

A Wait-Free Vector

A Versatile Approach to Wait-Free Progress and Concurrency

Steven Feldman
University of Central Florida
Feldman@knights.ucf.edu

Carlos Valera-Leon
University of Central Florida
cvaleraleon@knights.ucf.edu

Damian Dechev
University of Central Florida and
Sandia National Laboratories
dechev@eecs.ucf.edu

Abstract

The vector is a fundamental data structure, which provides access to a dynamically-resizable range of elements. Currently, there exist no wait-free vectors. The only non-blocking version only supports a subset of the sequential vector API and exhibits significant synchronization overhead caused by supporting opposing operations. As many applications operate in phases of execution, this overhead is unnecessary for the majority of the application.

To address the limitations of the non-blocking version, we present a new design that is wait-free, supports more of the operations provided by the sequential vector, and provides alternative implementations of key operations. These alternatives allow the developer to balance the performance and functionality of the vector as the requirements of the vector change throughout its use.

To allow for situations where a variable number of words may need to be modified while preserving wait-freedom, we present a descriptor-based recursive helping technique. To ensure correctness, we employ an association model that conditionally updates an address if a specific operation has not been applied. We believe that this approach can be applied to other non-blocking algorithms without requiring significant changes to the behavior of those algorithm.

Compared to the known non-blocking version and the concurrent vector found in Intel’s TBB library, our design outperforms or provides comparable performance in the majority of tested scenarios. Over all tested scenarios, the presented design performs an average of 4.97 and 1.54 times more operations per second, respectively. In a scenario designed to simulate the filling of a vector, performance improvement increases to 13.38 and 1.16 times.

This work presents the first ABA-free non-blocking vector that is capable of storing all elements contiguously. Unlike the other non-blocking approach, all operations are wait-free and bounds-checked.

Categories and Subject Descriptors D.1.3 [Programming Techniques]: Concurrent Programming—Parallel programming

Keywords vector, non-blocking, wait-free, concurrent

1. Introduction

Applications designed for many-core systems require the use of fine-grained synchronization in order to achieve strong scaling¹. Non-blocking algorithms are a class of algorithms designed without mutual exclusion that provide this fine-grained synchronization. These algorithms are classified based on their guarantee of progress. Wait-freedom is the strongest form of non-blocking design. It guarantees an operation will complete in a finite number of steps (even if it is continuously interrupted). It is well known that the methods by which many wait-free algorithms achieve their progress guarantee result in limited scalability and performance [9, 12]. This is often a consequence of supporting operations which implement procedures are in conflict with one another.

In this work, we present the design of a concurrent vector that uses a new methodology to prevent thread starvation and incorporates function models to achieve higher performance when in the majority of scenarios when a subset of vector operations are needed.

A vector is a sequence container that stores elements contiguously in memory [11]. In contrast to linked list and tree structures that must be traversed to access elements, this contiguous property allows for efficient $O(1)$ random access to its elements through the use of indices. When the size of the vector is equal to its capacity, a vector allocates a larger region of memory and copy the elements over to it. The presented implementation includes a new resize algorithm that facilitates concurrent access and modification during resize. Other known concurrent vectors [4, 10] use an array segment model to store elements. This avoids the copy-over problem associated with using a single array, but causes elements to be scattered across several arrays. We present an abstraction over the underlying memory model which allows the presented vector operations to function with either the segmented or continuous models.

Our design supports random access read (*at*), write (*write*), insert (*insertAt*) and erase (*eraseAt*) operations as well as appending to (*pushBack*) and removing from (*popBack*) the end of the vector. Unlike a sequential vector, a concurrent vector must maintain correctness when multiple

¹ Strong scaling is the scenario when the total problem size stays fixed while the number of processing elements are increased. The challenge is how to synchronize the work of the processing elements in a correct and efficient manner without “wasting” too many cycles on parallelism overhead.

threads are performing *pushBack* and *popBack* operations. Achieving this correctness adversely affects the complexity and performance of these two operations. As a result, other known concurrent vectors either sacrifice functionality or safety guarantees to achieve desired performance.

PushBack and *popBack* are not typically executed concurrently, but rather are used in different phases of an application's execution. Applications are not able to take advantage of such use patterns and must bear the cost of supporting both operations through the entire execution, instead of just the portions where they are necessary. In this work we describe how alternative function models can deliver high performance in phases which require less functionality and more functionality in phases that require it. To the best of our knowledge, no other work has proposed the use of different function models based on how functionality requirements change throughout an application. In addition to the *traditional model*, which allows each vector operation to be executed alongside one another, we provide two alternative models for the *pushBack* and *popBack* operations. These alternative models are designed to take advantage of cases where only one of these two operations is executing. We refer to these two alternative models as the *Compare-and-Swap (cas) model* and the *Fetch-and-Add (faa) model*. In contrast to the *cas model* where threads compete to append a value or remove the last value, the *faa model* assigns a position to each thread to perform its operation. Sec 6.3 discusses our methodology in designing these models and their compatibility with other vector operations.

A developer would be able to select one model at the start of the application and switch to another model later on. Because the in-memory representation of our vector is independent of these models, there is no cost or limit to the number of times a user can switch between models. Compared to the *traditional model*, the *cas model* and the *faa model* perform 6.48 and 11.95 times as many operations per second, respectively.

This approach in concurrency is inspired by the component aspect of the generative programming paradigm. In this paradigm an optimized software product is automatically generated from a set of reusable implementation components based on a set of requirements [3]. A generative optimizer would be able to automatically select when to use each model based on its understanding of the application and user pragmas. However such an optimizer is beyond the scope of this work, and the presented design necessitates that the developer explicitly selects which model to use.

We are aware of only one other non-blocking vector [4], which supports random access *read* and *write*, and both tail operations *popBack* and *pushBack*. Additionally, Intel's Thread Building Blocks (TBB) library provides a fine-grained locking vector that supports random access *read* and *write* and only the *pushBack* tail operation.

In contrast to the presented vector, neither supports *insertAt* or *eraseAt*. Further, the non-blocking vector does not support bounds-checking or protect against the ABA problem [5].

The specific contribution of this work are as follows:

- This is the first vector to provide a wait-free progress guarantee for all operations. This strong progress property makes it applicable for both real-time and many-core systems. Our design imposes few restrictions on the type of elements stored in the vector (Sec. 4).

