Reliable and Efficient Concurrent Synchronization for Embedded Real-Time Software

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

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

Abstract

The high degree of autonomy and increased complexity of future robotic spacecraft pose significant challenges in assuring their reliability and efficiency. To achieve fast and safe concurrent interactions in mission critical code, we survey the practical state-of-the-art nonblocking programming techniques. We study in detail two nonblocking approaches: (1) CAS-based algorithms and (2) Software Transactional Memory. We evaluate the strengths and weaknesses of each approach by applying each methodology for engineering the design and implementation of a nonblocking shared vector. Our study investigates how the application of nonblocking synchronization can help eliminate the problems of deadlock, livelock, and priority inversion and at the same time deliver a performance improvement in embedded real-time software.

1 Objectives

[1] Future space exploration projects, such as Mars Science Laboratory (MSL) [23], demand the engineering of some of the most complex embedded software systems. The notion of concurrency is of critical importance for the design and implementation of such systems. The process-oriented software development and certification protocols (such as [20]) often do not reach the level of detail of providing guidelines for the engineering of reliable concurrent software. In this work, we present a detailed survey of the state-of-the-art *nonblocking* programming techniques that can help in implementing *efficient* and *safe* concurrent interactions in mission critical embedded code.

[1]This is the authors' version of the work. It is posted here by permission of the publisher. Not for redistribution. The definitive version is published in Proceedings of 3rd IEEE International Conference on Space Mission Challenges for Information Technology (IEEE SMC-IT 2009), Pasadena, California, July 2009.

1.1 Parallelism and Complexity

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 optimized in some scenarios by utilizing fine-grained locks or context-switching. Often due to the resource limitations of flight-qualified hardware, optimized lock mechanisms are not a desirable alternative [18]. 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 behavior.

1.2 Nonblocking Synchronization

To achieve higher safety and gain performance, we suggest the application of *nonblocking synchronization*. A concurrent object is *nonblocking* if it guarantees that *some* process in the system will make progress in a *finite* amount of steps [14]. An object that guarantees that *each* process will make progress in a *finite* number of steps is defined as *wait-free*. *Obstruction-freedom* [13] is an alternative nonblocking condition that ensures progress if a thread eventually executes in *isolation*. It is the weakest nonblocking property and obstruction-free objects require the support of a contention manager to prevent livelocking. Nonblocking designs most commonly are founded on a set of atomic primitives supported by the hardware architecture.

The most ubiquitous and versatile data structure in the ISO C++ Standard Template Library [22] is *vector*, offering a combination of dynamic memory management and constant-time random access. Because of the vector's wide use and challenging parallel implementation of its nonblocking dynamic operations, we illustrate the efficiency of each nonblocking approach discussed in this work with respect to its applicability for the design and implementation of a shared nonblocking vector. A number of pivotal concurrent applications in the Mission Data System [15] framework employ a shared STL vector (in all scenarios protected by mutually exclusive locks). Such is the Data Management Service library described by Wagner in [24].

2 Engineering a Nonblocking Data Structure

Lock-free and wait-free algorithms exploit a set of portable atomic primitives such as the word-size Compare-and-Swap (CAS) instruction [9]. The design of nonblocking data structures poses significant challenges and their development and optimization is a current topic of research [8], [14]. The Compare-And-Swap (CAS) atomic primitive (commonly known as Compare and Exchange, *CMPXCHG*, on the Intel *x86* and *Itanium* architectures [16]) 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 [16]). The application of a CAS-controlled speculative manipulation of a shared location (L_i) is a fundamental programming technique in the engineering of nonblocking algorithms [14] (an example is shown in Algorithm 1). In our pseudocode we use

Algorithm 1 CAS-controlled speculative manipulation of L_i

```
1: repeat
2:     $value\_type\ A_i = L_i\hat{}$
3:     $value\_type\ B_i = fComputeB$
4: until $CAS(L_i, A_i, B_i) == Bi$
```

the symbols ^, &, and . to indicate pointer dereferencing, obtaining an object's address, and integrated pointer dereferencing and field access. When the value stored at L_i is the control value of a CAS-based speculative manipulation, we call L_i and $L_i\hat{}$ *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 [8]. 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.

