Lock-free Dynamically Resizable Arrays

Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup

Texas A&M University
College Station, TX 77843-3112
{dechev, peter.pirkelbauer}@tamu.edu, bs@cs.tamu.edu

Abstract. We present a first lock-free design and practical implementation of a dynamically resizable array (vector). The most extensively used container in the C++ Standard Library is *vector*, offering a combination of dynamic memory management and efficient random access. Our approach is based on a single 32-bit word atomic compare-and-swap (CAS) instruction and our implementation is portable to all systems supporting CAS, and more. It provides a flexible, generic, linearizable and highly parallelizable STL like interface, effective lock-free memory allocation and management, and fast execution. Our current implementation is designed to be most efficient on the most recent multi-core architectures. The test cases on a dual-core Intel processor indicate that our lock-free vector outperforms its lock-based STL counterpart and the latest concurrent vector implementation provided by Intel by a factor of 10. The implemented approach is also applicable across a variety of symmetric multiprocessing (SMP) platforms. The performance evaluation on an 8-way AMD system with non-shared L2 cache demonstrated timing results comparable to the best available lock-based techniques for such systems. The presented design implements the most common STL vector's interfaces, namely random access read and write, tail insertion and deletion, pre-allocation of memory, and query of the container's size.

Keywords: lock-free, STL, C++, vector, concurrency, real-time systems

1 Introduction

The ISO C++ Standard [19] does not mention concurrency or thread-safety (though it's next revision, C++0x, will [3]). Nevertheless, ISO C++ is widely used for parallel and multi-threaded software. Developers writing such programs face challenges not known in sequential programming: notably to correctly manipulate data where multiple threads access it. Currently, the most common synchronization technique is to use mutual exclusion locks. While conceptually simple, lock-based programs suffer performance and safety problems. A mutual exclusion lock guarantees thread-safety of a concurrent object by blocking all contending threads except the one holding the lock. This can seriously affect the performance of the system by diminishing its parallelism. The behavior of mutual exclusion locks can sometimes be optimized by using fine-grained locks, context-switching locks [2], and reader/writer locks [25]. However, the interdependence of processes implied by the use of locks – even efficient locks – introduces the dangers of deadlock, livelock, and priority inversion. To many systems, the problem with locks is one of difficulty of providing correctness more than one of performance.

The widespread use of multi-core symmetric multiprocessing (SMP) architectures and the hardware support of multi-threading in some modern hardware platforms pose the challenge to develop effective, practical and robust concurrent data structures. The main target of our design is to deliver most optimal performance for such systems (Section 4). In addition, many real-time and autonomous systems, such as the Mission Data Systems Project at the Jet Propulsion Laboratory [8], require effective fine-grained synchronization. In such systems, the application of locks is a significantly complex and challenging problem due to the hazards of priority inversion and deadlock. Furthermore, the implementation of distributed parallel containers and algorithms such as STAPL[2] can benefit from a shared lock-free vector. The use of non-blocking (lock-free) techniques have been suggested to prevent the interdependence of the concurrent processes introduced by the application of locks. By definition, a lock-free concurrent data structure guarantees that when multiple threads operate simultaneously on it, *some* thread will complete its task in a *finite* number of steps despite failures and waits experienced by other threads [13]. The vector is the most versatile and ubiquitous data structure in the C++ Standard Template Library (STL) [28]. It is a dynamically resizable array (dynamic table) that provides automatic memory management, random access, and tail element insertion and deletion with an amortized cost of O(1). [4].

This paper presents the following contributions:

(a) A first design and practical implementation of a lock-free dynamically resizable array. Our lock-free vector provides a set of common STL vector operations in a generic and linearizable [15] fashion. The presented design defines effective concurrent semantics and allows disjoint-access parallelism for some operations.
(b) A portable algorithm and implementation. Our design is based on the word-size compare-and-swap (CAS) instruction available on a large number of hardware platforms.
(c) A fast and space-efficient implementation. In particular, it avoids excessive copying [1] and on a variety of tests executed on an Intel dual-core architecture it outperforms its lock-based STL counterpart and the latest concurrent vector provided by Intel [18] by a factor of 10. In addition, a large number of tests executed on an 8-way AMD architecture with non-shared L2 cache indicate performance comparable to the best available lock-based techniques for such systems..
(d) An effective incorporation of non-blocking memory management and memory allocation schemes into the algorithm. Without such an integration a design is not useful in real-world applications.

The rest of the paper is structured like this: 2: Background, 3: Implementation, 4: Performance Evaluation, and 5: Conclusion.