- The concept of using different function models throughout an application's execution to balance performance and functionality.
 - We present three versions of the vector's tail operations, each with varying degree of performance and support for other concurrent vector operations.
 - We discuss how a developer can select a high performing model at the start of the application and switch to a model with more functionality as the use case of the vector changes. This narrows the effect on an application's performance caused by supporting a wide range of operations to the portion of execution that requires those operations.
- A wait-free copy-over algorithm that allows elements to be stored contiguously on a single array. Both the non-blocking vector and TBB's concurrent vector use a series of array segments which break the contiguous property.
- The first non-blocking vector to support bounds checking. This prevents a thread from removing a value from an empty vector or performing a random access operation on a position that is not within bounds. Designs which do not support bounds checking exhibit undefined behavior in these scenarios.
- The first non-blocking vector to support multi-position operations, such as *insertAt* and *eraseAt*. The design of these operations can be used to build other multi-position operations as well. One practical multi-position operation we propose is a map operation to atomically update each value in the vector.
- A novel descriptor-based recursive helping technique which allows threads help complete operations which may span an arbitrary number of words across several memory addresses.
- In a micro-benchmark, the presented design achieves higher performance than the best available concurrent vectors.
 - On average, in scenarios involving only *pushBack* operations, our design improves performance by a factor of 7.27.
 - On average, in scenarios involving *pushBack* and random access read operations, our design improves performance by a factor of 4.4.
 - Averaging all test scenarios, our design improves performance by a factor of 27.55.

2. Background

Algorithms designed for concurrent execution are susceptible to many dangers. These dangers include: dead-lock [9], live-lock [1], and thread starvation [9]. Non-blocking algorithm designs avoid mutual exclusion in an attempt to provide scalable performance and to prevent such dangers. If an algorithm is wait-free, then it is free from all three of the aforementioned dangers. However, if it is obstruction-free or lock-free it is susceptible to starvation, and if it is obstruction-free it may also become live-locked [9].

The design of non-blocking algorithms, is predominantly based on the compare and swap operation (*cas*). This operation atomically compares the contents of a memory address with an expected value, and if they match, it assigns a new value to that address before returning the read value. This operation enables a developer to reason about the contents

of a memory address before and after an operation. However, it also introduces the ABA problem, which may lead a thread to act incorrectly. ABA is not an acronym, but a description of an error that can occur in a concurrent environment. For example, a thread operating on an address, P , may act incorrectly if the value of P transitions from a value, A , to another value, B , and back to the original value, A , without the thread perceiving P to have held the value B [9]. Sec. 6 provides a specific example of how ABA can occur and how our design prevents it.

Linearizability is a correctness condition that allows a developer to reason about the correctness of a concurrent object. An operation is linearizable if it appears “to take effect instantaneously at some moment between its invocation and response” [9]. It allows for the construction of a valid sequential history from an initial state to the current state given a set of concurrent operations. This sequential history is constructed by ordering all operations by their invocation and response events. If two operations have overlapping events, then they are ordered by the point at which they linearized with respect to each other.

3. Related Work

A vector is a sequence container which stores elements contiguously in memory. This allows for elements to be efficiently accessed using indices. New elements can either be appended to the end of the vector with $O(1)$ amortized complexity or inserted at an arbitrary position with $O(N)$ amortized complexity. Likewise, elements can be removed from the end of the vector with $O(1)$ amortized complexity and erased from an arbitrary position with $O(N)$ amortized complexity. If the number of elements exceeds the capacity of a vector, a new region of memory is allocated and the elements are copied over.

We are aware of only one other non-blocking concurrent vector in literature [4], which presents a lock-free vector that supports concurrent random access *read* and *write* operations as well as *pushBack* and *popBack* operations. In this design, a single shared object is used to serialize *popBack* and *pushBack* operations. To complete either operation, a thread must first acquire the shared object. If this object is already owned, the thread must execute a helping procedure. In contrast to the presented approach, this design can not guarantee a tail operation will complete if new operations are continually added to the system. Random access *read* and *write* operations are able to execute concurrently without acquiring this shared object; however, there is no mechanism to prevent a thread from accessing a position that is not in bounds. This can lead to the case where a thread reads or writes to a position that is out of bounds resulting in undefined behavior. This lock-free approach approach puts the burden of bounds checking on the user, and it is unclear how a user can safely perform bounds checking. For example, if a thread were to check the size of the vector, another thread can immediately pop one or more elements. The first thread would be unaware of this and could access an out of bounds position. Further, this design is prone to the ABA problem [4, Sec. 3.5], which may lead to elements not being stored contiguously.

Outside of literature, Intel’s Thread Building Blocks (TBB) library presents an open source concurrent vector that supports a subset of the operations provided by the C++ Standard Library [10]. This vector supports semi-bounds checked random access operations and the *pushBack* operation, but does not support the *popBack* operation.

PushBack is performed by fetching and adding one to the size variable and writing the value at the fetched position. In contrast to the presented approach, their methodology does not provide a mechanism to distinguish between a position that holds a value and a position that has been assigned but not written to, allowing for the case where a thread may operate on obsolete data or a partially written value.

In contrast to the design of the C++ Standard Library’s vector, which provides a guarantee that elements are stored contiguously in memory, these designs use a series of array segments, allowing for concurrent growth without having to move elements to increase capacity. The design of our operations are independent of the underlying memory model and can be used on either the contiguous or the segmented memory model. We examine the pros and cons of these two memory models in Sec. 5 and present a wait-free copy-over procedure to support the contiguous array model.

4. Overview of Algorithms

The following sections present our implementation of the vector tail operations (*pushBack* and *popBack*), random access operations (*at* and *curite*), and multi-position operations (such as *insertAt* and *eraseAt*). For each operation, we provide a description of the operation and an informal reasoning of correctness and wait-free progress.

The following algorithms use descriptor objects [2] to indicate that an operation that is in progress. Descriptors are special objects in the vector which denote that an ongoing operation is affecting the element at the given index. A descriptor may be placed at an index where another logical value already exists. Each descriptor object we present contains one or more data members and a set of functions that include a *complete* and a *getValue* function. The *complete* function allows a thread to complete the operation that placed the descriptor object. Upon its return, it guarantees that the descriptor object has been removed. The *getValue* function allows a thread to determine the logical value of the address holding the descriptor object. If the descriptor object’s operation is in progress then the value that the descriptor object replaced is returned; otherwise, the result of the operation is returned.

