The ABA Problem in Multicore Data Structures with Collaborating Operations

Damian Dechev
Department of Electrical Engineering and Computer Science
University of Central Florida
Orlando, FL 32816
email: dechev@eecs.ucf.edu

***Abstract*—An increasing number of modern real-time systems and the nowadays ubiquitous multicore architectures demand the application of programming techniques for reliable and efficient concurrent synchronization. Some recently developed Compare-And-Swap (CAS) based nonblocking techniques hold the promise of delivering practical and safer concurrency. The ABA[2] problem is a fundamental problem to many CAS-based designs. Its significance has increased with the suggested use of CAS as a core atomic primitive for the implementation of portable lock-free algorithms. The ABA problem's occurrence is due to the intricate and complex interactions of the application's concurrent operations and, if not remedied, ABA can significantly corrupt the semantics of a nonblocking algorithm. The current state of the art leaves the elimination of the ABA hazards to the ingenuity of the software designer. In this work we provide the first systematic and detailed analysis of the ABA problem in lock-free Descriptor-based designs. We study the semantics of Descriptor-based lock-free data structures and propose a classification of their operations that helps us better understand the ABA problem and subsequently derive an effective ABA prevention scheme. We supplement our analysis with a statistical model of the probability for an ABA event in a concurrent system. Our ABA prevention approach outperforms by a large factor the use of the alternative CAS-based ABA prevention schemes. It offers speeds comparable to the use of the architecture-specific CAS2 instruction used for version counting. We demonstrate our ABA prevention scheme by integrating it into an advanced nonblocking data structure, a lock-free dynamically resizable array.**

***Index Terms*—nonblocking synchronization, collaborating threads, multicore computing**

I. Introduction

The modern ubiquitous multi-core architectures demand the design of programming libraries and tools that allow fast and reliable concurrency. In addition, providing safe and efficient concurrent synchronization is of critical importance to the engineering of many modern real-time systems. Lock-free programming techniques [15] have been demonstrated to be effective in delivering performance gains and preventing some hazards, typically associated with the application of mutual exclusion, such as deadlock, livelock, and priority inversion [8], [3]. As explained by Herlihy [15], a concurrent object is *nonblocking* if it guarantees that *some* process in the system will make progress in a *finite* number of steps. An object that guarantees that *each* process will make progress in a *finite* number of steps is defined as *wait-free* [15]. Lock-free algorithms exploit a set of portable atomic primitives such as the word-size Compare-and-Swap (CAS) instruction [9]. The design of nonblocking data structures poses significant challenges and their development and optimization is a current topic of research [8], [15]. To ease the implementation complexity, some approaches suggest the application of uncommon hardware primitives such as Load-Link/Store-Conditional (LL/SC) or a Double-CAS (DCAS) [5]. Most commonly, to provide a portable and practical nonblocking design, developers rely solely on the use of widely available hardware primitives such as the single-word CAS. The ABA problem [4] is a fundamental problem to many CAS-based nonblocking designs and its occurrence can seriously corrupt the semantics of a nonblocking algorithm [9], [21], [3]. While of a simple nature and derived from the application of a basic hardware primitive, the ABA problem's occurrence is due to the intricate and complex interactions of the application's concurrent operations. The importance of the ABA problem has been reiterated in the recent years with the application of CAS for the development of nonblocking programming techniques.

Avoiding the hazards of ABA imposes an extra challenge for a lock-free algorithm's design and implementation. To the best of our knowledge, the literature does not offer an explicit and detailed analysis of the ABA problem, its relation to the most commonly applied nonblocking programming techniques (such as the use of Descriptors [3]) and correctness guarantees, and the possibilities for its avoidance. Thus, at the present moment of time, eliminating the hazards of ABA in a nonblocking algorithm is left to the ingenuity of the software designer. In this work we study in details and define the conditions that lead to ABA in a nonblocking Descriptor-based design. Based on our analysis, we define a generic and practical technique, called the $\lambda\delta$ approach, for ABA avoidance for a lock-free Descriptor-based linearizable design (Section IV). We demonstrate the application of our approach by incorporating it in a complex and advanced nonblocking data structure, a lock-free dynamically resizable array (vector) [3]. The ISO C++ Standard Template Library [25] vector offers a combination of dynamic memory management and constant-time random access. We survey the literature for other known ABA prevention techniques (usually described

[2]ABA is not an acronym and is a shortcut for stating that a value at a shared location can change from A to B and then back to A

COLLABORATECOM 2011, October 15-18, Orlando, United States

DOI 10.4108/collaboratecom.2011.62

as a part of a nonblocking algorithm's implementation) and study in detail three known solutions to the ABA problem (Sections II-A and II-C). Our work discusses the assumed and desired semantics of a nonblocking container (Section III). In addition, we present the first statistical model describing the practical probability for an ABA event (Section II-D). Our performance evaluation (Section V) establishes that the single-word CAS-based $\lambda\delta$ approach is fast, efficient, and practical. The $\lambda\delta$ approach outperforms by a large factor the application of garbage collection for the safe management of each shared location (discussed in Section II-C) and offers speed of execution comparable to the direct application of the architecture-specific CAS2 instruction used for reference counting.

II. The ABA Problem

The Compare-And-Swap (CAS) atomic primitive (commonly known as Compare and Exchange, CMPXCHG, on the Intel *x86* and *Itanium* architectures [16]) is a CPU instruction that allows a processor to atomically test and modify a single-word memory location. CAS requires three arguments: a memory location (L_i), an old value (A_i), and a new value (B_i). The instruction atomically exchanges the value stored at L_i with B_i, provided that L_i's current value equals A_i. The result indicates whether the exchange was performed. For the majority of implementations the return value is the value last read from L_i (that is B_i if the exchange succeeded). Some CAS variants, often called Compare-And-Set, have a return value of type boolean. The hardware architecture ensures the atomicity of the operation by applying a fine-grained hardware lock such as a cache or a bus lock (as is the case for IA-32 [16]). The application of a CAS-controlled speculative manipulation of a shared location (L_i) is a fundamental programming technique in the engineering of nonblocking algorithms [15] (an example is shown in Algorithm 1).

