Scalable Nonblocking Concurrent Objects for Mission Critical Code

Damian Dechev
Texas A&M University,
College Station, TX 77843-3112, U.S.A.
dechev@tamu.edu

Bjarne Stroustrup
Texas A&M University,
College Station, TX 77843-3112, U.S.A.
bs@cs.tamu.edu

Abstract

The high degree of complexity and autonomy of future robotic space missions, such as Mars Science Laboratory (MSL), poses serious challenges in assuring their reliability and efficiency. Providing fast and safe concurrent synchronization is of critical importance to such autonomous embedded software systems. The application of nonblocking synchronization is known to help eliminate the hazards of deadlock, livelock, and priority inversion. The nonblocking programming techniques are notoriously difficult to implement and offer a variety of semantic guarantees and usability and performance trade-offs. The present software development and certification methodologies applied at NASA do not reach the level of detail of providing guidelines for the design of concurrent software. The complex task of engineering reliable and efficient concurrent synchronization is left to the programmer's ingenuity. A number of Software Transactional Memory (STM) approaches gained wide popularity because of their easy to apply interfaces, but currently fail to offer scalable nonblocking transactions. In this work we provide an in-depth analysis of the nonblocking synchronization semantics and their applicability in mission critical code. We describe a generic implementation of a methodology for scalable implementation of concurrent objects. Our performance evaluation demonstrates that our approach is practical and outperforms the application of nonblocking transactions by a large factor. In addition, we apply our Descriptor-based approach to provide a solution to the fundamental ABA problem. Our ABA prevention scheme, called the $\lambda\delta$ approach, outperforms by a large factor the use of garbage collection for the safe management of each shared location. It offers speeds comparable to the application of the architecture-specific CAS2 instruction used for version counting. The $\lambda\delta$ approach is an ABA prevention technique based on classification of concurrent operations and 3-step execution of a Descriptor object. A practical alternative to the application of CAS2 is particularly important for the engineering of embedded systems.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Algorithms, Languages, Reliability

Keywords nonblocking synchronization, C++, ABA problem prevention, software transactional memory, autonomous space software

OOPSLA'09, October 25–29, 2009, Orlando, Florida, USA.

1. Introduction

Robotic space mission projects, such as Mars Science Laboratory (MSL) [28], pose the challenging task of engineering some of the most complex real-time embedded software systems. The notion of concurrency is of critical importance for the design and implementation of such systems. The present software development and certification protocols do not reach the level of detail of offering guidelines for the engineering of reliable concurrent software. In this work, we provide a detailed analysis of the state-of-the-art *nonblocking* programming techniques and derive a generic implementation for scalable lightweight concurrent objects that can help in implementing *efficient* and *safe* concurrent interactions in mission critical code.

1.1 Nonblocking Objects

The most common technique for controlling the interactions of concurrent processes is the use of mutual exclusion locks. A mutual exclusion lock guarantees thread-safety of a concurrent object by blocking all contending threads trying to access it except the one holding the lock. In scenarios of high contention on the shared data, such an approach can seriously affect the performance of the system and significantly diminish its parallelism. For the majority of applications, the problem with locks is one of difficulty of providing *correctness* more than one of performance. The application of mutual exclusion locks poses significant safety hazards and incurs high complexity in the testing and validation of mission-critical software. Locks can be optimized in some scenarios by utilizing fine-grained locks or context-switching. Because of resource limitations, optimized lock mechanisms are not a desirable alternative for flight-qualified hardware [24]. Even for efficient locks, the interdependence of processes implied by the use of mutual exclusion introduces the dangers of *deadlock*, *livelock*, and *priority inversion*. The incorrect application of locks is hard to detect with the traditional testing procedures and a program can be deployed and used for a long period of time before the flaws become evident and eventually cause anomalous behavior.

To achieve higher safety and gain performance, we suggest the application of *nonblocking synchronization*. A concurrent object is *nonblocking* if it guarantees that *some* process in the system will make progress in a *finite* amount of steps [16]. An object that guarantees that *each* process will make progress in a *finite* number of steps is defined as *wait-free*. *Obstruction-freedom* [18] is an alternative nonblocking condition that ensures progress if a thread

eventually executes in *isolation*. It is the weakest nonblocking property and obstruction-free objects require the support of a contention manager to prevent livelocking.

1.2 Impact for Space Systems

Modern robotic space exploration missions, such as Mars Science Laboratory [28], are expected to embed a large array of advanced components and functionalities and perform a complex set of scientific experiments. The high degree of autonomy and increased complexity of such systems pose significant challenges in assuring the reliability and efficiency of their software. A survey on the challenges for the development of modern spacecraft software by Lowry [24] reveals that in July 1997 the Mars Pathfinder mission experienced a number of anomalous system resets that caused an operational delay and loss of scientific data. The follow-up analysis identified the presence of a priority inversion problem caused by the low-priority meteorological process blocking the the high-priority bus management process. The software engineers found out that it would have been impossible to detect the problem with the black box testing applied at the time. A more appropriate priority inversion inheritance algorithm had been ignored due to its frequency of execution, the real-time requirements imposed, and its high cost incurred on the slower flight-qualified computer hardware. The subtle interactions in the concurrent applications of the modern aerospace autonomous software are of critical importance to the system's safety and operation. The presence of a large number of concurrent autonomous processes implies an increased volume of interactions that are hard to predict and validate. Allowing fast and reliable concurrent synchronization is of critical importance to the design of autonomous spacecraft software.

1.3 Mission Data System

Mission Data System (MDS) [9] is the Jet Propulsion Laboratory's framework for designing and implementing complete end-to-end data and control autonomous flight systems. The framework focuses on the representation of three main software architecture principles:

(1) *System control*: a state-based control architecture with explicit representation of controllable states [8].

(2) *Goal-oriented operation*: control intent is expressed by defining a set of goals as part of a goal network [1].

(3) *Layered data management*: an integrated data management and transport protocols [29].

In MDS a state variable provides access to the data abstractions representing the physical entities under control over a continuous period of time, spanning from the distant past to the distant future. As explained by Wagner [29], the implementation's intent is to define a goal timeline overlapping or coinciding with the timeline of the state variables. Computing the guarantees necessary for achieving a goal might require the lookup of past states as well as the computation of projected future states. MDS employs the concept of goals to represent control intent. Goals are expressed as a set of temporal constraints [5]. Each state variable is associated with exactly one state estimator whose function is to collect all available data and compute a projection of the state value and its expected transitions. Control goals are considered to be those that are meant to control external physical states. Knowledge goals are those goals that represent the constraints on the software system regarding a property of a state variable. Not all states are known at all time. The most trivial knowledge goal is the request for a state to be known, thus enabling its estimator. A data state is defined as the information regarding the available state and goal data and its storage format and location. The MDS platform considers data states an integral part of the control system rather than a part of the system under control. There are dedicated state variables representing the data states. In addition, data states can be controlled through the definition of data goals. A data state might store information such as location, formatting, compression, and transport intent and status of the data. A data state might not be necessary for every state variable. In a simple control system where no telemetry is used, the state variable implementation might as well store the information regarding the variable's value history and its extrapolated states.

At its present state of design and implementation, MDS does not provide a concurrent synchronization mechanism for building safer and faster concurrent interactions. Elevating the level of efficiency and reliability in the execution of the concurrent processes is of particular significance to the implementation of the System Control and the Data Management modules of MDS. It is the goal of this paper to illustrate the trade-offs in the semantics and application of some advanced nonblocking techniques and analyze their applicability in MDS. The most ubiquitous and versatile data structure in the ISO C++ Standard Template Library (STL) [27] is *vector*, offering a combination of dynamic memory management and constant-time random access. Because of the vector's wide use and challenging parallel implementation of its nonblocking dynamic operations, we illustrate the efficiency of each nonblocking approach discussed in this work with respect to its applicability for the design and implementation of a shared nonblocking vector. A number of pivotal concurrent applications in the Mission Data System [20] framework employ a shared STL vector (in all scenarios protected by mutual exclusion locks). Such is the Data Management Service library described by Wagner in [29].