We distinguish references to descriptor objects from other values by placing a mark on the least significant bit of the reference. For brevity, this bit marking has been omitted from the pseudo code. In general, if a value has been determined to be a descriptor object then the next step would be to remove the mark on the least significant bit.

Our approach requires support for the atomic primitives *compare and swap (cas)*, *fetch and add (faa)*, and *fetch and or (fao)*. When used to store *machine word* size objects, our design reserves the least significant bit (LSB) for type identification to distinguish between descriptors and elements. If the contiguous memory model is used, an additional bit is required to support the concurrent resize procedure presented in Sec. 5. The constants we use to represent empty positions and uninitialized positions impose no addition restriction as they use the LSB. When used to store objects that are larger than a *machine word*, our design requires the use of a multi-word compare-and-swap algorithm [7].

In the tested implementation, we incorporated a memory reclamation scheme based on hazard pointers [13] and reference counting [6] to determine if an object is safe to be reused. The specific details of which are not within the scope of this work.

4.1 Wait-Free Progress

A key challenge to achieving wait-free progress is that there is no guarantee that a thread will execute a successful *cas* operation or observe a value indicating that it should return. For example, a thread may update a value with such frequency that it causes another thread to always fail in its *cas* operation. To guard against this potential danger, we have implemented a progress assurance scheme based on the announcement table presented by Herlihy [8]. Each thread is required to check for an announcement before commencing its own operation. If a thread reads an announcement, it acts according to the contents of the announcement. We use the methodology described by Kogan [12, 14] to reduce the cost of this check to $O(1)$. We use a novel descriptor-based recursive helping technique which allows threads to help complete operations which may span an arbitrary number of words across several memory addresses. As shown in Alg 4 L. 5, a thread makes an announcement when a fail threshold has been reached. After making an announcement, only $(\text{thread count} - 1)^2$ more operations can occur before either the operation is complete.

When using the *fast-path-slow-path methodology* [12], a thread examines the state of a shared object, checks the status of the operation record related to it, and executes the operation if it has not yet been completed. For simple operations, this avoids data races that could occur when multiple threads are executing the same operation. In the more complex operations presented, this procedure has several shortcomings. If it is uncertain which memory words will be affected by an operation, the same operation may be successfully executed on multiple memory locations, and values could be reused, leading to the ABA problem. An example of this occurring is shown in Sec. 6.

We present an association model that overcomes the limitations present in [12]. In this model, each operation record contains a control word which holds a reference to a descriptor object. This reference is initially null, and a *cas* operation is used to assign a value to it. Once the control word is assigned a value, the descriptor and the operation record are said to be associated. By design, it is not possible for the control word to change after it has been set. By using this association, it is possible to determine whether a descriptor object was placed correctly as part of an operation, or if it was placed in error due to thread delay. We use the association model in the construction of several of the vector's operations.

When a thread tries to access a vector element and instead reads a descriptor object, the thread will often need to perform a helping routine. While executing this helping routine, the thread may need to recursively complete the operations for other descriptors. We prevent indefinite recursive helping by requiring a thread to store a reference to its own operation's control word. Before helping to complete another thread's operation, it checks whether or not the value of its control word has changed to a non-null value. If so, some other thread has completed the operation, and the executing thread can return back to its own operation. A thread can also return back to its own operation if it reaches a recursive depth greater than the number of executing threads as this indicates that some operation it is currently helping has been completed.

For each vector operation, we describe the design of the operation record and any challenges faced while implementing it. For brevity, we omit the specific implementation of these operation records; however, the tested implementa-

tion of this algorithm is available upon request. The following algorithms are wait-free because they are composed of only wait-free functions and, using the progress assurance scheme, a limit can be placed on the number of times a thread can be prevented from making progress.

5. Data Structure Representation

The traditional in-memory representation of a vector is a single contiguous region of memory. When this region can no longer accommodate new elements, a new region must be allocated and the elements copied over. This design provides efficient random access operations and high data locality. However, the known concurrent vectors use a segmented memory model to store the vector's elements on a series of array segments. This model does not require elements to be copied over during a resize, but instead allows a thread to append a new array segment to a list of segments. Consequently, a thread must access this list of segments before it can access an element.

Known implementations of the array segment model depend on a static array of references to array segments. They provide $O(1)$ access by requiring that the first array segment be a power of two and the capacity of each subsequent array segment must be twice the capacity of the previous array. An element is accessed by using its index to compute the array segment that holds the element and the offset into that segment. The array segment is determined by taking the \log_2 of the element's index divided the vector's initial capacity and the offset into the array segment is determined by subtracting the sum of the previous array segments from the element's index.

To support the contiguous element model, we include a new wait-free resize procedure. To our knowledge, there are no other known wait-free resize procedures for this model. The use of this resize procedure increases the number of bits reserved by the vector from each element by one. This allows a thread to distinguish between a valid element and one that has been copied to a new vector. The presented algorithm focuses on increasing the vector's capacity and omits details related to decreasing the capacity. However, the presented methodology can be applied to reduce the vector to a specific size.

A thread attempting to resize the vector allocates a new internal vector object with the specified capacity and a reference to the old internal vector. All values up to the capacity of the old internal vector are initialized to a constant that indicates it has not been copied (*notCopied*) and the remaining positions are initialized to a constant representing the position does not hold a valid value (*notValue*). Next, the thread attempts to replace the reference to the old internal vector with a reference to the new one. If this fails, it indicates that the vector has been resized by some other thread. In this event the function returns the reference to the new array. Otherwise, it attempts to copy each element from the old vector to the new vector.

To copy an element from one internal vector to another, the thread uses an atomic *OR* operation to set the resize bit to 1. This prevents any thread from changing the value, as any future *cas* operations will fail and any subsequent reads will observe the bitmark. Next, the thread replaces the *notCopied* value on the new internal vector with the value from the old internal vector. Any thread that sees a *notCopied* value would perform the same procedure before continuing its operation. Similarly, if a thread reads a value with its resize bit set to 1, it will call the *getSpot* function

to get the address of the element on the new vector. This allows any thread performing an operation during a resize to copy only the elements pertinent to its operation.

