Design and application of distributed containers in a CPS software control library

Damian Dechev* and Pierre LaBorde

University of Central Florida,
4000 Central Florida Blvd.,
Orlando, FL 32816, USA
Email: dechev@eecs.ucf.edu
Email: pierrelaborde@knights.ucf.edu
*Corresponding author

Abstract: Non-blocking synchronisation is known to alleviate the hazards of deadlock, livelock, and priority inversion. We present the design and portable implementation of a lock-free dynamically resizable array. Our lock-free implementation of a shared vector outperforms its lock-based STL counterpart and the implementation provided by Intel by a factor of 10 or more. The ABA problem is fundamental to all CAS-based designs. We offer a solution, called the $\lambda\delta$ approach that is practical and efficient and offers speeds comparable to the direct application of the architecture-specific CAS2 instruction used for version counting. Our lock-free vector demonstrated high scalability when compared to the application of non-blocking transactions. We demonstrate the use of our non-blocking synchronisation methodology and our shared vector for the engineering of a framework for verification and semantics parallelisation of the mission data system's (MDS) goal networks which provide for testing and development of autonomous real-time flight applications.

Keywords: lock-free; non-blocking; vector; concurrency; data structures; parallel programming; embedded computing.

Reference to this paper should be made as follows: Dechev, D. and LaBorde, P. (2015) 'Design and application of distributed containers in a CPS software control library', *Int. J. Software Engineering, Technology and Applications*, Vol. 1, No. 1, pp.2–52.

Biographical notes: Damian Dechev is an Assistant Professor with the EECS Department, University of Central Florida, Orlando, FL, USA, and Founder of the Computer Software Engineering-Scalable and Secure Systems Laboratory. He received his PhD degree from Texas A&M University, College Station, TX, USA, in 2009, under the supervision of Dr. Bjarne Stroustrup. His current research interests include programming techniques and tools, practical non-blocking synchronisation, and exascale computing.

Pierre LaBorde received his MSc in Computer Science from the University of Central Florida in 2013. Since then, he has been a PhD student at the University of Central Florida. He is currently a member of the Computer Software Engineering: Scalable and Secure Systems Lab (CSE: S3) under the advisement of Dr. Damian Dechev. His research interests include parallel programming, real-time systems, machine learning, exascale simulation and computing, and distributed and cloud computing.

This paper is a revised and expanded version of a paper entitled 'Scalable nonblocking concurrent objects for mission critical code' presented at Proceedings of 24th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009), ACM SIGPLAN, Orlando, Florida, October 2009.

1 Introduction

Robotic space mission projects 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 (such as RTCA, 1992) 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 *non-blocking* 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 Non-blocking 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 mutually exclusive locks poses significant safety hazards and incurs high complexity in the testing and validation of mission-critical software. Locks can be optimised in some scenarios by utilising fine-grained locks or context-switching. Often due to the resource limitations of flight-qualified hardware, optimised lock mechanisms are not a desirable alternative (Lowry, 2002). 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 determine 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 behaviour.

To achieve higher safety and gain performance, we suggest the application of *non-locking synchronisation*. A concurrent object is *non-blocking* if it guarantees that *some* process in the system will make progress in a *finite* amount of steps (Herlihy and Shavit, 2008). An object that guarantees that *each* process will make progress in a *finite* number of steps is defined as *wait-free*. *Obstruction-freedom* (Herlihy et al., 2003) is an alternative non-blocking condition that ensures progress if a thread eventually executes in *isolation*. It is the weakest non-blocking 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 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 (2002) 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 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 synchronisation is of critical importance to the design of autonomous spacecraft software.

1.3 Mission data system

Mission data system (MDS) (Dvorak et al., 2007) 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 (Dvorak et al., 2002)

2 goal-oriented operation: control intent is expressed by defining a set of goals as part of a goal network (Barett et al., 2004)

3 layered data management: an integrated data management and transport protocols (Wagner, 2005).

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 (2005), 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 (TCs) (Dechev et al., 2008). 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 synchronisation 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 non-blocking techniques and analyse their applicability in MDS. The most ubiquitous and versatile data structure in the ISO C++ Standard Template Library (Stroustrup, 2000) 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 non-blocking dynamic operations, we illustrate the efficiency of each non-blocking approach discussed in this work with respect to its applicability for the design and implementation of a shared non-blocking vector. A number of pivotal concurrent applications in the MDS framework employ a shared STL vector (in all scenarios protected by mutually exclusive locks). Such is the Data Management Service library described by Wagner (2005).

2 Non-blocking 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 (Gifford and Spector, 1987). The design of non-blocking data structures poses significant challenges and their development and optimisation is a current topic of research (Fraser and Harris, 2007; Herlihy and Shavit, 2008). The CAS atomic primitive [commonly known as Compare and Exchange, CMPXCHG, on the Intel *x86* and *Itanium* architectures (Intel, 2007)] 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 (Intel, 2007)]. The application of a CAS-controlled speculative manipulation of a shared location (L_i) is a fundamental programming technique in the engineering of non-blocking algorithms (Herlihy and Shavit, 2008) (an example is shown in Algorithm 1). In our pseudocode we use the symbols ^, &, and . to indicate pointer dereferencing, obtaining an object's address, and integrated pointer dereferencing and field access. When the value stored at L_i is the control value of a CAS-based speculative manipulation, we call L_i and $\hat{L_i}$ *control location* and *control value*, respectively. We indicate the control value's type with the string value type. The size of value type must be equal or less than the maximum number of bits that a hardware CAS instruction can exchange atomically (typically the size of a single memory word). In the most common cases, value type is either an integer or a pointer value. In the latter case, the implementor might reserve two extra bits per each control value and use them for implementation-specific value marking (Fraser and Harris, 2007). 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.

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($L_i$, $A_i$, $B_i$)==$B_i$
```

Linearisability (Herlihy and Shavit, 2008) is a correctness condition for concurrent objects: a concurrent operation is linearisable 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 linearisation point. The implementations of many non-blocking data structures require the update of two or more memory locations in a linearisable fashion (Dechev et al., 2006; Fraser and Harris, 2007). 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. (2002) propose in 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 skips lists (Fraser, 2004). 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 non-blocking operations is the use of a *Descriptor Object* (Section 2.1).

A number of advanced Software Transactional Memory (STM) libraries provide non-blocking transactions with dynamic linearisable operations (Dice and Shavit, 2007; Spear et al., 2007). Such transactions can be utilised for the design of non-blocking containers (Spear et al., 2007). 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 3 and 5.1 describe in detail the implementation of a non-blocking shared vector using CAS-based techniques and STM, respectively. Section 6 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 linearisability requirement is stronger than the widely applied Lamport's sequential consistency model (Lamport, 1979). 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 non-blocking 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 (Dechev et al., 2006).

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 → 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 linearisable update of two memory locations:

1 a reference to a Descriptor Object (data type pointer to δ stored in a location L_δ)

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, initialised (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 initialised 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 4), all references to descriptors must be memory managed by a safe non-blocking garbage collection (GC) 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 ω_δ and, should the flag's value be still positive, it proceeds to executing the requested update ω_δ @ 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.

Algorithm 2 Two-step execution of a δ object

```
1:   Step 1: place a new descriptor in $L_\delta$
2:   repeat
3:       $\delta$ $\mu$OldDesc = $L_\delta^{\wedge}$
4:       if $\mu$OldDesc.WDpending == true then
5:           execute $\mu$OldDesc.WD
6:       end if
7:       value type $A_i = L_i^{\wedge}$
8:       value type $B_i$ = fComputeB
9:       $\omega_\delta$ WD = $f_{\omega\delta}()$
10:      WD.Target = $L_i$
11:      WD.OldElement = $A_i$
12:      WD.NewElement = $B_i$
13:      $\upsilon\delta$ DescData = $f_{\upsilon\delta}()$
14:      $\delta$ $\mu$NewDesc = $f_\delta$(DescData, WD)
15:      $\mu$NewDesc.WDpending = true
16:  until CAS($L_\delta$, $\mu$OldDesc, $\mu$NewDesc) == $\mu$NewDesc
17:
18:  Step 2: execute the write descriptor
19:  if $\mu$NewDesc.WDpending then
20:      CAS(WD.Target, WD.OldElement, WD.NewElement)
         WD.NewElement
21:      $\mu$NewDesc.WDPending = false
22:  end if
```

2.2 Non-blocking concurrent semantics

The use of a Descriptor Object provides the programming technique for the implementation of some of the complex non-blocking operations in a shared container, such as the push_back, pop_back, and reserve operations in a shared vector (Dechev et al., 2006). The use and execution of a Write Descriptor guarantees the linearisable update of two or more memory locations.

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 (Dechev et al., 2006) 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 (Dechev et al., 2006). 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 (Dechev et al., 2006), 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 non-blocking design (Herlihy and Shavit, 2008; Fraser and Harris, 2007), 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 *linearisable 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

c in case a given response precedes an invocation in the concurrent execution, then it must precede it in the derived sequential history.

When two δ-modifying operations ($O_{\delta1}$ and $O_{\delta2}$) are concurrent, according to Algorithm 2, $O_{\delta1}$ precedes $O_{\delta2}$ in the linearisation history if and only if $O_{\delta1}$ completes step 1, Algorithm 2 prior to $O_{\delta2}$.

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 linearisation history. The $\lambda\delta$-point does not necessarily need to coincide with the operation's linearisation point, τ_{lin}. As illustrated in Dechev et al. (2006), τ_{lin} can vary depending on the operations' concurrent interleaving. The linearisation point of a shared vector's (Dechev et al., 2006) 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 linearisable operation is that it must appear to execute in a single instant of time with respect to other concurrent operations. The linearisation 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 implementation, thus making it easier to statically argue about an operation's correctness. In the pseudo code in Algorithm 2 $\tau_{\lambda\delta} \equiv \tau_\delta$.

3 Descriptor-based shared vector

In this section, 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 CAS instruction. It provides a linearisable and highly parallelisable STL-like interface, lock-free memory allocation and management, and fast execution. 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.

3.1 Lock-free data structures

Recent research into the design of lock-free data structures includes linked-lists (Harris, 2001; Michael, 2002) double-ended queues (Michael, 2003; Sundell and Tsigas, 2004), stacks (Hendler et al., 2004), hash tables (Michael, 2002; Shalev and Shavit, 2003), and binary search trees (Fraser, 2004). 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 that can be effectively applied in embedded real-time applications.

3.2 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 (Alexandrescu and Michael, 2004), minimise 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++ (ISO/IEC 14882 International Standard, 1998).

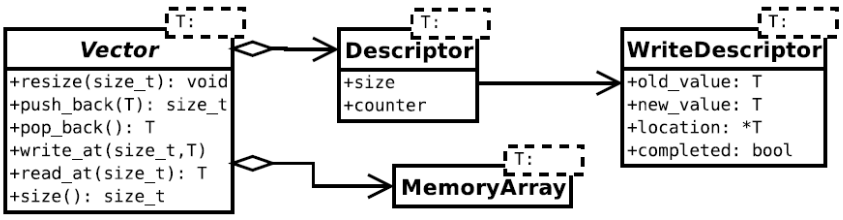
3.3 Algorithms

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.

3.3.1 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 (Cormen et al., 2001) 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.

Figure 1 Lock-free vector



Note: T denotes a data structure parameterised on T.

The UML diagram in Figure 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 3.3.3) and an optional reference to a 'Write Descriptor'. Our approach requires that data types bigger than word size is indirectly stored through pointers. Like Intel's concurrent vector (Intel, 2006), our implementation avoids storage relocation and its synchronisation hazards by utilising 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' [similar to a Barnes (1993) style announcement] 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*. Table 1 illustrates the implemented operations as well as their signatures, descriptor modifications, and runtime guarantees.

Table 1 Vector – operations

	Operations	*Memory locations*
push_back	*Vector × Elem → void*	*2: element, size*
pop_back	*Vector → Elem*	*1: size*
reserve	*Vector × size_t → Vector*	*n: all elements*
read	*Vector × size_t → Elem*	*none*
write	*Vector × size_t × Elem → Vector*	*1: element*
size	*Vector → size_t*	*none*

The remaining part of this section presents the generalised pseudo-code of the implementation and omits code necessary for a particular memory management scheme. 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 utilise 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 utilise the descriptor and their success is independent of the descriptor's value.
- *Reserve (increase allocated space):* In the case of concurrently executing *reserve* operations, only one succeeds per bucket, while the others deallocate the acquired memory.
- *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.

Algorithm 3 push_back *vector*, *elem*