Linearizability [14] is an important correctness condition for concurrent objects: a concurrent operation is linearizable if it appears to execute instantaneously in a given point of time between the time τ_1 of its invocation and the time τ_2 of its completion. The implementations of many nonblocking data structures require the update of two or more memory locations in a linearizable fashion [3], [8]. The engineering of such operations (e.g. *push_back* and *resize* in a shared dynamically resizable array) is critical and particularly challenging in a CAS-based design. Harris et al. propose in [10] a software implementation of a multiple-compare-and-swap (MCAS) algorithm based on CAS. This software-based MCAS algorithm has been applied by Fraser in the implementation of a number of lock-free containers such as binary search trees and skip lists [7]. 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. However, the MCAS implementation employs a number of techniques (such as pointer bit marking and the use of Descriptors) that are useful for the design of practical lock-free systems. A common programming technique applied for the implementation of the complex nonblocking operations is the use of a *Descriptor Object* [3], [8]. A Descriptor is an object that allows an interrupting thread help an interrupted thread complete successfully.

A number of advanced Software Transactional Memory (STM) libraries provide nonblocking transactions with dynamic linearizable operations [5], [21]. Such transactions can be utilized for the design of nonblocking containers [21]. As our performance evaluation demonstrates, the high cost of the extra level of indirection and the conflict detection and validation schemes in STM systems does not allow performance comparable to that of a hand-crafted lock-free container that relies solely on the application of portable atomic primitives. Sections 2.3 and 2.5 describe in detail the implementation of a nonblocking shared vector using CAS-based techniques and STM, respectively. Section 3 provides analysis of the suggested implementation strategies and discusses the performance evaluation of the two approaches.

2.1 Design Goals

In this section we synthesize the most desirable characteristics of a nonblocking shared vector.

a. thread-safety: the data should be accessible to multiple processors at all times

b. lock-freedom: guarantee (at least) lock-free progress of the container's operations

c. portability: do not rely on uncommon architecture-specific instructions

d. easy-to-use interfaces: offer the interfaces, functionality, and guarantees 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, levels of indirection, and costly conflict detection and validation schemes, minimize the time spent on redundant and speculative computations and the number of calls to costly atomic primitives

2.2 Implementation Concerns

We provide a brief summary of the most important implementation concerns for the practical and portable design of a nonblocking dynamic array. The following sections discuss the implementation issues related to guaranteeing portability, meeting the requirements for linearizability, preventing race conditions, coping with the ABA problem, and incorporating nonblocking memory management and allocation schemes.

2.2.1 Portability

Virtually at the core of every known synchronization technique is the application of a number of hardware atomic primitives. The semantics of such primitives vary depending on the specific hardware platform. There are a number of architectures that support some hardware atomic instructions that can provide greater flexibility such as the Load-Link/Store Conditional (LL/SC) supported by the PowerPC, Alpha, MIPS, and the ARM architectures or instructions that perform atomic writes to more than a single word in memory, such as the double-compare-and-swap instruction (DCAS) [4]. The hardware support for such atomic instructions can vastly simplify the design of a nonblocking algorithm as well as offer immediate solutions to a number of challenging problems such as the ABA problem [19]. However, to maintain portability across a large number of hardware platforms, the design and implementation of a nonblocking algorithm cannot rely on the support of such atomic primitives. The most common atomic primitive that is supported by a large majority of hardware platforms is the single-word CAS instruction.

2.2.2 Linearizability Guarantee

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

a. *Extra Level of Indirection:* Reference semantics [22] must be assumed in case the data being manipulated is larger than a memory word size or the approach relies on the application of smart pointers or garbage collection for each individual element in the shared container

b. *Barnes-style announcement [3]:* Often referred to as a *Descriptor Object*, a Barnes-style announcement stores a description of a pending operation on a given memory location. It allows the interrupting threads help the interrupted thread complete an operation rather than wait for its completion

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

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

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

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

2.2.3 Interfaces of the Concurrent Operations

According to the ISO C++ Standard [17], the STL containers' interfaces are inherently sequential. The next ISO C++ Standard [1] is going to include a concurrent memory model [2] and possibly a blocking threading library. In Table 2 we