Depending on the use case of the vector, one representation may be more suitable than the other. For example, systems with limited memory resources may choose the contiguous model as it supports resizing by a specific amount. This is in contrast to the segmented model which must resize in a specific manner, which could allow for a significant amount of wasted space in the event a large expansion occurs when there are a small number of elements left to be added. For other use cases, the resize procedure provided by the contiguous array model could be too costly and the memory utilization of the segmented model may not be a concern.

The presented vector operations are applicable for either memory model. In the description of the following algorithms we abstract the details of underlying memory model using the *getSpot* method which returns a specified position. Internally, *getSpot* calls the vector resize procedure if the position is beyond the vector’s current capacity.

6. Tail Operations

This section presents our implementation for the vector tail operations *popBack* and *pushBack*. Our design implements a wait-free multi-word compare and swap algorithm [7] to perform these operations without breaking the vector’s contiguous element property. We define the last element of the vector as any element followed by *notValue*. Because elements are contiguous, there can only be one element that is considered the last element.

| | | | | | | | | |
|-------|---|---|---|-----|---|-----------------|-----------------|-----|
| index | 0 | 1 | 2 | ... | n | n+1 | n+2 | ... |
| value | A | B | C | ... | X | <i>notValue</i> | <i>notValue</i> | ... |

Figure 1: Example of Contiguous Elements

Using Fig. 1, a *popBack* operation would attempt to change positions n and $n + 1$ to *notValue*. It must attempt to change both, in case the tail of the vector is moved by another thread. Similarly, a *pushBack* operation would attempt to change positions n and $n + 1$ to X and the value being pushed, respectively. The semantics of the presented algorithms ensure that these modifications are performed in a linearizable fashion, giving the appearance of atomicity.

Fig. 2 presents a visualization of how *popBack* and *pushBack* are performed. *PopBack* is the first operation applied in Fig. 2. This operation uses two types of descriptor objects, *popDescr* and *popSubDescr*. The *popDescr* consists solely of reference to a *popSubDescr* (child), this reference is initially null. The *popSubDescr* consists of a reference to a previously placed *popDescr* (parent) and the value that was replaced by the *popSubDescr* (value).

Let T_{Pop} denote the thread performing a *popBack* operation. To perform a *popBack*, T_{Pop} must first place a *popDescr* at the position following the last element (Alg. 1 L. 11). We see this in state Fig. 2.b, where the i^{th} position now holds a reference to the *popDescr* D_{Po} . The next step is to replace the last element of the vector with a *popSubDescr* (Alg. 3 L. 13), this is shown in Fig. 2.c, where the $(i - 1)^{th}$ position now holds a reference to the *popSubDescr* D_{Ps} . To ensure correct behavior, these two descriptors are associated using a *cas* operation (Alg. 3 L. 14). After associating the

two descriptors, T_{Pop} ’s operation linearizes, and the two descriptors are replaced by *notValue* (Alg. 3 L. 18 and L. 16).

PushBack is the second operation shown in Fig. 2.e. This operation uses a *pushDescr* to ensure that the element being pushed is placed correctly. The *pushDescr* contains the value to be pushed (value) and a state variable (state). The state is initially in UNDECIDED and can transition to either FAILED or PASSED.

Let T_{Push} denote the thread performing this operation. To perform a *pushBack*, T_{Push} must place a *pushDescr* at the position following the last element (Alg. 4 L. 16). This is shown in Fig. 2.f, where the $(i - 1)^{th}$ position now holds a reference to the *pushDescr* D_{Pu} . Next, T_{Push} checks position $(i - 2)$ to ensure D_{Pu} has been placed after the last element. If so, T_{Push} sets D_{Pu} ’s *state* member to *PASS*. If not, the *state* is set to *FAIL*, and T_{Push} must try again. T_{Push} completes its operation by replacing D_{Pu} with its logical value of X .

The last operation depicted in Fig. 2 is a delayed thread incorrectly helping a thread whose operation has already been completed. Let T_e be a thread that is helping to complete T_{Pop} ’s operation. T_e was suspended from execution just after determining D_{Po} is not associated. T_e resumes execution in Fig. 2.g. After placing D_e at position $(i - 1)$, T_e attempts to associate D_e with D_{Po} . T_e fails to associate the descriptors because D_{Po} is already associated with D_{Ps} . T_e , realizing the error it made, replaces D_e with X , the value that was originally replaced by D_e . This shows how ABA could occur with concurrent *pushBack* and *popBack* operations, and how our association prevents it from leading to incorrect behavior.

6.1 Correctness

We show that *pushBack* behaves as expected and the result is linearizable by examining the order of its operations. A value is only appended by replacing a *pushDescr* with that value. Since each descriptor object is used once and is placed in a single position, its internal value can only be appended once. After a *pushDescr* replaces *notValue*, the value to replace it with must be determined. This is done by examining the *state* member of the *pushDescr*. If it is unset, the thread will attempt to set the state using a *cas* operation. This guarantees that the state will be set once, and by design it will not change again. A thread sets the state to *PASS* if the position before the *pushDescr* holds an element. In this case, it is impossible for that element to be removed before the *pushBack* operation associated with the *pushDescr* has been completed. This is because removing the element requires placing a descriptor object at the following the position. However, that position already holds a *pushDescr*, which can only be removed after its operation has been completed. As such, it is impossible to incorrectly set the state to *PASS*. This guarantees that if a thread replaces a *pushBack* descriptor with a value, it did so while maintaining the contiguous element property of the vector. By examining the state of the *pushBack* descriptor, a thread knows if its operation succeeded or failed and whether it needs to retry. This design guarantees that the value being appended will be added to the vector exactly once and that it will not break the contiguous element property.