```
1:  repeat
2:      $desc_{current} \leftarrow$ vector.desc
3:      CompleteWrite(vector, $desc_{current}$.pending)
4:      bucket $\leftarrow$ HighestBit($desc_{current}$.size + FBS) $-$ HighestBit(FBS)
5:      if vector.memory[bucket] = NULL then
6:          AllocBucket(vector, bucket)
```

```
7:       end if
8:       writeop ← new WriteDesc(At(desc_current.size), elem,
         desc_current.size)
9:       desc_next ← new Descriptor(desc_current.size+1, writeop)
10:  until CAS(&vector.desc, desc_current, desc_next)
11:  CompleteWrite(vector, desc_next.pending)
```

Algorithm 4 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)
5:  end if
```

Algorithm 5 Size *vector*

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

Algorithm 6 Read *vector*, *i*

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

Algorithm 7 Write *vector*, *i*, *elem*

```
1:  At(vector, i)^ ← elem
```

Algorithm 8 pop_back *vector*

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

Algorithm 9 Reserve *vector*, *size*

```
1:  i ← HighestBit(vector.desc.size + FBS –1)-
    HighestBit(FBS)
2:  if i < 0 then
3:     i ← 0
```

```
4:  end if
5:  while $i$ < HighestBit(size + FBS – 1) – HighestBit(FBS)
    do
6:    $i \leftarrow i + 1$
7:    AllocBucket(vector, $i$)
8:  end while
```

Algorithm 10 At *vector*, *i*

```
1:  pos ← $i$ + FBS
2:  hibit ← HighestBit(pos)
3:  idx ← pos xor $2^{\text{hibit}}$
4:  return &vector.memory[hibit – HighestBit(FBS)][idx]
```

Algorithm 11 CompleteWrite *vector*, *writeop*

```
1:  if writeop.pending then
2:      CAS(At(vector, writeop.pos), writeop.value$_{\text{old}}$,
        writeop.value$_{\text{new}}$)
3:      writeop.pending ← false
4:  end if
```

3.3.2 Correctness

The main correctness requirement of the semantics of the vector's operations is linearisability. The definition of linearisability (Herlihy and Shavit, 2008) 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 $\{o_1, o_2 \ldots, o_n\} \in S_{vec}$. We outline a proof that operation o_i is linearisable.

- *Linearisation points*: For all non-descriptor-modifying operations the linearisation point is at the time instance τ_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 (TPs) τ_{desc} (Algorithm 1, line 10; Algorithm 6, line 6) and $\tau_{\text{writedesc}}$ (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 $o_c \in \{o_1, o_2 \ldots, o_n\}$, respectively. Similarly, time point τ_{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 linearisation point is either τ_{readelem} or τ_{desc} for the former case and τ_{readelem}, τ_{desc}, or $\tau_{\text{writedesc}}$ for the latter case.

- *Sequential semantics*: let S_c be the set of all concurrent operations $\{o_1, \dots, o_n\}$ in a time interval $[\tau_\alpha, \tau_\beta]$. If $\forall o_i \in S_c$, DescriptorModifying(o_i), the linearisation point for each operation is $\tau_{\text{desc}}(o_i)$. Similarly, if $\forall o_i \in S_c$, Non-DescriptorModifying(o_i), the linearisation point for each operation is $\tau_a(o_i)$. In these cases, the resulting sequential histories are directly derived from the temporal order of the linearisation 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 Non-DescriptorModifying(o_2).

  We emphasise that the presented ordering relations are $o_1 \setminus$ o$_2$ read write push_back $\tau_{\text{writedesc}}(o_1)$, $\tau_a(o_2)$ $\tau_{\text{readelem}}(o_1)$, $\tau_a(o_2)$ pop_back $\tau_{\text{desc}}(o_1)$, $\tau_a(o_2)$ $\tau_{\text{readelem}}(o_1)$, $\tau_a(o_2)$ 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 TPs $\tau_a(o_2)$, $\tau_a(o_3)$ occur between $\tau_{\text{readelem}}(o_1)$ and $\tau_{\text{writedesc}}(o_1)$ 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 linearisability 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.

Table 2 Linearisation points of o_1, o_2

$o_1 \setminus o_2$	*Read*	*Write*
push_back	$\tau_{\text{writedesc}}(o_1)$, $\tau_a(o_2)$	$\tau_{\text{readelem}}(o_1)$, $\tau_a(o_2)$
pop_back	$\tau_{\text{desc}}(o_1)$, $\tau_a(o_2)$	$\tau_{\text{readelem}}(o_1)$, $\tau_a(o_2)$

3.3.3 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 primarily reference counting as described by Michael and Scott (1995). 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 (2004a) hazard pointers and Herlihy et al.'s (2005) pass the buck (PTB) overcome the problem. To gain an insight about the impact of the application of such a sophisticated non-blocking GC approach, as opposed to reference counting, we have re-implemented and integrated Herlihy et al.'s PTB approach. Our performance analysis in Section 3.3.6 briefly compare the performance of two variations of our vector: one relying on reference counting and the other utilising PTB.
- *Allocator recent:* Research by Michael (2004b) and Gidenstam et al. (2005) presents implementations of true lock-free memory allocators. Due to its availability and performance, we selected Gidenstam's allocator for our performance tests.

3.3.4 ABA hazards

The semantics of the presented lock-free vector's operations can be corrupted by the occurrence of the ABA problem (Section 4). 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 this vector's implementation:

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. In Section 4, we analyse in depth the hazards of ABA and suggest a generic methodology for ABA avoidance.

Given the current implementation, to eliminate the ABA problem of 2 (in the absence of CAS2), we have incorporated a variation of Herlihy et al.'s (2005) PTB algorithm utilising 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 behaviour complies with the STL value-semantics (Stroustrup, 2000), 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.

3.3.5 Alternatives

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

- *Copy on write:* Alexandrescu and Michael (2004) present a lock-free map, where every write operation creates a clone of the original map, which insulates modifications from concurrent operations. Once completed, the pointer to the map'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. (2002) present a software multi-compare and swap (MCAS) implementation based on CAS instructions. While convenient, the MCAS operation is expensive (requiring 2M+1 CAS instructions). Thus, it is not the best choice for an effective implementation.

- *Contiguous storage:* Techniques similar to the ones used in our vector implementation could be applied to achieve a vector with contiguous storage. The difference is that the storage area can change during the data structure's lifetime. This requires resize to move all elements to the new location. Hence, storage and its capacity should become members of the descriptor. Synchronisation between *write* and *resize* operations is what makes this approach difficult. A straightforward solution is to apply descriptor-modifying semantics for all random access write operations as well as resize.

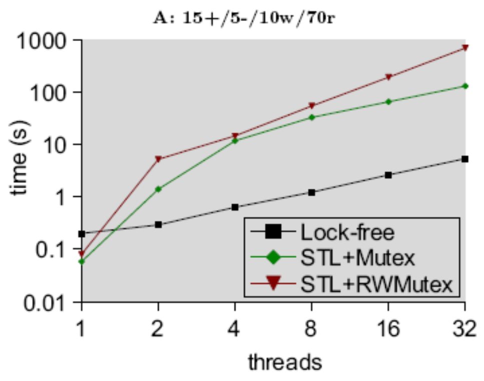
We discussed the descriptor- and non-descriptor modifying writes in the context of the two-level array and the contiguous storage vector. However, these write properties are not inherent in these two approaches. In the two-level array, it is possible to make each write operation descriptor modifying, 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.

3.3.6 Performance evaluation

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 OS X operating system. In our performance analysis, we compare the lock-free approach (with its integrated lock-free memory management and memory allocation) with the concurrent vector provided by Intel (2006) as well as an STL vector protected by a lock. For the latter scenario we applied different types of locking synchronisations – 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 synchronisation technique for a given distribution of operations. We utilise the locking synchronisation provided by Intel (2006).

Similarly to the evaluation of other lock-free concurrent containers (Fraser, 2004; Michael, 2002), 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 (2004), 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 2. Similarly, Figure 4 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.

Figure 2 Shared vector performance results A – Intel core duo (see online version for colours)



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. Figures 3 and 5 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 (intel, 2006), SpinLocks are volatile, unfair, and not scalable. They showed fast execution for the experiments with eight 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 5, the lower bound is about a $\frac{1}{10}$ of the lock-free vector.

Figure 3 Shared vector performance results B – Intel core duo (see online version for colours)

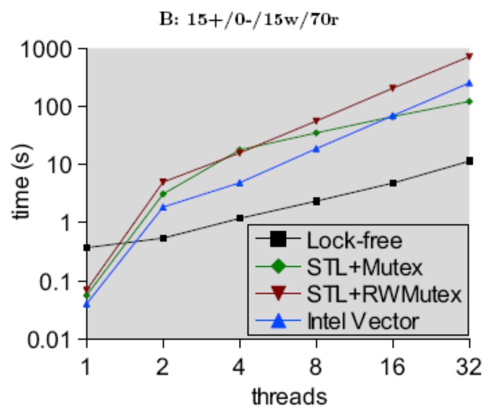


Figure 4 Shared vector performance results C – Intel core duo (see online version for colours)

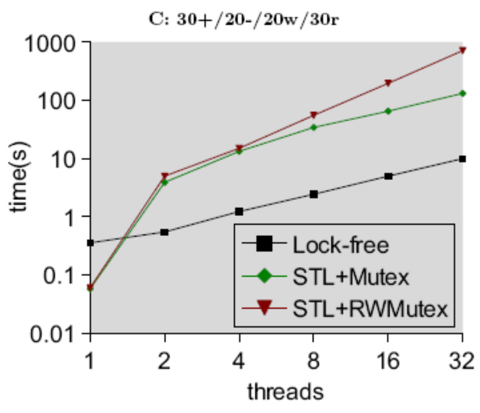
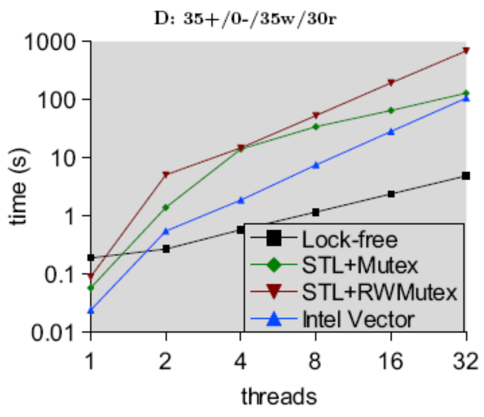


Figure 5 Shared vector performance results D – Intel core duo (see online version for colours)



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.

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.

4 The ABA problem

The ABA problem is a fundamental problem to all CAS-based designs. Its significance has increased with the suggested use of CAS as a core atomic primitive for the implementation of portable lock-free algorithms. Here, we offer a solution, called the $\lambda\delta$ approach, that by a large factor outperforms the use of GC for the safe management of each shared location. It offers speeds comparable to the direct application of the architecture-specific CAS2 instruction used for version counting. A practical alternative to the application of the architecture-specific CAS2 is particularly important to the majority of complex embedded systems. We offer an explicit and detailed analysis of the ABA problem, its relation to the most commonly applied non-blocking programming techniques and correctness guarantees, and the possibilities for its detection and avoidance. The current state of the art leaves the elimination ABA hazards to the ingenuity of the software designer. Here, we analyse the concurrent interactions that lead to ABA as well as ABAs relation to the most commonly described non-blocking programming techniques. We define a generic and practical condition for ABA avoidance for a lock-free linearisable design. We demonstrate our approach by integrating it into an advanced non-blocking data structure, a lock-free dynamically resizable array. Our performance evaluation establishes that the single word CAS-based $\lambda\delta$ approach delivers performance comparable to the use of CAS2.

The ABA problem can seriously corrupt the semantics of a non-blocking algorithm (Gifford and Spector, 1987; Michael, 2004a; Dechev et al., 2006). 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 non-blocking 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 non-blocking programming techniques 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 non-blocking algorithm is left to the ingenuity of the software designer. In this section, we study in detail and define the conditions that lead to ABA. We investigate the relationship between the ABA hazards and the most commonly applied non-blocking 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 a lock-free linearisable design (Section 4.3). We demonstrate the application of our approach by incorporating it in a complex and advanced non-blocking data structure, a lock-free dynamically resizable array (Section 3). We survey the literature for other known ABA prevention techniques (usually described as a part of a non-blocking algorithm's implementation) and study in detail three known solutions to the ABA problem (Section 4.1). Our work establishes the criteria we apply in our search for a generic and practical solution to the ABA problem (Section 4.2). Our performance evaluation (Section 4.4) establishes that the single word CAS-based $\lambda\delta$ approach is fast, efficient, and practical.