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

Table 1. Vector - Operations

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

Table 2. Interfaces of STL Vector

show a brief overview of some of the basic operations of an STL vector. Consider the sequence of operations applied to an instance, vec, of the STL vector: *vec[vec.size()-1]; vec.pop_back();*. In an environment with *concurrent* operations, we cannot have the guarantee that the element being deleted by the *pop_back* is going to be the element that had been read earlier by the invocation of *operator[]*. Such a *sequential* history is just one of the several legal sequential histories that can be derived from the concurrent execution of the above operations. While the STL interfaces have proven to be efficient and flexible for a large number of applications [22], to preserve the semantic behavior implied by the sequential definition of the STL semantics, one can either rely on a library with atomic transactions [5], [21] or alternatively define concurrent STL interfaces adequate with respect to the applied consistency model. In the example we have shown, it might be appropriate to modify the interface of the *pop_back* operation and return the element being deleted instead of the *void* return type specified in STL. Such an implementation efficiently combines two operations: reading the element to be removed from the container and removing the element. Should we prefer to keep the STL standard interface of $void$ *pop_back()*, the task of obtaining the value of the removed element in a concurrent nonblocking execution might be quite costly and difficult to implement. Based on the shared containers' usage, observing the possibilities for such combinations can deliver better usability and performance advantages in a nonblocking implementation. Other possibly beneficial combinations of operations are 1) CAS-based *read-modify-write* at location L_i that unifies a random access read and write at location L_i and 2) the $push\_back$ of a block of tail elements.

2.3 Overview of the Lock-free Operations

In this section we present a brief overview of the most critical lock-free algorithms employed by a CAS-based shared vector (see [3] for the full set of the operations of the first lock-free dynamically resizable array). To help tail operations update the size and the tail of the vector (in a linearizable manner), the design presented in [3] suggests the application of of a helper object, named "`Write Descriptor (WD)`" that announces a pending tail modifications and allows interrupting threads help the interrupted thread complete its operations. A pointer to the WD object is stored in the "`Descriptor`" together with the container's size and a reference counter required by the applied memory management scheme. The approach avoids storage relocation and its synchronization hazards by utilizing a two-level array. Whenever `push_back` exceeds the current capacity, a new memory block twice the size of the previous one is added. The remaining part of this section presents the pseudo-code of the tail operations (*push_back* and *pop_back*) and the random access operations (*read* and *write* at a given location within the vector's bounds).

Algorithm 2 $push\_back\ vector, elem$

```
1: repeat
2:     $desc_{current} \leftarrow vector.desc$
3:     $CompleteWrite(vector, desc_{current}.pending)$
4:     if vector.memory[bucket] = NULL then
5:         $AllocBucket(vector, bucket)$
6:     end if
7:     $wop \leftarrow$
       $new\ WriteDesc(At(desc_{current}.size)\hat{}, elem, desc_{current}.size)$
8:     $desc_{next} \leftarrow new\ Descriptor(desc_{current}.size + 1, wop)$
9: until $CAS(\&vector.desc, desc_{current}, desc_{next})$
10: $CompleteWrite(vector, desc_{next}.pending)$
```

Algorithm 3 Read $vector, i$

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

Algorithm 4 Write $vector, i, elem$

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

Algorithm 5 $pop\_back\ vector$

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

Algorithm 6 CompleteWrite $vector, wop$

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

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

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

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

2.3.1 The ABA Problem

The ABA problem [19] is fundamental to all CAS-based systems. The ABA problem can occur in the CAS-based design of a nonblocking dynamic array in a number of possible ways. One possible hazardous execution can happen like this: assume a thread T_0 attempts to perform a push_back; in the vector's "Descriptor", push_back stores an announcement declaring 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 *Descriptor Object*. 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. As a common technique for overcoming the ABA problem it has been suggested to use a version tag attached to each value. Such an approach demands the application of an atomic instruction such as a CAS2 (compare-and-swap two co-located words), a hardware primitive that is available on some modern Intel architectures. For our nonblocking implementation we cannot assume the availability of such atomic primitives since they are specific to a limited number of hardware platforms. ABA avoidance on CAS-based architectures has been typically limited to two possible approaches:

a. split a 32-bit memory word into a value and a counter portions (thus significantly limiting the usable address space or the range of values that can be stored) [6]

b. apply value semantics (by utilizing an extra level of indirection, i.e. create a unique pointer to each value to be stored) in combination with a memory management approach that disallows the reuse of potentially hazardous memory locations [12], [19] (thus impose a significant performance overhead)

To eliminate the ABA problem of (2), the authors in [3] suggest the application of a memory management scheme such as Herlihy et al.'s *Pass The Buck* algorithm [11] that utilizes a separate thread to periodically reclaim unguarded objects. The vector's vulnerability to (1) (in the absence of CAS2 or LL/SC), can be eliminated by requiring the data structure to copy all elements and store pointers to them.