2. Nonblocking Data Structures

Lock-free and wait-free algorithms exploit a set of portable atomic primitives such as the word-size Compare-and-Swap (CAS) instruction [13]. The design of nonblocking data structures poses significant challenges and their development and optimization is a current topic of research [12], [16]. The Compare-And-Swap (CAS) atomic primitive (commonly known as Compare and Exchange, CMPXCHG, on the Intel *x86* and *Itanium* architectures [21]) is a CPU instruction that allows a processor to atomically test and modify a single-word memory location. CAS requires three arguments: a memory location (L_i), an old value (A_i), and a new value (B_i). The instruction atomically exchanges the value stored at L_i with B_i, provided that L_i's current value equals A_i. The result indicates whether the exchange was performed. For the majority of implementations the return value is the value last read from L_i (that is B_i if the exchange succeeded). Some CAS variants, often called Compare-And-Set, have a return value of type boolean. The hardware architecture ensures the atomicity of the operation by applying a fine-grained hardware lock such as a cache or a bus lock (as is the case for IA-32 [21]). The application of a CAS-controlled speculative manipulation of a shared location (L_i) is a fundamental programming technique in the engineering of nonblocking algorithms [16] (an example is shown in Algorithm 1). In our pseudocode we

Algorithm 1 CAS-controlled speculative manipulation of L_i

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

use the symbols ^, &, and . to indicate pointer dereferencing, obtaining an object's address, and integrated pointer dereferencing and field access. When the value stored at L_i is the control value of a CAS-based speculative manipulation, we call L_i and L_i^ *control location* and *control value*, respectively. We indicate the con-

trol value's type with the string value_type. The size of value_type must be equal or less than the maximum number of bits that a hardware CAS instruction can exchange atomically (typically the size of a single memory word). In the most common cases, value_type is either an integer or a pointer value. In the latter case, the implementor might reserve two extra bits per each control value and use them for implementation-specific value marking [12]. This is possible if we assume that the pointer values stored at L_i are aligned and the two low-order bits have been cleared. In Algorithm 1, the function fComputeB yields the new value B_i. Typically, B_i is a value directly derived from the function's arguments and is not dependent on the value stored at the control location. A routine where B_i's value is dependent on A_i would be a *read-modify* routine in contrast to the *modify* routine shown in Algorithm 1.

Linearizability [16] is a correctness condition for concurrent objects: a concurrent operation is linearizable if it appears to execute instantaneously in a given point of time τ_{lin} between the time τ_{inv} of its invocation and the time τ_{end} of its completion. The literature often refers to τ_{lin} as a *linearization point*. The implementations of many nonblocking data structures require the update of two or more memory locations in a linearizable fashion [4], [12]. The engineering of such operations (e.g. push_back and resize in a shared dynamically resizable array) is critical and particularly challenging in a CAS-based design. Harris et al. propose in [14] a software implementation of a multiple-compare-and-swap (MCAS) algorithm based on CAS. This software-based MCAS algorithm has been applied by Fraser in the implementation of a number of lock-free containers such as binary search trees and skip lists [11]. The cost of the MCAS operation is expensive requiring $2M + 1$ CAS instructions. Consequently, the direct application of the MCAS scheme is not an optimal approach for the design of lock-free algorithms. A common programming technique applied for the implementation of the complex nonblocking operations is the use of a *Descriptor Object* (Section 2.1).

A number of advanced Software Transactional Memory (STM) libraries provide nonblocking transactions with dynamic linearizable operations [7], [26]. Such transactions can be utilized for the design of nonblocking containers [26]. As our performance evaluation demonstrates, the high cost of the extra level of indirection and the conflict detection and validation schemes in STM systems does not allow performance comparable to that of a hand-crafted lock-free container that relies solely on the application of portable atomic primitives. Sections 2.4 and 2.7 describe in detail the implementation of a nonblocking shared vector using CAS-based techniques and STM, respectively. Section 3 provides analysis of the suggested implementation strategies and discusses the performance evaluation of the two approaches.

2.1 The Descriptor Object

The consistency model implied by the linearizability requirement is stronger than the widely applied Lamport's sequential consistency model [23]. According to Lamport's definition, sequential consistency requires that the results of a concurrent execution are equivalent to the results yielded by *some* sequential execution (given the fact that the operations performed by each individual processor appear in the sequential history in the order as defined by the program). The pseudocode in Algorithm 2 shows the two-step execution of a Descriptor Object. In our nonblocking design, a Descriptor Object stores three types of information:

(a) Global data describing the state of the shared container ($\upsilon\delta$), e.g. the size of a dynamically resizable array [4].

(b) A record of a pending operation on a given memory location. We call such a record requesting an update at a shared location L_i from an old value, old_val, to a new value, new_val, a *Write Descriptor ($\omega\delta$)*. The shortcut notation we use is $\omega\delta$ @ L_i : old_val $\rightarrow$ new_val. The fields in the Write Descriptor Object store the target location as well as the old and the new values.

(c) A boolean value indicating whether $\omega\delta$ contains a pending write operation that needs to be completed.

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

1. we read the value of the current reference to δ stored in L_δ (line 3);
2. if the current δ object contains a pending operation, we need to help its completion (lines 4-5);
3. we record the current value, A_i, at L_i (line 7) and compute the new value, B_i, to be stored in L_i (line 8);
4. a new $\omega\delta$ object is allocated on the heap, initialized (by calling $f_{\omega\delta}$), and its fields Target, OldValue, and NewValue are set (lines 9-12);
5. any additional state data stored in a Descriptor Object must be computed (by calling $f_{\upsilon\delta}$). Such data might be a shared element or a container's size that needs to be modified (line 13);
6. a new Descriptor Object is initialized containing the new Write Descriptor and the new descriptor's data. The new descriptor's *pending operation* flag (WDpending) is set to true (lines 14-15);
7. we attempt a swap of the old Descriptor Object with the new one (line 16). Should the CAS fail, we know that there is another process that has interrupted us and meanwhile succeeded to modify L_δ and progress. We need to go back at the beginning of the loop and repeat all the steps. Should the CAS succeed, we proceed with Step 2 and perform the update at L_i.

The size of a Descriptor Object is larger than a memory word. Thus, we need to store and manipulate a Descriptor Object through a reference. Since the control value of Step 1 stores a pointer to a Descriptor Object, to prevent ABA (Section 2.4.1), all references to descriptors must be memory managed by a safe nonblocking garbage collection scheme. We use the prefix μ for all variables that require safe memory management. In Step 2 we execute the Write Descriptor, WD, in order to update the value at L_i. Any interrupting thread (after the completion of Step 1) detects the pending flag of $\omega\delta$ and, should the flag's value be still positive, it proceeds to executing the requested update $\omega\delta$ @ $L_i : A_i \rightarrow B_i$. There is no need to perform a CAS-based loop and the execution of a single CAS execution is sufficient for the completion of $\omega\delta$. Should the CAS from Step 2 succeed, we have completed the two-step execution of the Descriptor Object. Should it fail, we know that there is an interrupting thread that has completed it already.

2.2 Nonblocking Concurrent Semantics

The use of a Descriptor Object provides the programming technique for the implementation of some of the complex nonblocking operations in a shared container, such as the push_back, pop_back, and reserve operations in a shared vector [4]. The use and execution of a Write Descriptor guarantees the linearizable update of two or more memory locations.

