors with mutable fields. At a high level, a k-CAS operation O in this algorithm starts by creating a descriptor that contains its arguments. It then tries to lock each location a_i *for the operation O* by changing the contents of a_i from e_i to *des*, where *des* is a pointer to O's descriptor. If it successfully locks each location a_i, then it changes each a_i from *des* to n_i, and returns true. If it fails because a_i is locked for another operation, then it helps the other operation to complete (and unlock its addresses), and then tries again. If it fails because a_i contains an application value different from e_i, then the k-CAS fails, and unlocks each location a_j that it locked by changing it from *des* back to e_j, and returns false. (The same thing happens if O fails to lock a_i because the operation has already terminated.)

We now give a more detailed description of the algorithm. Pseudocode appears in Figure 2. A k-CAS operation creates its descriptor at line 5, and then invokes a function *k-CASHelp* to complete the operation. In addition to the arguments to its k-CAS operation, a k-CAS descriptor contains a 2-bit *state* field that initially contains *Undecided* and is changed to *Succeeded* or *Failed* depending on how the operation progresses. This *state* field is used to coordinate helpers.

Let p be a process performing (or helping) a k-CAS operation O that created a descriptor d. If p fails to lock some address a_i in d, then p attempts to change the *state* of d using CAS from *Undecided* to *Failed*. On the other hand, if p successfully locks each address in d, then p attempts to change the *state* of d using CAS from *Undecided* to *Succeeded*. Since the *state* field changes only from *Undecided* to either *Failed* or *Succeeded*, only the first CAS on the state field of d will succeed. The k-CAS implementation then uses a lock-free DCSS primitive (the one presented in Section 2.1.2) to ensure that p can lock addresses for O *only* while d's *state* is *Undecided*. This prevents helpers from erroneously performing successful CAS steps after the k-CAS operation is already over.

Recall that the DCSS algorithm allocates a descriptor for each DCSS operation. A k-CAS operation performs potentially *many* DCSS operations (at least k for a successful k-CAS), and also allocates its own k-CAS descriptor. The k-CAS algorithm need not be

```
1   type  k-CASdes : {STATE, ADDR₁, EXP₁, NEW₁,
          ADDR₂, EXP₂, NEW₂, . . . , ADDRₖ, EXPₖ, NEWₖ}

3   ▷ k-CAS ADT operations
4   k-CAS(a₁, e₁, n₁, a₂, e₂, n₂, . . . , aₖ, eₖ, nₖ) :
5     des := CreateNew(k-CASdes, Undecided, a₁, e₁, n₁, . . .)
6     fdes := flagged version of des
7     return  k-CASHelp(fdes)

9   k-CASRead(addr) :
10    loop
11      r := DCSSRead(addr)
12      if  r is flagged  then  k-CASHelp(r)
13      else exit loop
14    return  r

16  ▷ Private procedures
17  k-CASHelp(fdes) :
18    des := remove the flag from fdes
19    ▷ Use DCSS to store fdes in each of a₁, a₂, . . . , aₖ
20    ▷ only if des has STATE Undecided and aᵢ = eᵢ for all i
21    if  ReadField(des, STATE) = Undecided  then
22      state := Succeeded
23      for  i = 1...k  do
24  retry_entry :
25        a₁ := ReadField(des, STATE)
26        a₂ := ReadField(des, ADDRᵢ)
27        e₂ := ReadField(des, EXPᵢ)
28        val := DCSS(⟨des, STATE⟩, Undecided, a₂, e₂, fdes)
29        if  val is flagged  then
30          if  val ≠ fdes  then
31            k-CASHelp(val)
32            goto  retry_entry
33          else
34            if  val ≠ e₂  then
35              state := Failed
36              break
37      CASField(des, STATE, Undecided, state)

39    ▷ Replace fdes in a₁, ..., aₖ with n₁, ..., nₖ or e₁, ..., eₖ
40    state := ReadField(des, STATE)
41    for  i = 1...k  do
42      a = ReadField(des, ADDRᵢ)
43      if  state = Succeeded  then
44        new := ReadField(des, NEWᵢ)
45      else
46        new := ReadField(des, EXPᵢ)
47      CAS(a, fdes, new)
48    return  (state = Succeeded)
```

Figure 2 Code for the k-CAS algorithm of Harris et al. [17] using the *mutable descriptor* ADT.

aware of DCSS descriptors (or of the bit reserved in each memory location by the DCSS algorithm to flag values as DCSS descriptor pointers), since it can simply use the *DCSSRead* procedure described above whenever it accesses a memory location that might contain a DCSS descriptor. However, the k-CAS algorithm performs DCSS on the *state* field of a k-CAS descriptor, which is accessed using the k-CAS descriptor's *ReadField* operation. To allow DCSS to access the *state* field, we must modify DCSS slightly. First, instead of passing an address a_1 to DCSS, we pass a pointer to the k-CAS descriptor and the name of the *state* field (at line 28 of Figure 2). Second, we replace the read of $addr_1$ in DCSS (at line 26 of Figure 1) with an invocation of *ReadField*.

Since *k*-CAS descriptor pointers are temporarily stored in memory locations that normally contain application values, the *k*-CAS algorithm needs a way to determine whether a value in a memory location is an application value or a *k*-CAS descriptor pointer. In the DCSS algorithm, the solution was to reserve a bit in each memory location, and use this bit to *flag* the value contained in the location as a pointer to a DCSS descriptor. Similarly, the *k*-CAS algorithm reserves a bit in each memory location to flag a value as a *k*-CAS descriptor pointer. The *k*-CAS and DCSS algorithms need not be aware of each other's reserved bits, but they should not reserve the same bit (or else, for example, a DCSS operation could encounter a *k*-CAS descriptor pointer, and interpret it as a DCSS descriptor pointer).

When the *k*-CAS algorithm is used, some memory addresses may contain either values or descriptor pointers, so reads of such addresses must be replaced by a *k-CASRead* operation. This operation reads an address, and checks whether it contains a *k*-CAS descriptor pointer. If so, it helps the *k*-CAS operation to complete, and tries again. Otherwise, it returns the value it read. For further details, on the *k*-CAS algorithm refer to [17].

3   Weak descriptors

In this section we present a *weak descriptor* ADT that has weaker semantics than the mutable descriptor ADT, but can be implemented more efficiently (in particular, without requiring any memory reclamation for descriptors). We identify a class of algorithms that use the mutable descriptor ADT, and which can be transformed to use the weak descriptor ADT, instead.

We first discuss a restricted case where operation attempts only create a single descriptor, and we give an ADT, transformation and proof for that restricted case. (In the next section, we describe how the ADT and transformation can be modified slightly to support operation attempts that create multiple descriptors.)

3.1   Weak descriptor ADT

The weak descriptor ADT is a variant of the mutable descriptor ADT that allows some operations to *fail*. To facilitate the discussion, we introduce the concept of descriptor validity. Let *des* be a pointer returned by a *CreateNew* operation O by a process p, and d be the descriptor pointed to by *des*. In each configuration, d is either **valid** or **invalid**. Initially, d is valid. If p performs another *CreateNew* operation O' *after* O, then d becomes invalid immediately after O' (and will never be valid again).

We say that a *ReadField*(*des*, ...), *WriteField*(*des*, ...) or *CASField*(*des*, ...) operation is performed **on a descriptor** d, where *des* is a pointer to d. An operation on a valid (resp., invalid) descriptor is said to be valid (resp., invalid). Invalid operations have no effect on any base object, and return a special value $\perp$ (which is never contained in a field of any descriptor) instead of their usual return value. We say that a *CreateNew* operation O is performed **on a descriptor** d if O returns a pointer to d. Observe that a *CreateNew* operation is always valid. We say that a process p **owns** a descriptor d if it performed a *CreateNew* operation that returned a pointer *des* to d.

The semantics for *CreateNew* are the same as in the mutable descriptor ADT. The semantics for the other three operations are the same as in the mutable descriptor ADT, except that they can be invalid. As in the previous ADTs, these operations must be lock-free.

3.2 Transforming a class of algorithms to use the weak descriptor ADT

We now formally define a class of lock-free algorithms that use the mutable descriptor ADT, and can easily be transformed so that they use the weak descriptor ADT, instead. We say that a step s of an execution is *nontrivial* if it changes the state of an object o in shared memory, and *trivial* otherwise. In particular, all invalid operations are trivial, and an unsuccessful CAS or a CAS whose expected and new values are the same are both trivial. In the following, we abuse notation slightly by referring interchangeably to a descriptor and a pointer to it.

▶ **Definition 1.** Weak-compatible algorithms (WCA) are lock-free wasteful algorithms that use the mutable descriptor ADT, and have the following properties:

1. Each high-level operation attempt O by a process p may create (and initialize) a single descriptor d. Inside O, p may perform at most one invocation of a function $Help(d)$ (and p may not invoke $Help(d)$ outside of O).
2. A process may help any operation attempt O' by another process by invoking $Help(d')$ where d' is the descriptor that was created by O'.
3. If O terminates at time t, then any steps taken in an invocation of $Help(d)$ after time t are *trivial* (i.e., do not **change** the state of **any** shared object, incl. d).
4. While a process $q \neq p$ is performing $Help(d)$, q cannot change any variables in its private memory that are still defined once $Help(d)$ terminates (i.e., variables that are local to the process q, but are not local to $Help$).
5. All accesses (read, write or CAS) to a field of d occur inside either $Help(d)$ or O.

At a high level, properties 1 and 2 of WCA describe how descriptors are created and helped. Property 4 intuitively states that, whenever a process q finishes helping another process perform its operation attempt, q knows only that it finished helping, and does not remember anything about what it did while helping the other process. In particular, this means that q cannot pay attention to the return value of *Help*. We explain why this behaviour makes sense. If q creates a descriptor d as part of a high-level operation attempt O and invokes $Help(d)$, then q might care about the return value of *Help*, since it needs to compute the response of O. However, if q is just helping another process p's high-level operation attempt O, then it does not care about the response of *Help*, since it does not need to compute the response of O. The remaining properties, 3 and 5, allow us to argue that the contents of a descriptor are no longer needed once the operation that created it has terminated (and, hence, it makes sense for the descriptor to become invalid). In Section 4, we will study a larger class of algorithms with a weaker version of property 5.