Definition 6: The ABA problem is a false positive execution of a CAS-based speculation on a shared location L_i.

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 the a B_i. Afterwards, either P_k or any other process $P_j \neq P_1$ can eventually store A_i back to L_i. When P_1 resumes, its CAS loop succeeds (false positive execution) despite the fact that L_i's value has been meanwhile manipulated.

Table 3 ABA at L_i

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

Definition 7: A non-blocking algorithm is ABA-free if its semantics cannot be corrupted by the occurrence of ABA problem.

ABA-freedom is achieved when:

a occurrence of ABA is harmless to the algorithm's semantics or

b ABA is avoided.

The former scenario is uncommon and strictly specific to the algorithm's semantics. The latter scenario is the general case and in this work we focus on providing details of how to eliminate ABA.

4.1 Known ABA prevention techniques

A general strategy for ABA avoidance is based on the fundamental guarantee that no process $P_j (P_j \neq P_1)$ can possibly store A_i again at location L_i (step 3, Table 3). One way to satisfy such a guarantee is to require all values stored in a given control location to be *unique*. To enforce this uniqueness invariant we can place a constraint on the user and request each value stored at L_i be used only once (*Known Solution 1*). Enforcing this constraint can be facilitated if a programming language's type system supports uniqueness typing (Vries et al., 2008) that forbids the use of more than a single reference to an object. We are not familiar with any programming language or library that implements uniqueness typing in a concurrent environment. To achieve this goal, it would be necessary to design and apply a complex tool-chain of static and dynamic program analysis. For a large majority of concurrent algorithms, enforcing uniqueness typing would not be a suitable solution since their applications imply the usage of a value or reference more than once.

An alternative approach to satisfying the uniqueness invariant is to apply a version tag attached to each value. The usage of *version tags* is the most commonly cited solution for ABA avoidance (Gifford and Spector, 1987). The approach is effective, when it is possible to apply, but suffers from a significant flaw: a portable single-word CAS instruction is insufficient for the atomic update of a word-sized control value and a word-sized version tag. An effective application of a version tag (Detlefs et al., 2002) requires the hardware architecture to support a more complex atomic primitive that allows the atomic update of two memory location, such as compare-and-swap two (CAS2 co-located words) or DCAS (CAS2 memory locations). The availability of such atomic primitives might lead to much simpler, elegant, and efficient concurrent designs (in contrast to a CAS-based design). It is not desirable to suggest a CAS2/DCAS-based ABA solution for a CAS-based algorithm, unless the implementor explores the optimization possibilities of the algorithm upon the availability of CAS2/DCAS. A proposed hardware implementation (entirely built into a present cache coherency protocol) of an innovative alert-on-update (AOU) instruction (Spear et al., 2007) has been suggested by Spear et al. to eliminate the CAS deficiency of allowing ABA. The main drawbacks for using version tags is the fact that a large number of the current hardware architectures, such as the majority of real-time embedded systems (Volpe and Peters, 2003), do not support complex atomic primitives such as CAS2, DCAS, LL/SC, and AOU. A synchronisation scheme on such machines can rely only on the portable single-word CAS instruction.

In Reinholtz (2008), offers a technique for applying version tags using a 32-bit single-word memory swap (*Known Solution 2*). Similarly to the AtomicStampedReference in the Java Concurrency Library, Reinholtz's reference counting pointers (RCP) split a version counter into two half-words: a half-word used to store the control value's data value (an integer version counter in RCPs case) and a half-word used as a version tag. The limitations of this approach are:

a there is a limit of maximum $2^{16} - 1$ writes for each control location

b the range of values that can be represented in a control value is significantly decreased (by a factor of 2^{16}).

To guarantee the uniqueness invariant of a control value of type pointer in a concurrent system with dynamic memory usage, we face an extra challenge: even if we write a pointer value no more than once in a given control location, the memory allocator might reuse the address of an already freed object (A_i) and pose an ABA hazard. To prevent this scenario, all control values of pointer type must be guarded by a concurrent non-blocking GC scheme such as hazard pointers (Michael, 2004a) (that uses a list of hazard pointers per thread) or Herlihy et al.'s (2005) PTB algorithm (that utilises a dedicated thread to periodically reclaim unguarded objects). While enhancing the safety of a concurrent algorithm (when needed), the application of a complementary GC mechanism might come at a significant performance cost (see Section 6 for details).

A known approach for avoiding a false positive execution of the Write Descriptor from Algorithm 2 is the application of value semantics for all values of type value type (*Known Solution 3*). As discussed in Hendler et al. (2004) and Dechev et al. (2006), an ABA avoidance scheme based on value semantics relies on:

a *Extra level of indirection:* All values are stored in shared memory indirectly through pointers. Each write of a given value v_i to a shared location L_i needs to allocate on the heap a new reference to v_i (η_{v_i}), store (η_{v_i}) into L_i, and finally safely delete the pointer value removed from L_i. If the value type of v_i is pointer then η_{v_i}'s type is pointer to pointer.

b *Non-blocking GC:* All references stored in shared memory (such as η_{v_i}) need to be safely managed by a non-blocking GC scheme (e.g., hazard pointers, PTB).

As reflected in our performance test results (Section 6), the usage of both, an extra level of indirection as well as the heavy reliance on a non-blocking 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 non-blocking algorithm. However, the use of value semantics is the *only known approach* for ABA avoidance in the execution of a Write Descriptor Object. In Section 4.3, we present a three-step execution approach that helps us eliminate ABA, avoid the need for an extra level of indirection, and reduce the usage of the computationally expensive GC scheme.

4.2 Criteria

To provide a practical and generic solution to the ABA problem without incurring a prohibitive cost to the lock-free application, our search for a solution has been guided by the following design criteria:

a *Complexity and semantics preservation:* An ABA avoidance scheme should not incur extra algorithmic complexity and should preserve the application's non-blocking guarantees and correctness conditions. For example, a shared vector's tail operations have a complexity of $O(1)$ that must be preserved.

b *Dynamic and open memory usage:* Ability to support dynamic and open memory usage at a minimal cost.

c *Fast performance:* An ABA prevention scheme should make minimal usage of expensive GC and should not prevent disjoint-access parallelism. Some lock-free container's implementations provide a combination of lock-free (δ-modifying) and wait-free (non-δ-modifying) operations (Dechev et al., 2006). Wait-free operations are fast and progress regardless the contention on the shared memory. Preserving the wait-free semantics of such operations might be critical to the container's performance. While sometimes necessary to apply, the application of GC schemes must be limited to an absolute minimum.

d *Portability:* We assume the availability of single-word atomic read, write, and CAS instructions. We consider solutions based on multi-word CAS, AOU (Spear et al., 2007), or LL/SC to be platform-specific.

e *Unlimited data usage:* We prefer to avoid placing constraints on the usage of the data values. We assume that the data values stored in a shared container need not be unique, there is no restriction on the range of values (imposed by the ABA prevention algorithm), data elements can be written and read an arbitrary number of times to/from any location, and there is no restriction on the number of writer threads.

f *No extra levels of indirection:* A famous quote by David Wheeler states: "Any problem in computer science can be solved with another layer of indirection, but that usually will create another problem" (Stroustrup, 2000). As illustrated in Section 6, the application of an extra level of indirection suffers performance penalties, leads to heavy usage of the costly GC scheme, increases the complexity of the non-blocking algorithm, and is difficult to integrate in an already existing non-blocking implementation.

4.3 Implementing a λδ-modifying operation

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. Algorithm 12 demonstrates the occurrence of ABA in the execution of a δ object with two concurrent *δ-modifying* operations (O_{δ_1} and O_{δ_2}) and a concurrent write, O_i, to L_i. We assume that the δ object's implementation follows Algorithm 2. The execution of O_{δ_1} and 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 (line 1–2, Algorithm 12). Next, O_{δ_1} proceeds with instantiating a new δ object and replaces the old descriptor with the new one (line 3, Algorithm 12).

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 (line 4, Algorithm 12).

3 O_{δ_1} resumes and completes the execution of its δ object by storing B_i into L_i (line 5, Algorithm 12).

4 An interrupting operation, O_i, writes the value A_i into L_i (line 6, Algorithm 12).

5 O_{δ_2} resumes and executes $\omega\delta$ it has previously read, the $\omega\delta$'s CAS falsely succeeds (line 6, Algorithm 12).

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 from Algorithm 12 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.

Algorithm 12 ABA occurrence in the execution of a Descriptor Object

```
1:   $O_{\delta_1} : \tau_{read_\delta}$
2:   $O_{\delta_1} : \tau_{access_i}$
3:   $O_{\delta_1} : \tau_{\delta}$
4:   $O_{\delta_2} : \tau_{read_\delta}$
5:   $O_{\delta_1} : \tau_{wd}$
6:   $O_i : \tau_{write_i}$
7:   $O_{\delta_2} : \tau_{wd}$
```

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

Assume that in Algorithm 12 the O_{δ_1}'s $\lambda\delta$-point $\equiv \tau_{access_i}$. As shown in Algorithm 12, 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 *λδ-modifying* implementation of O_{δ_1} allows any *non-ωδ-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 *three-step descriptor execution* approach, described in Section 4.3, offers a solution based on Definition 8. In an implementation with two or more concurrent ωδ-executing operations, each *ωδ-executing* operation must be *λδ-modifying* in order to eliminate the hazard of a spurious execution of an *ωδ* that has been picked up by a collaborating operation. To effectively avoid the ABA hazard at L_i, we generalise two fundamental strategies:

a Guarantee that a Write Descriptor created by O_{δ_1}, or any other *ωδ-executing* operation, succeeds at most once. We refer to such a *δ* object as a *once-execute descriptor*. Definition 8 offers the basics for a solution of this type. In our example in Algorithm 12, a *once-execute-descriptor* strategy would cause the attempt to re-execute the write descriptor by O_{δ_1} (line 7, Algorithm 12) or any other operation to fail. Our three-step *δ* execution approach presented in Section 4.3 is one possible way of implementing a once-execute-descriptor.

b Guarantee that no concurrent interleaving of operations can write a value posing ABA hazard (such as B_i in Algorithm 12) at L_i. Relying on a methodology that employs unique values, such as *Known Solution 1*, is an approach of this type. Requiring uniqueness typing for ABA prevention is an overkill. The guarantee we need is that no thread can restore an old value A_i in a shared location L_i while there is an alive *ωδ* object in the system requesting *ωδ@*L_i: $A_i \rightarrow$ *any_value*$_i$. Modern mainstream programming languages do not yet explicitly support concurrency and lack the tools to express and enforce such a concurrent and dynamic correctness condition.

In Algorithm 13, we suggest a design strategy for the implementation of a $\lambda\delta$-modifying operation. Our approach is based on a three-step execution of the δ object. While similar to Algorithm 2, the approach shown in Algorithm 13 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 26) fail, we have to unroll the changes to 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 13, 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 initialisation. 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 13, 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 13, 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 13) 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 linearisable update of two shared memory locations (L_i and L_δ).