Algorithm 2 Two-step execution of a δ object

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

Definition 1: *An operation whose success depends on the creation and execution of a Write Descriptor is called an $\omega\delta$-executing operation.*

The operation push_back of a shared vector [4] is an example of an $\omega\delta$-executing operation. Such $\omega\delta$-executing operations have *lock-free* semantics and the progress of an individual operation is subject to the contention on the shared locations L_δ and L_i (under heavy contention, the body of the CAS-based loop from Step 1, Algorithm 2 might need to be re-executed). For a shared vector, operations such as pop_back do not need to execute a Write Descriptor [4]. Their progress is dependent on the state of the global data stored in the Descriptor, such as the size of a container.

Definition 2: *An operation whose success depends on the state of the $\upsilon\delta$ data stored in the Descriptor Object is a δ-modifying operation.*

A δ-modifying operation, such as pop_back, needs only update the shared global data (the data of type $\upsilon\delta$, such as size) in the Descriptor Object (thus pop_back seeks an atomic update of only one memory location: L_δ). Since an $\omega\delta$-executing operation by definition always performs an exchange of the entire Descriptor Object, every $\omega\delta$-executing operation is also δ-modifying. The semantics of a δ-modifying operation are *lock-free* and the progress of an individual operation is determined by the interrupts by other δ-modifying operations. An $\omega\delta$-executing operation is also δ-modifying but as is the case with pop_back, not all δ-modifying operations are $\omega\delta$-executing. Certain operations, such as the random access read and write in a vector [4], do not need to access the Descriptor Object and progress regardless of the state of the descriptor. Such operations are non-δ-modifying and have *wait-free* semantics (thus no delay if there is contention at L_δ).

Definition 3: *An operation whose success does not depend on the state of the Descriptor Object is a non-δ-modifying operation.*

2.2.1 Concurrent Operations

The semantics of a concurrent data structure can be based on a number of assumptions. Similarly to a number of fundamental studies in nonblocking design [16], [12], we assume the following premises: each processor can execute a number of operations. This establishes a *history* of invocations and responses and defines a *real-time order* between them. An operation O_1 is said to precede an operation O_2 if O_2*'s* invocation occurs after O_1*'s* response. Operations that do not have real-time ordering are defined as *concurrent*. A *sequential history* is one where all invocations have immediate responses. A *linearizable history* is one where: *a.* all invocations and responses can be reordered so that they are equivalent to a sequential history, *b.* the yielded sequential history must correspond to the semantic requirements of the sequential definition of the object, and *c.* in case a given response precedes an invocation in the concurrent execution, then it must precede it in the derived sequential history. It is the last requirement that differentiates the consistency model implied by the definition of linearizability with Lamport's sequential consistency model and makes linearizability stricter. When two δ-modifying operations (O_{δ_1} and O_{δ_2}) are concurrent, according to Algorithm 2, O_{δ_1} precedes O_{δ_2} in the linearization history if and only if O_{δ_1} completes Step 1, Algorithm 2 prior to O_{δ_2}.

Definition 4: *We refer to the instant of successful execution of the global Descriptor exchange at L_δ (line 16, Algorithm 2) as τ_δ.*

Definition 5: *A point in the execution of a δ object that determines the order of an $\omega\delta$-executing operation acting on location L_i relative to other writer operations acting on the same location L_i, is referred to as the $\lambda\delta$-point ($\tau_{\lambda\delta}$) of a Write Descriptor.*

The order of execution of the $\lambda\delta$-points of two concurrent $\omega\delta$-executing operations determines their order in the linearization history. The $\lambda\delta$-point does not necessarily need to coincide with the operation's linearization point, τ_{lin}. As illustrated in [4], τ_{lin} can vary depending on the operations' concurrent interleaving. The linearization point of a shared vector's [4] δ-modifying operation can be any of the three possible points: *a.* some point after τ_δ at which some operation reads data form the Descriptor Object, *b.* τ_δ or *c.* the point of execution of the Write Descriptor, τ_{wd} (the completion of Step 2, Algorithm 2). The core rule for a linearizable operation is that it must appear to execute in a single instant of time with respect to other concurrent operations. The linearization point need not correspond to a single fixed instruction in the body of the operation's implementation and can vary depending on the interrupts the operation experiences. In contrast, the $\lambda\delta$-point of an $\omega\delta$ object corresponds to a single instruction in the objects's implementation, thus making it easier to statically argue about an operation's correctness. In the pseudo code in Algorithm 2 $\tau_{\lambda\delta} \equiv \tau_\delta$.

2.3 Implementation Concerns

We provide a brief summary of the most important implementation concerns for the practical and portable design of a nonblocking data container.

2.3.1 Portability

Virtually at the core of every known synchronization technique is the application of a number of hardware atomic primitives. The semantics of such primitives vary depending on the specific hardware platform. There are a number of architectures that support some hardware atomic instructions that can provide greater flexibility such as the Load-Link/Store Conditional (LL/SC) supported by the PowerPC, Alpha, MIPS, and the ARM architectures or instructions that perform atomic writes to more than a single word in memory, such as the Double-Compare-And-Swap instruction (DCAS) [6]. The hardware support for such atomic instructions can vastly simplify the design of a nonblocking algorithm as well as offer immediate solutions to a number of challenging problems such as the ABA problem. To maintain portability across a large number of hardware platforms, the design and implementation of a nonblocking algorithm cannot rely on the support of such atomic primitives. The most common atomic primitive that is supported by a large majority of hardware platforms is the single-word CAS instruction. It is not desirable to suggest a CAS2/DCAS-based ABA solution for a CAS-based algorithm, unless the implementor explores the optimization possibilities of the algorithm upon the availability of CAS2/DCAS.

2.3.2 Linearizability Guarantee

In a CAS-based design, a major difficulty is meeting the linearizability requirements for operations that require the update of more than a single-word in the system's shared memory. To cope with this problem, it is possible to apply a combination of a number of known techniques:

a. *Extra Level of Indirection:* reference semantics [27] must be assumed in case the data being manipulated is larger than a memory word or the approach relies on the application of smart pointers or garbage collection for each individual element in the shared container.

b. *Descriptor Object:* a Descriptor stores a record of a pending operation on a given shared memory location. It allows the interrupting threads help the interrupted thread complete an operation rather than wait for its completion.

c. *Descriptive Log:* at the core of virtually all Software Transactional Memory implementations, the Descriptive Log stores a record of all pending reads and writes to the shared data. It is used for conflict detection, validation, and optimistic speculation.

d. *Transactional Memory:* a duplicate memory copy used to perform speculative updates that are invisible to all other threads until the linearization point of the entire transaction.

e. *Optimisitic Speculation:* complex nonblocking operations often employ optimistic speculative execution in order to carry out the memory updates on a local or duplicate memory copy and commit once there are no conflicts with interfering operations. It is necessary to employ a methodology for unrolling all changes performed by the speculating operation, should there be conflicts during the commit phase.

To illustrate the complexity of a nonblocking design of a shared vector, Table 1 provides an analysis of the number of memory locations that need to be updated upon the execution of some of its basic operations.