The transformation. Each algorithm in WCA can be transformed in a straightforward way into an algorithm that uses the weak descriptor ADT as follows. Consider any *ReadField* or *CASField* operation op performed by a high-level operation attempt O in an invocation of $Help(d)$, where d was created by a *different* high-level operation attempt O'. Note that op is performed while O is *helping* O'. After op, a check is added to determine whether op was invalid, in which case p returns from *Help* immediately. (In this case, *Help* does not need to continue, since op will be invalid only if O' has already been completed by the process that owns d or a helper.)

Example Algorithm: DCSS. Figure 3 shows code for the *DCSS* algorithm in Figure 1 that has been *transformed* to use the weak descriptor ADT. There, we include only the *DCSSHelp* procedure, since it is the only one that differs from Figure 1. The transformation adds lines 4, 6, 8, 11 and 15 to *check* whether the preceding invocations of *ReadField* are invalid.

```
1   DCSSHelp(fdes) :
2      des := Unflag(fdes)
3      a₁ := ReadField(des, ADDR₁)
4      if a₁ = ⊥ then return
5      a₂ := ReadField(des, ADDR₂)
6      if a₂ = ⊥ then return
7      e₁ := ReadField(des, EXP₁)
8      if e₁ = ⊥ then return
9      if *a₁ = e₁ then
10        n₂ := ReadField(des, NEW₂)
11        if n₂ = ⊥ then return
12        CAS(a₂, fdes, n₂)
13     else
14        e₂ := ReadField(des, EXP₂)
15        if e₂ = ⊥ then return
16        CAS(a₂, fdes, e₂)
```

■ **Figure 3** Applying the transformation to *DCSS*.

3.3 Correctness

We argue that our transformation takes a linearizable algorithm $\mathcal{A} \in$ WCA that uses mutable descriptors and produces a linearizable algorithm $\mathcal{A}'$ that uses weak descriptors. Consider any execution e' of the transformed algorithm $\mathcal{A}'$. We prove there exists an execution e of the original algorithm $\mathcal{A}$ that performs the *same* high-level operations, in the same order, and with the same responses, as in e'. We explain how this helps. Since e is a correct execution of the original algorithm $\mathcal{A}$, the high-level operations performed in e must respect the sequential specification(s) of the object(s) implemented in $\mathcal{A}$. Furthermore, since e' performs the same high-level operations, in the same order, and with the same responses, the high-level operations in e' must also respect the sequential specification(s) of the same object(s). Therefore, the transformed algorithm $\mathcal{A}'$ is correct.

We construct e as follows. By Property 5 of WCA, all *ReadField*, *WriteField* and *CAS-Field* operations occur in *Help*. Whenever a check by a process p follows a *ReadField* or *CASField* in e' that returns $\bot$ (because the operation attempt O being helped by p has already terminated), we replace that check by a consecutive sequence of steps in which p finishes its invocation of *Help*. All other checks immediately following *ReadField* or *CASField* are simply removed.

By Property 3 of WCA, none of the steps added to e change the state of any shared object. So, these steps will not change the behaviour of any other process. We also argue that none of these steps make any changes to p's private memory that persist after p finishes its invocation of *Help*. (I.e., any changes these steps make to p's private memory are *reverted* by the time p finishes its invocation of *Help*, so p's private memory is the same just after the invocation of *Help* as it was just before the invocation of *Help*.) So, these steps will not change the behaviour of p after it finishes its invocation of *Help*. Observe that, whenever a process performs a *ReadField* or *CASField* operation on a descriptor that it created, this operation will return a value different from $\bot$. This is due to Property 1 of WCA, and the definition of the weak descriptor ADT, which states that d becomes invalid only after O has terminated. Since p's invocation of *ReadField* or *CASField* returns $\bot$, p must therefore be performing *Help(d)* where d was created by a *different* process. Thus, Property 4 of WCA implies that, after p performs the sequence of steps to finish its invocation of *Help(d)*, its private memory has the same state as it did just before it invoked *Help*.

```
1    DCSSHelp(fdes) :
2       des := Unflag(fdes)
3       values := ReadImmutables(des)
4       if  values = ⊥ then return
5       ⟨a₁, e₁, a₂, e₂, n₂⟩ := values

7       if  *a₁ = e₁ then
8          CAS(a₂, fdes, n₂)
9       else
10         CAS(a₂, fdes, e₂)
```

Figure 4 Using *ReadImmutables* to optimize and streamline the transformed DCSS algorithm.

3.4   Reading immutable fields efficiently

If an invocation of *Help*(*des*) accesses many immutable fields of a descriptor, then we can optimize it by replacing many *ReadField* operations with a single, more efficient operation called *ReadImmutables*. This operation reads and returns *all* of a descriptor's immutable fields, unless the descriptor is invalid, in which case it returns ⊥.

To use *ReadImmutables* in *Help*(*des*), one can simply perform, at the beginning of *Help*, a *ReadImmutables* operation, followed by an *if*-statement that checks whether it the operation invalid, and, if so, returns immediately. Then, in the body of *Help*(*des*), each invocation of *ReadField*(*des*, *f*), where *f* is immutable, is replaced with a direct read from the set of values returned by *ReadImmutables*. We demonstrate this approach on the transformed pseudocode for DCSS in Figure 3. Figure 4 shows the result. Since all fields of a DCSS descriptor are immutable, *every* invocation of *ReadField* can be replaced with a direct read from the result of the *ReadImmutables* operation performed at line 3. (This will not be the case in an algorithm where the *Help* procedure reads mutable fields.) Since *ReadImmutables* replaces several invocations of *ReadField*, it has the added benefit of making code simpler and shorter.

4   Extended Weak Descriptors

In this section, we describe an extended version of the weak descriptor ADT, and an extended version of the transformation in Section 3.2. This extended transformation weakens property 5 of WCA so that *ReadField* operations on a descriptor *d* can also be performed *outside* of *Help*(*d*). At a high level, we handle *ReadField* operations performed outside of *Help* as follows. For *ReadField*s performed inside *Help*, we have seen that we can simply stop helping when ⊥ is returned. However, for *ReadField*s performed outside of *Help*, it is not clear, in general, how we should respond if ⊥ is returned. Intuitively, the goal is to find a value that *ReadField* can return so that the algorithm will behave the same way as it would if the descriptor were still valid. In some algorithms, just knowing that an operation has been completed gives us enough information to determine what a *ReadField* operation should return (as we will see below).

Extended weak descriptor ADT.   This ADT is the same as the weak descriptor ADT, except that *ReadField* is extended to take, as an additional argument, a default value *dv* that is returned instead of ⊥ when the operation is invalid. Observe that the weak descriptor ADT is a special case of the extended weak descriptor ADT where each argument *dv* to an invocation of *ReadField* is ⊥.

The extended transformation.   *CASField* and *WriteField* operations are handled the same way as in the WCA transformation. However, an invocation of *ReadField*(*des*, *f*) is

handled differently depending on whether it occurs inside an invocation of *Help*(*des*). If it does, it is replaced with an invocation of *ReadField*(*des*, *f*, ⊥) followed by the check, as in the WCA transformation. If not, it is replaced with an invocation of *ReadField*(*des*, *f*, *dv*), where the choice of *dv* is specific to the algorithm being transformed.

Let $\mathcal{A}$ be any algorithm that uses mutable descriptors, and satisfies properties 1-4 of WCA algorithms (see Definition 1), as well as a weaker version of property 5, called property 5′, which states: every write or CAS to a field of a descriptor *d* must occur in an invocation of *Help*(*d*). Let *e* be an execution of $\mathcal{A}$ and let *e*′ be an execution that is the same as *e*, except that one (arbitrary) descriptor *d* becomes invalid at some point *t* after the high-level operation attempt *O* that created *d* terminates. (When we say that *d* becomes invalid at time *t*, we mean that after *t*, each invocation of *ReadField*(*d*, *f*, *dv*) that is performed outside of *Help*(*d*) returns its default value *dv*.)

Let *O*′ be any high-level operation attempt in *e*′ which, after *t*, performs *ReadField* on *d* outside of *Help*(*d*). We say that an extended transformation is *correct for* $\mathcal{A}$ if, for all choices of *e*, *e*′, *d*, *t*, and *O*′, the exact same changes are performed by *O*′ in *e* and *e*′ to any variables that are still defined once *O*′ terminates (i.e., variables that are local to the process performing *O*′, but are not local to *O*′, and variables in shared memory), and *O*′ returns the same response in both executions. An algorithm $\mathcal{A}$ is an *extended weak-compatible algorithm* (and is in the class *EWCA*) if there is an extended transformation that is correct for $\mathcal{A}$.

4.1   Correctness

Consider any extended transformation which is correct for a linearizable algorithm $\mathcal{A}$ that uses mutable descriptors. We prove the result of applying this transformation to $\mathcal{A}$ is a linearizable algorithm $\mathcal{A}'$ that uses extended weak descriptors. Specifically, let *e*′ be any execution of $\mathcal{A}'$. We prove there is an execution *e* of $\mathcal{A}$ that performs the same high-level operations, in the same order, with the same responses, as in *e*′.

First, we define an execution e_0. Whenever a check in *e*′ by a process *p* in *Help*(*d*) determines that the preceding *ReadField* or *CASField* on a descriptor *d* is *invalid* (which means that the operation attempt being helped by *p* has already terminated), we replace that check by a consecutive sequence of steps in which *p* finishes its invocation of *Help*(*d*). By Property 3 of WCA, none of these added steps change the state of any shared variable. Moreover, by Property 4 of WCA, *p* does not change any variable that is still defined after its invocation of *Help*, so *p* has the same local state after *Help* in e_0 and *e*′. Whenever such a check determines that the preceding *ReadField* or *CASField* is *valid*, we simply remove this check. Observe that each invalid operation in e_0 is an invalid *ReadField* operation on some descriptor *d* performed outside of *Help*(*d*).