2 Background

As defined by Herlihy [13], a concurrent object is *lock-free* 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*. Lock-free and wait-free algorithms do not apply mutual exclusion locks. Instead they rely on a set of atomic primitives such as the word-size CAS instruction or Load-Linked (LL) / Store Conditional (SC) operations. Common CAS implementations require three arguments: a memory location,Mem, an old value,V_{old}, and a new value,V_{new}. The instruction's purpose is to atomically exchange the value stored in Mem with V_{new}, provided that its current value corresponds to V_{old}. The architecture ensures the atomicity of the operation by applying a fine-grained hardware lock such as a cache or a bus lock (e.g.: IA-32 [17]). The use of a hardware lock does not violate the non-blocking property as defined by Herlihy. Common locking synchronization methods such as semaphores, mutexes, monitors, and critical sections utilize the same atomic primitives to manipulate a control token. Such an application of the atomic instructions introduces interdependencies of the contending processes. In the most common scenario, lock-free systems utilize CAS in order to implement a speculative manipulation of a shared object. Each contending process speculates by applying a set of writes on a local copy of the shared data and attempts to CAS the shared object with the updated copy. Such an approach guarantees that from within a set of contending processes, there is at least one that succeeds within a finite number of steps. Therefore, the system is non-blocking and the performed operations could be implemented in a linearizable fashion at the expense of some extra work performed by the contending processes.

2.1 Pragmatic Lock-Free Programming

The practical implementation of lock-free containers is notoriously difficult: in addition to addressing the hazards of race conditions, the developer must also use non-blocking memory management and memory allocation schemes [14]. As explained in [1] and [5], a single-word CAS operation is inadequate for the practical implementation of a non-trivial concurrent container. A better design requires atomic update of several memory locations. The use of a double-compare-and-swap primitive (DCAS) has been suggest by Detlefs et al. in [5], however it is rarely supported by the hardware architecture.

A software implementation of a multiple-compare-and-swap (MCAS) algorithm, based on CAS, has been proposed by Harris et al. [11]. This software-based MCAS algorithm has been effectively applied by Fraser towards the implementation of a number of lock-free containers such as binary search trees and skip lists [7]. The cost of this MCAS operation is relatively expensive requiring $2M + 1$ CAS instructions. Consequently, the direct application of this MCAS scheme is not an optimal approach for the design of lock-free algorithms. However, the MCAS implementation employs a number of techniques (such as pointer bit marking and the use of descriptors) that are useful for the design of practical lock-free systems. As discussed by Harris et al., a descriptor is an object that allows an interrupting thread to help an interrupted thread to complete successfully.

2.2 Lock-Free Data Structures

Recent research into the design of lock-free data structures includes linked-lists [30], [10], [21], [6], double-ended queues [20], [29], stacks [12], hash tables [21], [27] and binary search trees [7]. The problems encountered include excessive copying, low parallelism, inefficiency and high overhead. Despite the widespread use of the STL vector in real-world applications, the problem of the design and implementation of a lock-free dynamic array has not yet been discussed. The vector's random access, data locality, and dynamic memory management poses serious challenges for its non-blocking implementation. Our goal is to provide an effecient and practical lock-free STL-style vector.

2.3 Design Principles

We developed a set of design principles to guide our implementation:

(a) *thread-safety*: all data can be shared by multiple processors at all times.
(b) *lock-freedom*: apply non-blocking techniques to provide an implementation of thread-safe C++ dynamic array based on the current C++ memory model.
(c) *portability*: do not rely on uncommon architecture-specific instructions.
(d) *easy-to-use interfaces*: offer the interfaces and functionality available in the sequential STL vector.
(e) *high level of parallelism*: concurrent completion of non-conflicting operations should be possible.
(f) *minimal overhead*: achieve lock-freedom without excessive copying and minimize the time spent on CAS-based looping as well as the number of calls to CAS.
(g) *simplicity*: keep the implementation simple to allow model-based testing and verification.

The lock-free vector's design and implementation provided follow the syntax and semantics of the ISO STL vector as defined in ISO C++ [19]. The discussed implementation techniques in the following section can be applied towards the lock-free design and implementation of the complete set of ISO C++ vector's interfaces.

3 Algorithms

In this section we define a concurrent semantic model of the vector's operations, provide a description of the design and the applied implementation techniques, outline a correctness proof based on the adopted semantic model, address concerns related to memory management, and discuss some alternative solutions to our problem. The presented algorithms have been implemented in ISO C++ and designed for execution on an ordinary multi-threaded shared-memory system supporting only single-word read, write, and CAS instructions.

