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 implementation of a dynamically resizable array (vector). The most extensively used container in the C++ Standard Template Library (STL) is *vector*, offering a combination of dynamic memory management and constant-time random access. Our approach is based on a single 32-bit word atomic compare-and-swap (CAS) instruction. It provides a linearizable and highly parallelizable STL-like interface, lock-free memory allocation and management, and fast execution. Our current implementation is designed to be most efficient on multi-core architectures. Experiments on a dual-core Intel processor with shared L2 cache indicate that our lock-free vector outperforms its lock-based STL counterpart and the latest concurrent vector implementation provided by Intel by a large factor. The performance evaluation on a quad dual-core AMD system with non-shared L2 cache demonstrated timing results comparable to the best available lock-based techniques. 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. Using the current implementation, a user has to avoid one particular ABA problem.

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

1 Introduction

The ISO C++ Standard [18] does not mention concurrency or thread-safety (though it's next revision, C++0x, will [4]). 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. 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 [17] or context-switching. However, the interdependence of processes implied by the use of locks – even efficient locks – introduces the dangers of deadlock, livelock, and

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 142–156, 2006.

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 architectures and the hardware support for multi-threading pose the challenge to develop practical and robust concurrent data structures. The main target of our design is to deliver good 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 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 has been suggested to prevent the interdependence of the concurrent processes introduced by the application of locks [13]. 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. The vector is the most versatile and ubiquitous data structure in the C++ STL [26]. It is a dynamically resizable array that provides automatic memory management, random access, and tail element insertion and deletion with an amortized cost of O(1).

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 linearizable [15] STL vector operations, which allow disjoint-access parallelism for random access read and write.
(b) A portable algorithm. 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. On a variety of tests executed on an Intel dual-core architecture it outperforms its lock-based STL counterpart and the concurrent vector provided by Intel by a factor of 10. The same tests executed on an 8-way AMD architecture with non-shared L2 cache indicate performance comparable to the best available lock-based techniques.
(d) An effective incorporation of non-blocking memory management and memory allocation schemes.

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] [14], a concurrent object is *non-blocking* 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*. Non-blocking (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. Common CAS implementations require three arguments: a memory location, Mem, an old value, V_{old}, and a new value,V_{new}. The instruction atomically exchanges the value stored in Mem with V_{new}, provided that its current value equals 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 [16]). 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 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.

2.1 Pragmatic Lock-Free Programming

The practical implementation of lock-free containers is known to be 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 [6], a single-word CAS operation is inadequate for the practical implementation of a non-trivial concurrent container. The use of a double-compare-and-swap primitive (DCAS) has been suggested by Detlefs et al. in [6], 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 [10], [20] double-ended queues [19], [27], stacks [12], hash tables [20], [25] 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 efficient 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 for our implementation.
(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 [1], minimize the time spent on CAS-based looping and the number of calls to CAS.

The lock-free vector's design and implementation provided follow the syntax and semantics of the ISO STL vector as defined in ISO C++ [18].

3 Algorithms

In this section we define a 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 Implementation Overview

The major challenges of providing a 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 [5] 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.

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 3.4) and an optional reference to a

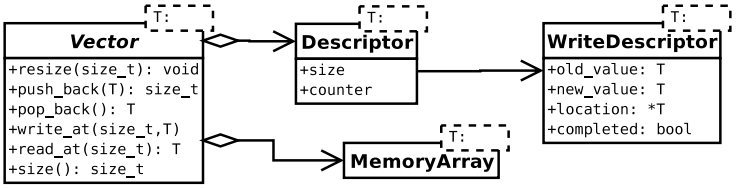

Fig. 1. Lock-free Vector. T denotes a data structure parameterized on T.

"`Write Descriptor`". Our approach requires that data types bigger than word size are indirectly stored through pointers. Like Intel's concurrent vector [24], our implementation avoids storage relocation and its synchronization hazards 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. The use of a "`Descriptor`" and "`WriteDescriptor`" (Barnes-style announcement [3]) 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 operation. Not counting memory management overhead, `push_back` executes *two successful CAS* instructions to update *two memory locations*.

3.2 Operations

Table 1 illustrates the implemented operations as well as their signatures, descriptor modifications, and runtime guarantees.