Algorithm 13 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:  $\omega\delta$ WD = $f_{\omega\delta}()$
5:  WD.Target = $L_i$
6:  WD.NewElement = $B_i$
7:  $\upsilon\delta$ DescData = $f_{\upsilon\delta}()$
8:  $\delta$ $\mu$NewDesc = $f_\delta$(DescData, WD)
9:  WD.TempElement = &NewDesc
10: $\mu$NewDesc.WDpending = true
11: repeat
```

```
12:     $A_i$ = ^$L_i$
13:     WD.OldElement = $A_i$
14: until CAS($L_i$, $A_i$, $\mu$NewDesc) == $\mu$NewDesc
15:
16: Step 2: place the new descriptor in $L_\delta$
17: bool unroll = false
18: repeat
19:     if unroll then
20:         CAS(WD.Target, $\mu$NewDesc, WD.OldElement)
21:         go to 3
22:     end if
23:     $\delta$ $\mu$OldDesc = ^$L_\delta$
24:     if $\mu$OldDesc.WDpending == true then
25:         execute $\mu$OldDesc.WD
26:     end if
27:     unroll = true
28: until CAS($L_\delta$, $\mu$OldDesc, $\mu$NewDesc) == $\mu$NewDesc
29:
30: Step 3: execute the Write Descriptor
31: if $\mu$NewDesc.WDpending then
32:     CAS(WD.Target, WD.TempElement, WD.NewElement)
        == WD.NewElement
33:     $\mu$NewDesc.WDPending = false
34: end if
```

4.4 Performance evaluation

We incorporated the presented ABA elimination approach in the implementation of the non-blocking dynamically resizable array as presented in Section 3. Our test results indicate that the $\lambda\delta$ approach offers ABA prevention with performance comparable to the use of the platform-specific CAS2 instruction to implement version counting. This finding is of particular value to the engineering of some embedded real-time systems where the hardware does not support complex atomic primitives such as CAS2 (Lowry, 2002). 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 X operating system. In our performance analysis we compare:

1 *λδ approach:* The implementation of a vector with a *λδ*-modifying push_back and a *δ*-modifying pop_back. Table 4 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 (2005) PTB algorithm. In addition, we use PTB to protect the Descriptor Objects for all of the tested approaches.

3 *CAS2-based approach:* The application of CAS2 for maintaining a reference counter for each element. A CAS2-based version counting implementation is easy to apply to almost any pre-existent CAS-based algorithm. While a CAS2-based solution is not portable 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 4 offers an overview of the shared vector's operations' relative cost in terms of number and type of atomic instructions invoked per operation.

Figure 6 Performance results A (see online version for colours)

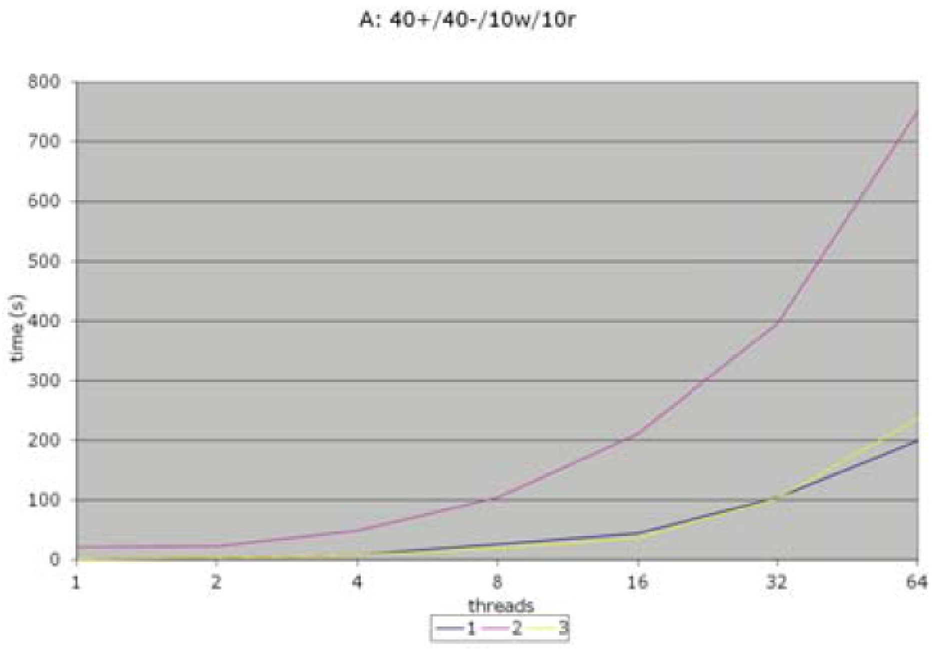


Figure 7 Performance results B (see online version for colours)

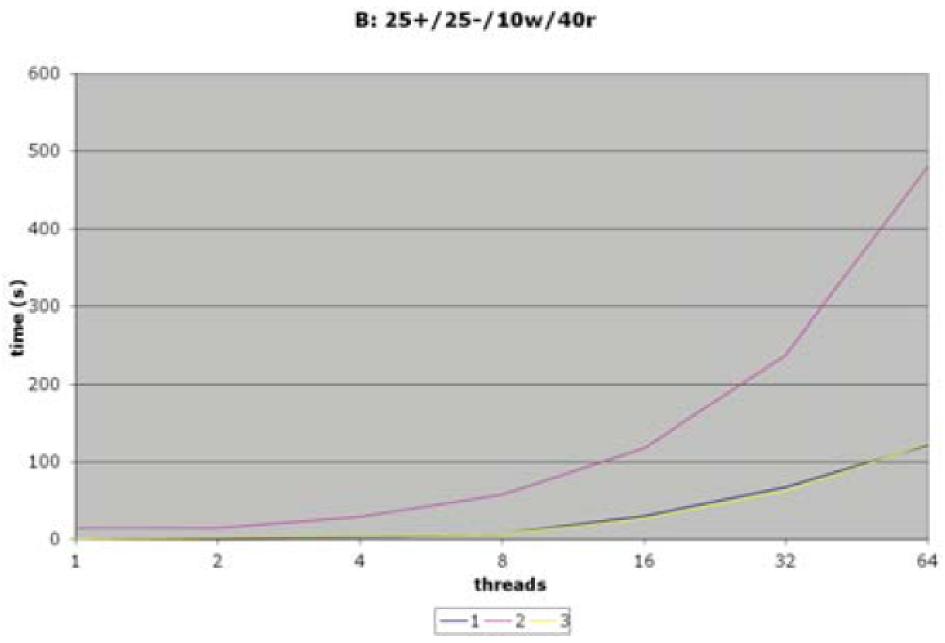


Figure 8 Performance results C (see online version for colours)

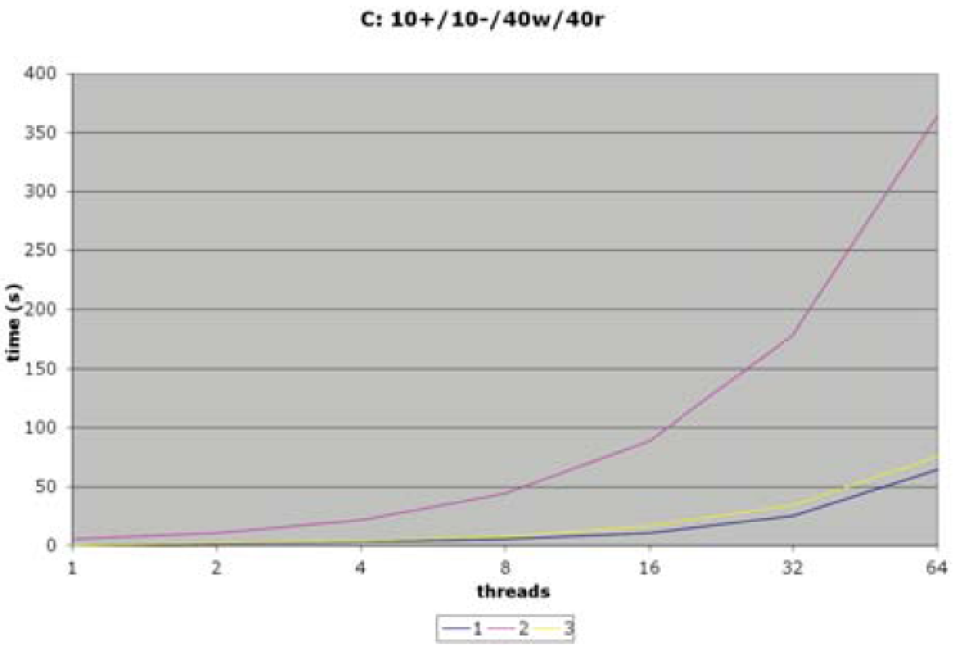


Figure 9 Performance results D (see online version for colours)

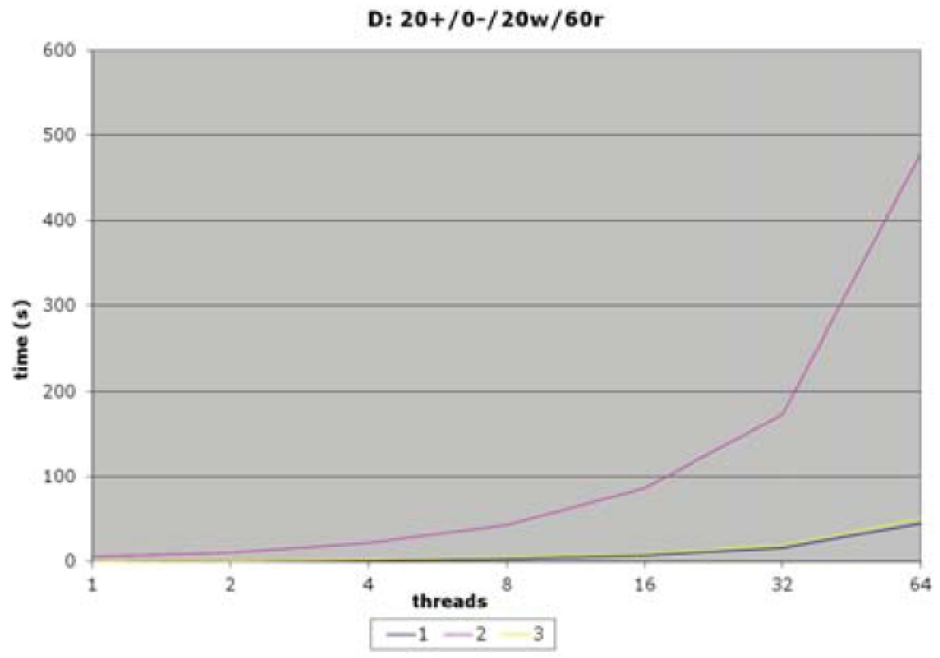


Table 4 A shared vector's operations cost (best case scenario)

		push_back	*pop_back*	*read_i*	*write_i*
1	λδ approach	3 CAS	1 CAS	Atomic read	Atomic write
2	All-GC	2CAS + GC	1CAS + GC	Atomic read	Atomic write + GC
3	CAS2-based	1CAS2 + 1CAS	1 CAS	Atomic read	1 CAS2

We varied the number of threads, starting from 1 and exponentially increased their number to 64. Each thread executed 500,000 lock-free operations on the shared container. We measured the execution time (in seconds) that all threads needed to complete. Each iteration of every thread executed an operation with a certain probability (+, –, random access w, random access r]. We show the performance graph for a distribution of +:40%, –:40%, w:10%, r:10% on Figure 6. Figure 7 demonstrates the performance results with less contention at the vector's tail, +:25%, –:25%, w:10%, r:40%. Figure 8 illustrates the test results with a distribution containing predominantly random access read and write operations, +:10%, –:10%, w:40%, r:40%. Figure 9 reflects our performance evaluation on a vector's use with mostly random access read operations: +:20%, –:0%, w:20%, r:60%, a scenario often referred to as the most common real-world use of a shared container (Fraser, 2004). 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 *λδ* approach delivers consistent performance gains in all possible operation mixes by a large factor, a factor of at least 3.5 in the cases with less contention at the tail and a factor of 10 or more when there is a high concentration of tail operations. These observations come as a confirmation to our expectations that introducing an extra level of indirection and the necessity to memory manage each individual element with PTB (or an alternative memory management scheme) to avoid ABA comes with a pricy performance overhead. The $\lambda\delta$ approach offers an alternative by introducing the notion of a $\lambda\delta$-point and enforces it though a three-step execution of the δ object. The application of version counting based on the architecture-specific CAS2 operation is the most commonly cited approach for ABA prevention in the literature (Hendler et al., 2004; Herlihy et al., 2005). Our performance evaluation shows that the $\lambda\delta$ approach delivers performance comparable to the use of CAS2-based version counting. CAS2 is a complex atomic primitive and its application comes with a higher cost when compared to the application of atomic write or a single-word CAS. In the performance tests we executed, we notice that in the scenarios where the random access write is invoked more frequently (Figures 8 and 9), the performance of the CAS2 version counting approach suffers a performance penalty and runs slower than the $\lambda\delta$ approach by about 12% to 20%. According to our performance evaluation, the $\lambda\delta$ approach is a systematic, effective, portable, and generic solution for ABA avoidance. 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.

5 STM-based non-blocking design

A variety of recent STM approaches (Dice and Shavit, 2007; Spear et al. 2007) 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 utilisation 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 non-blocking, unbounded, and dynamic transactions provides an alternative to CAS-based designs for the implementation of non-blocking 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 non-blocking designs require two levels of indirection in accessing data.

b *Linearisability:* The linearisability requirements are hard to meet for an unbounded and dynamic STM. To achieve efficiency and reduce the complexity, all known non-blocking STMs offer the less demanding obstruction-free synchronisation (Herlihy et al., 2003).

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 non-blocking and lock-based STMs often require a closed memory system (Dice and Shavit, 2007).

e *Vulnerability of large transactions:* In a non-blocking 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.

5.1 Obstruction-free descriptor vs. lock-free descriptor

To be able to reduce the complexity of implementing non-blocking transactions, the available non-blocking 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 14 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 14 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 14 should the attempted update at L_δ in step 2, Algorithm 14 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 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 stored in L_i. When an interrupting thread recognises an ω_δ object, it can simply ignore its presence and obtain the element (of type value type) by reading the field OldElement of the Write Descriptor.

Algorithm 14 Implementing a descriptor object with obstruction-free semantics