Algorithm 1 CAS-based speculative manipulation of L_i

```
1: repeat
2:     value_type A_i=^L_i
3:     value_type B_i = fComputeB
4: until CAS(L_i, A_i, B_i) == Bi
```

In our pseudocode we use the symbols ^, &, and . to indicate pointer dereferencing, obtaining an object's address, and integrated pointer dereferencing and field access. When the value stored at L_i is the target value of a CAS-based speculative manipulation, we call L_i and $\hat{}L_i$ *control location* and *control value*, respectively. We indicate the control value's type with the string value_type. The size of value_type must be equal or less than the maximum number of bits that a hardware CAS instruction can exchange atomically (typically the size of a single memory word). In the most common cases, value_type is either an integer or a pointer value. In the latter case, the implementor might reserve two extra bits per each control value and use them for implementation-specific value marking [8]. This is possible if we assume that the pointer values stored at L_i are aligned and the two low-order bits have been cleared during the initialization. In Algorithm 1, the function fComputeB yields the new value, B_i, to be stored at L_i.

Definition 1. *The ABA problem is a false positive execution of a CAS-based speculation on a shared location L_i.*

As illustrated in Table I, ABA can occur if a process P_1 is interrupted at any time after it has read the old value (A_i) and before it attempts to execute the CAS instruction from Algorithm 1. An interrupting process (P_k) might change the value at L_i to B_i. Afterwards, either P_k or any other process $P_j \neq P_1$ can eventually store A_i back to L_i. When P_1 resumes, its CAS loop succeeds (false positive execution) despite the fact that L_i's value has been meanwhile manipulated.

Definition 2. *A nonblocking algorithm is ABA-free when its semantics cannot be corrupted by the occurrence of ABA.*

ABA-freedom is achieved when: *a)* occurrence of ABA is harmless to the algorithm's semantics or *b)* ABA is avoided. The former scenario is uncommon and strictly specific to the algorithm's semantics. The latter scenario is the general case and in this work we focus on providing details of how to eliminate ABA.

A. Known ABA Avoidance Techniques I

A general strategy for ABA avoidance is based on the fundamental guarantee that no process P_j ($P_j \neq P_1$) can possibly store A_i again at location L_i (Step 3, Table I). One way to satisfy such a guarantee is to require all values stored in a given control location to be *unique*. To enforce this uniqueness invariant we can place a constraint on the user and request each value stored at L_i to be used only once (*Known Solution 1*). Enforcing this constraint can be facilitated if a programming language's type system supports uniqueness typing [27] that forbids the use of more than a single reference to an object. We are not familiar with any programming language or library that implements uniqueness typing in a concurrent environment. For a large majority of concurrent algorithms, enforcing uniqueness typing would not be a suitable solution since their applications imply the usage of a value or reference more than once.

An alternative approach to satisfying the uniqueness invariant is to apply a *version tag* attached to each value (or the use of an AtomicStampedReference in Java [15]). The usage of version tags is the most commonly cited solution for ABA avoidance [9]. The approach is effective, when it is possible to apply, but suffers from a significant flaw: a single-word CAS is insufficient for the atomic update of a word-sized control value and a word-sized version tag. An effective application of a version tag [6] requires the hardware architecture to support a more complex atomic primitive that allows the atomic update of two memory locations, such as CAS2 (compare-and-swap two co-located words) or DCAS (compare-and-swap two memory locations). The availability of such atomic primitives might lead to much simpler, elegant, and efficient concurrent designs (in contrast to a CAS-based design). It is not desirable

to suggest a CAS2/DCAS-based ABA solution for a CAS-based algorithm, unless the implementor explores the optimization possibilities of the algorithm upon the availability of CAS2/DCAS. A proposed hardware implementation (entirely built into a present cache coherence protocol) of an innovative Alert-On-Update (AOU) instruction [24] has been suggested by Spear et al. to eliminate the CAS deficiency of allowing ABA. Some suggested approaches [22] split a version counter into two half-words (*Known Solution 2*): a half-word used to store the control value and a half-word used as a version tag. Such techniques lead to severe limitations on the addressable memory space and the number of possible writes into the shared location. To guarantee the uniqueness invariant of a control value of pointer type in a concurrent system with dynamic memory usage, we face an extra challenge: even if we write a pointer value no more than once in a given control location, the memory allocator might reuse the address of an already freed object (A_i) and pose an ABA hazard. To prevent this scenario, all control values of pointer type must be guarded by a concurrent nonblocking garbage collection scheme such as Hazard Pointers [21] (that uses a list of hazard pointers per thread) or Herlihy et al.'s Pass The Buck algorithm [14] (that utilizes a dedicated thread to periodically reclaim unguarded objects). While enhancing the safety of a concurrent algorithm (when needed), the application of a complementary garbage collection mechanism might come at a significant performance cost (Section V).

B. The Descriptor Object

Linearizability is an important correctness condition for concurrent objects [15]. A concurrent operation is linearizable if it appears to execute instantaneously in a given point of time τ_{lin} between the time τ_{inv} of its invocation and the time τ_{end} of its completion. The literature often refers to τ_{lin} as a *linearization point*. The consistency model implied by the linearizability requirement is stronger than the widely applied Lamport's sequential consistency model [17]. According to Lamport's definition, sequential consistency requires that the results of a concurrent execution are equivalent to the results yielded by *some* sequential execution (given the fact that the operations performed by each individual processor appear in the sequential history in the order as defined by the program). The implementations of many nonblocking data structures require the update of *two or more memory locations* in a linearizable fashion [3], [8]. The engineering of such operations (e.g. push_back and resize in a dynamically resizable array) is particularly challenging in a CAS-based design. A common programming technique applied to guarantee the linearizability requirements for such operations is the use of a *Descriptor Object (δ object)* [3], [8]. The pseudocode in Algorithm 2 shows the generalized two-step execution of a Descriptor Object. Our definition of a Descriptor Object requires the Descriptor to store three types of information:

(1) Global data describing the state of the shared container ($\upsilon\delta$), e.g. the size of a dynamically resizable array [3].
(2) A record of a pending operation on a given memory location. We call such a record requesting an update at a shared location L_i from an old value, old_val, to a new value, new_val, a *Write Descriptor ($\omega\delta$)*. The shortcut notation we use is $\omega\delta$ @ L_i : old_val $\rightarrow$ new_val. The fields in the Write Descriptor Object store the target location as well as the old and the new values.
(3) A boolean value indicating whether $\omega\delta$ contains a pending write operation that needs to be completed.