3.1 Concurrent Semantics

The semantics of the vector's operations are based on a number of assumptions. In the presented design, we assume that each processor can execute a number of the vector's 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*. The vector's operations are of two types: those whose progress depends on the vector's descriptor and those who are independent of it. We refer to the former as *descriptor-modifying operations* and to the latter as *non-descriptor modifying*. Each vector's operation in a set of concurrent descriptor-modifying operations S_1 is thread-safe and lock-free. The execution of the operations in S_1 always leads to a legal and desirable sequential history. A *sequential history* is defined as a history where an operation's invocation is immediately followed by its response. The non-descriptor modifying operations such as random access read and write are implemented through the direct application of atomic read and write instructions on the shared data. In a set of non-descriptor modifying operations S_2, all operations are thread-safe and wait-free. The execution of the set of operations $S_3 = S_1 \cup S_2$ leads to legal but not always desirable sequential history. The reason is the possibility to concurrently execute an operation $O_1 \in S_1$ that reduces the size of the vector (such as a tail deletion) and an operation $O_2 \in S_2$. Such a scenario can result in a legal but undesirable sequential history in the case when operation O_1's response precedes operation O_2's response. This limitation can be eliminated by transforming all non-descriptor modifying operations into descriptor-modyfing. We prefer not to do so in order to preserve the efficiency and wait-freedom of the current non-descriptor modifying operations. Instead we define the following:

Usage Rule: Non-descriptor modifying operations that access the tail should never be executed concurrently with descriptor modifying operations that reduce the vector's size.

3.2 Implementation Overview

The major challenges of providing lock-free vector implementation stem from the fact that key operations need to atomically modify two or more non-colocated words. For example, the critical vector operation `push_back` increases the size of the vector and stores the new element. Moreover, capacity-modifying operations such as `reserve` and `push_back` potentially allocate new storage and relocate all elements in case of a dynamic table [4] implementation. Element relocation must not block concurrent operations (such as `write` and `push_back`) and must guarantee that interfering updates will not compromise data consistency. Therefore, an update operation needs to modify up to four vector values: size, capacity, storage, and a vector's element.

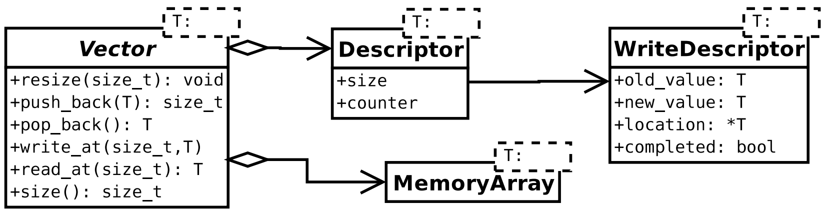

Fig. 1. Lock-free Vector. T denotes a data structure parameterized on T.

The UML diagram in Fig. 1 presents the collaborating classes, their programming interfaces and data members. Each vector object contains the memory locations of the data storage of its elements as well as an object named "`Descriptor`" that encapsulates the container's size, a reference counter required by the applied memory management scheme (Section 3.5) and an optional reference to a write descriptor. Our approach requires that datatypes bigger than word size are indirectly stored through pointers. Like Intel's concurrent vector [26], our implementation avoids the synchronization hazards during storage relocation by utilizing a two-level array. Whenever push_back exceeds the current capacity, a new memory block twice the size of the previous one is added. The semantics of the pop_back and push_back operations are guaranteed by the "Descriptor" object. Consider the case when a pop_back operation is interrupted by any matching number of push_back and pop_back operations. In a naïve implementation, the size of the vector would appear unchanged when the original pop_back resumes and the operation could produce an erroneous result. This is an instance of the ABA problem (Section 3.6). The use of a descriptor allows a thread-safe update of two memory locations thus eliminating the need for a DCAS instruction. An interrupting thread intending to change the descriptor, will need to complete any pending write. Not counting memory management overhead, this mechanism requires *two successful CAS* instructions to update *two memory locations* in the case of a push_back operation.

3.3 Operations

We have modeled the vector's interfaces according to the STL and implemented the following operations with the given interfaces and runtime guarantees (Tab. 1).

	Operations	Vector	Complexity
push_back	$Vector \times Elem \rightarrow void$	$Descriptor_t \rightarrow Descriptor_{t+1}$	$O(1) \times congestion$
pop_back	$Vector \rightarrow Elem$	$Descriptor_t \rightarrow Descriptor_{t+1}$	$O(1) \times congestion$
reserve	$Vector \times size\_t \rightarrow Vector$	$Descriptor_t \rightarrow Descriptor_t$	$O(1)$
read	$Vector \times size\_t \rightarrow Elem$	$Descriptor_t \rightarrow Descriptor_t$	$O(1)$
write	$Vector \times size\_t \times Elem \rightarrow Vector$	$Descriptor_t \rightarrow Descriptor_t$	$O(1)$
size	$Vector \rightarrow size\_t$	$Descriptor_t \rightarrow Descriptor_t$	$O(1)$

Table 1. Vector - Operations

The remaining part of this section presents the generalized pseudo-code of the implementation and hence omits code necessary for a particular memory management scheme. We use the symbols ^, &, and . to indicate pointer dereferencing, obtaining an object's address, and integrated pointer dereferencing and field access respectively. The function *HighestBit* returns the bit-number of the highest bit that is set in an integer value. On Intel's x86 processor *HighestBit* corresponds to the *BSR* assembly instruction. First_Bucket_Size is a constant representing the capacity of the first bucket and corresponds to eight in our implementation.