Table 1. Vector - Operations

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

The remaining part of this section presents the generalized pseudo-code of the implementation and 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 modern x86 architectures $HighestBit$ corresponds to the BSR assembly instruction. `FBS` is a constant representing the size of the first bucket and equals eight in our implementation.

Push_back (add one element to end). The first step is to complete a pending operation that the current descriptor might hold. In case that the storage capacity has reached its limit, new memory is allocated for the next memory bucket. Then, `push_back` defines a new "`Descriptor`" object and announces the current write operation. Finally, `push_back` uses 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.

Pop_back (remove one element from end). Unlike `push_back`, `pop_back` does not utilize a "`Write Descriptor`". It completes any pending operation of the current descriptor, reads the last element, defines a new descriptor, and attempts a CAS on the descriptor object.

Non-bound checking Read and Write at position i. The random access `read` and `write` do not utilize the descriptor and their success is independent of the descriptor's value.

Algorithm 1. pushback $vector, elem$

```
1: repeat
2:    desc_current ← vector.desc
3:    CompleteWrite(vector, desc_current.pending)
4:    bucket ← HighestBit(desc_current.size + FBS) − HighestBit(FBS)
5:    if vector.memory[bucket] = NULL then
6:       AllocBucket(vector, bucket)
7:    writeop ← new WriteDesc(At(desc_current.size)^, elem, desc_current.size)
8:    desc_next ← new Descriptor(desc_current.size + 1, writeop)
9: until CAS(&vector.desc, desc_current, desc_next)
10: CompleteWrite(vector, desc_next.pending)
```

Algorithm 2. AllocBucket $vector, bucket$

```
1: bucketsize ← FBS^(bucket+1)
2: mem ← new T[bucketsize]
3: if not CAS(&vector.memory[bucket], NULL, mem) then
4:    Free(mem)
```

Algorithm 3. Size $vector$

```
1: desc ← vector.desc
2: size ← desc.size
3: if desc.writeop.pending then
4:    size ← size − 1
5: return size
```

Reserve (increase allocated space). In the case of concurrently executing `reserve` operations, only one succeeds per bucket, while the others deallocate the acquired memory.

Algorithm 4. Read $vector, i$

1: **return** $At(vector, i)\hat{}$

Algorithm 5. Write $vector, i, elem$

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

Algorithm 6. popback $vector$

1: **repeat**
2: $desc_{current} \leftarrow vector.desc$
3: $CompleteWrite(vector, desc_{current}.pending)$
4: $elem \leftarrow At(vector, desc_{current}.size - 1)\hat{}$
5: $desc_{next} \leftarrow new\ Descriptor(desc_{current}.size - 1, NULL)$
6: **until** $CAS(\&vector.desc, desc_{current}, desc_{next})$
7: **return** $elem$

Algorithm 7. Reserve $vector, size$

1: $i \leftarrow HighestBit(vector.desc.size + \texttt{FBS} - 1) - HighestBit(\texttt{FBS})$
2: **if** $i < 0$ **then**
3: $i \leftarrow 0$
4: **while** $i < HighestBit(size + \texttt{FBS} - 1) - HighestBit(\texttt{FBS})$ **do**
5: $i \leftarrow i + 1$
6: $AllocBucket(vector, i)$

Algorithm 8. At $vector, i$

1: $pos \leftarrow i + \texttt{FBS}$
2: $hibit \leftarrow HighestBit(pos)$
3: $idx \leftarrow pos\ xor\ 2^{hibit}$
4: **return** $\&vector.memory[hibit - HighestBit(\texttt{FBS})][idx]$

Algorithm 9. CompleteWrite $vector, writeop$

1: **if** $writeop.pending$ **then**
2: $CAS(At(vector, writeop.pos), writeop.value_{old}, writeop.value_{new})$
3: $writeop.pending \leftarrow false$

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.

3.3 Semantics

The semantics of the vector's operations is based on a number of assumptions. 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* and to the latter as *non-descriptor modifying operations*. All of the vector's operations in the set of concurrent descriptor-modifying operations S_1 are thread-safe and lock-free. 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 the set of non-descriptor modifying operations S_2, all operations are thread-safe and wait-free. In this section, we omit the discussion of ABA problem related issues, typical to all CAS-based implementations. There are a number of well known techniques to overcome these problems, see section 3.5.