Let $d_1, d_2, ...$ be the sequence of descriptors created in e_0. We inductively construct a sequence $e_1, e_2, ...$ of executions such that e_i differs from e_{i-1} only in that descriptor d_i never becomes invalid in e_i. Specifically, for each high-level operation attempt *O*′ that performs an invalid *ReadField* operation on descriptor d_i outside of *Help*(d_i), consider the first such *ReadField* operation *R*. All of the steps of *O*′ prior to *R* are the same in e_i as in e_{i-1}. After *R*, *O*′ continues to take steps in e_i, but each *ReadField* operation that *O*′ performs on a field *f* of d_i returns the contents of *f* (instead of a default value). This may result in *O*′ executing completely different code paths in e_{i-1} and e_i. However, by the definition of an extended transformation that is correct for $\mathcal{A}$, *O*′ returns the same response in e_i and e_{i-1} and performs the *exact same changes* to any variables that are still defined once *O*′ terminates. Thus, for each variable *v* that is still defined once *O*′ terminates, we can schedule the sequence of changes to *v* in the exact same way in e_i and e_{i-1} (which implies

that any reads in e_{i-1} which see these changes can be scheduled appropriately in e_i).

Since the claim holds for all i, there is an execution e in which no descriptor becomes invalid (so e is an execution of $\mathcal{A}$), and the same high-level operation attempts are performed, in the same order, and with the same responses.

4.2   Multiple descriptors per operation attempt

In some lock-free algorithms, a high-level operation attempt can create several different descriptors, and potentially invoke a different *Help* procedure for each descriptor. We describe how to adjust the definitions above to support these kinds of algorithms. For simplicity, we think of there being a single *Help* procedure that checks the type of the descriptor passed to it, and behaves differently for different types.

In order to allow a high-level operation attempt to create multiple descriptors without simply invalidating the ones it previously created, we update the definition of valid and invalid descriptors. Let *des* be a pointer to a descriptor d of type T returned by a *CreateNew* operation C performed by process p. Initially, d is valid. If p performs another *CreateNew* operation C' with the *same descriptor type* T after C, then d becomes invalid immediately after C' (and will never be valid again).

With this definition of valid and invalid descriptors, it might initially seem like an operation cannot create multiple descriptors of the same type T. However, this turns out not to be a problem. If an operation should create multiple descriptors of type T, we can simply imagine creating multiple *clone* types $T_1, T_2, ...$ that have the exact same fields as T. To create k descriptors of type T, one would then create k clone types, and have an operation invoke *CreateNew* once for each clone type. (However, we are unaware of any algorithms in which a high-level operation attempt creates multiple descriptors of the same type.)

We also slightly modify Property 1 of (extended) weak-compatible algorithms, as follows, to accommodate the use of multiple descriptors. Each high-level operation attempt O by a process p may create (and initialize) a sequence D of descriptors, each with a **unique type**. Inside O, p may perform at most one invocation of a function $Help(d)$ for each $d \in D$ (and p may not invoke $Help(d)$ outside of O). Note that the proof for the extended weak transformation goes through unchanged.

4.3   Example Algorithm: k-CAS

In this section, we explain how the extended transformation is applied to the k-CAS algorithm presented in Section 2.2.1. Note that no invocations of *ReadField* on a DCSS descriptor *des* are performed outside of *HelpDCSS(des)*. There is only one place in the algorithm where an invocation I of *ReadField* on a k-CAS descriptor *des* is performed *outside* of *Help(des)* (the *Help* procedure for k-CAS). Specifically, I reads the *state* field of a k-CAS descriptor inside the modified version of *HelpDCSS*. Recall that the k-CAS algorithm passes a k-CAS descriptor pointer and the name of the *state* field as the first argument to DCSS at line 28 of Figure 2, and the DCSS algorithm is modified to use *ReadField* at line 26 of Figure 1 to read this *state* field. We choose the default value $dv = $ *Succeeded* for this invocation of *ReadField*. We explain why this extended transformation of the k-CAS algorithm is correct.

When I is performed at line 26 of *DCSSHelp* (in Figure 1), its response is compared with e_1, which contains *Undecided*. If I returns *Undecided*, then the CAS at line 28 is performed, and the process p performing I returns from *HelpDCSS*. Otherwise, the CAS at line 31 is performed, and p returns from *HelpDCSS*.

Suppose I is invalid. Then, we know the k-CAS operation attempt that created *des* has been completed. We use the following algorithm specific knowledge. After a k-CAS operation attempt has completed, its k-CAS descriptor has *state Succeeded* or *Failed* (and is never changed back to *Undecided*). (This can be determined by inspection of the code.) Thus, if I were valid, its response would *not* be *Undecided*, and p would perform the CAS at line 31 and return from *HelpDCSS*. Since $dv = Succeeded$, p does exactly the same thing when I is invalid. (Note that the exact value of *state* is unimportant. It is only important that it is not *Undecided*.)

4.4   Example Algorithm: LLX and SCX

In this section, we explain how the extended transformation is applied to the multiword synchronization primitives load-linked-extended (LLX) and store-conditional-extended (SCX) of Brown et al. [11]. Note that Brown et al. [10] also used these primitives to design a tree update template that can be followed to produce a fast lock-free implementation of any data structure based on a down-tree (a directed acyclic graph where each node has indegree one). Thus, by optimizing LLX and SCX, we also optimize the tree update template, and all of the data structures that have been implemented with it. Pseudocode for LLX and SCX using mutable descriptors is presented in Figure 5.

LLX and SCX operate on multi-field *data records*, which can be used to represent, e.g., nodes in a tree, or records in a table. Like descriptors, *data records* contain mutable and immutable fields. However, whereas descriptors are used only to facilitate helping, and are not part of a sequential data structure, data records are.

$LLX(r)$ attempts to take a snapshot of the mutable fields of a Data-record r. If it is concurrent with an SCX involving r, it may return FAIL, instead. Individual fields of a Data-record can also be read directly. An $SCX(V, R, fld, new)$ takes as its arguments a sequence V of Data-records, a subsequence R of V, a pointer fld to a mutable field of one Data-record in V, and a new value *new* for that field. The SCX tries to atomically: store the value *new* in the field that fld points to and *finalize* each Data-record in R. Once a Data-record is finalized, its mutable fields cannot be changed by any subsequent SCX, and any LLX of the Data-record will return FINALIZED instead of a snapshot.

Before a process invokes SCX, it must perform an $LLX(r)$ on each Data-record r in V. The last such LLX by the process is said to be *linked* to the SCX, and the linked LLX must return a snapshot of r (not FAIL or FINALIZED). An $SCX(V, R, fld, new)$ by a process modifies the data structure and returns TRUE only if no Data-record r in V has changed since its linked $LLX(r)$; otherwise the SCX fails and returns FALSE. Although LLX and SCX can fail, their failures are limited in such a way that they can be used to build data structures with lock-free progress. See [11] for a more formal specification of these primitives.

Each SCX operation creates a new descriptor called an SCX-record. LLX and SCX requires each Data-record r to have a dedicated field $r.des$ that stores a pointer to an SCX-record, and this field is only ever accessed by LLX and SCX operations. Each Data-record also has a *marked* bit which is accessed only by LLX and SCX. This field is used by SCX to finalize Data-records. We say that a Data-record is *marked* if its *marked* bit is set. SCX-records have two mutable fields: a 2-bit *state* field and an *allFrozen* bit. The *state* field contains one of three values: *InProgress*, *Committed* and *Aborted*.

The following properties of the LLX and SCX algorithm are relevant for our purposes.

P1. Before the first invocation of $Help(des)$ for an SCX O (performed by O or a helper) has been completed, the SCX-record *des* created by O has its *state* field set to *Committed* or *Aborted*, and, after this, the *state* field of *des* is never changed again.

type *SCXdes*
  ▷ Immutable descriptor fields
  NFREEZE   ▷ number of Data-records to be frozen
  NFINALIZE   ▷ number of Data-records to be finalized
  $V_1, V_2, ...$ ▷ Data-records to be frozen
  $R_1, R_2, ...$ ▷ Data-records to be finalized (must be a subsequence of $\langle V_1, V_2, ... \rangle$)
  $DES_1, DES_2, ...$ ▷ descriptor pointers read from $V_1.info, V_2.info, ...$
  FLD   ▷ pointer to a field of some V_i
  NEW   ▷ value to be written into the field FLD
  OLD   ▷ value previously read from the field FLD
  ▷ Mutable descriptor fields
  STATE   ▷ one of {InProgress, Committed, Aborted}
  ALLFROZEN   ▷ Boolean

```
1   LLX(r) by process p
2      marked_1 := r.marked
3      rinfo := r.info
4      state := ReadField(SCXdes, rinfo, STATE)
5      marked_2 := r.marked
6      if state = Aborted or (state = Committed and not marked_2) then     ▷ if r was not frozen at line 4
7         read r.m_1, ..., r.m_y  and record the values in local variables m_1, ..., m_y
8         if r.info = rinfo then                                           ▷ if r.info contains the same
9            store ⟨r, rinfo, ⟨m_1, ..., m_y⟩⟩ in p's local table           ▷ descriptor as on line 3
10           return ⟨m_1, ..., m_y⟩
11     if state = InProgress then Help(rinfo)
12     if marked_1 then return FINALIZED
13     else return FAIL
```

```
14  SCX(V = ⟨V_1, V_2, ..., V_k⟩, R = ⟨R_1, R_2, ..., R_l⟩, fld, new) by process p
15  ▷ Preconditions: (1) for each r in V, p has performed an invocation I_r of LLX(r) linked to this SCX
                      (2) new is not the initial value of fld
                      (3) for each r in V, no SCX(V', R', fld, new) was linearized before I_r was linearized
16     Let des_1, des_2, ..., des_k be the descriptor pointers for V_1, V_2, ..., V_k in p's local table of LLX results
17     Let old be the value for fld stored in p's local table of LLX results
18     des := CreateNew(SCXdes, {(NFREEZE, k), (NFINALIZE, l), (V_1, V_1), (V_2, V_2), ..., (V_k, V_k),
           (R_1, R_1), (R_2, R_2), ..., (R_l, R_l), (DES_1, des_1), (DES_2, des_2), ..., (DES_k, des_k),
           (FLD, fld), (NEW, new), (OLD, old), (STATE, InProgress), (ALLFROZEN, FALSE)})
19     return Help(des)
```

```
20  Help(des)
21     ▷ Freeze all Data-records in des.V to protect their mutable fields from being changed by other SCXs
22     ⟨nfreeze, nfinal, V_1, V_2, ..., V_nfreeze, R_1, R_2, ..., R_nfinal, des_1, des_2, ..., des_nfreeze, fld, new, old⟩ := ReadImmutables(SCXdes, des)
23     for i = 1..nfreeze do
24        if not CAS(V_i.info, des_i, des) then                           ▷ freezing CAS
25           if V_i.info ≠ des then
26              ▷ Could not freeze V_i because it is frozen for another SCX
27              if ReadField(SCXdes, des, ALLFROZEN) = TRUE then           ▷ frozen check step
28                 ▷ the SCX has already completed successfully
29                 return TRUE
30              else
31                 ▷ Atomically unfreeze all Data-records frozen for this SCX
32                 WriteField(SCXdes, des, STATE, Aborted)                 ▷ abort step
33                 return FALSE
34     ▷ Finished freezing Data-records (Assert: state ∈ {InProgress, Committed})
35     WriteField(SCXdes, des, ALLFROZEN, TRUE)                           ▷ frozen step
36     for i = 1..nfinal do
37        R_i.marked := TRUE                                              ▷ mark step
38     CAS(fld, old, new)                                                 ▷ update CAS
39     ▷ Finalize all R_i in R, and unfreeze all V_i in V that are not in R
40     WriteField(SCXdes, des, STATE, Committed)                         ▷ commit step
41     return TRUE
```

■ **Figure 5** Code for the *LLX* and *SCX* algorithm using the *mutable descriptor* ADT.