Push_back (add one element to end) Since push_back is a descriptor-modifying operation, the first step is to complete a pending operation that the current descriptor might hold. Second, in case that the storage capacity has reached its limit, new memory is allocated for the next memory bucket. Third, push_back defines a new "Descriptor" object and describes the current write operation. Finally, push_back uses a CAS to swap the previous "Descriptor" object with the new one. Should CAS fail, the routine is re-executed. After succeeding, push_back finishes by writing the element to the storage.

Pop_back (remove one element from end) Besides push_back, pop_back is the only other descriptor modifying operation. Unlike push_back, pop_back does not utilize a write descriptor. To provide a better correspondence to the concurrent semantics of the lock-free vector, we have slightly augmented the interface. In a sequential program, calling read_i(size-1) before pop_back reads and removes the last element. In a parallel environment, such semantics cannot be assumed. Hence, pop_back does not only remove the last element from the stack but also returns its value.

Reserve (increase allocated space) Reserve allocates new storage according to the description in Section 3.2. In the case of concurrently executing reserve operations, only one succeeds, while the others deallocate the acquired memory.

Algorithm 1 pushback $vector, elem$

```
repeat
    $descriptor_{current} \leftarrow vector.descriptor$
    $CompleteWrite(vector, descriptor_{current}.pending)$
    $bucket \leftarrow HighestBit(descriptor_{current}.size + \texttt{First\_Bucket\_Size}) - HighestBit(\texttt{First\_Bucket\_Size})$
    if vector.memory[bucket] = NULL then
        $AllocBucket(vector, bucket)$
    end if
    $writeop \leftarrow new\ WriteDesc(At(descriptor_{current}.size)\hat{}, elem, descriptor_{current}.size)$
    $descriptor_{next} \leftarrow new\ Descriptor(descriptor_{current}.size + 1, writeop)$
until $CAS(\&vector.descriptor, descriptor_{current}, descriptor_{next})$
$CompleteWrite(vector, descriptor_{next}.pending)$
```

Algorithm 2 popback $vector$

```
repeat
    $descriptor_{current} \leftarrow vector.descriptor$
    $CompleteWrite(vector, descriptor_{current}.pending)$
    $elem \leftarrow At(vector, descriptor_{current}.size - 1)\hat{}$
    $descriptor_{next} \leftarrow new\ Descriptor(descriptor_{current}.size - 1, NULL)$
until $CAS(\&vector.descriptor, descriptor_{current}, descriptor_{next})$
return $elem$
```

Algorithm 3 Read $vector, i$

```
return $At(vector, i)\hat{}$
```

Algorithm 4 Write $vector, i, elem$

```
$At(vector, i)\hat{} \leftarrow elem$
```

Algorithm 5 Reserve $vector, size$

```
$i \leftarrow HighestBit(vector.descriptor.size + \texttt{First\_Bucket\_Size} - 1) - HighestBit(\texttt{First\_Bucket\_Size})$
if $i < 0$ then
    $i \leftarrow 0$
end if
while $i < HighestBit(size + \texttt{First\_Bucket\_Size} - 1) - HighestBit(\texttt{First\_Bucket\_Size})$ do
    $i \leftarrow i + 1$
    $AllocBucket(vector, i)$
end while
```

Algorithm 6 At $vector, i$

```
$pos \leftarrow i + \texttt{First\_Bucket\_Size}$
$hibit \leftarrow HighestBit(pos)$
$idx \leftarrow pos\ xor\ 2^{hibit}$
return $\&vector.memory[hibit - HighestBit(\texttt{First\_Bucket\_Size})][idx]$
```

Algorithm 7 CompleteWrite $vector, writeop$

```
if $writeop.pending$ then
    $CAS(At(vector, writeop.pos), writeop.value_{old}, writeop.value_{new})$
    $writeop.pending \leftarrow false$
end if
```

Read and Write at position i (use of subscripting) The random access `read` and `write` do not utilize the descriptor and their success is independent of the descriptor's value. The provided design corresponds to the STL's vector subscripting operations that do not offer bound checking.