The use of a Descriptor allows an interrupting thread to help the interrupted thread complete an operation rather than wait for its completion. As shown in Algorithm 2, the technique is used to implement, using only two CAS instructions, a linearizable update of two memory locations: 1. a reference to a Descriptor Object (data type pointer to δ stored in a location L_δ) and 2. an element of type value_type stored in L_i. In Step 1, Algorithm 2, we perform a CAS-based speculation of a shared location L_δ that contains a reference to a Descriptor Object. The CAS-based speculation routine's purpose is to replace an existing Descriptor Object with a new one. Step 1 executes in the following fashion:

1. we read the value of the current δ reference stored in L_δ (line 3),
2. if the current δ object contains a pending operation, we need to help its completion (lines 4-5),
3. we record the current value, A_i, in location L_i (line 6) and compute the new value, B_i, to be stored in L_i (line 7),
4. a new $\omega\delta$ object is allocated on the heap, initialized (by calling $f_{\omega\delta}$), and its fields Target, OldValue, and NewValue are set (lines 8-11),
5. any state carrying data stored in a Descriptor Object must be computed (by calling $f_{\upsilon\delta}$). Such data might be a shared element or a container's size (line 12),
6. a new Descriptor Object is initialized containing the new Write Descriptor and the new descriptor's data. The new descriptor's *pending operation* flag (WDpending) is set to true (lines 13-14),
7. we attempt a swap of the old Descriptor Object with the new one (line 15). Should the CAS fail, we know that there is another process that has interrupted us and meanwhile succeeded to modify L_δ and progress. We need to go back at the beginning of the loop and repeat all the steps. Should the CAS succeed, we proceed with Step 2 and perform the update at L_i.

The size of a Descriptor Object is larger than a memory word. Thus, we need to store and manipulate a Descriptor Object through a reference. Since the control value of Step 1 stores a pointer to a Descriptor Object, to prevent ABA at L_δ, all references to descriptors must be memory managed by a safe nonblocking garbage collection scheme. We use the prefix μ for all variables that require safe memory management. In Step 2 we execute the Write Descriptor, WD, in order to update the value at L_i. Any interrupting thread (after the completion of Step 1) detects the pending flag of $\omega\delta$ and, should the flag's value be still positive, it proceeds to executing the requested

update $\omega\delta$ @ $L_i : A_i \rightarrow B_i$. There is no need to execute a CAS-based loop and the call to a single CAS is sufficient for the completion of $\omega\delta$. Should the CAS from Step 2 succeed, we have completed the two-step execution of the Descriptor Object. Should it fail, we know that there is an interrupting thread that has completed it already. A false positive execution of the CAS operation from Step 2 can lead to a *spurious write* of B_i into L_i, violate the operation's linearizability guarantee, and corrupt the semantics of a nonblocking algorithm. In the following sections (Sections II-C, IV) we discuss a number of possible techniques that help us avoid ABA in this scenario.

Algorithm 2 Two-step execution of a δ object

```
1:  Step 1: place a new descriptor in L_δ
2:  repeat
3:      δ μOldDesc = ^L_δ
4:      if μOldDesc.WDpending == true then
5:          execute μOldDesc.WD
6:      value_type A_i = ^L_i
7:      value_type B_i = fComputeB
8:      ωδ WD = f_ωδ()
9:      WD.Target = L_i
10:     WD.OldElement = A_i
11:     WD.NewElement = B_i
12:     υδ DescData = f_υδ()
13:     δ μNewDesc = f_δ(DescData, WD)
14:     μNewDesc.WDpending = true
15: until CAS(L_δ, μOldDesc, μNewDesc) == μNewDesc
16:
17: Step 2: execute the write descriptor
18: if μNewDesc.WDpending then
19:     CAS(WD.Target, WD.OldElement, WD.NewElement) == WD.NewElement
20:     μNewDesc.WDPending = false
```

C. Known ABA Avoidance Techniques II

A known approach for avoiding a false positive execution of the Write Descriptor from Algorithm 2 is the application of *value semantics* for all values of type value_type (*Known Solution 3*). As discussed in [13] and [3], an ABA avoidance scheme based on value semantics relies on:

a. *Extra level of indirection*: all values are stored in shared memory indirectly through pointers. Each write of a given value v_i to a shared location L_i needs to allocate on the heap a new reference to v_i (η_{v_i}), store η_{v_i} into L_i, and finally safely delete the pointer value removed from L_i.
b. *Nonblocking garbage collection (GC):* all references stored in shared memory (such as η_{v_i}) need to be safely managed by a nonblocking garbage collection scheme (e.g. Hazard Pointers, Pass The Buck).

As reflected in our performance test results (Section V), the usage of both an extra level of indirection as well as the heavy reliance on a nonblocking GC scheme for managing the Descriptor Objects *and* the references to value_type objects is very expensive with respect to the space and time complexity of a nonblocking algorithm. However, the use of value semantics is the **only known approach** for ABA avoidance in the execution of a Descriptor Object. In Section IV we present a 3-step execution approach that helps us eliminate ABA, avoid the need for an extra level of indirection, and reduce the usage of the computationally expensive GC scheme.

D. Statistical Analysis of ABA

In this section we develop a statistical model of the probability for an ABA event. We need the following sequence of events for ABA to occur.