2.4 STM-based Nonblocking Design

A variety of recent STM approaches [5], [21] claim safe and easy to use concurrent interfaces. The most advanced STM implementations allow the definition of efficient "large-scale" transactions, i.e. *dynamic* and *unbounded* transactions. *Dynamic transactions* are able to access memory locations that are not statically known. *Unbounded transactions* pose no limits on the number of locations being accessed. The basic techniques applied are the utilization of public records of concurrent operations and a number of conflict detection and validation algorithms that prevent side-effects and race conditions. To guarantee progress transactions help those ahead of them by examining the public log record. The availability of nonblocking, unbounded, and dynamic transactions provides an alternative to CAS-based designs for the implementation of nonblocking data structures. The complex designs of such advanced STMs often come with an associated cost:

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

b. *Linearizability:* The linearizability requirements are hard to meet for an unbounded and dynamic STM. To achieve efficiency and reduce the complexity, many STMs offer the less demanding *obstruction-free* synchronization [13]

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

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

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

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

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

2.5 RSTM-based Vector

The Rochester Software Transactional Memory (RSTM) [21] is a word- and indirection-based C++ STM library that offers obstruction-free nonblocking transactions. As explained by the authors in [21], while helping provide lightweight committing and aborting of transactions, the extra level of indirection can cause a dramatic performance degradation due to the more frequent capacity and coherence misses in the cache. In this section we employ the RSTM library (version 4) to build an STM-based nonblocking shared vector. In Algorithms 7, 8, 9, and 10, we present the RSTM-based implementation of the $read$, $write$, pop_back, and $push\_back$ operations, respectively. According to the RSTM API [21], access to shared data is achieved by utilizing 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 a an object opened for writing by a transaction, and 4) a *privatized object* (class $un\_ptr < T >$) representing an object that can be accessed by one thread at a time. These smart pointer templates can be instantiated only with data types derived from a core RSTM object class $stm :: Object$. Thus, we need to wrap each element stored in the shared vector in a class $STMVectorNode$ that derives from $stm :: Object$. Similarly, we define a Descriptor class $STMVectorDesc$ (derived from $stm :: Object$) that stores the container-specific data such as the vector's size and capacity. The tail operations need to modify (within a single transaction) the last element and the Descriptor object (of type $STMVectorDesc$) that is stored in a location L_{desc}. The vector's memory array is named with the string mem. In the pseudo-code in Algorithms 9 and 10 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 7 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 8 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 9 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 10 RSTM vector, operation $push\_back\ v$

```
1: $BEGIN\_TRANSACTION$
2: $sh\_ptr$<STMVectorNode> nv =
   $new\ sh\_ptr$<STMVectorNode>($new$ STMVectorNode(v))
3: $sh\_ptr$<STMVectorDesc> desc =
   $new\ sh\_ptr$<STMVectorDesc>
   ($new$ STMVectorDesc($L_{desc}- > size + 1$))
4: mem[size] = nv
5: $L_{desc}$ = desc
6: $END\_TRANSACTION$
```

3 Analysis and Results

To evaluate the performance of the discussed synchronization techniques, in this section we analyze the performance of three approaches for the implementation of a shared vector:
(1) the RSTM-based nonblocking vector implementation as presented in Section 2.5
(2) an RSTM lock-based execution of the vector's transactions. RSTM provides an option of running the transactional code in a lock-based mode using redo locks [21]. Though blocking and not meeting our goals for safe and reliable synchronization, we include the lock-based RSTM vector execution to gain additional insight about the relative performance gains or penalties that the discussed nonblocking approaches offer when compared to the execution of a lock-based, STM-based container
(3) the hand-crafted CAS-based algorithms design as presented in Section 2.3
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 10.5.6 operating system. We designed our experiments by generating a workload of the various operations. We