```
1:  Step 1: update $L_i$
2:  $\delta$ $\mu$OldDesc = $\hat{L_\delta}$
3:  value type $A_i = \hat{L_i}$
4:  value type $B_i$ = fComputeB
5:  $\omega\delta$ WD = $f_{\omega\delta}()$
6:  WD.Target = $L_i$
7:  WD.NewElement = $B_i$
8:  $\upsilon\delta$ DescData = $f_{\upsilon\delta}()$
9:  $\delta$ $\mu$NewDesc = $f_\delta$(DescData, WD)
10: WD.TempElement = &NewDesc.WD
11: repeat
12:     $Ai = \hat{L_i}$
13:     WD.OldElement = $A_i$
14: until CAS($L_i$, $A_i$, $\mu$NewDesc.WD) == $\mu$NewDesc.WD
15:
16: Step 2: place the new descriptor in $L_\delta$
17: bool unroll = false
18: repeat
19:     if unroll then
20:         CAS(WD.Target, $\mu$NewDesc, WD.OldElement)
21:         goto 1
22:     end if
23:     unroll = true
24: until CAS($L_\delta$, $\mu$OldDesc, $\mu$NewDesc) == $\mu$NewDesc
25:
26: Step 3: execute the write descriptor
27: 27: CAS(WD.Target, WD.TempElement, WD.NewElement)
```

Obstruction-free objects following a design similar to Algorithm 14 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.

5.2 RSTM-based vector

The Rochester Software Transactional Memory (RSTM) (Spear et al., 2007) is a word- and indirection-based C++ STM library that offers obstruction-free non-blocking transactions. As explained by the authors in Spear et al. (2007), 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 non-blocking shared vector. We chose to use RSTM because of its flexible and efficient object-oriented C++ design, demonstrated high performance when compared to alternative STM techniques, and the availability of non-blocking transactions. In Algorithms 15, 16, 17, and 18, we present the RSTM-based implementations of the read, write, pop_back, and push_back operations, respectively. According to the RSTM API (Spear et al., 2007), access to shared data is achieved by utilising four classes of shared pointers:

1 a *shared object* (class sh_ptr <T>) representing on 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

4 a *privatised 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 17 and 18 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 15 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 16 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 17 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 18 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
```

6 Analysis and results

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

1 The RSTM-based non-blocking vector implementation as presented in Section 5.1.

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 (Spear et al., 2007). Though blocking and not meeting our goals for safe and reliable synchronisation, we include the lock-based RSTM vector execution to gain additional insight about the relative performance gains or penalties that the discussed non-blocking 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 3.

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 OS X operating system. 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 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 (+, –, random access w, random access r). We show the performance graph for a distribution of +:10%, –:10%, w:40%, r:40% on Figure 10. Figure 11 demonstrates the performance results in a read-many-write-rarely scenario, +:10%, –:10%, w:10%, r:70%. Figure 12 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 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 non-blocking 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 (Spear et al., 2007; Dice and Shavit, 2007), we suggest that a shared vector design based on the utilisation of non-blocking CAS-based algorithms can better serve the demands for safe and reliable concurrent synchronization in mission critical code.

Figure 10 Performance results 1 (see online version for colours)

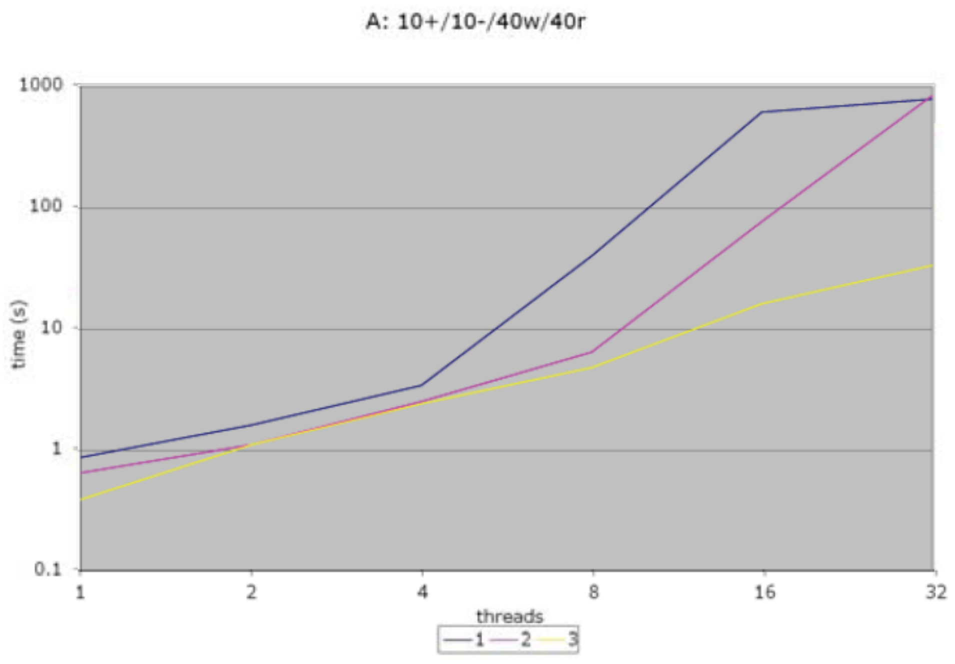


Figure 11 Performance results 2 (see online version for colours)

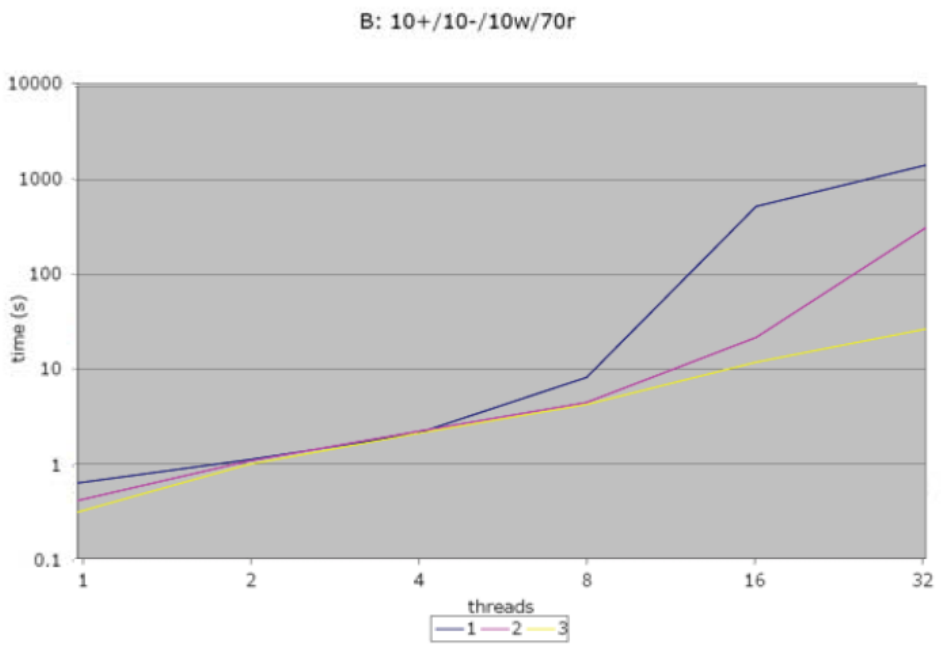
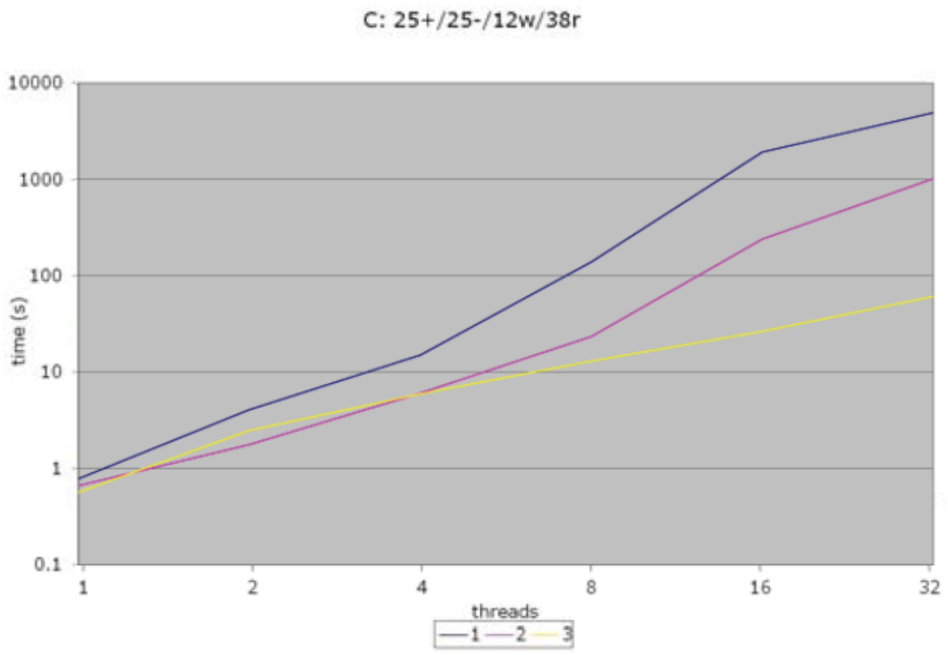


Figure 12 Performance results 3 (see online version for colours)



7 Verification and semantic parallelisation of goal networks

In this section, we describe the design, implementation, and application of a first *concurrency* and *time* centred framework for verification and semantic parallelisation of real-time C++ within the JPL MDS framework. The end goal of the industrial project that motivated our work is to provide certification artefacts and accelerated testing of the complex software interactions in autonomous flight systems. The process of software certification establishes the level of confidence in a software system in the context of its *functional* and *safety* requirements. A software certificate contains the evidence required for the system's independent assessment by an authority having minimal knowledge and trust in the technology and tools employed (Denney and Fischer, 2005). Providing such certification evidence may require the application of a number of software development, analysis, verification, and validation techniques (Lowry, 2002). The dominant paradigms for software development, assurance, and management at NASA rely on the principle "test what-you-fly and fly-what-you-test". This methodology had been applied in a large number of robotic space missions at the Jet Propulsion Laboratory. For such missions, it has proven suitable in achieving adherence to some of the most stringent standards of man-rated certification such as the DO-178B (RTCA, 1992), the Federal Aviation Administration (FAA) software standard. Its level A certification requirements demand 100% coverage of all high and low level assurance policies. Some future space exploration projects suggest the engineering of some of the most complex man-rated software systems. As stated in the Columbia Accident Investigation Board Report, the inability to thoroughly apply the required certification protocols had been determined to be a contributing factor to the loss of STS-107, Space Shuttle Columbia.

Schumann and Visser's (2006) discussion suggests that the current certification methodologies are prohibitively expensive for systems of such complexity. A detailed analysis by Lowry (2002) indicates that at the present moment the certification cost of mission-critical space software exceeds its development cost. The challenges of certifying and re-certifying avionics software has led NASA to initiate a number of advanced experimental software development and testing platforms, such as the MDS (Rasmussen et al., 2005), as well as a number of program synthesis, modelling, analysis, and verification techniques and tools, such as The JavaPathFinder (Brat et al., 2005), the CLARAty project (Volpe et al., 2001), Project Golden Gate (Dvorak et al., 2004), The New Millenium Architecture Prototype (New-MAAP) (Dvorak, 2002). The high cost and demands of man-rated certification have motivated the experimental development of several accelerated testing platforms (Boehm et al., 2004). A great number of the experimental faster-than-real-time flight software simulators require the parallelization of previously sequential real-time algorithms.