	Operations	Memory Locations
push_back	$Vector \times Elem \rightarrow void$	*2: element, size*
pop_back	$Vector \rightarrow Elem$	*1: size*
reserve	$Vector \times size\_t \rightarrow Vector$	*n: all elements*
read	$Vector \times size\_t \rightarrow Elem$	*none*
write	$Vector \times size\_t \times Elem \rightarrow Vector$	*1: element*
size	$Vector \rightarrow size\_t$	*none*

Table 1. Vector - Operations

2.3.3 Interfaces of the Concurrent Operations

According to the ISO C++ Standard [22], the STL containers' interfaces are inherently sequential. The next ISO C++ Standard [2] is going to include a concurrent memory model [3] and possibly a blocking threading library. In Table 2 we show a brief overview of some of the basic operations of an STL vector according to the current standard of the C++ programming language. Consider the sequence of operations applied to an instance, vec, of the STL vector: vec[vec.size()-1]; vec.pop_back();. In an environment with *concurrent* operations, we cannot have the guarantee that the element being deleted by thepop_back is going to be the element that had been read earlier by the invocation of operator[]. Such a *sequential* history is just one of the several legal sequential histories that can be derived from the concurrent execution of the above operations. While the STL interfaces have proven to be efficient and flexible for a large number of applications [27], to preserve the semantic behavior implied by the sequential definitions of STL, one can either rely on a library with atomic transactions [7], [26] or alternatively define concurrent STL interfaces adequate with respect to the applied consistency model. In the example we have shown, it might be appropriate to modify the interface of the pop_back operation and return the element being deleted instead of the void return type specified in STL. Such an implementation efficiently combines two operations: reading the element to be removed from the container and removing the element. The ISO C++ implementation of pop_back() returns void so that the operation is optimal (and does not perform extra work) in the most general case: the deletion of the tail element. Should we prefer to keep the STL standard interface of void pop_back() in a concurrent implementation, the task of obtaining the value of the removed element in a concurrent nonblocking execution might be quite costly and difficult to implement. By analyzing the shared containers' usage, we have observed that combining related operations can deliver better usability and performance advantages in a nonblocking implementation. Other possibly beneficial combinations of operations are 1) CAS-based read-modify-write at location L_i that unifies a random access read and write at location L_i and 2) the push_back of a block of tail elements. Furthermore, as demonstrated by Dechev et al. [5], the application of static analysis tools such as The Pivot [5] can help in shifting some part of the concurrent implementation's complexity from the run-time of the system to the compile-time program analysis stage. The Pivot is a compiler-independent platform for static analysis and semantics-based transformation of the complete ISO C++ programming language and some advanced language features proposed for the next generation C++ [2].

Operation	Description
size_type size() const	Number of elements in the vector
size_type capacity() const	Number of available memory slots
void reserve($size\_type\ n$)	Allocation of memory with capacity n
bool empty() const	true when size = 0
T* operator[] ($size\_type\ n$) const	returns the element at position n
T* front()	returns the first element
T* back()	returns the last element
void push_back(const T&)	inserts a new element at the tail
void pop_back()	removes the element at the tail
void resize(n, t = T())	modifies the tail, making size = n

Table 2. Interfaces of STL Vector

2.4 Descriptor-based Shared Vector

In this section we present a brief overview of the most critical lock-free algorithms employed by a Descriptor-based shared vector (see [4] for the full set of the operations of the first lock-free dynamically resizable array). To help tail operations update the size and the tail of the vector (in a linearizable manner), the design presented in [4] suggests the application of of a helper object, named Write Descriptor (WD) that announces a pending tail modifications and allows interrupting threads help the interrupted thread complete its operations. A pointer to the WD object is stored in the Descriptor together with the container's size and a reference counter required by the applied memory management scheme. The approach 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 remaining part of this section presents the pseudo-code of the tail operations (push_back and pop_back) and the random access operations (read and write at a given location within the vector's bounds).

Push_back (add one element to end): The first step is to complete a pending operation that the current descriptor might hold. In

Algorithm 3 push_back $vector, elem$

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

Algorithm 4 read $vector, i$

```
1: return At(vector, i)^
```

Algorithm 5 write $vector, i, elem$

```
1: At(vector, i) ^ ← elem
```

Algorithm 6 pop_back $vector$

```
1: repeat
2:     desc_current ← vector.desc
3:     CompleteWrite(vector, desc_current.pending)
4:     elem ← At(vector, desc_current.size-1)
5:     desc_next ← new Descriptor(desc_current.size-1, NULL)
6: until CAS(& vector.desc, desc_current, desc_next)
7: return elem
```

Algorithm 7 CompleteWrite $vector, wop$

```
1: if wop.pending then
2:     CAS(At(vector, wop.pos), wop.value_old, wop.value_new)
3:     wop.pending ← false
4: end if
```

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.

2.4.1 The ABA Problem

The ABA problem [25] is fundamental to all CAS-based systems. The ABA problem is a false positive execution of a CAS-based speculation on a shared location L_i (Algorithm 1). While of a simple nature and derived from the application of a basic hardware primitive, the ABA problem's occurrence is due to the intricate and complex interactions of the application's concurrent operations. The importance of the ABA problem has been reiterated in the recent years with the application of CAS for the development of nonblocking programming techniques. Avoiding the hazards of ABA imposes an extra challenge for a lock-free algorithm's design and implementation. To the best of our knowledge, the literature *does not offer an explicit and detailed analysis of the ABA problem*, its relation to the most commonly applied nonblocking programming techniques (such as the use of Descriptors) and correctness guarantees, and the possibilities for its detection and avoidance. Thus, at the present moment of time, eliminating the hazards of ABA in a nonblocking algorithm is left to the ingenuity of the software designer. As illustrated in Table 3, ABA can occur if a process P_1 is interrupted at any time after it has read the old value (A_i) and before it attempts to execute the CAS instruction from Algorithm 1. An interrupting process (P_k) might change the value at L_i to a new value, B_i. Afterwards, either P_k or any other process $P_j \neq P_1$ can eventually store A_i back to L_i. When P_1 resumes in Step 4, its CAS loop succeeds (false positive execution) despite the fact that L_i's value has been meanwhile manipulated.

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

Table 3. ABA at L_i

The ABA problem can corrupt the semantics of a Descriptor-based design of an unprotected nonblocking dynamic array [4]. As a common technique for overcoming the ABA problem it has been suggested to use a version tag attached to each value [15]. Such an approach demands the application of the architecture-specific CAS2 instruction. ABA avoidance on CAS-based architectures has been typically limited to two possible approaches:

1. split a 32-bit memory word into a value and a counter portions (thus significantly limiting the usable address space or the range of values that can be stored) [10],
2. apply value semantics (by utilizing an extra level of indirection, i.e. create a unique pointer to each value to be stored) in combination with a memory management approach that disallows the reuse of potentially hazardous memory locations [17], [25] (thus impose a significant performance overhead).

As reflected in our performance test results (Section 2.5), the usage of both, an extra level of indirection as well as the heavy reliance on a nonblocking GC scheme for managing the Descriptor Objects *and* the references to value_type objects, is very expensive with respect to the space and time complexity of a nonblocking algorithm. In the following section we present a 3-step execution of the Descriptor Object approach that helps us eliminate ABA, avoid the need for an extra level of indirection, and reduce the usage of the computationally expensive GC scheme.

2.5 ABA Prevention for Descriptor-based Lock-free Designs

In this section we study in detail and define the conditions that lead to ABA in a nonblocking Descriptor-based design. We investigate the relationship between the ABA hazards and the most commonly applied nonblocking programming techniques and correctness guarantees. Based on our analysis, we define a generic and practical condition, called the $\lambda\delta$ approach, for ABA avoidance for lock-free linearizable designs. The $\lambda\delta$ approach provides ABA prevention by the categorization of the concurrent operations. For complex operations (in a Descriptor-based design), the $\lambda\delta$ approach identifies the critical point in an operation's execution and suggests a 3-step execution strategy for ABA avoidance.

Let us designate the point of time when a certain δ-modifying operation reads the state of the Descriptor Object by τ_{read_δ}, and the instants when a thread reads a value from and writes a value into a location L_i by τ_{access_i} and τ_{write_i}, respectively. Table 4 demonstrates the occurrence of ABA in the execution of a δ object

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

Table 4. ABA occurrence in the execution of a Descriptor Object

with two concurrent δ-modifying operations (O_{δ_1} and O_{δ_2}) and a concurrent write, O_i, at L_i. We assume that the δ object's implementation follows Algorithm 2. The execution of O_{δ_1}, O_{δ_2}, and O_i proceeds in the following manner:

(1) O_{δ_1} reads the state of the current δ object as well as the current value at L_i, A_i (Steps 1-2, Table 4). Next, O_{δ_1} proceeds with instantiating a new δ object and replaces the old descriptor with the new one (Step 3, Table 4).