P2. A marked Data-record remains marked forever.

P3. A marked Data-record cannot point to an *SCX*-record with *state* = *Aborted*.

P4. Each time the *des* field of a Data-record changes, it changes to a new value that has never previously been stored there (to avoid the ABA problem).

There is only one place in the code where an invocation I of *ReadField*(*SCX*-record, d, f, dv) can occur outside of *Help*(*des*): at line 4 of *LLX* in Figure 5. I reads the *state* field of d. We choose the default value dv = *Committed* for I. We give a rigorous, but straightforward, proof that this extended transformation of *LLX* and *SCX* is correct.

Let e be an execution of the original *LLX* and *SCX* algorithm $\mathcal{A}$, and let e' be an execution that is the same as e, except that one arbitrary *SCX*-record d becomes invalid at some point t after the *SCX* operation attempt O that created d terminates. Let O' be any *LLX* in e which, after t, performs an invocation I of *ReadField* on d outside of *Help*(d). We must prove that O' performs the exact same changes in e and e' to any variables that are still defined after O' terminates, and returns the same response in both executions.

Since I is invalid in e', by definition, the *SCX* O that created d must have terminated before I. Thus, by P1, I must return *Committed* or *Aborted* in e. If I returns *Committed* in e, then I returns the same response in e and e', so O' is exactly the same in both executions. Now, suppose I returns *Aborted* in e. We consider three cases, depending on where O' returns in e.

Case 1: O' returns at line 10 in e. If $marked_2$ = FALSE, then O' behaves exactly the same way in e and e'. So, suppose $marked_2$ = TRUE. Then, O' will enter the if-statement at line 6 in e, but not in e'. In this case, O' saw that the Data-record r pointed to an *SCX*-record with *state* = *Aborted* when it performed line 3, and that r was marked when it performed line 5. By P3, r cannot simultaneously be marked and point to an *SCX*-record with *state* = *Aborted*, so $r.des$ must change between these two lines. By P4, it must change to a value different from $r\,des$, so the if-block at line 8 will not be executed in e. However, this contradicts our assumption that O' returns at line 10.

Case 2: O' returns FINALIZED at line 12 in e. Observe that O' does not execute line 9 in e (since it would then return at the following line). We first prove that O' does not execute line 9 in e'. Since O' sees $marked_1$ = TRUE just before returning at line 12 in e, P2 implies that $marked_2$ = TRUE (in both e and e'). Since I returns *Committed* in e', O' will not enter the if-block at line 6 in e'. Thus, O' reaches line 11 in both e and e'.

Since I returns *Committed* in e', and we have assumed I returns *Aborted* in e, O' will not invoke *Help* at line 11 in e or e'. Therefore, O' does not change any variable that is still defined after it terminates. So, it suffices to prove that O' returns FINALIZED (at line 12) in e'. However, this is immediate from the fact that $marked_1$ = TRUE in O' in e (and, hence, in e').

Case 3: O' returns FAIL at line 13 in e. The proof is similar to the previous case, except $marked_1$ = FALSE in O' in e, so when O' reaches line 12, it will enter the *else*-block and return FAIL in both e and e'.

5 Implementing the extended weak descriptor ADT

We give a brief high-level overview of our implementation. It uses largely known techniques (similar to [24]), and is not the main contribution of this work. Each process p uses a *single* descriptor object $D_{T,p}$ in shared memory to represent *all* descriptors of type T that it ever creates. The descriptor object $D_{T,p}$ conceptually represents p's *current* descriptor of type T. At different times in an execution, $D_{T,p}$ represents different *abstract descriptors* created

by p. We store a sequence number in $D_{T,p}$ that is incremented every time p performs *CreateNew(T, −)*. Instead of using traditional descriptor pointers, we represent each descriptor pointer as a pair of fields stored in a single word. These fields contain the name of the process who owns the descriptor, and a sequence number that indicates which invocation of *CreateNew* conceptually created this descriptor. When a descriptor pointer is passed to an operation O on the abstract descriptor, O compares the sequence number in *des* with the current sequence number in $D_{T,p}$ to determine whether the operation is valid or invalid. Thus, incrementing the sequence number in $D_{T,p}$ effectively makes all abstract descriptors of type T that were previously created by p *invalid*. Mutable fields are stored in a single word alongside a sequence number, so they can be updated with CAS, preventing invalid operations from making changes. (If the mutable fields and a sequence number cannot fit in one word, then one can use multiple words and attach the sequence number to each word.)

5.1   Detailed description

Complete pseudocode appears in Figure 6. We start by describing the data types and shared variables. Each descriptor contains zero or more immutable fields, and zero or more mutable fields (which are determined by the *descriptor type*), as well as a sequence number field *seq*. Recall that $D_{T,p}$ represents different abstract descriptors at different times. Note that the immutable fields of $D_{T,p}$ are only immutable for as long as $D_{T,p}$ represents the same abstract descriptor. When $D_{T,p}$ is reused, so that it represents a different abstract descriptor, its immutable fields can be reinitialized. Usually very few bits are required for the mutable fields, since they exist solely to capture the state of an ongoing operation (and it is inefficient to frequently change the state of a descriptor). (Every lock-free algorithm we are aware of uses at most a small constant number of bits for its mutable fields.) Consequently, we think of the sequence field and the mutable fields of a descriptor d as being packed together in a single word *mutables* of d (with subfields for the sequence field and each mutable field). (Note that, if more space is needed for mutable fields in some future algorithm, we can eliminate this assumption about the size of *mutables*, as we explain below.) We use $d.f$ to denote an immutable field f of d, $d.mutables$ to denote the field *mutables* of d, and $d.mutables.f$ to denote a mutable field f of d.

Since *mutables* fits in a single word, it can be modified atomically using CAS. By having CAS atomically operate on a mutable field and the sequence number, we can ensure that a descriptor changes only if its sequence number has not changed.

We now describe the operations. An invocation of *CreateNew(T, ...)* by process p first increments the sequence number of $D_{T,p}$, then initializes all of its fields, then increments the sequence number again and returns a new descriptor pointer (with the up-to-date sequence number). Observe that the descriptor pointers returned by *CreateNew* always have even sequence numbers, and the sequence number of a descriptor is odd while it is being initialized by *CreateNew*. Consequently, while a descriptor is being initialized, its sequence number does not match any descriptor pointer in the system, so no process can read or modify the descriptor's fields.

Note that this approach of incrementing a sequence number twice has been used in different contexts such as in transactional memory, where the least significant bit represents whether the sequence number is locked or unlocked. Here, the idea is slightly different, since the least significant bit represents whether the descriptor is currently being reused and initialized, or is safe to access. (Nevertheless, in some sense, one can think of the bit indicating whether the descriptor is currently being initialized as a sort of lock. It does not prevent other processes from making progress (since operations on the descriptor will

```
 1    ▷ Data types
 2    Descriptor of type T :
 3       mutables = ⟨seq, mut₁, mut₂, ...⟩                                              ▷ Mutable fields
 4       imm₁, imm₂, ...                                                                ▷ Immutable fields

 6    ▷ Shared variables
 7       D_{T,p}  for each descriptor type T and process p

 9    ▷ ADT operations
10    CreateNew(T, v₁, v₂, ...) by process p :
11       oldseq  :=  D_{T,p}.mutables.seq
12       D_{T,p}.mutables.seq  :=  oldseq + 1
13       for  each  field f in D_{T,p}
14          let  value  be the corresponding value in {v₁, v₂, ...}
15          if  f is immutable  then
16             D_{T,p}.f := value
17          else
18             D_{T,p}.mutables.f := value
19       D_{T,p}.mutables.seq  :=  oldseq + 2
20       return  ⟨p, oldseq + 2⟩

22    ReadField(des, f, dv) :
23       ⟨q, seq⟩ := des
24       if  f is immutable  then
25          result := D_{T,q}.f
26       else
27          result := D_{T,q}.mutables.f
28       if  seq ≠ D_{T,q}.mutables.seq then return  dv
29       return  result

31    ReadImmutables(des) :
32       ⟨q, seq⟩ := des
33       for  each  f in  des
34          if  f is immutable  then  add D_{T,q}.f to result
35       if  seq ≠ D_{T,q}.mutables.seq then return  ⊥
36       return  result

38    WriteField(des, f, value) :
39       ⟨q, seq⟩ := des
40       loop
41          exp := D_{T,q}.mutables
42          if  exp.seq ≠ seq  then  return
43          new := exp
44          new.f := value
45          if  CAS(&D_{T,q}.mutables, exp, new)  then  return

47    CASField(des, f, fexp, fnew) :
48       ⟨q, seq⟩ := des
49       loop
50          exp := D_{T,q}.mutables
51          if  exp.seq ≠ seq  then  return  ⊥
52          if  exp.f ≠ fexp  then  return  exp.f
53          new := exp
54          new.f := fnew
55          if  CAS(&D_{T,q}.mutables, exp, new)  then
56             return  fnew
```

Figure 6 Pseudocode for the *extended weak descriptor* ADT implementation.

terminate, but will simply be invalid), but it does prevent them from accessing fields of the descriptor as they are being changed.)