In Perrow (1999), studies the risk factors in the modern high technology systems. His work identifies two significant sources of complexity in modern systems: *interactions* and *coupling*. The systems most prone to accidents are those with *complex* interactions and *tight* coupling. With the increase of the size of a system, the number of functions it has to serve, as well as its interdependence with other systems, its interactions become more incomprehensible to human and machine analysis and this can cause unexpected and anomalous behaviour. Tight coupling is defined by the presence of time-dependent processes, strict resource constraints, and little or no possible variance in the execution sequence. Perrow classifies space missions in the riskiest category since both hazard factors are present. We argue that the notions of *concurrency* and *time* are the most critical elements in the design and implementation of an embedded autonomous space system. According to a study on concurrent models of computation for embedded software by Lee and Neuendorffer (2005), the major contributing factors to the development and design complexity of such systems are the underlying sequential memory models and the lack of first class representation of the notions of time and concurrency in the applied programming languages.

In this section, we present the design and implementation of a first *concurrency* and *time* centred framework for *verification* and *semantic parallelisation* of real-time C++ within the JPL MDS framework. Our notion of *semantic parallelisation* implies the thread-safe concurrent execution of system algorithms that utilise shared data, based on the application's semantics and invariants. As a practical industrial-scale application, we demonstrate the parallelisation and verification of the MDS' goal networks, a critical component of the JPLs MDS.

7.1 Temporal constraint networks

A temporal constraint network (TCN) defines the goal-oriented operation of a control system in the context of a system under control. The TCNs application is at the core of the Jet Propulsion Laboratory's MDS (Rasmussen et al., 2005) state-based and goal-oriented unified architecture for testing and development of mission software. The framework's state- and model-based methodology and its associated systems engineering processes and development tools have been successfully applied on a number of test applications including the physical rovers Rocky 7 and Rocky 8 and a simulated Entry, Descent, and Landing (EDL) component for the Mars Curiosity mission. A TCN consists of a set of TCs and a set of TPs. In this model of goal-driven operation, a time point is defined as an interval of time when the configuration of the system is expected to satisfy a property predicate. The width of the interval corresponds to the temporal uncertainty inherent in the satisfaction of the predicate. Similarly, TCs have an associated interval of time corresponding to the acceptable bounds on the interactions between the control system and the system under control during the performance of a specific activity. A TCN graph topology represents a snapshot at a given time of the known set of activities the control system has performed so far, is currently engaged in, and will be performing in the near future up to the horizon of the elaborated plan initially created as a solution for a set of goals. The topology of a TCN must satisfy a number of invariants.

a A TCN is a directed acyclic graph where the vertices represent the set of all TPs (S_{tps}) and the edges the set of all TCs (S_{tcs}).

b For each time point $TP_i \in S_{tps}$, there is a set of temporal constraints that are immediate successors (S_{suuc_i}) of TP_i and a set, S_{pred_i}, consisting of all of TP_i's immediate predecessors.

c Each temporal constraint $TC_j \in S_{tcs}$ has exactly one successor TP_{succ_j} and one predecessor TP_{pred_j}.

d For each pair {TP_i, TC_j}, where $TP_i \equiv TC_{succ_j}$, $TC_j \in S_{pred_i}$ must hold. The reciprocal invariant must also be valid, namely for each pair of {TP_i, TC_j} such that $TP_i \equiv TC_{pred_j}$, $TC_j \in S_{succ_i}$.

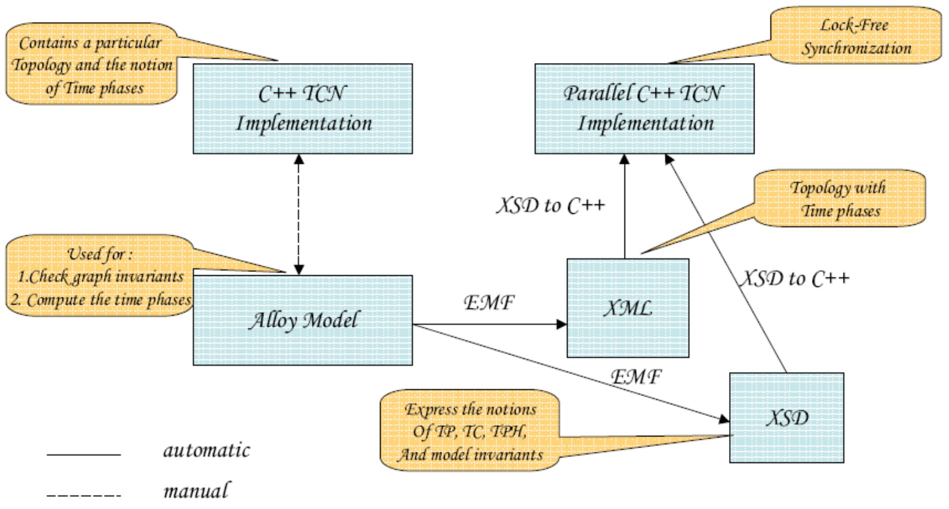
e The firing window of a time point $TP_i \in S_{tps}$ is represented by the pair of time instances $\{TP_{min_i}, TP_{max_i}\}$. Assuming that the current moment of time is represented by T_{now}, then $TP_{min_i} \le T_{now} \le TP_{max_i}$, for every $TP_i \in S_{tps}$.

Any implementation (in C++, Java or another programming language) must operate under the assumptions that the basic TCN invariants are satisfied. Thus, prior to implementing a solution to the TCN constraint propagation problem, it is necessary to guarantee the correctness and consistency of the topology of the goal network.

7.2 Verification and automatic parallelisation framework

In this section, we describe the design, implementation, and practical application of our framework for verification and semantic parallelization of real-time C++ within JPLs MDS framework (Figure 13). The input to the framework is the MDS mission planning and execution module that is based on the definition of TCNs. At the core of the most recent implementations at JPL of this critical module is an optimized iterative algorithm for the real-time propagation of TCs, developed and described by Lou (2002). Constraint propagation poses performance challenges and speed bottlenecks due to the algorithm's frequent execution and the necessary real-time update of the goal network's topology. Our goal is, given the implementation of the optimised iterative propagation scheme and the topology of a particular goal network, to establish the correctness of the core TCN semantic invariants (see Section 7.1) and *automatically* derive an implementation that can be executed concurrently on one of the JPLs experimental testbeds for accelerated testing (Boehm et al., 2004). Our approach for achieving concurrent execution is based on the idea of identifying Time Phases within a goal network, which allow the semantic parallelisation of the constraint propagation algorithm. In our work, we define *semantic parallelisation* as the thread-safe concurrent execution of an algorithm (whose operation is dependent on shared data), derived from the application's semantics and invariants. In the following sections we describe how we reach our goal of verification and semantic parallelization of the mission planning and control module by constructing and executing a formal verification model in alloy (Jackson, 2006) that represents the implementation's core semantics and functionality. We refine a formal modelling and analysis methodology, initially suggested by Rouquette (2008), that helps us analyse the logical properties of the goal network model and automatically derive a meta-model for our parallel solution.

Figure 13 A framework for verification and semantic parallelisation (see online version for colours)


Contains a particular Topology and the notion of Time phases
C++ TCN Implementation
Lock-Free Synchronization
Parallel C++ TCN Implementation
XSD to C++
Topology with Time phases
Used for : 1.Check graph invariants 2. Compute the time phases
Alloy Model
EMF
XML
XSD to C++
EMF
XSD
Express the notions Of TP, TC, TPH, And model invariants
automatic
manual

7.2.1 *The problem of TCN constraint propagation*

A classic solution to the problem of constraint propagation in TCN is the direct application of Floyd-Warshall's all-pairs-shortest-path algorithm (Cormen et al., 2001), offering a complexity of $O(N^3)$, where N is the number of TPs in the TCN topology. Since by definition, the goal of the TCN propagation algorithm is to compute the real-time values of the network's TCs, the algorithm is frequently executed and, given the massive scale of a real world goal network, can cause significant bottleneck for the overall system's performance. In Lou (2002), describes an innovative and effective TCN propagation scheme with a complexity close to linear. Lou's TCN propagation is based on the concept of alternating forward and backward propagation passes. A forward pass updates the time interval at each time point by considering only its incoming TCs (Algorithm 19). Similarly, a backward pass recomputes the time windows at each time point by considering only its outgoing TCs (Algorithm 20). The scheme utilises a shared container, named a *propagation queue*, to keep track of all TPs whose successor TPs' windows are about to be updated next (during a forward pass) and all TPs whose predecessor TPs' windows are about to be updated next (during a backward pass). A forward pass begins by selecting all TPs with no predecessors and inserts them into the propagation queue. A backward pass begins by selecting all TPs with no successors and inserts them into the propagation queue. Each iteration is carried out until:

a An iteration completes without updating any temporal constraints (thus indicating that there are no more updates to be performed during the pass). In this case, the TCN topology is considered to be temporally consistent.

b The iteration has stumbled upon a time window of negative value and the algorithm terminates with the outcome of having a temporally inconsistent network.

As stated by Lou (2002), prior to the execution of the optimised propagation scheme, it is critical to guarantee the validity of the core TCN invariants for the topology of the particular goal network. For example, the propagation scheme operates under the assumption that the goal network graph is cycle free. Should there be cycles, the propagation would enter into an endless loop.

Algorithm 19 Forward pass. Arguments: a reference to the time point about to the updated (tp) and a reference to the global data structure recording the state updates (vstate)

```
1:  $min_{tmp} \leftarrow$ tp.min
2:  $max_{tmp} \leftarrow$ tp.max
3:  **for** $j$ = 0 to tp.preds_size **do**
4:      $min_{tmp} \leftarrow$ std::max($min_{tmp}$, tp.preds[j].pred.min +
        tp.preds[j].min)
5:      $max_{tmp} \leftarrow$ std::min($max_{tmp}$, tp.preds[j].pred.max +
        tp.preds[j].max)
6:  **end for**
7:  **if** tp.min! = mintmp **then**
8:      ASSERT ( tp.min < $min_{tmp}$)
9:      tp.min $\leftarrow min_{tmp}$
10:     vstate.aIncr(vstate.count) {atomically increment the
        state vector's counter}
11: **end if**
12: **if** tp.max! = maxtmp then
13:     ASSERT (tp.max > $max_{tmp}$)
14:     tp.max $\leftarrow max_{tmp}$
15:     vstate.aIncr(vstate.count) {atomically increment the
        state vector's counter}
16: **end if**
17: **return** !($min_{tmp} > max_{tmp}$)
```

Alogrithm 20 Backward Pass. Arguments: a reference to the time point about to the updated (tp) and a reference to the global data structure recording the state updates (vstate)

```
1:  $min_{tmp} \leftarrow$ tp.min
2:  $max_{tmp} \leftarrow$ tp.max
3:  **for** $j$ = 0 to tp.succs size **do**
4:      $min_{tmp} \leftarrow$ std::max($min_{tmp}$, tp.succs[j].succ.min − tp.succs[j].max)
5:      $max_{tmp} \leftarrow$ std::min($max_{tmp}$, tp.succs[j].succ.max − tp.succs[j].min)
6:  **end for**
7:  **if** tp.min! = $min_{tmp}$ **then**
8:      ASSERT (tp.min < $min_{tmp}$)
9:      tp.min $\leftarrow min_{tmp}$
10:     vstate.aIncr(vstate.count) {atomically increment the
        state vector's counter}
11: **end if**
```

```
12: **if** tp.max! = $\max_{tmp}$ **then**
13:     ASSERT (tp.max > $\max_{tmp}$)
14:     tp.max ← $\max_{tmp}$
15:     vstate.aIncr(vstate.count) {atomically increment the
        state vector’s counter}
16: **end if**
17: **return** !($\min_{tmp}$ > $\max_{tmp}$)
```

7.2.2 Modelling, formal verification, and automatic parallelisation

Alloy (Jackson, 2006) is a lightweight formal specification and verification tool for the automated analysis of user-specified invariants on complete or partial models. The Alloy Analyzer is implemented as a front-end, performing the role of a model-finder, to a boolean SAT-solver. Formal verification and modelling of JPLs flight software has been previously demonstrated to be effective and successful by Gluck and Holzmann (2002). We use the Alloy specification language (Jackson, 2006) to formally represent and check the semantics of the temporal constraint networks library (Algorithm 21) and its main invariants (Algorithm 22). In our C++ goal networks implementation we have applied generic programming techniques and concepts (Dos Reis and Stroustrup, 2005), so that we can maintain a higher level of expressiveness. As a result we have achieved a significant similarity in the way the main TCN notions and invariants are expressed in our actual implementation and the Alloy verification models.