Algorithm 8 Size *vector*

```
$descriptor \leftarrow vector.descriptor$
$descriptor \leftarrow descriptor.size$
if $descriptor.writeop.pending$ then
    $size \leftarrow size - 1$
end if
return $size$
```

Algorithm 9 AllocBucket *vector*, *bucket*

```
$bucketsize \leftarrow \texttt{First\_Bucket\_Size}^{bucket+1}$
$mem \leftarrow new\ T[bucketsize]$
if $not\ CAS(\&vector.memory[bucket], NULL, mem)$ then
    $Free(mem)$
end if
```

Size (read number of elements) The `size` operations returns the size stored in the "`Descriptor`" minus a potential pending write operation at the end of the vector.

3.4 Correctness

The main correctness requirement for the semantics of the vector's operations is linearizability [15]. A concurrent operation is linearizable if it appears to execute instantaneously in a given moment of time between the time point t_1 of its invocation and the time point t_2 of its completion. The definition of linearizability implies that each concurrent history yields responses that are equivalent to the responses of some legal sequential history for the same requests. Additionally, the order of the operations within the sequential history should be consistent with the real-time order. In the context of the operations' semantics discussed earlier in Section 3.1, our proof of linearizability is straightforward by distinguishing the points where the execution of our operations appears to be atomic with respect to other operations.

Linearization Points

Let us assume that there is an operation $O_i \in S_{vec}$, where S_{vec} is the set of all possible vector's operations. We assume that O_i can be executed concurrently with n other operations $\{O_{op1}, O_{op2} ..., O_{opn}\} \in S_{vec}$. We provide a proof that operation O_i is linearizable. Assume O_i is a non-descriptor-modifying operation such as a random read or write. These operations perform a single atomic access directly to the shared data and their linearization point occurs naturally at the time point t_a when the atomic `read` or `write` instruction is executed. Assume O_i is a descriptor-modifying operation such as a `push_back` or a `pop_back`. Such operations are carried out in two stages: modify the "`Descriptor`" variable and then update the data structure's contents. Let us define that the time points t_{call}, and t_{return} represent the moments of time when O_i has been called and when it returns, respectively. Similarly, time points t_{desc} and $t_{writedesc}$ denote the instances of time when O_i executes an atomic update to the vector's "`Descriptor`" variable and when O_i's write descriptor is completed by O_i itself or another concurrent operation $O_c \in \{O_{op1}, O_{op2} ..., O_{opn}\}$, respectively. If the set $S_{tps} = \{T_{tp1}, T_{tp2} ..., T_{tpn}\}$ represents all instances of time in between t_{call} and t_{return}, it is clear that $t_{desc} \in S_{tps}$ and $t_{writedesc} \in S_{tps}$. As mentioned earlier, the `pop_back` operation does not need to utilize a write descriptor. In this case the linearization point of O_i is t_{desc}. In the case when O_i is a `push_back`, its linearization point is $t_{writeop}$.

Non-blocking We prove the non-blocking property of our implementation by showing that out of n threads at least one makes progress. Since the progress of non-descriptor modifying operations is independent, they are wait-free. Thus, it suffices to consider an operation O_1, where O_1 is either a `push_back` or `pop_back`. A write descriptor can be simultaneously read by n threads. While one of them will successfully perform the write descriptor's operation (O_2), the others will fail and not attempt it again. This failure is nonsignificant for the outcome of operation O_1. The first thread attempting to change the descriptor will succeed, which guarantees the progress of the system.

Testing We have executed a large number of test scenarios with our implementation. In addition, we have formalized the semantics of the lock-free vector as well as the atomic primitives using the SPIN model checker's modeling language Promela[16]. We have utilized SPIN in order to perform exhaustive testing and simulation of the vector's operations.

3.5 Memory Management

Our algorithms do not require the use of a particular memory management scheme. Assuming a garbage collected environment significantly reduces the complexity of the implementation. However, we are unaware of any available general lock-free garbage collector for C++.

Object Reclamation Our concrete implementation uses reference counting as described by Valois et al [30] and [24]. The scheme has two inherent drawbacks: each access to a managed object requires atomic modification of a counter. Second, a timing window allows objects to be reclaimed while a different thread is about to increase the counter. Consequently, objects cannot be freed but only recycled. We do not regard the latter problem as significant, since our usage pattern of the algorithm guarantees an upper bound of $2n$ number of objects in the system, where n is the number of concurrent threads operating on the data structure. Alternatively, we could also integrate Michael's hazard pointers [22], which use fewer atomic CAS instructions, reduce contention on the counter, and allow reclamation and arbitrary memory-reuse. This method requires to scan all threads for published pointers in use before objects can be freed. Although a correct selection of its parameters yields an amortized constant time for these scans, this technique does not uniformely distribute the overhead across the operations. The pass the bucket algorithm described by Herlihy et al. in [14] offers a solution presenting equivalent trade-offs in contrast to the reference counting schemes.