The point at which a *pushBack* operation linearizes is after correctly placing a *pushDescr* (Alg. 4 L. 16). However, this is not realized until the thread has determined that it was placed correctly (Alg. 5 L. 3). Any thread that reads a

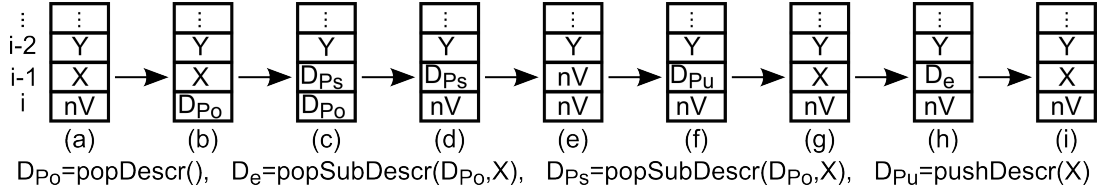


Figure 2: Visualization of the *popBack* and *pushBack* operations.

pushDescr placed at the tail determines its logical value to be the value being pushed.

We show the correctness of the *popBack* operation by showing that it removes a single element and that element was the last element in the vector. For the sake of contradiction, let us assume it is possible to remove an element that is not the last element. By the semantics of the presented algorithm, a value can only be removed by first replacing it with a *popSubDescr*. If the assumption is true, it implies that the *popSubDescr* was not placed before the *popDescr* or that the *popValue* did not replace *notValue*. Either scenario violates the semantics of the presented algorithms, so our initial assumption must be incorrect, and the element removed must have been the last element.

Now let us assume more than one value was removed during a single *popBack* operation. This implies that multiple threads read different non-null values when loading *popDescr*'s *child* member. This contradicts the semantics of the presented algorithms because the *child* member can only transition from null to non-null. As such the value of a *popDescr*'s *child* member can ever be one non-null value, so this assumption must also be incorrect.

The linearization point for *popBack* is the *cas* operation used to associate the *popDescr* and *popSubDescr* (Alg. 3 L. 14). After this point, any thread reading the value of the address holding those descriptors will return *notValue*.

6.2 Wait-Freedom

This section describes how we apply the progress assurance scheme described in Sec. 4.1 to the *popBack* and *pushBack* operations. The while loops in Alg. 1- 5 typically terminate upon reading a value that indicates the operation should fail or performing a successful *cas* operation. In the rare event where descriptor objects are continually removed, a failure threshold causes the loop to be terminated and the operation to be completed by an operation record.

The operation record for a *pushBack* operation consists of a reference to a *pushDescr* and, and the record for a *popBack* operation consists of a reference to a *popDescr*. Both references are initially null. These operations are completed in a similar manner as described above with the following differences. The *pushDescr* and *popDescr* objects contain a reference to the operation record the thread is executing. If a descriptor is placed correctly, a thread associates it with the operation record and replaces it with its logical value. The logical value of a descriptor is the result of the operation only if it is associated with the operation record; otherwise, its logical value is the value it originally replaced.

The while loops in Alg. 1- 5 terminate when the operation record is associated with a descriptor object. After an operation record has been placed in the announcement table, only a finite number of operations can occur before either the operation has been completed or all other threads are helping to complete the operation. If all threads are help-

ing to complete the same operation, some thread will successfully place a descriptor object and associate it with the operation record. This allows us to derive an upper limit on the number of iterations that a thread will perform on the while loops in Alg. 1- 5. As such, they are wait-free because an upper limit can be derived for all loops contained within and they are composed of only wait-free functions.

6.3 Alternative Tail Operation Models

Unlike the lock-free vector (LFvec), our *pushBack* and *popBack* model supports bounds checking, which prevents the case where a thread may incorrectly read the contents of a position greater than the current size. However, it does share a common deficiency with LFvec, which is the inability to diffuse and/or reduce the contention caused by supporting these opposing operations. It is in our opinion, that unless an auxiliary structure (such as Herlihy's stack exchange [9]) is used or the vector's memory representation is significantly transformed, it is unlikely for both operations to co-exist and exhibit strong scaling in a many core system.

To address this deficiency we present two alternative models of each operation that exhibit better scaling and performance. These alternatives are designed to be executed in isolation from their opposing tail operation. We believe that a number of algorithms which use vectors can take advantage of these alternative implementations during phases where elements are solely being pushed or popped from the vector.

6.3.1 Compare-and-Swap Tail Operations

Alg. 6(*cas_popBack*) and Alg. 7(*cas_pushBack*) present an implementation of *pushBack* and *popBack* in which other threads can safely access elements while elements are being pushed or popped from the vector. In contrast to Intel's Thread Building Block's vector (TBBvec), our design guarantees that if a thread reads the value at a position, either a valid value is returned, or false is returned indicating it is out of bounds.

The *cas_pushBack* operation loops until it replaces *notValue* with the value being pushed; after which it increments the size variable. The *cas_popBack* operation loops until it replaces a value with *notValue*; after which it decrements the size variable. If the size variable is less than zero, it returns false. As discussed in Sec. 4.1, the number of times a thread retries an operation is limited by our progress assurance scheme.

The linearization point of these operations is the successful execution of the *cas* operation. We sketch an informal proof of correctness by examining how the vector is modified. By the nature of the operations executing, the vector's tail can move in a single direction. As such, we can reason that if we observe an element followed by *notValue*, and we succeed at applying the operation, it is impossible for our

operation to break the contiguous element property. If we assume a thread incorrectly removed an element that was not at the tail, then this implies that the tail moved as a result of the addition of one or more elements. However, this is in contradiction to the restriction we placed on the types of operations allowed to execute concurrently (i.e. no *pushBack* or *insertAt* operations). A similar proof can be used to demonstrate the correctness of *pushBack* operation.

6.3.2 Fetch and Add Tail Operations

Alg. 8 (*faa_pushBack*) and Alg. 9 (*faa_popBack*) present an implementation of *pushBack* and *popBack* in which a thread is assigned a position to complete its operation. This assignment is accomplished by performing a fetch and add (*faa*) on the size variable which adjusts the size monotonically. This allows the algorithms to be implemented without loops or descriptor objects. Additionally, due to the short amount of time a thread accesses the size variable, it is unlikely that it becomes a bottleneck. This approach mirrors the approach used in TBBvec to append new elements. However, in contrast to TBBvec, our random access operations can be designed to detect if a position has been assigned but not fully initialized (See Sec. 7).