We utilise the Alloy Analyzer to implement our semantic parallelisation approach. Our method for semantic parallelisation of the goal network is based on the observation that in a topology we can identify groups of TPs that would allow the concurrent execution of the propagation passes. A possible criterion for identifying such groups would be to identify the TPs in a topology that allow disjoin-access to the shared data. Given the method used to compute the time window $[\mathrm{TP}_{\min_i}, \mathrm{TP}_{\max_i}]$ for each $\mathrm{TP}_i \in \mathrm{S}_{\mathrm{tps}}$, we have observed that the functionally-independent TPs are the TPs that are equidistant (with respect to the longest path) from the root of the graph. Thus, in our methodology, we define a *Time Phase Tph_i* as the set of the TPs $(\mathrm{S}_{\mathrm{Tph}_i})$ in a topology that are equidistant, with respect to the longest path, from the root of the graph. In such a way, by definition, the computations of $[\mathrm{TP}_{\min_a}, \mathrm{TP}_{\max_a}]$ and $[\mathrm{TP}_{\min_b}, \mathrm{TP}_{\max_b}]$ for every pair of $\{\mathrm{TP}_a, \mathrm{TP}_b\}$, such that $\mathrm{TP}_a \in \mathrm{S}_{\mathrm{Tph}_i}$ and $\mathrm{TP}_b \in \mathrm{S}_{\mathrm{Tph}_i}$, are mutually independent and allow disjoin-access to the shared data. With the support of Alloy Analyzer we define and identify the time phases in a goal network graph (Algorithm 23 and Algorithm 24). Figure 14 provides an example of a goal network containing 15 TPs and six-time phases.

Algorithm 21 Definition of the notions of temporal constraint and time point

```
1: sig TC {declaration of the Temporal Constraint signature}
2:     tc_pred: one TP,
3:     tc_succ: one TP
4: sig TP {declaration of the Time Point signature}
5:     tp_preds: set TC,
6:     tp succs: set TC
```

Algorithm 22 Main TCN invariants expressed in the alloy specification language

```
1: all tc:TC | tc in tc.tc pred.tp succs
2: all tc:TC | tc in tc.tc succ.tp preds
3: all tc:TP | some tp.tp preds ⇒ tp.tp preds.tc succ = tp
4: all tc:TP | some tp.tp succs ⇒ tp.tp succs.tc pred = tp
5: no ∧(tc pred.tp preds) & iden {check for cycles}
6: no ∧(tc succ.tp succs) & iden {check for cycles}
```

Algorithm 23 Definition of the notions of time phase and TCN (with time phases)

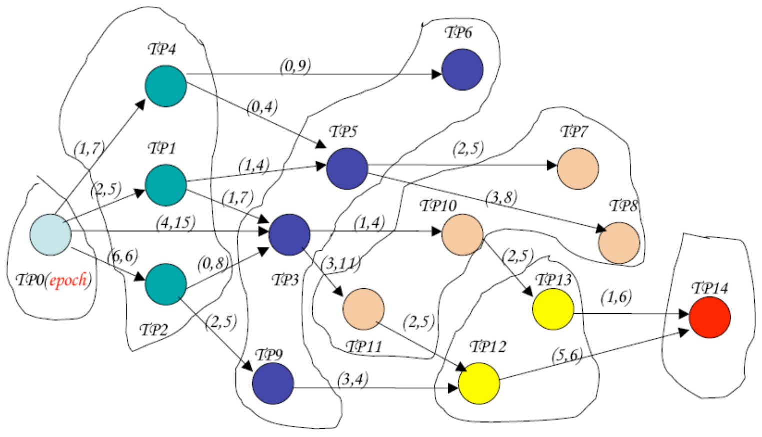
```
1: sig Tph {declaration of the Time Phase signature}
2:     events: set TP,
3:     next: lone Tph,
4:     tcn: one TCN
5: sig TCN {declaration of the TCN signature}
6: epoch: TP,
7: tps: set TP,
8: tcs: set TC,
9: init: one Tph
```

Algorithm 24 Main time phase invariants expressed in the alloy specification language

```
1: all p:Tph
2:     p.events.tp succs.tc succ in p.∧next.events
3:     p.events.tp preds.tc pred in p.∧ ~next.events
4:     p in p.tcn.init.*next
5:     p.events in p.tcn.tps
6:     no p.events & p.∧(next).events
```

Having identified the time phases in our temporal constraint network specification in Alloy, the aim of the rest of our tool-chain is to *automatically* derive the C++ implementation of the parallel solution through a number of code transformation techniques. Following Rouquette's (2008) methodology for model transformation through the application of the object constraint language (OCL) and the eclipse modelling framework (EMF), we are able to automatically derive an intermediary XML and XSD representations of the graph's topology and the TCN semantic notions, respectively. We apply an XML parser (XercesC) and a CodeSynthesis XSD transformation tool to deliver the C++ implementation of the goal network and our parallel propagation method.

Figure 14 A parallel TCN topology with 15 time points and six-time phases (see online version for colours)



To achieve higher safety and better performance, our parallel propagation scheme employs a number of innovative multi-processor synchronisation techniques. In our implementation we have encountered and addressed the following challenges:

1 Achieving low-overhead parallelisation. Our experiments indicated that the wide-spread Pthreads are computationally expensive when applied to the parallel propagation algorithm. Given the frequent real-time changes in the graph topology, employing a thread per iteration for the computations of each time phase comes at a prohibitive cost. To avoid this problem, we have incorporated in our design the application of the Intel tasks from the Threading Building Blocks Library (Intel, 2006). Our experiments indicate that the Intel tasks provide low-cost overhead when applied in the concurrent execution of the forward and backward passes of the propagation scheme.

2 Allowing fast and safe access to the shared data. The parallel algorithm requires the safe and efficient concurrent synchronisation of its shared data: the propagation queue and the vector containing control data (reflecting the updates during an iteration). By the definition of our algorithm, the propagation queue is synchronised by allowing only disjoint-access writes. While the access to the shared vector is less frequent, its concurrent synchronisation is more challenging since we do not have a guarantee that the concurrent writes would be disjoint. The application of mutual exclusion locks is a possible but likely an ineffective solution due to the risks of deadlock, livelock, and priority inversion. Moreover, the interdependency of processes implied by the use of locks diminishes the parallelism of a concurrent system. A lock-free object guarantees that within a set of contending processes, there is at least one process that will make progress within a finite number of steps. We have employed the implementation of the lock-free vector described in Chapter 3 in order to meet our goals for thread-safe and effective non-blocking synchronization. The lock-free vector provides the functionality of the popular STL C++ vector as

well as linearisable and safe operations with complexity of $O(1)$ and fast execution (outperforming the STL vector protected by a mutex by a factor of 10 or more).

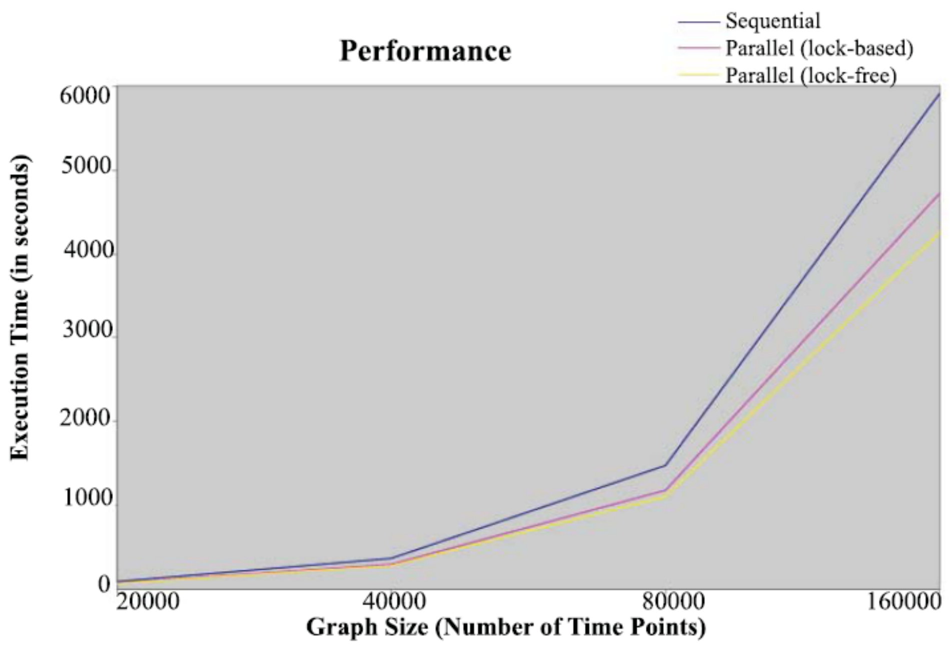
A number of graph properties, in a particular TCN topology, have a significant impact on the application and performance of the parallel propagation scheme. We expect better performance (with respect to the sequential propagation scheme) when:

1. The computational load per time point is high. This is the case of a real-world massive-scale goal network. For instance, instructing Mars Curiosity to autonomously find its way in a Martian crater, probe the soil, capture images, and communicate to Mission Control will result in a goal network containing tens or hundreds of thousands of TPs. In a small experimental graph topology with a low computational cost per time point (such as a few arithmetic operations), a single processor computation will perform best (when we take into account the parallelisation overhead).
2. Time phases with large number of TPs: a topology implying a sequential ordering of the planned events will not benefit from a parallel propagation scheme. The parallel propagation algorithm is beneficial to goal networks representing a large number of highly interactive concurrent system processes.

7.3 Framework application for accelerated testing

The presented design and implementation of our parallel propagation technique enable the incorporation of the optimized propagation approach described by Lou (2002) in an experimental framework for accelerated testing currently still under development at NASA. Accelerated testing platforms suggest a paradigm shift in the certification process employed by NASA from system testing with the actual flight hardware and software to accelerated cost-effective certification using hardware simulators and distributed software implementations. Such frameworks aim faster-than-real-time testing and analysis of the complex software interactions in JPLs autonomous flight systems. A number of these platforms require automated refactoring of previously sequential code into modular parallel implementations. Preliminary results reported in academic work (Boehm et al., 2004) as well as experience reports from a number of commercial tools (such as Simics by Virtutech and ADvantage BEACON by Applied Dynamics International) suggest the possible speedup of the flight system testing by a significant factor. We have followed Rouquette's methodology (Rouquette, 2008) that suggests the application of formal modelling and validation techniques that provide certification evidence for a number of functional dependencies in order to compensate for the added hazards in establishing the fidelity of the simulators. Due to the incomplete status of the accelerated testing framework as well as the lack of the actual flight hardware, it is difficult to measure a priori the effect of our parallel propagation scheme in achieving acceleration (with respect to the execution on the actual flight hardware) in the process of flight software testing. To gain insight of the possible performance gains and the algorithm's behaviour we ran performance tests on a conventional Intel IA-32 SMP machine with two 2.0GHz processor cores with 1 GB shared memory and 4 MB L2 shared cache running the MAC OS X operating system. In our performance analysis we have measured the execution time in seconds of two versions of our parallel propagation algorithm (one applying mutually exclusive locks and the other relying on non-blocking synchronisation) and the original sequential scheme presented by Lou (2002). In the experiments (Figure 15), we have generated a number of TCN graph topologies (each consisting of 4 to 8 Time Phases), in a manner similar to the pseudo-random graph generation methodology described in Dick et al. (1998). In the presented results on Figure 15, the *x*-axis represents the average measured execution time (in seconds) of each propagation scheme and the *y*-axis represents the number of TPs in the exponentially increasing graph size (starting with a graph of 20,000 TPs and reaching a TCN having 160,000 TPs). In the experimental setup we observed that the parallel propagation algorithm offers effective execution and a considerable speedup in all scenarios on our dual-core platform. We measured performance acceleration reaching 28% in the case of the non-blocking implementation and 20% for our algorithm relying on mutually exclusive locks. Lock-free algorithms deliver significant speedup in applications utilising shared data under high contention (Dechev et al., 2006). In a scenario like our parallel TCN propagation scheme with medium or low contention on the shared data, besides safety and prevention of priority inversion and deadlock, a lock-free implementation can guarantee better scalability.