S1. P_a succeeds at performing a write at L_δ.
(P_a can only be a push operation for an ABA to be possible).
S2. P_a is interrupted.
S3. P_b reads L_δ
(P_b must be a tail operation to pose an ABA hazard).
S4 P_b is interrupted.
S5. P_a resumes and updates L_i from its A to B.
S6. P_k (any thread different than P_b) stores A to L_i.
P_k could be a write operation only and the value of the write is A.
However, we can have the same effect if P_k does not execute a write but instead we have an immediate sequence of:
S6A. P_k executes a pop
S6B. P_k or any thread different than P_b executes a push with a value A.
S7. P_b resumes and runs into ABA.

This sequence of events involves several very low probability events:

E1. Processor P_a needs to be interrupted right after updating the descriptor L_δ, but before completing the push operation to change L_i.
E2. Similarly, P_b should be interrupted right after reading L_δ, but before it executes the descriptor stored at L_δ or has the chance to complete its own tail operation.
E3. **E2** should happen before P_a resumes.
E4. The write operation of **E1** and the read operation of **E2** should be at the same descriptor location.
E5. When P_b resumes, L_i should be first updated by P_a, and then updated back to its original value.

An interrupt is a low probability event by itself, however what makes these events highly unlikely is that two interrupts should come between the small interval of updating a descriptor, and accessing (first a write and a read) to the global address. Note that **E1** and **E2** are similar events, and **E3** essentially ties these events together to the same miniscule time interval. Thus if the probability of **E1** is x, probability of both happening essentially at the same time is x^2.

Moreover, we need these events to write to and read from the same descriptor variable (while there is only one global Descriptor Object in the vector's design [3], we might have several Descriptor Objects that are read by different processes and pending). Let D be the size of the descriptor set, and $p_w(d)$ and $p_r(d)$ be the probability of P_a writing on , and reading from descriptor $L - d$, respectively. Then this probability will be

$$\sum_{d=1}^{D} p_w(d) p_r(d)$$

Assuming all descriptor events are equally likely to be accessed, this probability will be $\sum_{d=1}^{D} \frac{1}{D^2} = \frac{1}{D}$.

E5 requires P_a to resume before P_b, and then another processor to restore the original value to L_i. The probability of this event is inversely proportional to the size of the alphabet for L_i, and the size of the active set that can be used for writes. This means that L_i should take values from a finite, small-sized alphabet for an ABA to have a practical probability to occur, and values should be repeated. Note that the repetition

requirement also excludes algorithms where the stack is used to identifiers of tasks that are ready to execute, which is a common usage in many graph algorithms and kernels, such as breadth-first or depth-first search, finding strongly connected components, etc. Even when the same value stored multiple times on the stack in these algorithms, they happen with long time gaps. For simplicity if we assume a processor can write to any of N positions with equal likelihood, and any of the K values with equally likelihood, the probability for a write to restore the original value would be would be $\frac{1}{NK}$.

At the time of this work, we did not have statistics for quantifying this probability. But to give the reader an idea about how small this probability is, we will provide the following argument. If the probability of an interrupt is 10^{-4}, that is an interrupt occurs at every 10^4 instructions. Probability of this interrupt coming between an descriptor update and write will be orders of magnitude smaller, say 10^{-8}. Two of these interrupts happening back to back would be 10^{-16}. If there are $D = 100$ descriptor positions, $K = 100$ different values for L_i, $N = 100$ is the size of the active set then the probability of this event will be 10^{-22}. This means if we have a computer that executes 10^{15} instructions per second, we expect an ABA event to occur at every 10^7 seconds, which corresponds to once in every 4 months.

III. Descriptor-based Operations Classification

The practical implementation of a hand-crafted lock-free container is notoriously difficult. Recent research into the design of lock-free data structures includes linked-lists ([11], [20]), double-ended queues ([19], [26]), stacks [13], hash tables ([20], [23]), binary search trees [7], and a dynamically resizable array [3]. For example, a shared vector's random access, data locality, and dynamic memory management pose serious challenges for its nonblocking implementation [3]. The use of a Descriptor Object provides the programming technique for the implementation of some of the complex nonblocking operations in a shared container, such as the push_back, pop_back, and reserve operations in a shared vector [3]. The use and execution of a Write Descriptor guarantees the linearizable update of two or more memory locations. Here, to better understand the interactions among these operations and the cause of ABA, we classify the operations in a nonblocking Descriptor-based design.

Definition 3. *An operation whose success depends on the creation and execution of a Write Descriptor is called an $\omega\delta$-executing operation.*

The operation push_back of a shared vector [3] is an example of an $\omega\delta$-executing operation. Such $\omega\delta$-executing operations have *lock-free* semantics and the progress of an individual operation is subject to the contention on the shared location L_i (under heavy contention, the body of the CAS-based loop from Step 1, Algorithm 2 might need to be re-executed). For a shared vector, operations such as pop_back do not need to execute a Write Descriptor Object [3]. Their progress is dependent on the state of the global data stored in the Descriptor Object, such as the size of a container.

Definition 4. *An operation whose success depends on the state of the $\upsilon\delta$ data stored in the Descriptor Object is a δ-modifying operation.*

A δ-modifying operation, such as pop_back, needs only update the shared global data (the size of type $\upsilon\delta$) in the Descriptor Object (thus pop_back seeks an atomic update of only one memory location: L_δ). Since an $\omega\delta$-executing operation by definition always performs an exchange of the entire Descriptor Object, every $\omega\delta$-executing operation is also δ-modifying. The semantics of a δ-modifying operation are *lock-free* and the progress of an individual operation is determined by the interrupts by other δ-modifying operations. An $\omega\delta$-executing operation is also δ-modifying but as is the case with pop_back, not all δ-modifying operations are $\omega\delta$-executing. Certain operations, such as the random access read and write in a vector [3], do not need to access the Descriptor Object and progress regardless of the state of the descriptor. Such operations are non-δ-modifying and have *wait-free* semantics (thus no delay if there is contention at L_δ).