Allocator Recent research by Michael [23] and Gidenstam [9] presents implementations of true lock-free memory allocators. Due to its availability and performance, we selected Gidenstam's allocator for our performace tests.

3.6 The ABA Problem

The ABA problem is fundamental to all CAS-based systems. In our current implementation we have not incorporated a remedy to prevent it. The ABA problem occurs when a thread T_1 reads a value A from a shared object and then an interrupting thread T_2 modifies the value of the shared object from A to B and then back to A. When T_1 resumes, it erroneously assumes that the object has not been modified. Given such behavior, there is a serious risk that T_2 's execution is going to violate the correctness of the object's semantics. Practical solutions to the ABA problem include the use of hazard pointers [22] or the association of a version counter to each element in the platforms supporting a double-word compare-and-swap primitive (CAS2) such as IA-32 [17].

The semantics of the lock-free vector's operations can be corrupted by the occurrence of the ABA problem. Consider the following execution: assume a thread T_0 attempts to perform a *push_back* and stores in the vector's "`Descriptor`" object a write-descriptor announcing that the value of the object at position i should be changed from A to B. Then a thread T_1 interrupts and reads the write-descriptor. Later, after T_0 resumes and successfully completes the operation, a third thread T_2 can modify the value at position i from B back to A. When, T_1 resumes its CAS is going to succeed and erroneously execute the update from A to B. There are two particular instances when the ABA problem can affect the correctness of the vector's operations:

(1) the user intends to store a memory address value A in multiple locations.

(2) the memory allocator reuses the address of an already freed object.

The use of tags or version counters eliminates the dangers in both cases. However, this solution is applicable only on a limited set of architectures supporting a CAS2 instruction. The application of Michael's hazard pointers prevents the reuse of memory addresses that are likely to cause the ABA problem. Thus, hazard pointers solve the problem stated in Scenario 2. There are a number of possibilities in order to eliminate the vector's vulnerability to the ABA problem in the former scenario. One option is to require the vector to copy the objects and store pointers to them. Such behavior complies with the STL value-semantics [28], however it can incur higher overhead in some cases due to the additional heap allocation and object construction. In a lock-free system, both the object construction and heap allocation can execute concurrently with other operations. In a scenario where the vector is under high contention, the additional overhead would not cause a significant delay of the operation's performance. In our future work we intend to evaluate the impact on performance of such ABA prevention techniques.

3.7 Alternatives

In this section we discuss several alternative designs for lock-free vectors.

Copy on Write In [1] Alexandrescu and Michael present a lock-free map, where every write operation creates a clone of the original map, which insulates modifications from concurrent operations. Once completed, the pointer to the map-representation is redirected from the original to the new map. The same idea could be adopted to implement a vector. Since the complexity of any write operation deteriorates to $O(n)$ instead of $O(1)$, this scheme would be limited to applications exhibiting read-often but write-rarely access patterns.

Using Software DCAS In [11] Harris et al. present a software multi-compare and swap (MCAS) implementation based on CAS instructions. While convenient, the MCAS operation is expensive (requiring $2M + 1$ CAS instructions). Thus, it is not the best choice for an effective implementation.

Contiguous storage Techniques similar to the ones used in our vector implementation could be applied to achieve a vector with contiguous storage. Compared to the presented approach, the major difference is that the storage area can change during lifetime. This requires `resize` to move all elements to the new location. Hence, storage and its capacity should become members of the descriptor. Synchronization between `write` and `resize` operations is what makes this approach difficult. One solution would make every vector-modification change the descriptor. In such case, the success of `resize` depends on the absence of concurrent writes. Furthermore, making all write operations modify the descriptor implies serialization of all writes and therefore diminishes the desired disjoint-access parallelism.

We discussed the descriptor- and non-descriptor modifying writes in the context of the two-level array and the contiguous storage vector. However, these write properties are not inherent in these two approaches. In the two-level array, it is possible to make each write operation descriptor-modifying, thus ensure a write within bounds. In the contiguous storage approach, element relocation could replace the elements with marked pointers to the new location. Every access to these marked pointers would get redirected to the new storage.

4 Performance Evaluation

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 OS 10.4.6 operating system. In our performance analysis, we compare the lock-free approach (with its integrated lock-free memory management and memory allocation) with the most recent concurrent vector provided by Intel [18] as well as an STL vector protected by a lock. For the latter scenario we applied different types of locking synchronizations - an operating system dependent mutex, a reader/writer lock, a spin lock, as well as a queuing lock. We used this variety of lock-based techniques in order to contrast our non-blocking implementation to the best available locking synchronization technique for a given distribution of operations. We utilize the locking synchronization provided by Intel [18].