An invocation of *ReadField*($des, f, default$) by p reads the value v of the mutable or immutable field f from $D_{T,p}$ followed by its sequence number s. If s matches the sequence number in the descriptor pointer *des*, then v is returned. Otherwise, *default* is returned.

ReadImmutables is similar to *ReadField*, except it reads all immutable fields, instead of a single field, and it returns $\perp$ instead of *default*.

An invocation I of *WriteField*($des, f, value$) by p performs a sequence of one or more *attempts*. In each attempt, it reads the contents *old* of *mutables*, including the sequence number s, from $D_{T,p}$, then checks whether s matches the sequence number in the descriptor pointer *des*. If the sequence numbers do not match, then the abstract descriptor represented by *des* is invalid, so I returns without changing f. Otherwise, I uses CAS to try to change $D_{T,p}.mutables$ from *old* to *new*, which is a copy of *old* in which the contents of field f have been changed (locally) to contain *value*. Observe that this CAS will succeed only if the sequence number in $D_{T,p}.mutables$ matches the sequence number in *des*. If the CAS succeeds, then I returns. Otherwise, I performs another attempt.

Note that *WriteField* is less efficient than performing a direct write to memory. However, since mutable fields are used merely to encode the status of an ongoing operation, there are usually very few changes to a descriptor.

CASField is quite similar to *WriteField*. The only differences are (1) *CASField* has different return values and, (2) in each attempt, it performs an additional check to determine whether $old.f$ is equal to *fexp*, and, if not, returns $old.f$.

5.2   Practical considerations

One might wonder, in an algorithm with multiple types of descriptors, why the type of a descriptor is not also encoded in descriptor pointers. In algorithms that use multiple descriptor types, any time the original algorithm accesses a field of a descriptor, it typically must know what kind of descriptor it is accessing (if, for no other reason, to compute the address of the desired field within the descriptor). In such algorithms, it would not be necessary for descriptor pointers to carry this extra information. For algorithms that access descriptors without knowing their exact types, one can include the descriptor type in descriptor pointers.

Some lock-free algorithms "steal" up to three bits from pointers to encode additional information, typically to distinguish between application values and (potentially, various types of) descriptors. To accommodate such algorithms, one can slightly shrink the sequence number in our descriptor pointers, and reserve the three lowest-order bits for use by other algorithms.

One obvious way to store the descriptors for each thread is to create an array for each descriptor type, with a slot containing a descriptor for each process. In this kind of implementation, it is extremely important to pad each slot to avoid false sharing [5]. We suggest allocating at least two cache lines for each descriptor (128 bytes on modern Intel and AMD machines).

To improve efficiency, modern Intel and AMD processors implement a relaxed memory model called total store order (TSO) that allows certain steps in a program to be executed out of order. Specifically, a read that occurs after a write in a program can actually be executed *before* the write, as long as the read and write are not accessing the same address. This can render a concurrent algorithm incorrect if it requires a write by a process p to be visible to other processes *before* p performs a subsequent read. One can prevent this reordering by

placing a memory fence (or barrier) between the write and read. CAS instructions also act as memory fences. Our implementation does not require any memory fences (beyond those implied by CAS instructions). This is an attractive property, since memory fences incur high overhead.

Our implementation uses unbounded sequence numbers. However, in practice, sequence numbers are bounded, and they may wrap around. If wraparound occurs, then two invocations of *CreateNew* might return the same descriptor pointer. This can cause an *ABA problem* if the high-level algorithm that uses descriptors relies on the uniqueness of descriptor pointers returned by *CreateNew*.

We argue that the sequence number can be made sufficiently large on modern systems for this to be a non-issue. Consider a system with a 64-bit word size. Recall that a sequence number appears both in each descriptor pointer, and also in the *mutables* field of each descriptor. A descriptor pointer contains only a process name and a sequence number, so if n bits are reserved for the process name, then $64 - n$ bits remain for the sequence number. The *mutables* field contains the descriptor's mutable fields and a sequence number, so if m bits are reserved for mutable fields, then $64 - m$ bits remain for the sequence number. Thus, if we use 14-bit process names (as the Linux kernel does), and the mutable fields of each descriptor fit in at most 14 bits, then 50 bits remain for the sequence number. We are unaware of any algorithm that requires more than three bits for mutable fields in its descriptors, so this is realistic. In this case, a single process must perform 2^{50} operations to trigger even a single wraparound. If we assume that a single process can perform one million operations per second, this will take 35 years of continuous execution. If this is still a concern, then one can use double-wide CAS (DWCAS), which is implemented on modern Intel and AMD systems, instead of CAS, to atomically operate on two adjacent words (containing a much larger sequence number).

Although we are unaware of any current lock-free algorithms that use more than three bits for mutable fields in descriptors, some future algorithm may use more. If the mutable fields of a descriptor cannot fit in the same word as a sequence number, then our approach must be modified. If the mutable fields and a sequence number can fit in two adjacent words, then one can simply use DWCAS instead of CAS. Otherwise, one can store mutable fields in their own separate words, and *replicate* the sequence number, storing a copy in the word adjacent to each mutable field. To change a mutable field, one would then perform DWCAS on the word containing the mutable field, and its adjacent sequence number. When the descriptor is reused, instead of incrementing a single sequence number, one would increment all sequence numbers.

In order to choose how many bits should be devoted to the process name in descriptor pointers, one must know an upper bound on the number of processes. We stress that this is not an onerous constraint, because the upper bound does not need to be tight. Note that one need not initially allocate descriptors for all processes that *could* be running in the system. It is straightforward to allocate a descriptor for a process the first time it invokes *CreateNew* (potentially even in batches, to amortize the cost and improve control over memory layout).

5.3 Correctness

We now prove that our implementation is linearizable. We first give the linearization points for all operations.

- Each invocation of *CreateNew* is linearized at the increment of the sequence number at line 12.
- If an invocation I of *ReadField*(des, f, dv) returns at line 28, then it is linearized at the read of the sequence number at the same line. If I returns at line 29, then it is linearized at the preceding read of the field f: for immutable fields this is line 25, and for mutable fields this is line 27.
- Each invocation of *ReadImmutables* is linearized at the read of the sequence number at line 35.
- If an invocation I of *WriteField*($des, f, value$) returns at line 42, then it is linearized at the last read of the sequence number at the same line. If I returns at line 45, then it is linearized at the successful CAS at the same line.
- If an invocation I of *CASField*($des, f, fexp, fnew$) returns at line 51, then it is linearized at the last read of the sequence number at the same line. If I returns at line 56, then it is linearized at the successful CAS at the previous line. If I returns at line 52, then it is linearized at the last read at the same line.

▶ **Observation 1.** The sequence number of $D_{T,p}$ (also denoted $D_{T,p}.mutables.seq$) is written only by p in invocations of *CreateNew*($T, -$).

▶ **Observation 2.** Every descriptor pointer returned by *CreateNew* has an even sequence number, and the linearization point of *CreateNew* always changes the sequence number of the descriptor to an odd number.

▶ **Observation 3.** The sequence number returned by a *CreateNew*($T, -$) operation by p is $2 + v$ where v is the sequence number returned by p's previous *CreateNew*($T, -$) operation, or $v = 0$ if p has not performed *CreateNew*($T, -$).

We now prove that the above linearization points are correct. Let e be an execution of our implementation of extended weak descriptors. Let $O_1, O_2 \cdots O_k$ be the extended weak descriptor operations executed in e in the order they are linearized. Note that we prove correctness assuming unbounded sequence numbers. The implications of bounded sequence numbers were considered above.

▶ **Theorem 2.** *The responses of $O_1, O_2 \cdots O_k$ respect the semantics of the extended weak descriptor ADT.*

Proof. By strong induction on the sequence of extended weak descriptor operations that terminate in e. Base case: the claim vacuously holds when no operations have returned. Induction step: assume the return values of $O_1, O_2 \cdots O_{i-1}$ follow the semantics of the extended weak descriptor ADT. Let p be the process that performs O_i, and T be the type of descriptor on which O_i is performed.

Suppose O_i is a *CreateNew*($T, -$) operation. By Observation 3, O_i returns a unique descriptor pointer.

In each of the following cases, O_i takes a descriptor pointer des as one of its arguments. Let q and seq be the process name sequence number in des, respectively. Let O_{init} be the *CreateNew*($T, -$) by q that returned des. Since des is returned by O_{init} before it is passed to any operation, O_{init} is linearized before O_i.

Suppose O_i is a *ReadField* that returns the default value at line 28, a *CASField* that returns $\bot$ at line 51 or a *ReadImmutables* that returns $\bot$ at line 35. We argue that *des* is invalid when O_i is linearized. In each case, O_i returns after seeing that the sequence number of $D_{T,q}$ no longer contains *seq*. Thus, this sequence number must change after *des* is returned by O_{init}, and before O_i is linearized. By Observation 1, this change to the sequence number of $D_{T,q}$ must be performed by a *CreateNew(T, −)* operation O_{change} by q (which occurs after O_{init}, and before O_i is linearized). O_{change} changes the sequence number twice, and is linearized at the first change. Thus, O_{change} is linearized after O_{init} and before O_i, which means that *des* is *invalid* when O_i is linearized.

Now suppose O_i is a *ReadField* that returns at line 29, a *CASField* that returns at line 56, or a *ReadImmutables* that returns at line 36. We first argue that *des* is valid when O_i is linearized. In each case, O_i sees that the sequence number of $D_{T,q}$ is *seq* at some time t, which is either when O_i is linearized, or is after O_i is linearized. By Observation 1 and Observation 3, whenever the sequence number of $D_{T,q}$ is changed, it is changed to a new value that it never previously contained. Thus, since the sequence number of $D_{T,q}$ contains *seq* when O_{init} terminates, and it contains *seq* at time t, it contains *seq* at all times after O_{init} terminates and before t. Hence, the sequence number of $D_{T,q}$ contains *seq* when O_i is linearized. By Observation 1, q does not perform any *CreateNew(T, −)* after O_{init} and before t, so *des* is *valid* when O_i is linearized.