Definition 5. *An operation whose success does not depend on the state of the Descriptor Object is a non-δ-modifying operation.*

A. Concurrent Operations

Similarly to a number of fundamental studies in nonblocking design [15], [8], we assume the following premises: each processor can execute a number of operations. This establishes a *history* of invocations and responses and defines a *real-time order* between them. An operation O_1 is said to precede an operation O_2 if O_2*'s* invocation occurs after O_1*'s* response. Operations that do not have real-time ordering are defined as *concurrent*. A *sequential history* is one where all invocations have immediate responses. A *linearizable history* is one where: *a.* all invocations and responses can be reordered so that they are equivalent to a sequential history, *b.* the yielded sequential history must correspond to the semantic requirements of the sequential definition of the object, and *c.* in case a given response precedes an invocation in the concurrent execution, then it must precede it in the derived sequential history. It is the last requirement that differentiates the consistency model implied by the definition of linearizability with Lamport's sequential consistency model and makes linearizability stricter. When two δ-modifying operations (O_{δ_1} and O_{δ_2}) are concurrent [15], according to Algorithm 2, O_{δ_1} precedes O_{δ_2} in the linearization history if and only if O_{δ_1} completes Step 1, Algorithm 2 prior to O_{δ_2}.

Definition 6. *We refer to the instant of successful execution of the global Descriptor exchange at L_δ (line 15, Algorithm 2) as τ_δ.*

Definition 7. *A point in the execution of a δ object that determines the order of an $\omega\delta$-executing operation acting on*

location L_i relative to other writer operations acting on the same location L_i, is referred to as the $\lambda\delta$-point ($\tau_{\lambda\delta}$) of a Write Descriptor.

The order of execution of the $\lambda\delta$-points of two concurrent $\omega\delta$-executing operations determines their order in the linearization history. The $\lambda\delta$-point does not necessarily need to coincide with the operation's linearization point, τ_{lin}. The core rule for a linearizable operation is that it must appear to execute in a single instant of time with respect to other concurrent operations. The linearization point need not correspond to a single fixed instruction in the body of the operation's implementation and can vary depending on the interrupts the operation experiences. In contrast, the $\lambda\delta$-point of an $\omega\delta$ object corresponds to a single instruction in the object's implementation. In the pseudo code in Algorithm 2 $\tau_{\lambda\delta} \equiv \tau_{\delta}$.

Let us designate the point of time when a certain δ-modifying operation reads the state of the Descriptor Object by τ_{read_δ}, and the instants when a thread reads a value from and writes a value into a location L_i by τ_{access_i} and τ_{write_i}, respectively. Table II demonstrates the occurrence of ABA in the execution of a δ object with two concurrent δ-modifying operations (O_{δ_1} and O_{δ_2}) and a concurrent write, O_i, to L_i. We assume that the δ object's implementation follows Algorithm 2. The placement of the $\lambda\delta$-point plays a critical role for achieving ABA safety in the implementation of an $\omega\delta$-executing operation. As shown in Table II, at time τ_{wd} when O_{δ_2} executes the write descriptor, O_{δ_2} has no way of knowing whether O_{δ_1} has completed its update at L_i or not. Since O_{δ_1}'s $\lambda\delta$-point $\equiv \tau_{\delta}$, the only way to know about the status of O_{δ_1} is to read L_δ. Using a single-word CAS operation prevents O_{δ_2} from atomically checking the status of L_δ and executing the update at L_i.

Definition 8. *A concurrent execution of one or more non-$\omega\delta$-executing δ-modifying operations with one $\omega\delta$-executing operation, O_{δ_1}, performing an update at location L_i is ABA-free if O_{δ_1}'s $\lambda\delta$-point $\equiv \tau_{access_i}$. We refer to an $\omega\delta$-executing operation where its $\lambda\delta$-point $\equiv \tau_{access_i}$ as a $\lambda\delta$-modifying operation.*

Assume that in Table II the O_{δ_1}'s $\lambda\delta$-point $\equiv \tau_{access_i}$. As shown in Table II, the ABA problem in this scenario occurs when there is a hazard of a spurious execution of O_{δ_1}'s Write Descriptor. Having a $\lambda\delta$-modifying implementation of O_{δ_1} allows any non-$\omega\delta$-executing δ-modifying operation such as O_{δ_2} to check O_{δ_1}'s progress while attempting the atomic update at L_i requested by O_{δ_1}'s Write Descriptor. Our *3-step descriptor execution* approach, discussed in Section IV, offers a solution based on Definition 8. In an implementation with two or more concurrent $\omega\delta$-executing operations, each $\omega\delta$-executing operation must be $\lambda\delta$-modifying in order to eliminate the hazard of a spurious execution of an $\omega\delta$ that has been picked up by a collaborating operation. To effectively avoid the ABA hazard at L_i during a Descriptor-based linearizable update of L_δ and L_i (see Algorithm 2), we generalize two fundamental strategies:

(a) Guarantee that a Write Descriptor created by O_{δ_1}, or any other $\omega\delta$-executing operation, succeeds at most once. We refer to such a δ object as a *once-execute-descriptor*. $Definition\ 8$ offers the condition leading to a solution of this type. In our example in Table II, a *once-execute-descriptor* strategy would cause the attempt to re-execute the write descriptor by O_{δ_1} (Step 7, Table II) or by any other operation to fail. Our 3-step δ execution approach presented in Section IV is one possible way of implementing a *once-execute-descriptor*.

(b) Guarantee that no concurrent interleaving of operations can lead to a write of a value posing ABA hazard (such as B_i in Table II) at L_i. Relying on a methodology that employs unique values, such as *Known Solution* 1, as well as the application of Semantically Enhanced Containers [2] are approaches of this type. Requiring uniqueness typing for ABA prevention is an overkill. The guarantee we need is that no thread can restore an old value A_i in a shared location L_i while there is an alive $\omega\delta$ object in the system requesting $\omega\delta$ @ $L_i : A_i \rightarrow$ $\mathsf{any\_value}_i$. Modern mainstream programming languages do not provide the tools to express and enforce such a concurrent and dynamic correctness condition.