Similarly to the evaluation of other lock-free concurrent containers [7] and [21], we have designed our experiments by generating a workload of various operations (`push_back`, `pop_back`, random access `write`, and `read`). In the experiments, we varied the number of threads, starting from 1 and exponentially increased their number to 32. Every active thread executed 500,000 operations on the shared vector. We measured the CPU time (in seconds) that all threads needed in order 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 use per-thread linear congruential random number generators where the seeds preserve the exact sequence of operations within a thread across all containers. We executed a number of tests with a variety of distributions and found that the differences in the containers' performances is generally preserved. As discussed by Fraser [7], it has been observed that in real-world concurrent application, the read operations dominate and account to about 70% to 75% of all operations. For this reason we illustrate the performance of the concurrent vectors with a distribution of `+`:15%, `-`:5%, `w`:10%, `r`:70% on Figure 2A. Similarly, Figure 2C demonstrates the performance results with a distribution containing predominantly writes, `+`:30%, `-`:20%, `w`:20%, `r`:30%. In these diagrams, the number of threads are plotted along the x-axis, while the time needed to complete all operations is shown along the y-axis. Both axes use locarithmic scale.

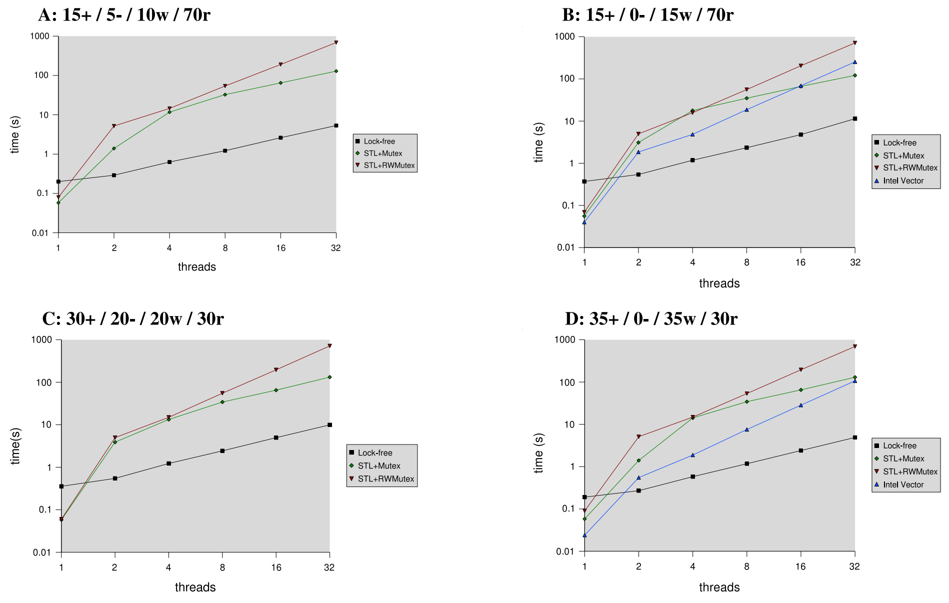

Fig. 2. Performace Results - Intel Core Duo

The current release of Intel's concurrent vector does not offer `pop_back` or any alternative to it. To include its performance results in our analysis, we excluded the `pop_back` operation from a number of distributions. Figure 2B and 2D present two of these distributions. For clarity we do not depict the results from the QueuingLock and SpinLock implementations. According to our observations, the QueuingLock performance is consistently slower than the other lock-based approachs. As indicated in [18], SpinLocks are volatile, unfair, and unscalable. They showed fast execution for the experiments with 8 threads or lower, however their performance significantly deteriorated with the experiments conducted with 16 or more active threads. To find a lower bound for our experiments we timed the tests with a non-thread safe STL-vector with preallocated memory for all operations. For example, in the scenario described in figure 2D, the lower bound is about a $\frac{1}{10}$ of the lock-free vector.

Our non-blocking implementation of the STL vector consistently outperforms the alternative lock-based approaches in all possible operation mixes by a significantly large factor (generally a factor of 10). It has also proved to be scalable as demonstrated by the performance analysis. These large performance gains have been consistently observed in all tested distributions of operations. Lock-free algorithms are particularly beneficial to shared data under high contention. It is expected that in a scenario with low contention, the performance gains will not be as considerable as in the case of high contention.