Correctness. The main correctness requirement of the semantics of the vector's operations is linearizability [15]. A concurrent operation is linearizable if it appears to execute instantaneously in some moment of time between the time point t_{inv} of its invocation and the time point t_{resp} of its response. Firstly, this definition implies that each concurrent history yields responses that are equivalent to the responses of some legal sequential history for the same requests. Secondly, the order of the operations within the sequential history must be consistent with the real-time order. Let us assume that there is an operation $o_i \in S_{vec}$, where S_{vec} is the set of all the vector's operations. We assume that o_i can be executed concurrently with n other operations $\{o_1, o_2..., o_n\} \in S_{vec}$. We outline a proof that operation o_i is linearizable.

Linearization Points. For all non-descriptor-modifying operations the linearization point is at the time instance t_a when the atomic `read` (Algorithm 4, line 1) or `write` (Algorithm 5, line 1) of the element is executed. Assume o_i is a descriptor-modifying operation. It is carried out in two stages: modify the "`Descriptor`" variable and then update the data structure's contents. Let us define time points t_{desc} (Algorithm 1, line 10; Algorithm 6, line 6) and $t_{writedesc}$ (Algorithm 9, line 2) 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 $o_c \in \{o_1, o_2..., o_n\}$, respectively. Similarly, time point $t_{readelem}$ (Algorithm 1, line 7; Algorithm 6, line 4) defines when o_i reads an element. o_i is either a `pop_back` or `push_back` operation. The linearization point is either $t_{readelem}$ or t_{desc} for the former case and $t_{readelem}$, t_{desc}, or $t_{writedesc}$ for the latter case.

Sequential Semantics. Let S_c be the set of all concurrent operations $\{o_1, ..., o_n\}$ in a time interval $[t_\alpha, t_\beta]$. If $\forall o_i \in S_c, DescriptorModifying(o_i)$, the linearization point for each operation is $t_{desc}(o_i)$. Similarly, if $\forall o_i \in S_c$, $NonDescriptorModifying(o_i)$, the linearization point for each operation is $t_a(o_i)$. In these cases, the resulting sequential histories are directly derived from the temporal order of the linearization points. In the remaining cases, the

derivation of a sequential history is significantly more complex. It is possible to transform all non-descriptor modifying operations into descriptor modifying in order to simplify the vector's sequential semantics. Given our current implementation, this can be achieved in a straightforward manner. We have chosen not to do so in order to preserve the efficiency and wait-freedom of the current non-descriptor modifying operations. Table 2 determines the linearization points for each pair of concurrent operations (o_1, o_2) where $DescriptorModifying(o_1)$ and $NonDescriptorModifying(o_2)$.

Table 2. Linearization Points of o_1, o_2

$o_1 \backslash o_2$	read	write
push_back	$t_{writedesc}(o_1), t_a(o_2)$	$t_{readelem}(o_1), t_a(o_2)$
pop_back	$t_{desc}(o_1), t_a(o_2)$	$t_{readelem}(o_1), t_a(o_2)$

We emphasize that the presented ordering relations are not transitive. Consider an example with three operations o_1 (`push_back`), o_2 (`write`), and o_3 (`read`), which access the same element. We assume that time points $t_a(o_2), t_a(o_3)$ occur between $t_{readelem}(o_1)$ and $t_{writedesc}(o_1)$ as well as that o_2 returns before the invocation of o_3. The resulting sequential history is o_1, o_2, o_3. It is derived from the real-time ordering between o_2 and o_3, and the pair-wise ordering relation between `push_back` and `write` in Table 2. A thorough linearizability proof for even the simplest data structure is non trivial and a further detailed elaboration is beyond the scope of this presentation.

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 insignificant for the outcome of operation o_1. The first thread attempting to change the descriptor will succeed, which guarantees the progress of the system.

3.4 Memory Management

Our algorithms do not require the use of a particular memory management scheme. A garbage collected environment would have significantly reduced the complexity of the implementation (by moving key implementation problems inside the GC implementation). However, we do not know of any available general lock-free garbage collector for C++.

Object Reclamation. Our concrete implementation uses reference counting as described by Michael and Scott [23]. The major drawback of this scheme is that 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. Alternatives such as Michael's hazard pointers [21] and Herlihy et al.'s pass the buck [14] overcome the problem.