Figure 15 Performance analyses (see online version for colours)



Notes: x-axis represents the number of TPs in each experimental TCN topology, y-axis represents the execution time in seconds of each of the three propagation algorithms.

The notions of time and concurrency are of critical importance for the design and development of autonomous space systems. The current certification methodologies do not reach the level of detail of providing guidelines for the development and validation of concurrent and real-time software. The increasing number of complex interactions and tight coupling of the future autonomous space systems pose significant challenges for their development and man-rated certification. A number of platforms for accelerated testing suggest a paradigm shift by applying a combination of modelling and verification methods, code generation tools, and software parallelisation for establishing a cost-effective and reliable certification process. In the light of the challenges posed by the design and development of these highly experimental approaches, we presented in this work a first time and concurrency-centred framework for validation and semantic parallelization of real-time C++ within JPLs MDS framework. We demonstrated the application of our framework in the validation of the semantic invariants of the Temporal Constraint Network Library. TCNs are at the core of the mission planning and control architecture of the MDS framework. In addition, we presented an approach for automatic semantic parallelisation of the propagation scheme establishing the consistency of the TCs in a goal network. Our parallel propagation scheme is based on the identification of time phases within a goal network and is implemented through the application of model transformation and formal analysis techniques to the model specifications of the TCN semantics. We have relied on innovative lock-free synchronisation techniques to achieve better performance and higher safety of our parallel implementation. Our preliminary tests indicate that our parallel propagation approach can support cost-effective and reliable flight software certification of control modules based on massive real-world goal networks.

8 Conclusions

Future robotic space missions are expected to embed a large array of highly complex and autonomous processes. Achieving safe and efficient concurrent synchronisation is of critical importance to the engineering of future robotic spacecraft software. In this work we, studied the state-of-the-art non-blocking semantics and programming techniques and their applicability in mission critical code. While difficult to design and implement, non-blocking data structures are known to eliminate the hazards of deadlock, livelock, and priority inversion (associated with the application of mutual exclusion). The STL C++ vector is a widely popular data structure that is also commonly used in the MDS and its Data Management Services Module. Its lock-free implementation is challenging due to its dynamic memory management, random access operations with complexity of $O(1)$, and tail access and update operations with complexity $O(1)$. In this work, we showed the first design and implementation of a lock-free shared vector. Our lock-free vector is portable (using only single word read, write, and CAS operations), fast (outperforming the best optimized lock-based approaches by a factor of 10 or more), and scalable (having the ability to efficiently handle heavy contention and demonstrating a significant performance and semantic advantage over the application of non-blocking transactions). The ABA problem is a fundamental problem to all CAS-based systems. At the present moment of time, the literature does not provide a generic and practical ABA solution and coping with the hazards of ABA is left to the ingenuity of the software designer. To achieve an ABA-free implementation, we introduced the $\lambda\delta$ approach, a generic methodology for ABA avoidance for lock-free linearisable designs. According to our performance evaluation, the $\lambda\delta$ approach offers speeds of execution comparable to the application of the architecture-specific CAS2 instruction (used for version counting). This result indicates that the $\lambda\delta$ approach is of particular importance to the application of non-blocking synchronisation in embedded robotic missions, where we cannot rely on the hardware support of complex atomic primitives. In a case study, we demonstrated the application of our lock-free shared vector that played a pivotal role for the design and implementation of a concurrency and time-centred framework for verification and semantic parallelisation of MDS goal networks.

References

Alexandrescu, A. and Michael, M. (2004) 'Lock-free data structures with hazard pointers', C++ User Journal, November.

Barett, A. et al. (2004) 'Mission planning and execution within the mission data system', *Proceedings of the International Workshop on Planning and Scheduling for Space*.

Barnes, G. (1993) 'A method for implementing lock-free shared-data structures', *SPAA '93: Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, pp.261–270, ACM Press, New York, NY, USA.

Boehm, H. et al. (2004) 'Using empirical testbeds to accelerate technology maturity and transition: the SCRover experience', *ISESE '04: Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pp.117–126, IEEE Computer Society, Washington, DC, USA.

Brat, G. et al. (2005) 'Experimental evaluation of verification and validation tools on Martian Rover software', *Formal Methods in Systems Design Journal*, September, Vol. 25, Nos. 2–3, pp.167–198.

Columbia Accident Investigation Board (xxxx) *Columbia Accident Investigation Board Report*, Vol. 1.

Cormen, T. et al. (2001) *Introduction to Algorithms*, MIT Press, Cambridge, MA, USA.

Dechev, D., Pirkelbauer, P. and Stroustrup, B. (2006) 'Lock-free dynamically resizable arrays', in Shvartsman, A.A. (Ed.): *OPODIS*, Vol. 4305 of Lecture Notes in Computer Science, pp.142–156, Springer.

Dechev, D., Rouquette, N., Pirkelbauer, P. and Stroustrup, B. (2008) 'Verification and semantic parallelization of goal-driven autonomous software', *Proceedings of ACM Autonomics 2008: 2nd International Conference on Autonomic Computing and Communication Systems*.

Denney, E. and Fischer, B. (2005) 'Software certification and software certification management systems', *SoftCement05. In Proceedings of the 2005 ASE Workshop on Software Certificate Management*.

Detlefs, D. et al. (2002) 'Lock-free reference counting', *Distrib. Comput.*, Vol. 15, No. 4, pp.255–271, 2002.

Dice, D. and Shavit, N. (2007) 'Understanding tradeoffs in software transactional memory', *Proc. of the 2007 International Symposium on Code Generation and Optimization (CGO)*.

Dick, R.P., Rhodes, D.L. and Wolf, W. (1998) 'TGFF: task graphs for free', *CODES/CASHE '98: Proceedings of the 6th International Workshop on Hardware/Software Codesign*, pp.97–101, IEEE Computer Society, Washington, DC, USA.

Dos Reis, G. and Stroustrup, B. (2005) *Specifying C++ Concepts*, ISO WG21 N1886.

Dvorak, D. (2002) 'Challenging encapsulation in the design of high-risk control systems', *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*.

Dvorak, D. et al. (2004) 'Project Golden Gate: towards real-time java in space missions', *The Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04)*.

Dvorak, D., Ingham, M., Morris, J. and Gersh, J. (2007) 'Goal-based operations: an overview', *Proceedings of AIAA Infotech 2007*.

Dvorak, D., Rasmussen, R. and Starbird, T. (2002) 'State knowledge representation in the mission data system', *Proceedings of IEEE Aerospace Conference*.

Fraser, K. (2004) *Practical Lock-Freedom*, Technical Report UCAM-CL-TR-579, February, University of Cambridge, Computer Laboratory.

Fraser, K. and Harris, T. (2007) 'Concurrent programming without locks', *ACM Trans. Comput. Syst.*, Vol. 25, No. 2, p.5.

Gidenstam, A. et al. (2005) 'Allocating memory in a lock-free manner', *ESA*, pp.329–342.

Gifford, D. and Spector, A. (1987) 'Case study: IBM's system/360-370 architecture', *Commun. ACM*, Vol. 30, No. 4, pp.291–307.

Gluck, R.P. and Holzmann, G. (2002) 'Using SPIN model checker for flight software verification', *Proceedings of the 2002 IEEE Aerospace Conference*.

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

Harris, T.L., Fraser, K. and Pratt, I.A. (2002) 'A practical multi-word compare-and-swap operation', *Proceedings of the 16th International Symposium on Distributed Computing*.

Hendler, D., Shavit, N. and Yerushalmi, L. (2004) 'A scalable lock-free stack algorithm', *SPAA '04: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pp.206–215, ACM Press, New York, NY, USA.

Herlihy, M. and Shavit, N. (2008) *The Art of Multiprocessor Programming*, March, Morgan Kaufmann, 30 Corporate Drive, Suite 400, Burlington, MA 01803, USA.

Herlihy, M. et al. (2005) 'Nonblocking memory management support for dynamic-sized data structures', *ACM Trans. Comput. Syst.*, Vol. 23, No. 2, pp.146–196.

Herlihy, M., Luchangco, V. and Moir, M. (2003) 'Obstruction-free synchronization: double-ended queues as an example', *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, p.522, IEEE Computer Society, Washington, DC, USA.

Intel (2006) *Reference for Intel Threading Building Blocks, Version 1.0*, April.

Intel. (2007) *IA-32 Intel Architecture Software Developer's Manual*, Vol. 3, System Programming Guide.

ISO/IEC 14882 International Standard (1998) *Programming Languages C++*, American National Standards Institute, September.

Jackson, D. (2006) *Software Abstractions: Logic, Language and Analysis*, The MIT Press, 55, Hayward Street, Cambridge, MA 02142.

Lamport, L. (1979) 'How to make a multiprocessor computer that correctly executes programs', *IEEE Transactions on Computers*, September, Vol. C-28, No. 9, pp.690–691.

Lee, E.A. and Neuendorffer, S. (2005) 'Concurrent models of computation for embedded software', *IEE Proceedings on Computers and Digital Techniques*, March.

Lou, J. (2002) *An Efficient Algorithm for Propagation of Temporal Constraint Networks*, NASA Tech Brief Vol. 26, No. 4 from JPL New Technology Report NPO-21098, April.

Lowry, M.R. (2002) 'Software construction and analysis tools for future space missions', in Katoen, J-P. and Stevens, P. (Eds.): *TACAS*, Vol. 2280 of Lecture Notes in Computer Science, pp.1–19, Springer.

Michael, M. (2003) 'CAS-based lock-free algorithm for shared Deques', *Euro-Par 2003: The Ninth Euro-Par Conference on Parallel Processing*, LNCS, Vol. 2790, pp.651–660.

Michael, M.M. (2002) 'High performance dynamic lock-free hash tables and list-based sets', *SPAA '02: Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pp.73–82, ACM Press, New York, NY, USA.

Michael, M.M. (2004a) 'Hazard pointers: safe memory reclamation for lock-free objects', *IEEE Trans. Parallel Distrib. Syst.*, Vol. 15, No. 6, pp.491–504.

Michael, M.M. (2004b) 'Scalable lock-free dynamic memory allocation', *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pp.35–46, ACM Press, New York, NY, USA.

Michael, M.M. and Scott, M.L. (1995) *Correction of a Memory Management Method for Lock-Free Data Structures*, Technical Report TR599.

Perrow, C. (1999) *Normal Accidents*, September, Princeton University Press, 41 William St, Princeton, NJ 08540.

Rasmussen, R., Ingham, M. and Dvorak, D. (2005) 'Achieving control and interoperability through unified model-based engineering and software engineering', *AIAA Infotech at Aerospace Conference*.

Reinholtz, K. (2008) 'Atomic reference counting pointers', C++ User Journal, December.

Rouquette, N. (2008) 'Analyzing and verifying UML models with OCL and alloy', *EclipseCon 2008*.

RTCA (1992) *Software Considerations in Airborne Systems and Equipment Certification (DO-178B)*.

Schumann, J. and Visser, W. (2006) 'Autonomy software: V&V challenges and characteristics', *Proceedings of the 2006 IEEE Aerospace Conference*.

Shalev, O. and Shavit, N. (2003) 'Split-ordered lists: lock-free extensible hash tables', *PODC '03: Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing*, pp.102–111, ACM Press, New York, NY, USA.

Spear, M. et al. (2007) 'Alert-on-update: a communication aid for shared memory multiprocessors', *PPoPP '07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp.132–133, ACM, New York, NY, USA.

Spear, M. et al. (2007) 'Nonblocking transactions without indirection using alert-on-update', *SPAA '07: Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pp.210–220, ACM, New York, NY, USA.

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

Sundell, H. and Tsigas, P. (2004) 'Lock-free and practical doubly linked list-based Deques using single-word compare-and-swap', *OPODIS*, pp.240–255.

Volpe, R. and Peters, S. (2003) 'Rover technology development and mission infusion for the 2009 mars science laboratory mission', *7th International Symposium on Artificial Intelligence, Robotics, and Automation in Space*, May.

Volpe, R. et al. (2001) 'The CLARATy architecture for robotic autonomy', *IEEE Aerospace Conference*, March.

Vries, E. et al. (2008) 'Uniqueness typing simplified', *Implementation and Application of Functional Languages: 19th International Workshop, IFL 2007*, Freiburg, Germany, 27–29 September 2007.

Wagner, D. (2005) 'Data management in the mission data system', *Proceedings of the IEEE System, Man, and Cybernetics Conference*.