(2) O_{δ_1} is interrupted by O_{δ_2}. O_{δ_2} reads L_δ and finds the $WDpending$ flag's value to be $true$ (Step 4, Table 4).

(3) O_{δ_1} resumes and completes the execution of its δ object by storing B_i into L_i (Step 5, Table 4).

(4) An interrupting operation, O_i, writes the value A_i into L_i (Step 6, Table 4).

(5) O_{δ_2} resumes and executes $\omega\delta$ it has previously read, the $\omega\delta$'s CAS falsely succeeds (Step 6, Step 4).

The placement of the $\lambda\delta$-point plays a critical role for achieving ABA safety in the implementation of an $\omega\delta$-executing operation. The $\lambda\delta$-point in Table 4 guarantees that the $\omega\delta$-executing operation O_{δ_1} completes before O_{δ_2}. However, at the time τ_{wd} when O_{δ_2} executes the write descriptor, O_{δ_2} has no way of knowing whether O_{δ_1} has completed its update at L_i or not. Since O_{δ_1}'s $\lambda\delta$-point $\equiv \tau_\delta$, the only way to know about the status of O_{δ_1} is to read L_δ. Using a single-word CAS operation prevents O_{δ_2} from atomically checking the status of L_δ and executing the update at L_i.

Definition 6: *A concurrent execution of one or more non-$\omega\delta$-executing δ-modifying operations with one $\omega\delta$-executing operation, O_{δ_1}, performing an update at location L_i is* ABA-free *if O_{δ_1}'s $\lambda\delta$-point $\equiv \tau_{access_i}$. We refer to an $\omega\delta$-executing operation where its $\lambda\delta$-point $\equiv \tau_{access_i}$ as a $\lambda\delta$-modifying operation.*

Assume that in Table 4 the O_{δ_1}'s $\lambda\delta$-point $\equiv$ τ_{access_i}. As shown in Table 4, the ABA problem in this scenario occurs when there is a hazard of a spurious execution of O_{δ_1}'s Write Descriptor. Having a $\lambda\delta$-modifying implementation of O_{δ_1} allows any non-$\omega\delta$-executing δ-modifying operation such as O_{δ_2} to check O_{δ_1}'s progress while attempting the atomic update at L_i requested by O_{δ_1}'s Write Descriptor. Our *3-step descriptor execution* approach offers a solution based on Definition 6. In an implementation with two or more concurrent $\omega\delta$-executing operations, each $\omega\delta$-executing operation must be $\lambda\delta$-modifying in order to eliminate the hazard of a spurious execution of an $\omega\delta$ that has been picked up by a collaborating operation.

In Algorithm 8 we suggest a design strategy for the implementation of a $\lambda\delta$-modifying operation. Our approach is based on a 3-step execution of the δ object. While similar to Algorithm 2, the approach shown in Algorithm 8 differs by executing a fundamental additional step: in Step 1 we store a pointer to the new descriptor in L_i prior to the attempt to store it in L_δ in Step 2. Since all δ objects are memory managed, we are guaranteed that no other thread would attempt a write of the value μNewDesc in L_i or any other shared memory location. The operation is $\lambda\delta$-modifying because, after the new descriptor is placed in L_i, any interrupting writer thread accessing L_i is required to complete the remaining two steps in the execution of the Write Descriptor. However, should the CAS execution in Step 2 (line 28) fail, we have to unroll the changes at L_i performed in Step 1 by restoring L_i's old value preserved in WD.OldElement (line 20) and retry the execution of the routine (line 21). To implement Algorithm 8, it is necessary to distinguish between objects of type value_type and δ. A possible solution is to require that all value_type values are pointers and all pointer values stored in L_i are aligned with the two low-order bits cleared during their initialization. That way, we can use the two low-order bits for designating the type of the pointer values. Subsequently, every read must check the type of the pointer obtained from a shared memory location prior to manipulating it. Once an operation succeeds at completing Step 1, Algorithm 8, location L_i contains a pointer to a δ object that includes both: L_i's previous value of type value_type and a write descriptor WD that provides a record for the steps necessary for the operation's completion. Any non-δ-modifying operation, such as a random access read in a shared vector, can obtain the value of L_i (of type value_type) by accessing WD.OldElement (thus going through a *temporary* indirection) and ignore the Descriptor Object. Upon the success of Step 3, Algorithm 8, the temporary level of indirection is eliminated. Such an approach would preserve the wait-free execution of a non-δ-modifying operation. The $\omega\delta$ data type needs to be amended to include a field TempElement (line 9, Algorithm 8) that records the value of the temporary δ pointer stored in L_i. The cost of the $\lambda\delta$ operation is 3 CAS executions to achieve the linearizable update of two shared memory locations (L_i and L_δ). The implementation of our $\lambda\delta$-modifying operation as shown in Algorithm 8 is similar to the execution of Harris et al.'s MCAS algorithm [14]. Just like our $\lambda\delta$-modifying approach, for an MCAS update of L_δ and L_i, the cost of Harris et al.'s MCAS is at least 3 executions of the single-word CAS instruction. Harris et al.'s work on MCAS [14] brings forward a significant contribution in the design of lock-free algorithms, however, it lacks any analysis of the hazards of ABA and the way the authors manage to avoid it.

2.5.1 Performance Evaluation of the ABA Prevention Scheme

We incorporated the presented ABA elimination approach in the implementation of the nonblocking dynamically resizable array as discussed in Section 2.4. Our test results indicate that the $\lambda\delta$ approach offers ABA prevention with performance comparable to the use of the platform-specific CAS2 instruction to implement version counting. This finding is of particular value to the engineering of some embedded real-time systems where the hardware does not support complex atomic primitives such as CAS2 [24]. We ran performance tests on an Intel IA-32 SMP machine with two 1.83 GHz processor cores with 512 MB shared memory and 2 MB L2 shared cache running the MAC 10.5.6 operating system. In our performance analysis we compare:

(1) $\lambda\delta$ *approach:* the implementation of a vector with a $\lambda\delta$-modifying push_back and a δ-modifying pop_back. Table 5 shows that in this scenario the cost of push_back is three single-word CAS operations and pop_back's cost is one single-word CAS instruction.

(2) *All-GC approach:* the use of an extra level of indirection and memory management for each element. Because of its performance and availability, we have chosen to implement and apply Herlihy et al.'s Pass The Buck algorithm [19]. In addition, we use Pass The Buck to protect the Descriptor Objects for all of the tested approaches.

Algorithm 8 Implementing a $\lambda\delta$-modifying operation through a three-step execution of a δ object

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

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

Table 5 offers an overview of the shared vector's operations' relative cost in terms of number and type of atomic instructions invoked per operation. We varied the number of threads, starting from 1

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

Table 5. A Shared Vector's Operations Cost (Best Case Scenario)

and exponentially increased their number to 64. Each thread executed 500,000 lock-free operations on the shared container. We measured the execution time (in seconds) that all threads needed to complete. Each iteration of every thread executed an operation with a certain probability (push_back (+), pop_back (−), random access write (w), random access read (r)). We show the performance graph for a distribution of +:40%, −:40%, w:10%, r:10% on Figure 1. Figure 2 demonstrates the performance results with less contention at the vector's tail, +:25%, −:25%, w:10%, r:40%. Figure 3 illustrates the test results with a distribution containing predominantly random access read and write operations, +:10%, −:10%, w:40%, r:40%. Figure 4 reflects our performance evaluation on a vector's use with mostly random access read operations:

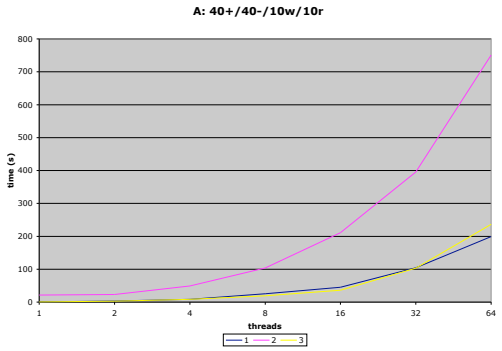

Figure 1. Performance Results A

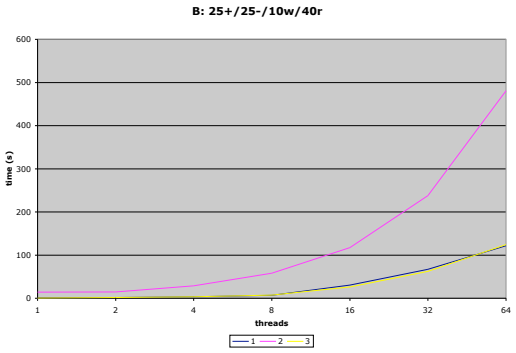

Figure 2. Performance Results B

+:20%, −:0%, w:20%, r:60%, a scenario often referred to as the most common real-world use of a shared container [11]. The number of threads is plotted along the x-axis, while the time needed to complete all operations is shown along the y-axis. According to the performance results, compared to the *All-GC* approach, the $\lambda\delta$ approach delivers consistent performance gains in all possible operation mixes by a large factor, a factor of at least 3.5 in the cases with less contention at the tail and a factor of 10 or more when there is a high concentration of tail operations. These observations come as a confirmation to our expectations that introducing an extra level of indirection and the necessity to memory manage each individual element with PTB (or an alternative memory management scheme) to avoid ABA comes with a pricy performance overhead. The $\lambda\delta$ approach offers an alternative by classifying the concurrent operations and introducing the notion of a $\lambda\delta$-point. The application of version counting based on the architecture-specific CAS2 operation is the most commonly cited approach for ABA prevention in the literature [15], [19]. Our performance evaluation shows that the $\lambda\delta$ approach delivers performance comparable to the use of CAS2-based version counting. CAS2 is a complex atomic primitive and its application comes with a higher cost when compared to the application of atomic write or a single-word CAS. In the performance tests executed, we notice that in the scenarios where the random access write is invoked more frequently (Figures 3 and 4), the per-

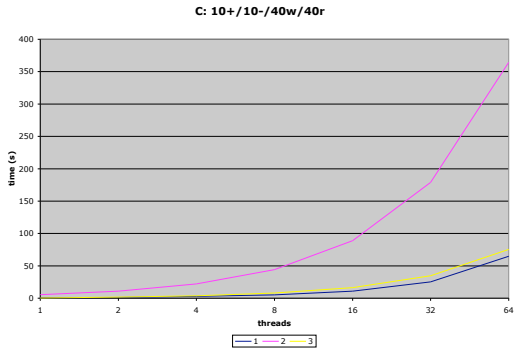

Figure 3. Performance Results C

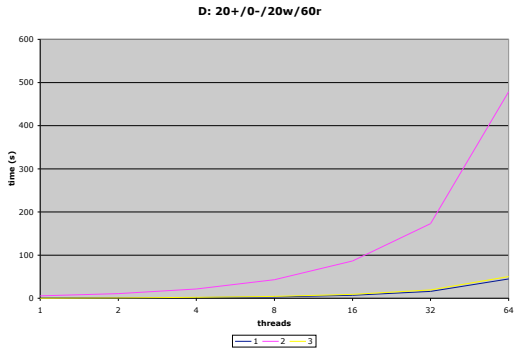

Figure 4. Performance Results D

formance of the CAS2 version counting approach suffers a performance penalty and runs slower than the $\lambda\delta$ approach by about 12% to 20%. According to our performance evaluation, the $\lambda\delta$ approach is a systematic, effective, portable, and generic solution for ABA avoidance. The $\lambda\delta$ scheme does not induce a performance penalty when compared to the architecture-specific application of CAS2-based version counting and offers a considerable performance gain when compared to the use of *All-GC*.

2.6 STM-based Nonblocking Design

A variety of recent STM approaches [7], [26] claim safe and easy to use concurrent interfaces. The most advanced STM implementations allow the definition of efficient "large-scale" transactions, i.e. *dynamic* and *unbounded* transactions. *Dynamic transactions* are able to access memory locations that are not statically known. *Unbounded transactions* pose no limits on the number of locations being accessed. The basic techniques applied are the utilization of public records of concurrent operations and a number of conflict detection and validation algorithms that prevent side-effects and race conditions. To guarantee progress transactions help those ahead of them by examining the public log record. The availability of nonblocking, unbounded, and dynamic transactions provides an alternative to CAS-based designs for the implementation of nonblocking data structures. The complex designs of such advanced STMs often come with an associated cost:

a. *Two Levels of Indirection:* A large number of the nonblocking designs require two levels of indirection in accessing data.

b. *Linearizability:* The linearizability requirements are hard to meet for an unbounded and dynamic STM. To achieve efficiency and reduce the complexity, all known nonblocking STMs offer the less demanding *obstruction-free* synchronization [18].

c. *STM-oriented Programming Model:* The use of STM requires the developer to be aware of the STM implementation and apply an STM-oriented Programming Model. The effectiveness of such programming models is a topic of current discussions in the research community.

d. *Closed Memory Usage:* Both nonblocking and lock-based STMs often require a *closed memory system* [7].

e. *Vulnerability of Large Transactions:* In a nonblocking implementation large transactions are a subject to interference from contending threads and are more likely to encounter conflicts. Large blocking transactions can be subject to time-outs, requests to abort or introduce a bottleneck for the computation.

f. *Validation:* A validation scheme is an algorithm that ensures that none of the transactional code produces side-effects. Code containing I/O and exceptions needs to be reworked as well as some class methods might require special attention. Consider a class hierarchy with a base class *A* and two derived classes *B* and *C*. Assume *B* and *C* inherit a virtual method *f* and B's implementation is side-effect free while C's is not. A validation scheme needs to disallow a call to C's method *f*.

With respect to our design goals, the main problems associated with the application of STM are meeting the stricter requirements posed by the lock-free progress and safety guarantees and the overhead introduced by the application of an extra level of indirection and the costly conflict detection and validation schemes.

2.6.1 Obstruction-free Descriptor vs Lock-free Descriptor

To be able to reduce the complexity of implementing nonblocking transactions, the available nonblocking STM libraries often provide the weaker obstruction-free progress guarantee. Even for experienced software designers, understanding the subtle differences between lock-free and obstruction-free designs is challenging. To better illustrate how obstruction-free objects differ from lock-free objects, in Algorithm 9 we demonstrate the implementation of an obstruction-free Descriptor Object. While similar to the execution of a lock-free Descriptor Object (Section 2.1), the obstruction-free Descriptor object from Algorithm 9 differs in two significant ways:

(1) *No thread collaboration when executing the Write Descriptor:* interrupting threads need not help interrupted threads complete. Obstruction-free execution guarantees that a thread will complete eventually in isolation. Thus, every time a thread identifies an interrupt it can simply repeat its update routine until the sequence of instructions completes without interrupts. Intuitively, a larger number of instructions in the execution routine implies a higher risk of interrupts.

(2) *Unrolling:* when an attempted update at L_δ fails, the operation needs to invoke a mechanism for unrolling any modifications it had performed to shared memory. In our obstruction-free Descriptor, the Write Descriptor stores the necessary information to execute an undo of Step 1, Algorithm 9 should the attempted update at L_δ in Step 2, Algorithm 9 fail. The unrolling approach requires that we store two types of objects in a shared location L_i: elements of type value_type and Write Descriptors of

Algorithm 9 Implementing a Descriptor Object with obstruction-free semantics

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

type $\omega\delta$. To distinguish between these two types of objects, we need to employ bit marking of the unused low-order bits for all Write Descriptors objects temporarily stored at L_i. When an interrupting thread recognizes an $\omega\delta$ object, it can simply ignore its presence and obtain the element (of type value_type) by reading the field OldElement of the Write Descriptor.