We now argue that the response of O_i is correct if it is a *ReadField(des, f, dv)*. The proof is similar when O_i is a *CASField* or *ReadImmutables*.

If f is immutable, then it is changed only by *CreateNew(T, −)* operations by q. Since *des* is valid when O_i is linearized, O_{init} performs the last change to f before O_i is linearized. Recall that O_i start after O_{init} terminates. Thus, the write of f in O_{init} happens before the invocation of O_i, and O_i will return the value written to f by O_{init}.

If f is mutable, then let O_{change} be the operation that performs the last change to f before O_i is linearized, and v be the value that it stores in f. Observe that O_i returns v. We show that O_{change} is the last operation that changes f and is linearized before O_i. If O_{change} is the same as O_{init}, then we are done. Otherwise, since we have argued that q does not perform any *CreateNew(T, −)* after O_{init} and before t, O_{change} must be a *WriteField* or *CASField*. In each case, O_{change} can change f only once, with a successful CAS (at line 55 or line 45). Since O_{change} is linearized at this CAS, it is linearized before O_i. Moreover, since we have assumed that O_{change} is the last operation to change f before O_i, no other operation that changes f linearized after O_{change} and before O_i.   ◄

5.4   Progress

Suppose, to obtain a contradiction, that there is an execution in which processes take infinitely many steps, but only finitely many (extended weak descriptor) operations terminate. Then, after some time t, no operation terminates, which means there is at least one operation O in which a process takes infinitely many steps. By inspection of Figure 6, O must be a *WriteField* or *CASField* operation. Suppose O is a *WriteField* operation. Then, each time O executes line 42, it sees *old.seq = seq*, and each time it executes line 45, its CAS fails and returns *old* without changing $D_{T,q}.mutables$. Observe that the CAS will fail only if $D_{T,q}.mutables$ changes after it is read at line 41 and before the CAS at line 45. Thus, $D_{T,q}.mutables$ changes infinitely many times in the execution. Since $D_{T,q}.mutables$ can be changed only by *WriteField* or *CASField* operations, and any operation that changes $D_{T,q}.mutables$ terminates, there must be infinitely many terminating *WriteField* or *CASField* operations, which is a contradiction. The proof is similar when O is a *CASField*.

6 Experiments

Our experiments were run on two large-scale systems. The first is a 2-socket Intel E7-4830 v3 with 12 cores per socket and 2 hyperthreads (HTs) per core, for a total of 48 threads. Each core has a private 32KB L1 cache and 256KB L2 cache (which is shared between HTs on a core). All cores on a socket share a 30MB L3 cache. The second is a 4-socket AMD Opteron 6380 with 8 cores per socket and 2 HTs per core, for a total of 64 threads. Each core has a private 16KB L1 data cache and 2MB L2 cache (which is shared between HTs on a core). All cores on a socket share a 6MB L3 cache.

Since both machines have multiple sockets and a non-uniform memory architecture (NUMA), in all of our experiments, we pinned threads to cores so that the first socket is filled first, then the second socket is filled, and so on. Furthermore, within each socket, each core has one thread pinned to it before hyperthreading is engaged. Consequently, our graphs clearly show the effects of hyperthreading and NUMA.

For example, on the Intel machine, from thread counts 1 to 12 all threads are running on a single socket and at most one thread is pinned to each core. **(socket 1: no HTs; socket 2: empty)**. From 13 to 24, all threads are running on a single socket and cores either have one or two threads pinned to them **(socket 1: HTs; socket 2: empty)**. From 25 to 36, each core on the first socket has two threads pinned to it, and the remaining threads are each pinned to unique cores on the second socket **(socket 1: HTs; socket 2: no HTs)**. Finally, from 37 to 48, each core on the first socket has two threads pinned to it, and cores on the second socket have one or two threads pinned to them **(socket 1: HTs; socket 2: HTs)**.

Both machines have 128GB of RAM. Each runs Ubuntu 14.04 LTS. All code was compiled with the GNU C++ compiler (G++) 4.8.4 with build target x86_64-linux-gnu and compilation options `-std=c++0x -mcx16 -O3`. Thread support was provided by the POSIX Threads library. We used the Performance Application Programming Interface (PAPI) library [12] to collect statistics from hardware performance counters to determine cache miss rates, stall times, instructions retired, and so on.

The system (glibc) allocator was found to have poor scaling and overall performance. Instead, we used jemalloc 4.2.1, a fast user-space allocator designed to minimize contention and improve scalability [15]. The library was dynamically linked with `LD_PRELOAD`, which is the recommended method. This allocator was found to yield vastly superior performance for all algorithms, in all benchmarks. We also tried the tcmalloc allocator from Google's Perftools library, which is another common choice for concurrency-friendly allocation. However, performance with tcmalloc was substantially worse for all algorithms than with jemalloc.

On the AMD machine, transparent huge-pages were disabled manually in the jemalloc implementation by changing the default allocation chunk size from 2^{21} to 2^{19} using the environment parameter setting `MALLOC_CONF=lg_chunk:19`. This maintained or improved the performance for all algorithms in all workloads, and did not change the performance relationship between any pair of algorithms. The same change did not improve performance on the Intel machine (for any algorithm or workload), so the original chunk size was used.

For read-heavy workloads, it was necessary to force distribution of pages across NUMA nodes to get consistently high performance. To achieve this, we used `numactl -interleave=all` for all workloads. (Doing this did not negatively impact the performance of any workload, but its benefit was less noticeable for write-heavy workloads.)
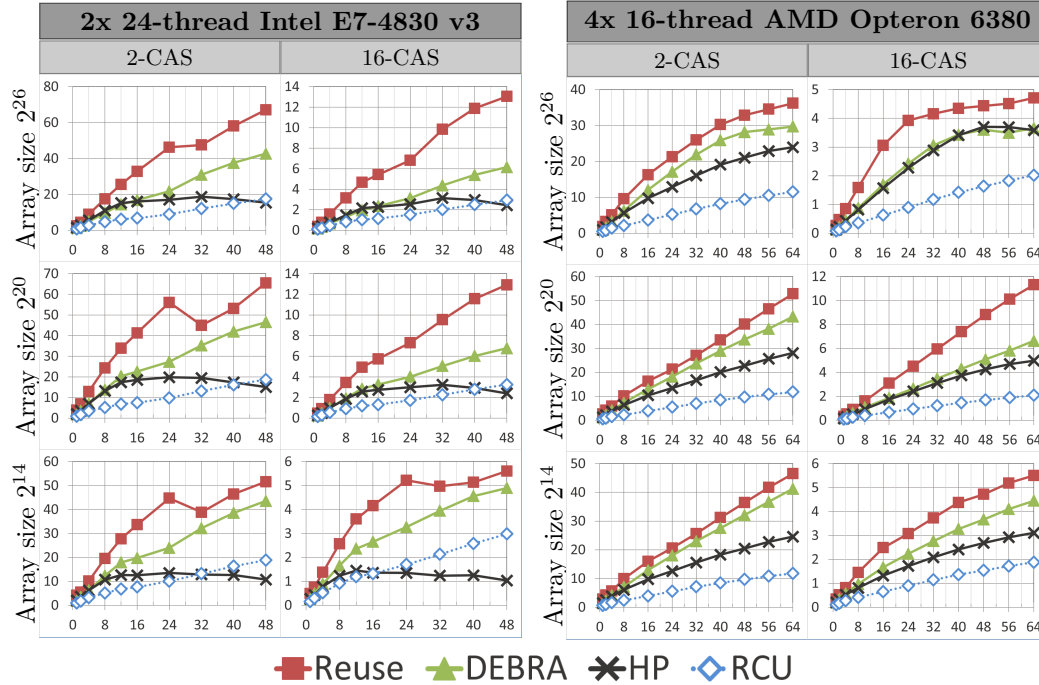
Figure 7 Results for a k-**CAS microbenchmark**. The x-axis represents the number of concurrent threads. The y-axis represents operations per microsecond.

6.1 k-**CAS microbenchmark**

In order to compare our reusable descriptor technique with algorithms that reclaim descriptors, we implemented k-CAS with several memory reclamation schemes. Specifically, we implemented a lock-free memory reclamation scheme that aggressively frees memory called *hazard pointers* [26], a (blocking) epoch-based reclamation scheme called *DEBRA* [7], and reclamation using the read-copy-update (RCU) primitives [13] (also blocking). We use *Reuse* as shorthand for our reusable descriptor based algorithm, and *DEBRA*, *HP* and *RCU* to denote the other algorithms.

The paper by Harris et al. also describes an optimization to reduce the number of DCSS descriptors that are allocated by embedding them in the k-CAS descriptor. We applied this optimization, and found that it did not significantly improve performance. Furthermore, it complicated reclamation with hazard pointers. Thus, we did not use this optimization.

Methodology.   We compared our implementations of k-CAS using a simple array-based microbenchmark. For each algorithm $A \in \{Reuse, DEBRA, HP, RCU\}$, array size $S \in \{2^{14}, 2^{20}, 2^{26}\}$ and k-CAS parameter $k \in \{2, 16\}$, we run ten timed *trials* for several thread counts n. In each trial, an array of a fixed size S is allocated and each entry is initialized to zero. Then, n concurrent threads run for one second, during which each thread repeatedly chooses k uniformly random locations in the array, reads those locations, and then performs a k-CAS (using algorithm A) to increment each location by one.

As a way of validating correctness in each trial, each thread keeps track of how many successful k-CAS operations it performs. At the end of the trial, the sum of entries in the array must be k times the total number of successful k-CAS operations over all threads.

Results.   The results for this benchmark appear in Figure 7. Error bars are not drawn on the graphs, since more than 97% of the data points have a standard deviation that is less

than 5% of the mean (making them essentially too small to see).

Overall, *Reuse* outperforms every other algorithm, in every workload, on both machines. Notably, on the Intel machine, its throughput is *2.2 times* that of the next best algorithm at 48 threads with $k = 16$ and array size 2^{26}. On the AMD machine, its throughput is 1.7 times that of the next best algorithm at 64 threads with $k = 16$ and array size 2^{20}.