Allocator. Recent research by Michael [22] and Gidenstam [9] presents implementations of true lock-free memory allocators. Due to its availability and performance, we selected Gidenstam's allocator for our performance tests.

3.5 The ABA Problem

The ABA problem is fundamental to all CAS-based systems [21]. 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`; in the vector's "`Descriptor`", `push_back` stores 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 multiple times.
(2) the memory allocator reuses the address of an already freed object.

A universal solution to the ABA problem is to associate a version counter to each element on platforms supporting CAS2. However, because of hardware requirements of our primary application domain, we cannot currently assume availability of CAS2.

To eliminate the ABA problem of (2) (in the absence of CAS2), we have incorporated a variation of Herlihy et al.'s pass the buck algorithm [14] utilizing a separate thread to periodically reclaim unguarded objects.

The vector's vulnerability to (1) (in the absence of CAS2), can be eliminated by requiring the data structure to copy all elements and store pointers to them. Such behavior complies with the STL value-semantics [26], however it can incur significant 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. However, for significant applications, our vector can be used because the application programmer can avoid ABA problem (1). For example, a vector of unique elements (e.g. a vector recording live or active objects) does not suffer this problem. Similarly, a vector that has a "growth phase" (using `push_back`) that is separate from a "write phase" (using assignment to elements) (e.g., an append-only vector) is safe.

3.6 Alternatives

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

Copy on Write. 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 [1]. Once completed, the pointer to the

map's 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. Harris et al. present a software multi-compare and swap (MCAS) implementation based on CAS instructions [11]. 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. 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. A straightforward solution is to apply descriptor-modifying semantics as discussed in section 3.3.

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 [17] 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 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 [17].

Similarly to the evaluation of other lock-free concurrent containers [7] and [20], 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 are 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 is plotted along the x-axis, while the time needed to complete all operations is shown along the y-axis. Both axes use logarithmic scale.

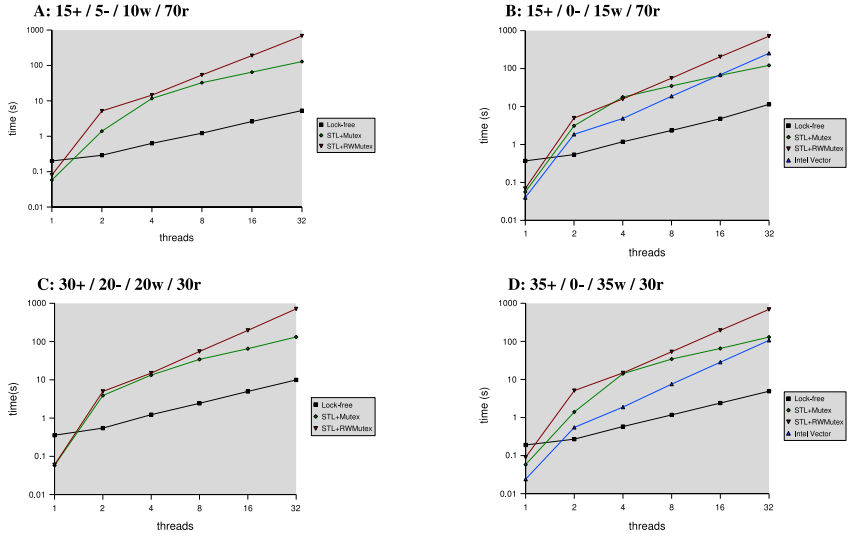

Fig. 2. Performance Results - Intel Core Duo

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 approaches. As indicated in [17], SpinLocks are volatile, unfair, and not scalable. 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 pre-allocated

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.

Under contention our non-blocking implementation consistently outperforms the alternative lock-based approaches in all possible operation mixes by a significantly large factor. It has also proved to be scalable as demonstrated by the performance analysis. 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.