7. Random Access Operations

This section presents our implementation for the random access read (*at*) and modification (*cwrite*) operations. These implementations are bounds checked on both the capacity of the vector and the tail of the vector. Any attempt to access a position that is not within the bounds of the vector causes the function to return false. To identify whether or not a position is within bounds, a thread first checks if the position is less than the capacity. Then it checks whether or not the logical value of a position is *notValue*. If it is *notValue*, that position is considered out of bounds.

7.1 At

Alg. 10 presents our implementation of the vector's *at* operation. Depending on the types of operations currently executing, a thread may attempt to access a position that holds a descriptor object. We handle this by calling the *getValue* function which returns a value based on the state of the operation associated with the descriptor. If the operation is in progress or the descriptor was placed in error, the value of the address before the descriptor was placed is returned. Otherwise, the result of the operation is returned.

The *at* algorithm is wait-free because it contains no loops and it only calls other wait-free functions. It linearizes, in respect to all other operation, on the loading of the value at the specified position (Alg. 10 L. 2), unless the value is a descriptor. If the value is not a descriptor, then it linearizes on the loading of the value (Alg. 10 L. 2). If it is a descriptor, it linearizes upon determining its logical value (Alg. 10 L. 5).

7.2 Conditional Write

Alg. 11 presents a wait-free implementation of a vector-specific *cas* operation (*cwrite*). The use of descriptor objects can cause a *cas* operation to fail, even if the logical value of the address matches the expected value. Our implementation overcomes this by detecting if the current value is a descriptor object and if so removing that descriptor object. When a non-descriptor object is read the thread compares the expected value with the current value. If they match, the thread attempts to replace the current value with the new

value. If this is successful, the thread returns true (Alg. 11 L. 4). Otherwise the current value is re-examined. If the current value does not match the expected value, the operation linearizes on the load of that value, and returns false (Alg. 11 L. 8).

To achieve wait-free behavior, we use an operation record (*WriteOp*) and a descriptor object (*WriteHelper*). A thread replaces the current value at the address with a reference to a *WriteHelper* then attempts to associate it with a *WriteOp*. A *WriteHelper* holds a reference to the *WriteOp* and the value it replaced. A *WriteOp* holds the address to operate on, expected value, new value, and a *WriteHelper* reference. The result of the operation can then be determined by the value replaced by an associated *WriteHelper*.

The *cwrite* operation linearizes, in respect to all other operation, on the successful *cas* operation (Alg. 11 L. 4), unless the operation record was used. If it is used, it linearizes when a thread associates the *WriteOp* with a *WriteHelper*.

7.3 Interaction with Fetch-and-Add Based Tail Operations

Using these random access operations in conjunction with the fetch and add based tail operations may lead to unexpected behavior. For example, if a thread performing a *faa_pushBack* has been assigned a position, *i*, at which to place its value, but has not written its value yet, then this position would be considered out of bounds. If subsequent *faa_pushBack* operations assign values to position greater than *i*, *i* will still be considered out of bounds until it has a value written to it.

8. Multi-Position Operations

This section provides an overview of how we implement wait-free linearizable multi-position operations, such as the shift operations *insertAt* and *eraseAt* and the *map* operation². We are aware of no other non-blocking or fine-grained locking vectors that support such operations. In a sequential vector, the cost of performing a shift operations is very high and in a non-blocking vector it is even greater. This functionality is provided for the sake of completeness of the vector's API.

Our multi-position operation is based on using an operation record that contains a state variable, a reference to a *doubly linked list descriptor* (*DLDescr*), a set of positions, and a function that determines the new value for each position. A *DLDescr* is composed of a reference to the operation record that describes the operation, a reference to a previously placed *DLDescr*, a reference to the *DLDescr* that is placed next, and the value replaced by the *DLDescr*. A multi-position operation executes in two phases: the first phase replaces the value at each position with a *DLDescr*. The second phase replaces the *DLDescr* at each position with the result of the operation. *DLDescrs* must be placed in an ascending order, or there is a risk of cyclic dependencies between two or more operations.

To place a *DLDescr*, the current value is examined. If it is a *pushDescr* or *popSubDescr*, its state is set to *PASS*, it is removed, and the position is re-examined³. If it is a *popDescr*, its state is set to *FAIL*, it is removed, and the position is re-examined. If it is another type of descriptor object, its complete function is called, and the position is re-examined. Otherwise the thread attempts to replace the

²A higher-order function that applies a given function to each element or subset of elements in the vector.

³Preventing a cyclic dependency from forming.

current value with a *DLDescr* that holds a reference to the operation, the previous *DLDescr*, and the value read. If the thread failed to replace the value, it re-examines the current value. We limit the number of retries using the progress assurance scheme discussed in Sec. 4.1. After successfully placing a descriptor, the thread attempts to associate the descriptor with the previously placed *DLDescr*. If this association fails, this indicates the operation has already been completed by some other thread.

9. Experimental Evaluation

In this section we compare the performance of the presented wait-free vector (WFvec) to that of TBB’s vector (TBBvec) [10] and the lock-free vector (LFvec) [4]. The *fetch and add* and *compare and swap* based tail operation models are represented in the following graphs as WFfaa and WFcas, respectively. We examine how the different designs are affected by thread contention, complexity, and the types of operations executing.

9.1 Testing Methodology

Our testing procedure consists of a main thread that initializes the vector with an initial capacity of one million and pre-fills it with ten thousand elements. Then it spawns a set of worker threads and when all threads are ready, it signals the start of execution and sleeps for five seconds. Upon waking, it signals the end of the execution and computes the result. Each worker thread repeatedly executes vector operations according to a specified distribution until it has received a signal to stop.

All tests were conducted on a 64-core ThinkMate RAX QS5-4410 server running Ubuntu 12.04 LTS. It is a NUMA system with four AMD Opteron 6272 CPUs (16 cores per chip @2.1 GHz) and 314 GB of shared memory. For each algorithm tested a separate executable was compiled with GCC 4.8 and the options `-std=c++11` and `-O3`. The following performance results is the average performance over ten executions.

9.2 Analysis

Fig. 3 presents the performance of each algorithm when using different ratios of random access read operations to *pushBack* operations. As shown in graph 3a, WFfaa algorithm consistently outperforms other implementations when there are solely *pushBack* operations executing on the vector. This test is similar to the phase of an application where values are being accumulated for later processing. On average, WFfaa performs 1.16 times as many operations per second as TBBvec, and 13.38 times as many as LFvec. Compared to WFcas algorithm and WFvec, WFfaa algorithm performs 2.3 and 22.12 times as many operations respectively.