IV. ABA-free Execution of the Descriptor Object

In Algorithm 3 we suggest a design strategy for the implementation of a $\lambda\delta$-modifying operation. Our approach is based on a 3-step execution of the δ object. While similar to Algorithm 2, the approach shown in Algorithm 3 differs by executing a fundamental additional step: in Step 1 we store a pointer to the new descriptor in L_i prior to the attempt to store it in L_δ in Step 2. Since all δ objects are memory managed, we are guaranteed that no other thread would attempt a write of the value μNewDesc in L_i or any other shared memory location. The operation is $\lambda\delta$-modifying because, after the new descriptor is placed in L_i, any interrupting writer thread accessing L_i is aware of the Write Descriptor stored at L_i and is required to complete the remaining two steps in the execution of the Write Descriptor. However, should the CAS execution in Step 2 (line 26) fail, we have to unroll the changes at L_i performed in Step 1 by restoring L_i's old value preserved in WD.OldElement (line 20) and retry the execution of the routine (line 21). To implement Algorithm 3, we have to be able to distinguish between objects of type value_type and δ. A possible solution is to require that all value_type variables are pointers and all pointer values stored in L_i are aligned with the two low-order bits cleared during their initialization. That way, we can use the two low-order bits for designating the type of the pointer values. Subsequently, every read must check the type of the pointer obtained from a shared memory location prior to manipulating it. Once an operation succeeds at completing Step 1, Algorithm 3, location L_i contains a pointer to a δ object that includes both: L_i's previous value of type value_type and a write descriptor WD that provides a record for the steps necessary for the

operation's completion. Any non-δ-modifying operation, such as a random access read in a shared vector, can obtain the value of L_i (of type value_type) by accessing WD.OldElement (thus going through a *temporary* indirection) and ignore the Descriptor Object. Upon the success of Step 3, Algorithm 3, the temporary level of indirection is eliminated. Such an approach would preserve the wait-free execution of a non-δ-modifying operation. The $\omega\delta$ data type needs to be amended to include a field TempElement (line 9, Algorithm 3) that records the value of the temporary δ pointer stored in L_i. The cost of the $\lambda\delta$ operation is 3 CAS executions to achieve the linearizable update of two shared memory locations (L_i and L_δ). We stress that in Algorithm 3, our assumption is that the competing operations both attempt to store an element at a location L_i. It is possible to think of a scenario where only the operation that succeeds first stores its update at L_i and then any interrupted operation would have to change its update location to L_{i+1} or any subsequent location [3]. In this case, we suggest that in Step 1, we perform an additional step: we would have to check whether L_i contains a pending Descriptor object and if it does, help it complete. That way, we would prevent concurrent interleaving that might possibly skip the removal of the temporary level of indirection at L_i and cause an extra use of the costly lock-free garbage collection scheme. Such an implementation is similar to the execution of Harris et al.'s MCAS algorithm [12]. In any scenario, just like our $\lambda\delta$-modifying approach, for an MCAS update of L_δ and L_i, the cost of Harris et al.'s MCAS is at least 3 executions of the single-word CAS instruction. Harris et al.'s work on MCAS [12] brings forward a significant contribution in the design of lock-free algorithms, however, it lacks any analysis of the hazards of ABA and the way the authors manage to avoid it.

Algorithm 3 Implementing a $\lambda\delta$-modifying operation through a three-step execution of a δ object

```
1: Step 1: place a new descriptor in L_i
2: value_type B_i = fComputeB
3: value_type A_i
4: ωδ WD = f_ωδ()
5: WD.Target = L_i
6: WD.NewElement = B_i
7: vδ DescData = f_vδ()
8: δ μNewDesc = f_δ(DescData, WD)
9: WD.TempElement = &NewDesc
10: μNewDesc.WDpending = true
11: repeat
12:     A_i = ^L_i
13:     WD.OldElement = A_i
14: until CAS(L_i, A_i, μNewDesc) == μNewDesc
15:
16: Step 2: place the new descriptor in L_δ
17: bool unroll = false
18: repeat
19:     if unroll then
20:         CAS(WD.Target, μNewDesc, WD.OldElement)
21:         goto 3
22:     δ μOldDesc = ^L_δ
23:     if μOldDesc.WDpending == true then
24:         execute μOldDesc.WD
25:     unroll = true
26: until CAS(L_δ, μOldDesc, μNewDesc) == μNewDesc
27:
28: Step 3: execute the Write Descriptor
29: if μNewDesc.WDpending then
30:     CAS(WD.Target, WD.TempElement, WD.NewElement) == WD.NewElement
31:     μNewDesc.WDPending = false
```

V. Performance Evaluation

To evaluate the performance of the ABA-free programming techniques discussed in this work, we incorporated the presented ABA elimination approaches in the implementation of a nonblocking dynamically resizable array [3]. Our test results indicate that the $\lambda\delta$ approach offers ABA prevention with performance comparable to the use of the platform-specific CAS2 instruction to implement version counting. This finding is of particular value to the engineering of some embedded real-time systems where the hardware does not support complex atomic primitives such as CAS2 [18]. We ran performance tests on an Intel IA-32 SMP machine with two 1.83GHz processor cores with 512 MB shared memory and 2 MB L2 shared cache running the MAC 10.5.6 operating system. In our performance analysis we compare:

(1) $\lambda\delta$ *approach*: the implementation of a vector with a $\lambda\delta$-modifying push_back and a δ-modifying pop_back. In this scenario the cost of push_back is 3 single-word CAS operations and pop_back's cost is one single-word CAS instruction. Table VII offers an overview of the shared vector's operations' relative cost in terms of number and type of atomic instructions invoked per operation.

(2) *All-GC approach*: the application of Known Solution 3 (Section II-C), namely the use of an extra level of indirection and memory management for each element. Because of its performance and availability, we have chosen to implement and apply Herlihy et al.'s Pass The Buck algorithm [14]. In addition, we use Pass The Buck to protect the Descriptor Objects for all of the tested approaches.