Obstruction-free objects that follow a design similar to Algorithm 9 eliminate the overhead an interrupting thread might experience when helping an interrupted thread. However, in scenarios with high contention, obstruction-free objects might experience frequent interrupts that could result in poor scalability and even livelocking.

2.7 RSTM-based Vector

The Rochester Software Transactional Memory (RSTM) [26] is a word- and indirection-based C++ STM library that offers obstruction-free nonblocking transactions. As explained by the authors in [26], while helping provide lightweight committing and aborting of transactions, the extra level of indirection can cause a dramatic performance degradation due to the more frequent capacity and coherence misses in the cache. In this section we employ the RSTM library (version 4) to build an STM-based nonblocking shared vector. We chose RSTM for our experiment because of its flexible and efficient object-oriented C++ design, demonstrated high performance when compared to alternative STM techniques, and the availability of nonblocking transactions. In Algorithms 10, 11, 12, and 13, we present the RSTM-based implementations of the read, write, pop_back, and push_back operations, respectively. According to the RSTM API [26], access to shared data is achieved by utilizing four classes of shared pointers: 1) a *shared object* (class sh_ptr¡T¿) representing an object that is untouched by a transaction, 2) a *read only object* (class rd_ptr¡T¿) referring to an object that has been opened for reading, 3) a *writable object* (class wr_ptr¡T¿) pointing to an object opened for writing by a transaction, and 4) a *privatized object* (class un_ptr¡T¿) representing an object that can be accessed by one thread at a time. These smart pointer templates can be instantiated only with data types derived from a core RSTM object class stm::Object. Thus, we need to wrap each element stored in the shared vector in a class STMVectorNode that derives from stm::Object. Similarly, we define a Descriptor class STMVectorDesc (derived from stm::Object) that stores the container-specific data such as the vector's size and capacity. The tail operations need to modify (within a single transaction) the last element and the Descriptor object (of type STMVectorDesc) that is stored in a location L_{desc}. The vector's memory array is named with the string mem. In the pseudo-code in Algorithms 12 and 13 we omit the details related to the management of mem (such as the resizing of the shared vector should the requested size exceed the container's capacity).

Algorithm 10 RSTM vector, operation read location p

```
1: BEGIN_TRANSACTION
2: rd_ptr< STMVectorNode > rp(mem[p])
3: result = rp→value
4: END_TRANSACTION
5: return result
```

Algorithm 11 RSTM vector, operation write v at location p

```
1: BEGIN_TRANSACTION
2: wr_ptr< STMVectorNode > wp(mem[p])
3: wp→val = v
4: sh_ptr< STMVectorNode > nv =
   new sh_ptr< STMVectorNode >(wp)
5: mem[p] = nv
6: END_TRANSACTION
```

Algorithm 12 RSTM vector, operation pop_back

```
1: BEGIN_TRANSACTION
2: rd_ptr< STMVectorNode > rp(mem[L_desc→size-1])
3: sh_ptr< STMVectorDesc > desc =
   new sh_ptr< STMVectorDesc >
   (new STMVectorDesc(L_desc→size-1))
4: result = rp→value
5: L_desc = desc
6: END_TRANSACTION
7: return result
```

Algorithm 13 RSTM vector, operation push_back v

```
1: BEGIN_TRANSACTION
2: sh_ptr< STMVectorNode > nv =
   new sh_ptr< STMVectorNode >(new STMVectorNode(v))
3: sh_ptr< STMVectorDesc > desc =
   new sh_ptr< STMVectorDesc >
   (new STMVectorDesc(L_desc→size+1))
4: mem[size] = nv
5: L_desc = desc
6: END_TRANSACTION
```

3. Analysis and Results

To evaluate the performance of the discussed synchronization techniques, we analyze the performance of three approaches for the implementation of a shared vector:

(1) The RSTM-based nonblocking vector implementation as presented in Section 2.7.

(2) An RSTM lock-based execution of the vector's transactions. RSTM provides the option of running the transactional code in a lock-based mode using redo locks [26]. Though blocking and not meeting our goals for safe and reliable synchronization, we include the lock-based RSTM vector execution to gain additional insight about the relative performance gains or penalties that the discussed nonblocking approaches offer when compared to the execution of a lock-based, STM-based container.

(3) The hand-crafted Descriptor-based approach as presented in Section 2.4.

We ran performance tests on an Intel IA-32 SMP machine with two 1.83 GHz processor cores with 512 MB shared memory and 2 MB L2 shared cache running the MAC 10.5.6 operating system.

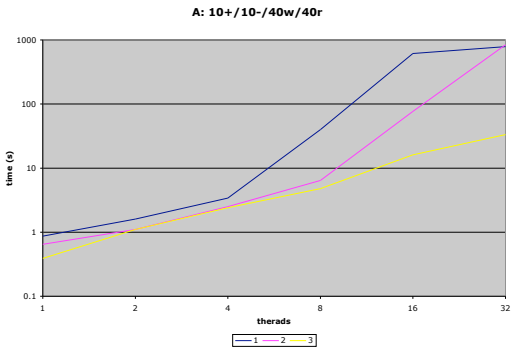

Figure 5. Performance Results I

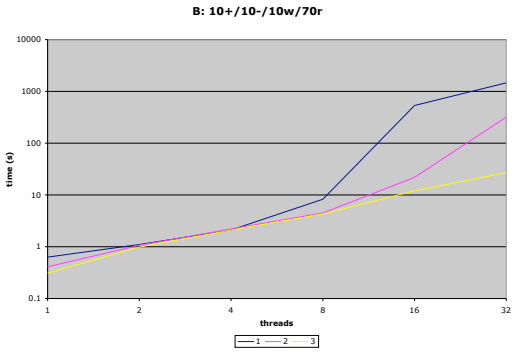

Figure 6. Performance Results II

We designed our experiments by generating a workload of the various operations. We varied the number of threads, starting from 1 and exponentially increased their number to 32. Each thread executed 500,000 lock-free operations. We measured the execution time (in seconds) that all threads needed to complete. Each iteration of every thread executed an operation with a certain probability (`push_back` (`+`), `pop_back` (`-`), random access `write` (`w`), random access `read` (`r`)). We show the performance graph for a distribution of `+`:10%, `-`:10%, `w`:40%, `r`:40% on Figure 5. Figure 6 demonstrates the performance results in a read-many-write-rarely scenario, `+`:10%, `-`:10%, `w`:10%, `r`:70%. Figure 7 illustrates the test results with a distribution `+`:25%, `-`:25%, `w`:12%, `r`:38%. The number of threads is plotted along the x-axis, while the time needed to complete all operations is shown along the y-axis. To increase the readability of the performance graphs, the y-axis uses a logarithmic scale with a base of 10. Our test results indicate that for the

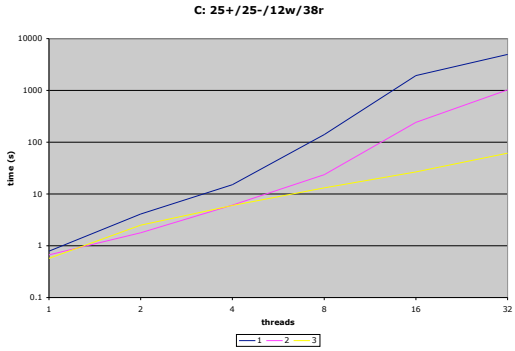

Figure 7. Performance Results III

large majority of scenarios the hand-crafted CAS-based approach outperforms by a significant factor the transactional memory approaches. The Descriptor-based approach offers simple application and fast execution. The STM-based design offers a flexible programming interface and easy to comprehend concurrent semantics. The main deterrent associated with the application of STM is the overhead introduced by the extra level of indirection and the application of costly conflict detection and validation schemes. According to our performance evaluation, the nonblocking RSTM vector demonstrates poor scalability and its performance progressively deteriorates with the increased volume of operations and active threads in the system. In addition, RSTM transactions offer obstruction-free semantics. To eliminate the hazards of livelocking, the software designers need to integrate a contention manager with the use of an STM-based container. Because of the limitations present in the state of the art STM libraries [26], [7], we suggest that a shared vector design based on the utilization of nonblocking CAS-based algorithms can better serve the demands for safe and reliable concurrent synchronization in mission critical code.