Both TBBvec and WFfaa performed similarly, while WFvec and LFvec algorithms performed poorly. We attribute this poor performance to the cost of supporting conflicting operations, which causes both algorithms to incur significant synchronization penalties during concurrent operations. The WFcas performs better than WFvec due to its simpler design and smaller critical section. However, its design does not distribute the contention like that of the *faa* based approaches, which reduces its scalability.

Fig. 3b shows the performance of the vectors when only 10% of operations are *pushBack* and the majority of operations are random access reads. This test is similar to phase

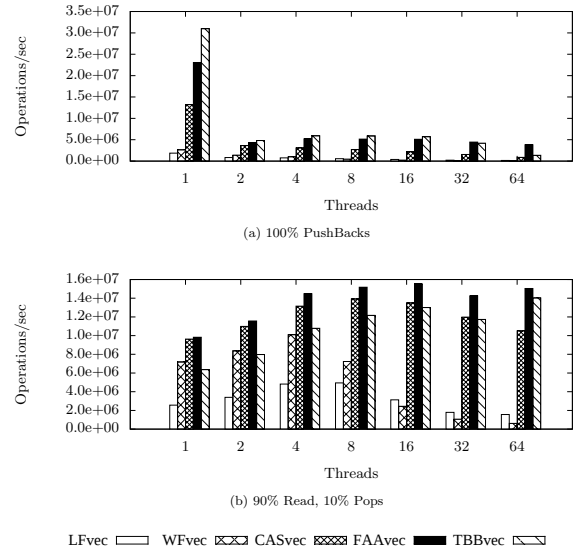


Figure 3: Random Access Reads and PushBack Operations

of an application where most threads are actively processing the accumulated elements in the vector. In this graph, each implementation exhibits a similar scaling pattern of increasing in performance up until 8 to 16 threads after which losing performance. WFvec’s and LFvec’s *pushBack* begin to lose performance sooner, while the fetch and add based approaches maintain scalability for longer. This loss in performance is explained by the fact that each processor on the system has 16 cores. Testing on a system which supports a higher number of cores on a single processor may produce different scaling patterns.

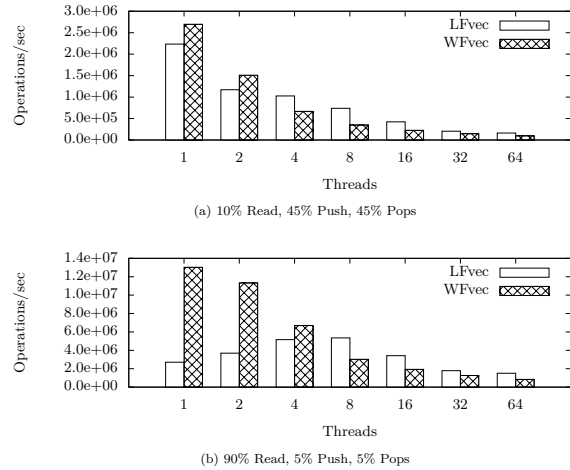


Figure 4: Random Access Reads and Tail Operations

Fig. 4 presents the performance of the vectors when there are interleaved *pushBack*, *popBack*, and random access read operations. TBBvec, WFfaa, and WFcas are not included in this test as they do not support concurrent *pushBack* and *popBack*. In this scenario, LFvec outperforms WFvec by a factor of 1.46, with both approaches scaling poorly as the number of threads increases. The methodology used by LFvec to support both operation has significant safety con-

cerns (Sec. 3) and we believe that the marginal performance benefit achieved by such a design does not justify the risk.

10. Conclusion

We presented a new concurrent vector that provides improvements over existing designs. These improvements include: providing a stronger guarantee of progress (wait-freedom), stronger safety properties (ABA-freedom and bounds checking), and support for more operations (*insertAt* and *eraseAt*). We developed a technique which uses association between operation records and descriptor objects to achieve wait-free progress when operations may access a variable number of words in memory." Our shift operations provide a technique for the development of other multi-position vector operations, empowering developers to operate on the vector in a more expressive manner.

We created a new resize procedure which allows elements to be contiguous in memory. This procedure facilitates concurrent access to the vector without requiring threads to help complete the entire resize; instead only requiring a thread to copy elements pertinent to its operation. The design of our vector operations are not limited to this contiguous model, but can also be implemented using the segmented memory model.

We compared the presented vector with other known approaches using a series of micro-benchmarks and found the presented approach to be most performant in the majority of cases. In comparison to the only existing non-blocking vector, our new design performed on average 4.97 times more operation per second and 13.38 times when filling the vector.

We identified how supporting opposing operations has an adverse affect on both the complexity and performance of the vector operations. We then presented a set of memory-layout-compatible alternative function models for tail operations which allows the developer to overcome this performance issue by leveraging the natural phases of execution in an application. Our analysis revealed that the throughput of the *pushBack* operation was increased by 22x as a result of applying this new model.

Our future goals include identifying data structures which we can simplify by applying our association model to provide wait-free progress and identifying structures where separating functionality into function models with relaxed semantics can improve performance.

Acknowledgments

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

- [1] A. Babich. Proving total correctness of parallel programs. *Software Engineering, IEEE Transactions on*, SE-5(6):558–574, Nov 1979.
- [2] G. Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, SPAA '93, pages 261–270, New York, NY, USA, 1993. ACM.
- [3] K. Czarneci and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

- [4] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Lock-free dynamically resizable arrays. In *Proceedings of the 10th international conference on Principles of Distributed Systems*, OPODIS'06, pages 142–156, Berlin, Heidelberg, 2006. Springer-Verlag.
- [5] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Understanding and effectively preventing the aba problem in descriptor-based lock-free designs. *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, 0:185–192, 2010.
- [6] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele, Jr. Lock-free reference counting. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, PODC '01, pages 190–199, New York, NY, USA, 2001. ACM.
- [7] S. Feldman, P. LaBorde, and D. Dechev. A wait-free multi-word compare-and-swap operation. *International Journal of Parallel Programming*, pages 1–25, 2014.
- [8] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, Nov. 1993.
- [9] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Elsevier Science, 2011.
- [10] Intel Corporation. Reference for Intel Threading Building Blocks (<http://threadingbuildingblocks.org/>). Retrieved 02/02/2014.
- [11] ISO/IEC 14882 Standard for Programming Language C++. *Programming languages: C++*. American National Standards Institute, September 2011.
- [12] A. Kogan and E. Petrank. A methodology for creating fast wait-free data structures. *SIGPLAN Not.*, 47(8):141–150, Feb. 2012.
- [13] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, June 2004.
- [14] S. Timnat and E. Petrank. A practical wait-free simulation for lock-free data structures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '14, pages 357–368, New York, NY, USA, 2014. ACM.