(3) *CAS2-based approach:* the application of CAS2 for maintaining a reference counter for each element. A CAS2-based version counting implementation is easy to apply to almost any pre-existent CAS-based algorithm. While a CAS2-based solution is not portable, we believe that the approach is applicable for a large number of modern architectures. For this reason, it is included in our performance evaluation. In the performance tests, we apply CAS2 (and version counting) for updates at the shared memory locations at L_i and a single-word CAS to update the Descriptor Object at L_δ.

Similarly to the evaluation of other lock-free algorithms [7], we designed our experiments by generating a workload of the various operations. We varied the number of threads, starting from 1 and exponentially increased their number to 64. Each thread executed 500,000 lock-free operations on the shared container. We measured the execution time (in seconds) that all threads needed to complete. Each iteration of every thread executed an operation with a certain probability (`push_back` (+), `pop_back` (−), random access `write` (`w`), random access `read` (`r`)). We show the performance graph for a distribution of +:40%, −:40%, `w`:10%, `r`:10% on

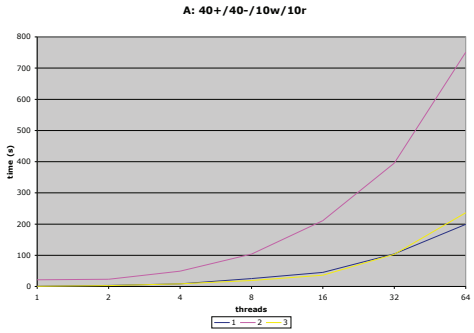

Fig. 1. Performance Results A

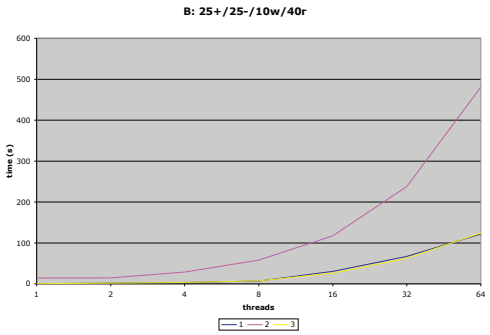

Fig. 2. Performance Results B

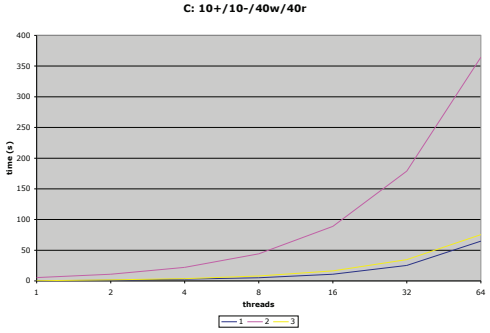

Fig. 3. Performance Results C

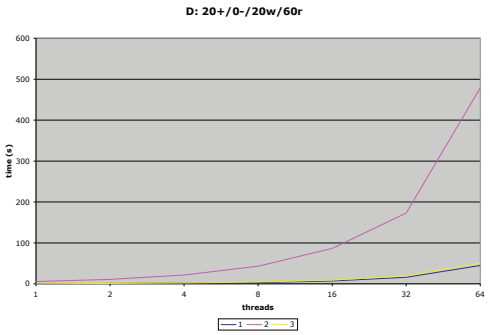

Fig. 4. Performance Results D

Figure 1. Figure 2 demonstrates the performance results with less contention at the vector's tail, +:25%, −:25%, w:10%, r:40%. Figure 3 illustrates the test results with a distribution containing predominantly random access read and write operations, +:10%, −:10%, w:40%, r:40%. Figure 4 reflects our performance evaluation on a vector's use with mostly random access read operations: +:20%, −:0%, w:20%, r:60%, a scenario often referred to as the most common real-world use of a shared container [7]. The number of threads is plotted along the x-axis, while the time needed to complete all operations is shown along the y-axis. According to the performance results, compared to the *All-GC* approach, the $\lambda\delta$ approach delivers consistent performance gains in all possible operation mixes by a large factor, a factor of at least 3.5 in the cases with less contention at the tail and a factor of 10 or more when there is a high concentration of tail operations. These observations come as a confirmation to our expectations that introducing an extra level of indirection and the necessity to memory manage each individual element with PTB (or an alternative memory management scheme) to avoid ABA comes with a pricy performance overhead. The $\lambda\delta$ approach offers an alternative by introducing the notion of a $\lambda\delta$-point and enforces it though a 3-step execution of the δ object. The application of version counting based on the architecture-specific CAS2 operation is the most commonly cited approach for ABA prevention in the literature. Our performance evaluation shows that the $\lambda\delta$ approach delivers performance comparable to the use of CAS2-based version counting. CAS2 is a complex atomic primitive and its application comes with a higher cost when compared to the application of atomic write or a single-word CAS. In the performance tests we executed, we notice that in the scenarios where random access write is invoked more frequently (Figures 3 and 4), the performance of the CAS2 version counting approach suffers a performance penalty and runs slower than the $\lambda\delta$ approach by about 12% to 20%. According to our performance evaluation, the $\lambda\delta$ approach is a systematic, effective, portable, and generic solution for ABA avoidance for Descriptor-based nonblocking designs. The $\lambda\delta$

scheme does not induce a performance penalty when compared to the architecture-specific application of CAS2-based version counting and offers a considerable performance gain when compared to the use of *All-GC*.

