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.

1998 ACM Subject Classification D.1.3 [Programming Techniques]: Concurrent Programming

Keywords and phrases Concurrency, data structures, lock-free, synchronization, descriptors

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.4

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., [27, 13, 22, 25]), 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 [12, 16, 26, 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 situ-

* Full paper submitted to Arxiv. It is also available at: `http://tbrown.pro`

ations 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).

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 *CAS-Field* 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 [23] and DEBRA+ [6] 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., [9]), 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 $\perp$. 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 $\perp$ 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 3.1). 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 demonstrate our approach by transforming a wasteful implementation of a double-compare-single-swap (DCSS) primitive [14].

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 demonstrate our approach by transforming a wasteful implementation of a k-compare-and-swap (k-CAS) primitive [14]. In the full paper, we also transform the LLX and SCX primitives of Brown et al. [9], and provide proofs for all of our transformations. 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, 7, 15].

We use mostly known techniques to produce an efficient, provably correct implementation of our extended weak descriptor ADT. 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. This entirely eliminates dynamic allocation *and* memory reclamation for descriptors, 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 5). 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 [6], hazard pointers [23] and read-copy-update (RCU) [11] by up to 2.3x, 3.3x and 5.0x, respectively.

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.

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 [12], but in Harris' implementation of multi-word CAS from single-word CAS, high-level operations temporarily replace application values with pointers to descriptors [14].

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*. We require the immutable descriptor ADT operations to be lock-free, so they can be used to implement lock-free data structures.

Example Algorithm: DCSS. We use the double-compare single-swap (*DCSS*) algorithm of Harris et al. [14] 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.

```
1  $DCSS(a_1, e_1, a_2, e_2, n_2)$ :
2      $des := CreateNew(DCSSdes, a_1, e_1, a_2, e_2, n_2)$
3      $fdes := flag(des)$
4      loop
5          $r := CAS(a_2, e_2, fdes)$
6          if $r$ is flagged then $DCSSHelp(r)$
7          else exit loop
8      if $r = e_2$ then $DCSSHelp(fdes)$
9      return $r$

11 $DCSSRead(addr)$ :
12     loop
13         $r := *addr$
14         if $r$ is flagged then $DCSSHelp(r)$
15         else exit loop
16     return $r$
```

```
17 type $DCSSdes$ :
       $\{\mathrm{ADDR}_1, \mathrm{EXP}_1, \mathrm{ADDR}_2, \mathrm{EXP}_2, \mathrm{NEW}_2\}$

21 $DCSSHelp(fdes)$ :
22     $des := unflag(fdes)$
23     $a_1 := ReadField(des, \mathrm{ADDR}_1)$
24     $a_2 := ReadField(des, \mathrm{ADDR}_2)$
25     $e_1 := ReadField(des, \mathrm{EXP}_1)$
26     if $*a_1 = e_1$ then
27         $n_2 := ReadField(des, \mathrm{NEW}_2)$
28         $CAS(a_2, fdes, n_2)$
29     else
30         $e_2 := ReadField(des, \mathrm{EXP}_2)$
31         $CAS(a_2, fdes, e_2)$
```

Figure 1 Code for the *DCSS* algorithm of Harris et al. [14] using 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 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 typically 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.

Example Algorithm: k-CAS. A $k\text{-}CAS(a_1, ..., a_k,\ e_1, ..., e_k,\ n_1, ..., n_k)$ 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. [14] is an example of a lock-free algorithm that has descriptors with mutable fields. At a high level, a k-CAS operation O 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.)

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) 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 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. Second, we replace the read of $addr_1$ in DCSS 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 [14].

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 (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 and transformation 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.)

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.

3.1 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.)

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. Details appear in the full paper.

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 3.1. 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 $\perp$ is returned. However, for *ReadField*s performed outside of *Help*, it is not clear, in general, how we should respond if $\perp$ 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 $\perp$ 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 $\perp$.

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*$(des, f, \perp)$ 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 $\mathcal{A}$.

Multiple descriptors per operation attempt. In some lock-free algorithms, an operation can create several different types of descriptors, and invoke different *Help* procedures. 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. To support such algorithms, we make the following minor changes. We redefine *CreateNew* so it only invalidates previous descriptors of the *same type*. We also update Property 1 as follows: Each high-level operation attempt O by a process p may create 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 may not invoke *Help*(d) outside of O). Details appear in the full paper.

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. 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, 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 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*.)

Example Algorithm: LLX and SCX. In the full paper, we also transform a wasteful implementation of the LLX and SCX primitives of Brown et al. [9].

Implementing the extended weak descriptor ADT. We give a brief high-level overview, here. Details appear in the full paper. It uses largely known techniques (similar to [21]), 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 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.

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 to collect statistics from hardware counters to determine cache miss rates, stall times, etc. We used the scalable allocator jemalloc 4.2.1, which greatly improved performance.

***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* [23], a (blocking) epoch-based reclamation scheme called *DEBRA* [6], and reclamation using the read-copy-update (RCU) primitives [11] (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.