A. Appendix

Algorithm 1 bool wait-free popBack(&value)

```

1: ph=new PopDescr()
2: pos=size.load()
3: while true do
4:   if failures++ > LIMIT then
5:     return announce_op(PopOp(value))
6:   else if pos==0 then
7:     return false
8:   spot=getSpot(pos)
9:   expected=spot.load()
10:  if expected==notValue then
11:    if spot.cas(expected,ph) then
12:      res=ph.complete(pos)
13:      if res then
14:        value=ph.child.value
15:        decrementSize()
16:        return true
17:      else
18:        ph=new PopDescr()
19:        pos--
20:   else if isDescr(expected) then
21:     expected.complete(vec, pos)
22:   else
23:     pos++

```

Algorithm 2 bool PopSubDescr.complete(*pos*)

```
1: spot=getSpot(pos)
2: ph=this.ph
3: ph.child.cas(null,this)
4: if ph.child.load() == this then
5:   spot.cas(this,notValue)
6: else
7:   spot.cas(this,this.value)
8: return ph.child.load() == this
```

Algorithm 3 bool PopDescr.complete(*pos*)

```
1: spot=getSpot(pos-1)
2: while !this.child.load() do
3:   if failures++ > LIMIT then
4:     this.child.cas(null,FAILED)
5:   else
6:     expected=spot.load()
7:     if expected==notValue then
8:       this.child.cas(null,FAILED)
9:     else if isDescr(expected) then
10:      expected.complete(vec, pos-1)
11:    else
12:      psh=new PopSubDescr(this, expected)
13:      if spot.cas(expected,psh) then
14:        this.child.cas(null,psh)
15:        spot2=getSpot(pos)
16:        spot2.cas(this,notValue)
17:        if this.child.load() == psh then
18:          spot.cas(psh,notValue)
19:        else
20:          spot.cas(this,expected)
21: return this.child.load() != FAILED
```

Algorithm 4 wait-free pushBack()

```
1: ph=new PushDescr(value)
2: pos=size.load()
3: while true do
4:   if failures++ > LIMIT then
5:     return announce_op(PushOp(value))
6:   spot=getSpot(pos)
7:   expected=spot.load()
8:   if expected==notValue then
9:     if pos==0 then
10:      if spot.cas(expected, value) then
11:        incrementSize()
12:        return 0
13:     else
14:       pos++
15:       spot=getSpot(pos)
16:     if spot.cas(expected,ph) then
17:       res=ph.complete(pos)
18:       if res then
19:         incrementSize()
20:         return (pos-1)
21:     else
22:       ph=new PushDescr(value)
23:       pos--
24:   else if isDescr(expected) then
25:     expected.complete(vec, pos)
26:   else
27:     pos++
```

Algorithm 5 bool PushDescr.complete(*pos*)

```
1: spot=getSpot(pos)
2: spot2=getSpot(pos-1)
3: current=spot2.load()
4: while !this.state.load() && isDescr(current) do
5:   if failures++ > LIMIT then
6:     this.state.cas(-1, FAILED)
7:   current.complete(pos-1)
8:   current=spot2.load()
9: if !this.state then
10:  if current == notValue then
11:    this.state.cas(-1, FAILED)
12:  else
13:    this.state.cas(-1, PASS)
14: if this.state== PASS then
15:  spot.cas(this, this.value)
16: else
17:  spot.cas(this, notValue)
18: return this.state== PASS
```

Algorithm 6 cas_popBack(&*value*)

```
1: pos=size.load()-1
2: while true do
3:   if failures++ < LIMIT then
4:     return announce_op(CasPopOp())
5:   else if pos<0 then
6:     return false
7:   else
8:     spot=getSpot(pos)
9:     cur=spot.load()
10:    if cur!=notValue and spot.cas(cur, notValue) then
11:      decrementSize()
12:      value=cur
13:      return true
14:    pos--
```

Algorithm 7 cas_pushBack(*value*)

```
1: pos=size.load()
2: while true do
3:   if failures++ < LIMIT then
4:     return announce_op(CasPushOp(value))
5:   spot=getSpot(pos)
6:   cur=spot.load()
7:   if cur==notValue and spot.cas(cur, value) then
8:     incrementSize()
9:     return pos
10:  pos++
```

Algorithm 8 faa_popBack(&*value*)

```
1: pos=decrementSize()
2: if pos > 0 then
3:   value=getSpot(pos-1).load()
4:   getSpot(pos-1).store(notValue)
5: else
6:   incrementSize()
7: return (pos > 0)
```

Algorithm 9 faa_pushBack(*value*)

```
1: pos=incrementSize()
2: getSpot(pos-1).store(value)
3: return pos
```

Algorithm 10 bool at(*pos*, &*value*)

```
1: if pos <= capacity.load() then
2:   spot=getSpot(pos)
3:   temp=spot.load()
4:   if isDescr(temp) then
5:     temp=temp.getValue(vec, pos)
6:   if temp!=notValue then
7:     value=temp
8:   return true
9: return false
```

Algorithm 11 bool cwrite(*pos*, &*old*, *new*)

```
1: if pos < capacity.load() then
2:   spot=getSpot(pos)
3:   for failures < LIMIT; failures++ do
4:     value=spot.load()
5:     if isDescr(value) then
6:       value.complete()
7:     else if value==old then
8:       if spot.cas(value, new) then
9:         return true
10:    else
11:      old=value
12:      return false
13: return announce_op(WriteOp(pos, old, new))
14: return false
```