On the Intel machine, with $k = 2$, NUMA effects are quite noticeable for *Reuse* in the jump from 24 to 32 threads, as threads begin running on the second socket. According the statistics we collected with PAPI, this decrease in performance corresponds to an increase in cache misses. For example, with $k = 2$ and an array of size 2^{26} in the Intel machine, jumping from 24 threads to 25 increases the number of L3 cache misses per operation from 0.7 to 1.6 (with similar increases in L1 and L2 cache misses and pipeline stalls). We believe this is due to cross-socket cache invalidations.

From the three graphs for $k = 2$ on Intel, we can see that the effect is more severe with larger absolute throughput (since the additive overhead of a cache miss is more significant). Conversely, the effect is masked by the much smaller throughput of the slower algorithms, and by the substantially lower throughputs in the $k = 16$ case, except when the array is of size 2^{14}. In the array of size 2^{14}, contention is extremely high, since each of the 48 threads are accessing 16 k-CAS addresses, each of which causes contention on the entire cache line of 8 words, for a total of 6144 array entries contended at any given time. Thus, cache misses become a dominating factor in the performance on two sockets. These effects were not observed on the AMD machine. There, the number of cache misses is not significantly different when crossing socket boundaries, which suggests a robustness to NUMA effects that is not seen on the Intel machine.

Interestingly, absolute throughputs on the AMD machine are larger with array size 2^{20} than with sizes 2^{14} and 2^{26}. This is because the 2^{20} array size represents a sweet spot with less contention than the 2^{14} size and better cache utilization than the 2^{26} size. For example, with 64 threads and $k = 16$, *Reuse* incurred approximately 50% more cache misses with size 2^{26} than with size 2^{20}, and approximately 50% of operations helped one another with size 2^{14}, whereas less than 1% of operations helped one another with size 2^{20}.

Note, however, that this is not true on the Intel machine. There, 2^{26} is almost always as fast as 2^{20}, because of the very large shared L3 cache (which is 5x larger than on the AMD machine). This is reflected in the increased number of cycles where the processor is stalled (e.g., waiting for cache misses to be served) when moving from size 2^{20} to 2^{26}. On the Intel machine, stalled cycles increase by 85% per operation, whereas on the AMD machine they increase by a whopping 450% per operation.

6.1.1   Memory usage in the k-CAS benchmark

We studied memory usage for all algorithms, in all workloads, on both systems, but we only show results for array size 2^{26} and $k = 16$, because the other graphs are very similar. These results appear in Figure 8. In particular, we are interested in the descriptor footprint, i.e., the maximum amount of memory ever occupied by descriptors in an execution. Unfortunately, computing the descriptor footprint exactly would require excessive synchronization between threads. Thus, we approximate the descriptor footprint by computing the descriptor footprint *for each thread*, and then summing those individual footprints. (This is only an approximation, since different threads may hit their peak memory usage for descriptors at different times.) The graph in Figure 8 contains the results of this approximation.

These results were obtained as follows. Each thread used three private variables: *totalFree*, *totalMalloc* and *maxFootprint*. Each time a thread invoked `free`, it incremented
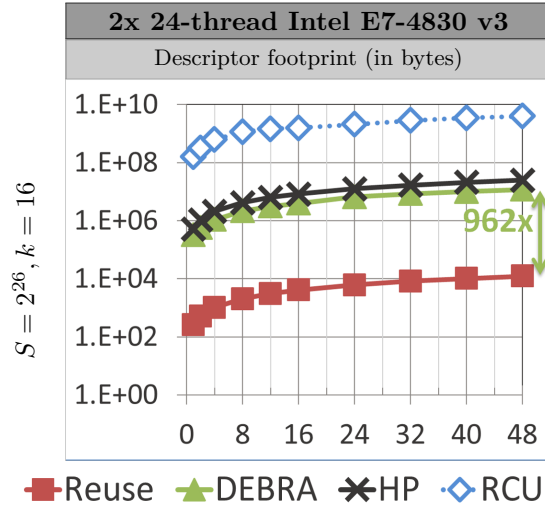
Figure 8 Memory usage for the k-**CAS microbenchmark**. The x-axis represents the number of concurrent threads. *Note the logarithmic scale.*

totalFree by the size of the descriptor being freed. Each time a thread invoked `malloc`, it incremented *totalMalloc* by the size of the descriptor being allocated, and then set $maxFootprint = max\{maxFootprint, totalMalloc - totalFree\}$. The per-thread *maxFootprint*s are then summed to obtain the data points in the graph.

Note that the y-axis is a logarithmic scale. The results show that *DEBRA* and *HPs* use almost **three orders of magnitude** more memory than *Reuse* at their peaks, and *RCU* uses nearly three orders of magnitude more memory than *DEBRA* and *HPs*. *RCU*'s memory usage is significantly higher because reclamation is delayed significantly longer than in the other algorithms.

6.2 BST microbenchmark

Unlike in the k-CAS algorithm, where memory reclamation was only needed for descriptors, in the BST, memory reclamation is always needed for nodes. To compare our technique with different memory reclamation options, we implemented four variants of the BST algorithm: *DEBRA/DEBRA*, *DEBRA/Reuse*, *RCU/RCU* and *RCU/Reuse*. Here, an algorithm named X/Y uses X to reclaim nodes and Y for descriptors. For example, *DEBRA/Reuse* uses DEBRA to reclaim nodes and has reusable descriptors.

Unfortunately, we could not create a variant of the BST using hazard pointers. As part of the *finalizing* mechanism, this BST implementation *marks* nodes before deleting them. Furthermore searches are allowed to traverse marked nodes, regardless of whether they have been deleted, and subsequently succeed. These algorithmic properties make it infeasible to use hazard pointers [7].

Methodology. We compared our BST variants using a simple randomized microbenchmark. For each algorithm $A \in \{DEBRA/DEBRA, DEBRA/Reuse, RCU/RCU, RCU/Reuse\}$, key range size $K \in \{10^5, 10^6\}$ and update rate $U \in \{100, 0\}$, we run ten timed *trials* for several thread counts n. Each trial proceeds in two phases: *prefilling* and *measuring*. In the prefilling phase, n concurrent threads perform 50% *Insert* and 50% *Delete* operations on keys drawn uniformly randomly from $[0, K)$ until the size of the tree converges to a steady state (containing approximately $K/2$ keys). Next, the trial enters the measuring phase,
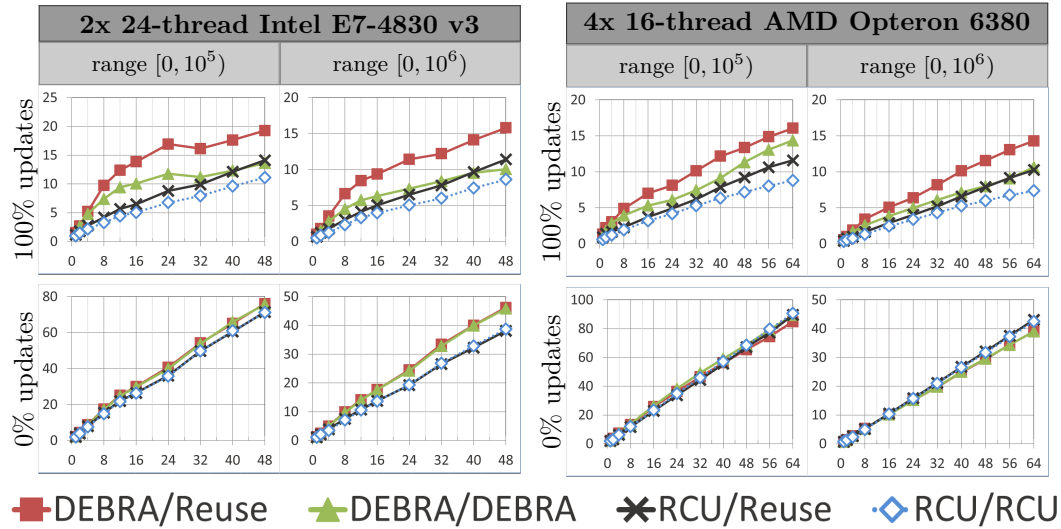
Figure 9 Results for a **BST microbenchmark**. The x-axis represents the number of concurrent threads. The y-axis represents operations per microsecond.

during which threads begin counting how many operations they perform. (These counts are eventually summed over all threads and reported in our graphs.) In this phase, each thread instead performs $(U/2)\%$ *Insert*, $(U/2)\%$ *Delete* and $(100 - U)\%$ *Find* operations on keys drawn uniformly from $[0, K)$ for one second.

As a way of validating correctness in each trial, each thread maintains a *checksum*. Each time a thread inserts a new key, it adds the key to its checksum. Each time a thread deletes a key, it subtracts the key from its checksum. At the end of the trial, the sum of all thread checksums must be equal to the sum of keys in the tree.

Results. The results for this benchmark appear in Figure 9. The *Reuse* variants perform at least as well as the pure reclamation variants in every case, and significantly outperform the reclamation variants in the 100% update workload. Most notably, on the Intel machine with key range $[0, 10^6]$ and 48 threads, *DEBRA/Reuse* outperforms *DEBRA/DEBRA* by 57%, and *RCU/Reuse* outperforms *RCU/RCU* by 33%. As expected, *Reuse* does not perform significantly faster than the reclamation variants in the workloads with no updates. This is because searches do not create descriptors. However, crucially, our transformation does not impose any overhead on searches, either.

6.3   Studying sequence number wraparound

We performed experiments on the larger AMD machine to study how frequently errors occur when sequence numbers of varying bit-widths experience wraparound. For each bit-width $B \in \{2, 3, 4, ..., 48\}$, we performed 200 trials in which 64 threads run for 100 milliseconds before terminating. Each trial was the same as a trial in our BST experiments with 100% updates and key range $[0, 10^5)$.