VI. Conclusion and Future Work

In this work we studied the ABA problem and the conditions leading to its occurrence in a Descriptor-based lock-free linearizable design. We offered a systematic and generic solution, called the $\lambda\delta$ approach, that outperforms by a significant factor the use of garbage collection for the safe management of each shared location and offers speed of execution comparable to the application of the architecture-specific CAS2 instruction used for version counting. Having a practical alternative to the application of the architecture-specific CAS2 is of particular importance to the design of some modern embedded systems [18]. We defined a condition for ABA-free synchronization that allows us to reason about the ABA safety of a lock-free algorithm. We presented a practical, generic, and portable implementation of the $\lambda\delta$ approach and evaluated it by integrating the $\lambda\delta$ technique into a nonblocking shared vector. The literature *does not offer a detailed analysis of the ABA problem* and the general techniques for its avoidance in a lock-free linearizable design. At the present moment of time, the challenges of eliminating ABA are left to the ingenuity of the software designer. The goal of our work is to deliver a guide for ABA comprehension and prevention in Descriptor-based lock-free linearizable algorithms. For the practical application of Descriptor-based nonblocking techniques in real-time systems, it is important to study the service-time bounds of the operations within the context of the Descriptor's CAS-based retry loop. Anderson et al. [1] present a fundamental approach for such formal timing analysis. In our future work we plan to utilize a model-checker [10] to express the $\lambda\delta$ condition as well as apply Anderson et al.'s [1] approach to derive the timing guarantees for our ABA prevention approach.

References

[1] J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. *ACM Trans. Comput. Syst.*, 15(2):134–165, 1997.

[2] D. Dechev, P. Pirkelbauer, N. Rouquette, and B. Stroustrup. Semantically Enhanced Containers for Concurrent Real-Time Systems. In *IEEE ECBS 2009*, 2009.

[3] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Lock-Free Dynamically Resizable Arrays. In A. A. Shvartsman, editor, *OPODIS*, volume 4305 of *Lecture Notes in Computer Science*, pages 142–156. Springer, 2006.

[4] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Understanding and Effectively Preventing the ABA Problem in Descriptor-based Lock-free Designs. In *In the Proceedings of the 13th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'10)*, 2010.

[5] D. Detlefs, C. H. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. L. S. Jr. Even better DCAS-based concurrent deques. In *International Symposium on Distributed Computing*, pages 59–73, 2000.

[6] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. S. Jr. Lock-free reference counting. *Distrib. Comput.*, 15(4):255–271, 2002.

[7] K. Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, Feb. 2004.

[8] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2):5, 2007.

[9] D. Gifford and A. Spector. Case study: IBM's system/360-370 architecture. *Commun. ACM*, 30(4):291–307, 1987.

[10] P. Gluck and G. Holzmann. Using SPIN Model Checker for Flight Software Verification. In *Proceedings of the 2002 IEEE Aerospace Conference*, 2002.

[11] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC '01: Proceedings of the 15th International Conference on Distributed Computing*, pages 300–314, London, UK, 2001. Springer-Verlag.

[12] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Symposium on Distributed Computing*, 2002.

[13] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA 2004*, pages 206–215, New York, NY, USA, 2004. ACM Press.

[14] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2):146–196, 2005.

[15] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008.

[16] Intel. IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide, 2004.

[17] L. Lamport. How to make a multiprocessor computer that correctly executes programs, September 1979.

[18] M. R. Lowry. Software Construction and Analysis Tools for Future Space Missions. In J.-P. Katoen and P. Stevens, editors, *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2002.

[19] M. Michael. CAS-Based Lock-Free Algorithm for Shared Deques. In *Euro-Par 2003: The Ninth Euro-Par Conference on Parallel Processing, LNCS volume 2790*, pages 651–660, 2003.

[20] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82, New York, NY, USA, 2002. ACM Press.

[21] M. M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[22] K. Reinholtz. Atomic Reference Counting Pointers, C++ User Journal. December 2008.

[23] O. Shalev and N. Shavit. Split-ordered lists: lock-free extensible hash tables. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 102–111, New York, NY, USA, 2003. ACM Press.

[24] M. F. Spear, A. Shriraman, H. Hossain, S. Dwarkadas, and M. L. Scott. Alert-on-update: a communication aid for shared memory multiprocessors. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 132–133, New York, NY, USA, 2007. ACM.

[25] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[26] H. Sundell and P. Tsigas. Lock-Free and Practical Doubly Linked List-Based Deques Using Single-Word Compare-and-Swap. In *OPODIS*, pages 240–255, 2004.

[27] E. Vries, R. Plasmeijer, and D. M. Abrahamson. Uniqueness Typing Simplified. In *Implementation and Application of Functional Languages: 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007.*, 2008.

VII. Appendix: Tables

Step	*Action*
Step 1	P_1 reads A_i from L_i
Step 2	P_k interrupts P_1; P_k stores the value B_i into L_i
Step 3	P_j stores the value A_i into L_i
Step 4	P_1 resumes; P_1 executes a false positive CAS

TABLE I
ABA at L_i

Step	*Action*
Step 1	O_{δ_1}: τ_{read_δ}
Step 2	O_{δ_1}: τ_{access_i}
Step 3	O_{δ_1}: τ_δ
Step 4	O_{δ_2}: τ_{read_δ}
Step 5	O_{δ_1}: τ_{wd}
Step 6	O_i: τ_{write_i}
Step 7	O_{δ_2}: τ_{wd}

TABLE II
ABA occurrence in the execution of a Descriptor Object

ABA prevention approach / operation	push_back	pop_back	read_i	write_i
1. $\lambda\delta$ approach	3 CAS	1 CAS	atomic read	atomic write
2. All-GC approach	2 CAS + GC	1 CAS + GC	atomic read	atomic write + GC
3. CAS2-based approach	1 CAS2 + 1 CAS	1 CAS	atomic read	1 CAS2

TABLE III
A Shared Vector's Operations Cost (Best Case Scenario)

operation	$push$	pop	$read$	$write$
$push$	$ABA\ free$	ABA	$ABA\ free$	ABA
pop	$ABA\ free$	ABA	$ABA\ free$	$ABA\ free$
$read$	$ABA\ free$	$ABA\ free$	$ABA\ free$	$ABA\ free$
$write$	ABA	$ABA\ free$	$ABA\ free$	ABA

TABLE IV
ABA-free and ABA-prone interleaving of two concurrent operations