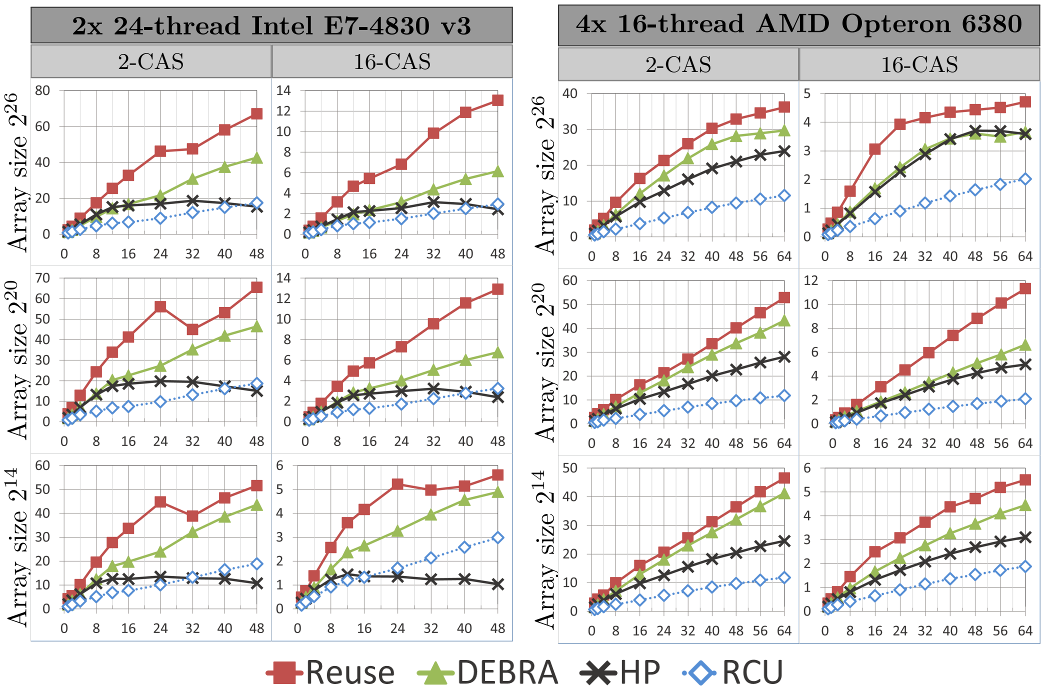

Figure 2 Results for a k**-CAS microbenchmark**. The x-axis represents the number of concurrent threads. The y-axis represents operations per microsecond.

Results. The results for this benchmark appear in Figure 2. 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.

Several additional experiments appear in the full paper, including empirical studies of memory usage, and of the performance of a transformed LLX and SCX implementation.

6 Related Work

Several papers have presented universal constructions or strong primitives for non-blocking algorithms in which operations create descriptors [17, 2, 1, 24, 14, 20, 18, 21, 3, 9]. A subset of these algorithms employ ad-hoc techniques for reusing descriptors [17, 2, 1, 24, 21, 20, 18]. 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 [14, 12, 3, 16, 26, 9, 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 [8, 19]. 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 [8, 19] 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 [5] 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.

7 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 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

We thank Faith Ellen for her gracious help in proving correctness for our transformations, and her insightful comments. Some of this work was done while Trevor was a student at the University of Toronto, and while Maya was visiting him there. This work was supported by the Israel Science Foundation (grant 1749/14), the Natural Sciences and Engineering Research Council of Canada, and Global Affairs Canada. Maya Arbel-Raviv is supported in part by the Technion Hasso Platner Institute Research School.

References

1 Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast. In *Proceedings of STOC '95*, pages 538–547, 1995.

2 James H. Anderson and Mark Moir. Universal constructions for multi-object operations. In *Proceedings of PODC '95*, pages 184–193, 1995.

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. In *Proceedings of SPAA '93*, pages 261–270, 1993.

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

6 Trevor Brown. Reclaiming memory for lock-free data structures. In *Proceedings of PODC '15*, pages 261–270, 2015.

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

8 Trevor Brown. A template for implementing fast lock-free trees using HTM. In *Proceedings of PODC '17*, pages 293–302, 2017.

9 Trevor Brown, Faith Ellen, and Eric Ruppert. Pragmatic primitives for non-blocking data structures. In *Proceedings of PODC '13*, pages 13–22, 2013.

10 Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *Proceedings of PPoPP '14*, pages 329–342, 2014.

11 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.

12 Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of PODC '10*, pages 131–140, 2010.

13 Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of DISC '01*, pages 300–314, 2001.

14 Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of DISC '02*, pages 265–279, 2002.

15 Meng He and Mengdu Li. Deletion without rebalancing in non-blocking binary search trees. In *Proceedings of OPODIS '16*, pages 34:1–34:17, 2017.

16 Shane V. Howley and Jeremy Jones. A non-blocking internal binary search tree. In *Proceedings of SPAA '12*, pages 161–171, 2012.

17 Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of PODC '94*, pages 151–160, 1994.

18 Prasad Jayanti and Srdjan Petrovic. Efficiently implementing a large number of LL/SC objects. In *Proceedings of OPODIS'05*, pages 17–31, 2005.

19 Yujie Liu, Tingzhe Zhou, and Michael Spear. Transactional acceleration of concurrent data structures. In *Proceedings of SPAA '15*, pages 244–253, 2015.

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

21 Virendra Jayant Marathe and Mark Moir. Toward high performance nonblocking software transactional memory. In *Proceedings of PPoPP '08*, pages 227–236, 2008.

22 Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of SPAA '02*, pages 73–82, 2002.

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

24 Mark Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of PODC '97*, pages 219–228, 1997.

25 Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of PPoPP '14*, pages 317–328, 2014.

26 Niloufar Shafiei. Non-blocking patricia tries with replace operations. In *Proceedings of ICDCS '13*, pages 216–225, 2013.

27 John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of PODC '95*, pages 214–222, 1995.