4. Conclusion

In this work we investigated how the application of nonblocking synchronization can help deliver more efficient and reliable concurrent synchronization in mission critical applications, such as the Mission Data System Project. We studied the challenging process of how to design and implement a nonblocking data container by applying 1) CAS-based synchronization and 2) Software Transactional Memory. We discussed the principles of nonblocking synchronization and demonstrated the application of both approaches by showing the implementation of a lock-free shared vector. Our performance evaluation concluded that while difficult to design, CAS-based algorithms offer fast and scalable performance and in a large majority of scenarios outperform the alternative STM-based nonblocking or lock-based approaches by a significant factor. To facilitate the design of future lock-free algorithms and data structures, we provided a generic implementation of a Descriptor Object. Understanding the subtle differences between the two main types of nonblocking objects, lock-free objects and obstruction-free objects, is a challenging task even to the most experienced software engineers. To better illustrate the advantages and shortcomings of each progress guarantee, we provided an obstruction-free design of the Descriptor object and contrasted its execution to its lock-free counterpart. In addition, we studied the ABA problem and the conditions

leading to its occurrence in a Descriptor-based lock-free linearzibale design. We offered a systematic and generic solution, called the $\lambda\delta$ approach, that outperforms by a significant factor the use of garbage collection for the safe management of each shared location and offers speed of execution comparable to the application of the architecture-specific CAS2 instruction used for version counting. The literature does not offer a detailed analysis of the ABA problem and the general techniques for its avoidance in a lock-free linearizable design. Currently, the challenges of eliminating ABA is left to the ingenuity of the software designer. Having a practical alternative to the application of the architecture-specific CAS2 is of particular importance to the design of some modern embedded systems such as Mars Science Laboratory. This paper aimed at delivering better understanding of the advantages (over mutual exclusion) as well as the usability and performance trade-offs of the modern nonblocking programming techniques that can be of critical importance for the engineering of reliable and efficient concurrent flight software.

Acknowledgments

We thank our colleagues William K. Reinholtz, Nicloas Rouquette, and David Wagner from the Jet Propulsion Laboratory's Mission Data System team and Peter Pirkelbauer and Jaakko Järvi from Texas A&M University for their collaboration and provided directions on this work.

References

[1] Barett, A., Knight, R., Morris, J. and Rasmussen, R., Mission Planning and Execution Within the Mission Data System, Proceedings of the International Workshop on Planning and Scheduling for Space, 2004.

[2] Becker, P., Working Draft, Standard for Programming Language C++, ISO WG21 N2009, `http://www.open-std.org/JTC1/SC22/WG21/`, April, 2006.

[3] Boehm, H. and Adve, S., Foundations of the C++ Concurrency Memory Model, PLDI '08: Proceedings of the ACM SIGPLAN 2008 conference on Programming language design and implementation, ACM Press, 2008.

[4] Dechev, D., Pirkelbauer, P. and Stroustrup, B., Lock-Free Dynamically Resizable Arrays, Proceedings of 10th International Conference on Principles of Distributed Systems, OPODIS 2006, Bordeaux, France, December 12-15, 2006, Lecture Notes in Computer Science, Springer, Volume 4305, pages 142-156.

[5] Dechev, D., Rouquette, N., Pirkelbauer, P. and Stroustrup, B., Verification and Semantic Parallelization of Goal-Driven Autonomous Software, Proceedings of ACM Autonomics 2008: 2nd International Conference on Autonomic Computing and Communication Systems, 2008.

[6] Detlefs, D., Flood, C., Garthwaite, A., Martin, P., Shavit, N. and Steele Jr., G., Even Better DCAS-Based Concurrent Deques, International Symposium on Distributed Computing, DISC 2000.

[7] Dice, D. and Shavit, N., Understanding Tradeoffs in Software Transactional Memory, Proceedings of the 2007 International Symposium on Code Generation and Optimization (CGO), San Jose, CA, 2007.

[8] Dvorak, D., Rasmussen, R. and Starbird, T., State Knowledge Representation in the Mission Data System, Proceedings of IEEE Aerospace Conference, 2002.

[9] Dvorak, D., Ingham, M., Morris, J., and Gersh, J., Goal-Based Operations: An Overview, Proceedings of AIAA Infotech, 2007.

[10] Dvorak, D. and Reinholtz, K. Hard real-time: C++ versus RTSJ, OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, 2004.

[11] Fraser, K., Practical lock-freedom, Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, 2004, ISSN 1476-2986.

[12] Fraser, K. and Harris, T., Concurrent programming without locks, ACM Trans. Comput. Syst., Volume 25, Number 2, 2007, ISSN 0734-2071.

[13] Gifford, D. and Spector, A., Case study: IBM's system/360-370 architecture, Commun. ACM, Volume 30, Number 4, 1987, ISSN 0001-0782, pages 291-307.

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

[15] Hendler, D., Shavit, N. and Yerushalmi, L., A scalable lock-free stack algorithm, SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures, Barcelona, Spain.

[16] Herlihy, M. and Shavit, N., The Art of Multiprocessor Programming, Morgan Kaufmann, March, 2008, ISBN 0123705916.

[17] Herlihy, M., Luchangco, V. and Moir, M., The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized, Lock-Free Data Structures, DISC '02: Proceedings of the 16th International Conference on Distributed Computing, 2002.

[18] Herlihy, M., Luchangco, V. and Moir, M., Obstruction-Free Synchronization: Double-Ended Queues as an Example, ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems, IEEE Computer Society, 2003, Washington, DC, USA.

[19] Herlihy, M., Luchangco, V., Martin, P. and Moir, M., Nonblocking memory management support for dynamic-sized data structures, ACM Trans. Comput. Syst., Volume 23, Number 2, 2005, ISSN 0734-2071, pages 146-196.

[20] Ingham, M., Rasmussen, R., Bennett, M. and Moncada, A., Engineering Complex Embedded Systems with State Analysis and the Mission Data System, Proceedings of First AIAA Intelligent Systems Technical Conference, 2004.

[21] Intel, IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide, `http://www.intel.com/cd/software/products/asmo-na/eng/272688.htm`, 2004.

[22] ISO/IEC 14882 International Standard, American National Standards Institute, Programming languages, C++, September, 1998.

[23] Lamport, L., How to make a multiprocessor computer that correctly executes programs, IEEE Trans. Comput., September, 1979.

[24] Lowry, M., Software Construction and Analysis Tools for Future Space Missions, International Conference on Tools and Algorithms for Construction and Analysis of Systems 2002, Lecture Notes in Computer Science, Springer, Volume 2280, pages 1-19.

[25] Maged M., Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects, IEEE Trans. Parallel Distrib. Syst., Volume 15, Number 6, 2004, ISSN 1045-9219, pages 491-504.

[26] Spear, M., Shriraman, A., Dalessandro, L., Dwarkadas, S. and Scott, M., Nonblocking transactions without indirection using Alert-On-Update, `http://www.cs.rochester.edu/research/synchronization/rstm/v4api.shtml`, Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures SPAA 2007, San Diego, California, USA.

[27] Stroustrup, B., The C++ Programming Language, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000, ISBN 0201700735.

[28] Volpe, R. and Peters S., Rover Technology Development and Mission Infusion for the 2009 Mars Science Laboratory Mission, 7th International Symposium on Artificial Intelligence, Robotics, and Automation in Space, Nara, Japan, May, 2003.

[29] Wagner, D., Data Management in the Mission Data System, Proceedings of the IEEE System, Man, and Cybernetics Conf., 2005.