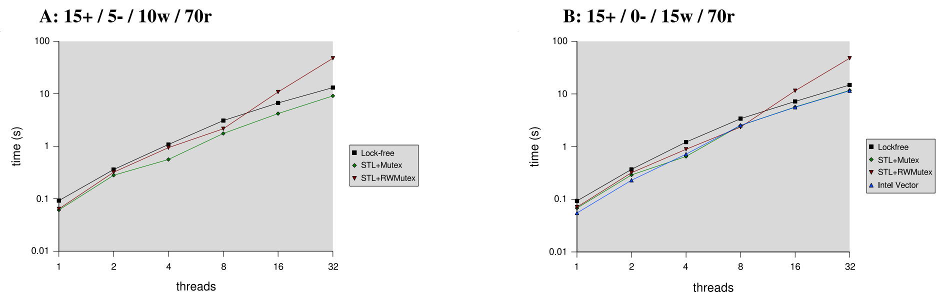

Fig. 3. Performace Results - AMD 8-way Opteron

On systems without shared L2 cache, shared data structures suffer from performance degradation due to cache coherency problems. To test the applicability of our approach on such architecture we have performed the same experiments on an AMD 2.2GHz 8-way Opteron architecture with 1 MB L2 cache for each processor and 4GB shared RAM running the MS Windows 2003 operating system. (Fig.3). The applied lock-free memory allocation scheme is not available for MS Windows. For the sake of our performance evaluation we applied a regular lock-based memory allocator. The experimental results on this architecture lack the impressive performance gains we have observed on the dual-core L2 shared-cache system. However, the graph (Fig.3) demonstrates that the performance of our lock-free approach on such architectures is comparable to the performance of the best lock-based alternatives. An essential direction in our future work is to further optimize the effectiveness of our approach on such hardware architectures. This can be achieved by eliminating the bottlenecks introduced by the reference counting scheme applied for memory management. In the current scheme, each access to the container's tail requires a CAS-based manipulation of the descriptor's reference counter, invalidating copies in other caches. The memory latencies increase even more by the enqueing and dequeuing of descriptors at the head of a single freelist. We expect that the application of an alternative memory management scheme eliminate these problems [14] [22].

5 Conclusion

We presented a first practical and portable design and implementation of a lock-free dynamically resizable array. We developed a highly efficient algorithm that supports disjoint-access parallelism and incurs minimal overhead. To provide a practical implementation, our approach integrates non-blocking memory management and memory allocation schemes. We compared our implementation to the best available concurrent lock-based vectors on a dual-core system and have observed an overall speed-up of a factor of 10. The applied memory management is costly, thus we intend to integrate a better scheme. In addition, it is our goal to define the concurrent semantics of the remaining STL vector interface and incorporate them in our implementation.

References

1. A. Alexandrescu and M. Michael. Lock-free data structures with hazard pointers. *C++ User Journal*, November 2004.
2. P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: A Standard Template Adaptive Parallel C++ Library. In *Int. Wkshp on Adv. Compiler Technology for High Perf. and Embedded Processors*, pages 193–208, Cumberland Falls, Kentucky, aug 2001.

3. P. Becker. Working Draft, Standard for Programming Language C++, ISO WG21N2009, April 2006.
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms.* MIT Press, Cambridge, MA, USA, 2001.
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. M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 50–59, New York, NY, USA, 2004. ACM Press.
7. K. Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, Feb. 2004.
8. D. Garlan, W. K. Reinholtz, B. Schmerl, N. D. Sherman, and T. Tseng. Bridging the gap between systems design and space systems software. In *SEW '05: Proceedings of the 29th Annual IEEE/NASA on Software Engineering Workshop*, pages 34–46, Washington, DC, USA, 2005. IEEE Computer Society.
9. A. Gidenstam, M. Papatriantafilou, and P. Tsigas. Allocating memory in a lock-free manner. In *ESA*, pages 329–342, 2005.
10. 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.
11. 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.
12. D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 206–215, New York, NY, USA, 2004. ACM Press.
13. M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, 1993.
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. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
16. G. Holzmann. *The Spin Model Checker, Primer and Reference Manual.* Addison-Wesley, Reading, Massachusetts, 2003.
17. Intel. Ia-32 intel architecture software developer's manual, volume 3: System programming guide, 2004.
18. Intel. Reference for Intel Threading Building Blocks, version 1.0, April 2006.
19. ISO/IEC 14882 International Standard. *Programming languages C++.* American National Standards Institute, September 1998.
20. 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.
21. 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.
22. M. M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.
23. M. M. Michael. Scalable lock-free dynamic memory allocation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 35–46, New York, NY, USA, 2004. ACM Press.
24. M. M. Michael and M. L. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, 1995.
25. M. Reiman and P. E. Wright. Performance analysis of concurrent-read exclusive-write. *SIGMETRICS Perform. Eval. Rev.*, 19(1):168–177, 1991.
26. A. Robison, Intel Corporation. Personal communication, April 2006.
27. 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.
28. B. Stroustrup. *The C++ Programming Language.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
29. 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.
30. J. D. Valois. Lock-free linked lists using compare-and-swap. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222, New York, NY, USA, 1995. ACM Press.