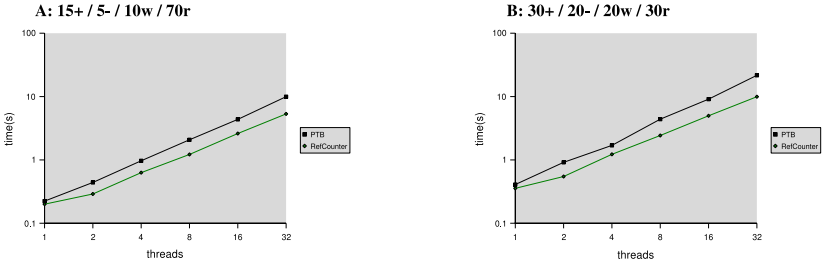

Fig. 3. Performance Results - Alternative Memory Management

As discussed in section 3.4, we have incorporated two different memory management approaches with our lock-free implementation, namely Michael and Scott's reference counting scheme (RefCount) and Herlihy et al.'s pass the buck technique (PTB). We have evaluated the vector's performance with these two different memory management schemes (Fig. 3).

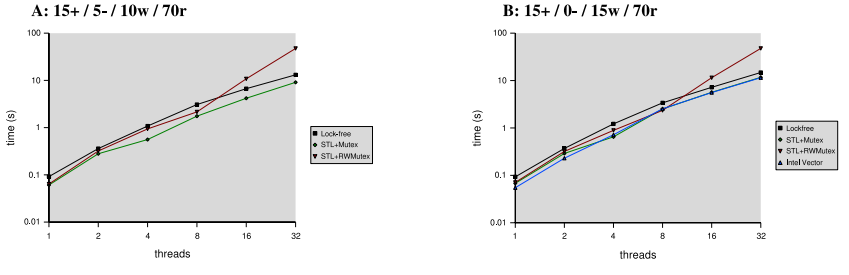

Fig. 4. Performance Results - AMD 8-way Opteron

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 quad dual core Opteron architecture with 1 MB L2 cache and 4GB shared RAM running the MS Windows 2003 operating system. (Fig.4). 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.4) demonstrates that the performance of our lock-free approach on such architectures is comparable to the performance of the best lock-based alternatives.

5 Conclusion

We presented a first practical and portable design and implementation of a lock-free dynamically resizable array. We developed an 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. An essential direction in our future work is to further optimize the effectiveness of our approach on various hardware architectures and to eliminate the remaining ABA problem. In addition, it is our goal to precisely specify the concurrent semantics of the remaining STL vector interface and incorporate them in our implementation.

Acknowledgements

We thank Kirk Reinholtz, Herb Sutter, Olga Pearce, Yuriy Solodkyy, Luke Wagner, and the anonymous referees for their helpful suggestions as well as the Adobe Photoshop team for providing us with access to their multi-core machines.

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 *LCPC '01*, pages 193–208, Cumberland Falls, Kentucky, Aug 2001.
3. G. Barnes. A method for implementing lock-free shared-data structures. In *SPAA '93: Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 261–270, New York, NY, USA, 1993. ACM Press.
4. P. Becker. Working Draft, Standard for Programming Language C++, ISO WG21N2009, April 2006.
5. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 2001.
6. D. Detlefs, C. H. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. L. Steele. Even better DCAS-based concurrent deques. In *DISC '00*, pages 59–73, 2000.
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*, 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 2005: LNCS, volume 3669*, pages 329–342, 2005.
10. T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC '01*, 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 *DISC '02*, 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, 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. Intel. Ia-32 intel architecture software developer's manual, volume 3: System programming guide, 2004.
17. Intel. Reference for Intel Threading Building Blocks, version 1.0, April 2006.
18. ISO/IEC 14882 International Standard. *Programming languages C++*. American National Standards Institute, September 1998.
19. M. Michael. CAS-Based Lock-Free Algorithm for Shared Deques. In *Euro-Par '03, LNCS volume 2790*, pages 651–660, 2003.
20. M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82, New York, NY, USA, 2002. ACM Press.
21. M. M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.
22. M. M. Michael. Scalable lock-free dynamic memory allocation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conf. on Programming language design and implementation*, pages 35–46, New York, NY, USA, 2004. ACM Press.
23. M. M. Michael and M. L. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, 1995.
24. A. Robison, Intel Corporation. Personal communication, April 2006.
25. 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.
26. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
27. H. Sundell and P. Tsigas. Lock-Free and Practical Doubly Linked List-Based Deques Using Single-Word Compare-and-Swap. In *OPODIS 2004: Principles of Distributed Systems, 8th Int. Conf., LNCS, volume 3544*, pages 240–255, 2005.