We identified three different types of errors in these trials. First, at the end of a trial, the sum of the checksums maintained by all threads would fail to match the sum of keys in the tree. Second, threads would enter infinite loops due to the tree structure being corrupted, e.g., because a cycle was introduced. (We identified this type of error by waiting until some thread had run twice as long as it should have.) Third, an invalid memory access would

Bits	E[time until error]
16	4.5 hours
24	116 years
32	26078192 years
48	$> 10^{18}$ years
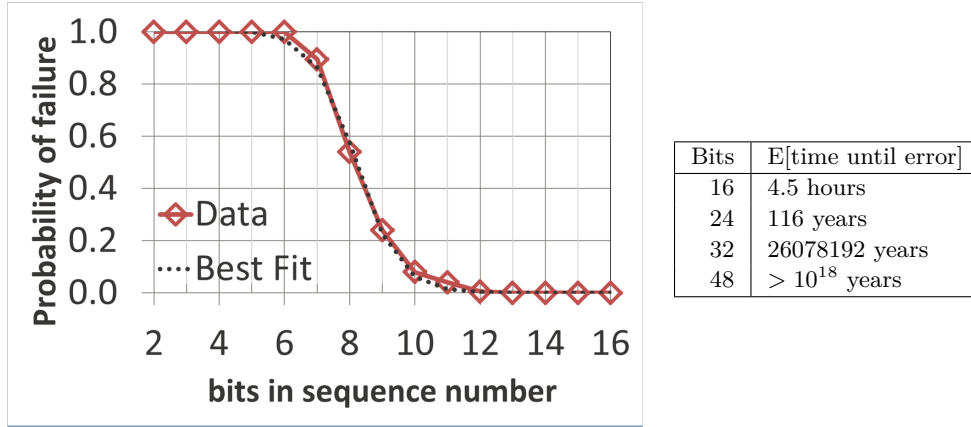
Figure 10 Experiment studying sequence number wraparound.

cause immediate program failure (e.g., due to segmentation fault or bus error).

For each B value, we divided the number of failed runs by 200 to estimate the probability of a trial failing. A graph showing the resulting estimated probability distribution appears in Figure 10. For small B values, trials frequently experienced errors. However, for $B \geq 13$, we did not observe a single error in 200 trials (despite the fact that wraparound consistently occurred in every trial). For $B \geq 16$, trials were not sufficiently long for wraparound to consistently occur. The results appear in Figure 10.

As is common in physics when studying unknown functions, we make an educated guess that the distribution is sigmoidal, which means it is of the form $f(x) = a/(1 + e^{-b(x-c)})$ for constants a, b and c. We determined a sigmoidal curve of best fit from the data, obtaining the function $f(x) = 1/(1 + e^{1.53969(x-8.199181)})$, which is plotted as the *Best Fit* curve on the graph in Figure 10. As the graph shows, the error between the best fit curve and the measured data is extremely small. Although we do not have a justification for the shape of this distribution, we think it is worthwhile to put forth a hypothesis and study its consequences.

We used $f(x)$ to extrapolate on the data to estimate the expected time until an error occurs in this workload for several bit-widths that would be impractical to test experimentally. These extrapolations appear in the table on the right of Figure 10. They should be taken with a grain of salt, since the error in our estimation likely grows quickly with B. However, the extrapolations suggest that even $B = 32$ would be quite safe for this workload. To our knowledge, this kind of experimental exploration of the practicality of unbounded sequence numbers has not previously been done.

7 Related Work

Several papers have presented universal constructions or strong primitives for non-blocking algorithms in which operations create descriptors [20, 2, 1, 27, 17, 23, 21, 24, 3, 11]. A subset of these algorithms employ ad-hoc techniques for reusing descriptors [20, 2, 1, 27, 24, 23, 21]. The rest assume descriptors will be allocated for each operation and eventually reclaimed.

Most of the ad-hoc techniques for reusing descriptors have significant downsides. Some are complex and tightly integrated into the underlying algorithm, or rely on highly specific algorithmic properties (e.g., that descriptors contain only a single word). Others use synchronization primitives that atomically operate on large words, which are not available on modern systems, and are inefficient when implemented in software. Yet others introduce

high space overhead (e.g., by attaching a sequence number to *every* memory word). Some techniques also incur significant runtime overhead (e.g., by invoking expensive synchronization primitives just to *read fields* of a descriptor). Furthermore, these techniques give, at best, a vague idea of how one might reuse descriptors for arbitrary algorithms, and it would be difficult to determine how to use them in practice. Our work avoids all of these downsides, and provides a concrete approach for transforming a large class of algorithms.

Barnes [4] introduced a technique for producing non-blocking algorithms that can be more efficient (and sometimes simpler) than the universal constructions described above. With Barnes' technique, each operation creates a new descriptor. Creating a new descriptor for each operation allows his technique to avoid the ABA problem while remaining conceptually simple. Each operation conceptually locks each location it will modify by installing a pointer to its descriptor, and then performs it modifications and unlocks each location. Barnes' technique is the inspiration for the class WCA. Many algorithms have since been introduced using variants of this technique [17, 14, 3, 19, 29, 11, 10]. Several of these algorithms are quite efficient in practice despite the overhead of creating and reclaiming descriptors. Our technique can significantly improve the space and time overhead of such algorithms.

Recent work has identified ways to use hardware transactional memory (HTM) to reduce descriptor allocation [9, 22]. Currently, HTM is supported only on recent Intel and IBM processors. Other architectures, such as AMD, SPARC and ARM have not yet developed HTM support. Thus, it is important to provide solutions for systems with no HTM support. Additionally, even with HTM support, our approach is useful. Current (and likely future) implementations of HTM offer no progress guarantees, so one must provide a lock-free fallback path to guarantee lock-free progress. The techniques in [9, 22] accelerate the HTM-based code path(s), but do nothing to reduce descriptor allocations on the fallback path. In some workloads, many operations run on the fallback path, so it is important for it to be efficient. Our work provides a way to accelerate the fallback path, and is orthogonal to work that optimizes the fast path.

The *long-lived renaming* (LLR) problem is related to our work (see [6] for a survey), but its solutions do not solve our problem. LLR provides processes with operations to *acquire* one unique resource from a pool of resources, and subsequently *release* it. One could imagine a scheme in which processes use LLR to reuse a small set of descriptors by invoking *acquire* instead of allocating a new descriptor, and eventually invoking *release*. Note, however, that a descriptor can safely be released only once it can no longer be accessed by any other process. Determining when it is safe to release a descriptor is as hard as performing general memory reclamation, and would also require delaying the release (and subsequent acquisition) of a descriptor (which would increase the number of descriptors needed). In contrast, our weak descriptors eliminate the need for memory reclamation, and allow immediate reuse.

8   Conclusion

We presented a novel technique for transforming algorithms that throw away descriptors into algorithms that reuse descriptors. Our experiments show that our transformation yields significant performance improvements for a lock-free *k*-CAS algorithm. Furthermore, our transformation reduces peak memory usage by nearly three orders of magnitude over the next best implementation.

We also applied our transformation to a lock-free implementation of *LLX* and *SCX*, and studied its performance by doing rigorous experiments on a lock-free binary search tree that

uses *LLX* and *SCX*. These experiments demonstrated a significant performance advantage for our transformed algorithm in workloads that perform many updates. Our transformed *LLX* and *SCX* algorithm has the potential to improve the performance of many algorithms that use *LLX* and *SCX*.

We believe our transformation can be used to improve the performance and memory usage of many other algorithms that throw away descriptors. Moreover, we hope that our extended weak descriptor ADT will aid in the design of more efficient, complex algorithms, by allowing algorithm designers to benefit from the conceptual simplicity of throwing away descriptors without paying the practical costs of doing so.

Acknowledgments

References

1 Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast. STOC '95, pages 538–547.

2 James H. Anderson and Mark Moir. Universal constructions for multi-object operations. PODC '95, pages 184–193.

3 Hagit Attiya and Eshcar Hillel. Highly concurrent multi-word synchronization. *Theoretical Computer Science*, 412(12):1243–1262, 2011.

4 Greg Barnes. A method for implementing lock-free shared-data structures. SPAA '93, pages 261–270.

5 William J. Bolosky and Michael L. Scott. False sharing and its effect on shared memory performance. SEDMS '93, pages 57–71.

6 Alex Brodsky, Faith Ellen, and Philipp Woelfel. Fully-adaptive algorithms for long-lived renaming. *Distributed Computing*, 24(2):119, 2011.

7 Trevor Brown. Reclaiming memory for lock-free data structures. PODC '15, pages 261–270.

8 Trevor Brown. *Techniques for Constructing Efficient Data Structures*. PhD thesis, University of Toronto, 2017.

9 Trevor Brown. A template for implementing fast lock-free trees using HTM. PODC '17, pages 293–302, 2017.

10 Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. PPoPP '14, pages 329–342. Full version available from `http://tbrown.pro`.

11 Trevor Brown, Faith Ellen, and Eric Ruppert. Pragmatic primitives for non-blocking data structures. PODC '13, pages 13–22. Full version available from `http://tbrown.pro`.

12 Shirley Browne, Jack Dongarra, Nathan Garner, George Ho, and Philip Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

13 Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais, and Jonathan Walpole. User-level implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems*, 23(2):375–382, 2012.

14 Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. PODC '10, pages 131–140.

15   Jason Evans. A scalable concurrent malloc (3) implementation for freebsd. In *Proc. of the BSDCan Conference, Ottawa, Canada*, 2006.

16   Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. DISC '01, pages 300–314.

17   Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. DISC '02, pages 265–279.

18   Meng He and Mengdu Li. Deletion without rebalancing in non-blocking binary search trees. OPODIS '16.

19   Shane V. Howley and Jeremy Jones. A non-blocking internal binary search tree. SPAA '12, pages 161–171.

20   Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. PODC '94, pages 151–160.

21   Prasad Jayanti and Srdjan Petrovic. Efficiently implementing a large number of LL/SC objects. OPODIS'05, pages 17–31.

22   Yujie Liu, Tingzhe Zhou, and Michael Spear. Transactional acceleration of concurrent data structures. SPAA '15, pages 244–253, New York, NY, USA, 2015. ACM.

23   Victor Luchangco, Mark Moir, and Nir Shavit. Nonblocking k-compare-single-swap. *Theory of Computing Systems*, 44(1):39–66, January 2009.

24   Virendra Jayant Marathe and Mark Moir. Toward high performance nonblocking software transactional memory. PPoPP '08, pages 227–236, New York, NY, USA, 2008. ACM.

25   Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. SPAA '02, pages 73–82.

26   Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, June 2004.

27   Mark Moir. Practical implementations of non-blocking synchronization primitives. PODC '97, pages 219–228.

28   Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. PPoPP '14, pages 317–328.

29   Niloufar Shafiei. Non-blocking patricia tries with replace operations. ICDCS '13, pages 216–225.

30   John D. Valois. Lock-free linked lists using compare-and-swap. PODC '95, pages 214–222.