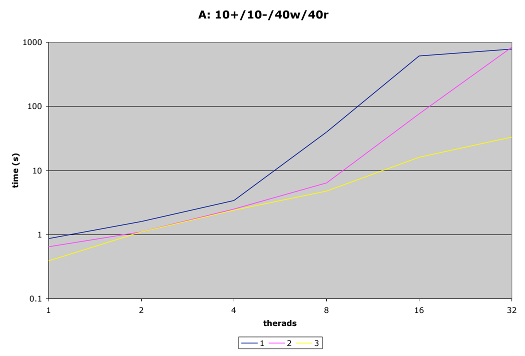

Figure 1. Performance Results A

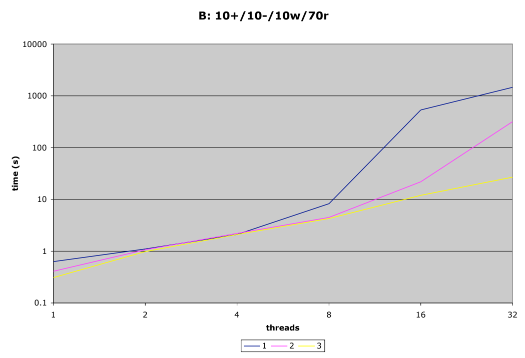

Figure 2. Performance Results B

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 (`push_back` (+), `pop_back` (−), random access `write` (`w`), random access `read` (`r`)). We show the performance graph for a distribution of +:10%, −:10%, `w`:40%, `r`:40% on Figure 1. Figure 2 demonstrates the performance results in a read-many-write-rarely scenario, +:10%, −:10%, `w`:10%, `r`:70%. Figure 3 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 approach from [3] offers simple application and fast execution. The STM-based design offers a flexible programming interface and easy to comprehend concurrent semantics. The main deterrent associated with the application of STM is the overhead introduced by the extra level of indirection and the application of costly conflict detection and validation schemes. According to our performance evaluation, the nonblocking RSTM vector demonstrates poor scalability and its performance progressively deteriorates with the increased volume of operations and active threads in the system. In addition, RSTM transactions offer obstruction-free semantics. To eliminate the hazards of livelocking, the software designers need to integrate a contention manager with the use of an STM-based container. Because of the limitations present in the state of the art STM libraries [21], [5], we suggest that a shared vector design based on the utilization of nonblocking CAS-based algorithms can better serve the demands for safe and reliable concurrent synchronization in mission critical code.

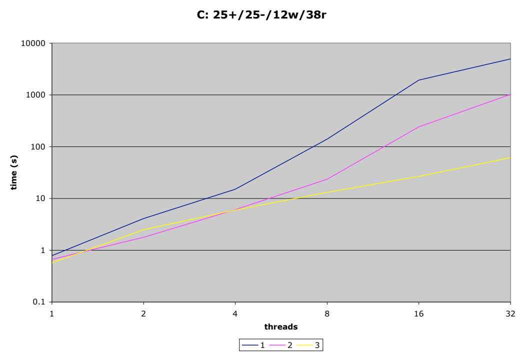

Figure 3. Performance Results C

4 Impact for Space Systems

Modern robotic space exploration missions, such as the Mars Science Laboratory [23], 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 [18] reveals that in July 1997 The Mars Pathfinder mission experienced a number of anomalous system resets that caused an operational delay and loss of scientific data. The follow-up analysis identified the presence of a priority inversion problem caused by the low-priority meteorological process blocking the the high-priority bus management process. The software engineers found out that it would have been impossible to detect the problem with the black box testing applied at the time. A more appropriate priority inversion inheritance algorithm had been ignored due to its frequency of execution, the real-time requirements imposed, and its high cost incurred on the slower flight-qualified computer hardware. The subtle interactions in the concurrent applications of the modern aerospace autonomous software are of critical importance to the system's safety and operation. The presence of a large number of concurrent autonomous processes implies an increased volume of interactions that are hard to predict and validate. Allowing fast and reliable concurrent synchronization is of critical importance to the design of autonomous spacecraft software.

5 Conclusion

In this work we investigated how the application of nonblocking synchronization can help eliminate the problems of deadlock, livelock, and priority inversion in embedded real-time mission critical software. We studied the challenging process of how to design and implement a nonblocking data container by applying 1) CAS-based synchronization and 2) Software Transactional Memory. We discussed the principles of nonblocking synchronization and demonstrated the application of both approaches by showing the implementation of a lock-free shared vector. Our performance evaluation concluded that while difficult to design, CAS-based algorithms offer fast and scalable performance and in a large majority of scenarios outperform the alternative STM-based nonblocking or lock-based approaches by a significant factor. This paper aimed at delivering better understanding of the advantages (over mutual exclusion) as well as the usability and performance trade-offs of the modern nonblocking programming techniques that can be of critical importance for the engineering of reliable and efficient concurrent flight software.

References

[1] P. Becker. Working Draft, Standard for Programming Language C++, ISO WG21 N2009, April 2006.

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

[3] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Lock-Free Dynamically Resizable Arrays. In A. A. Shvartsman, editor, *OPODIS*, volume 4305 of *Lecture Notes in Computer Science*, pages 142–156. Springer, 2006.

[4] D. Detlefs, C. H. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. L. S. Jr. Even better DCAS-based concurrent deques. In *International Symposium on Distributed Computing*, pages 59–73, 2000.

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

[6] D. Dvorak and W. Reinholtz. Hard real-time: C++ versus RTSJ. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 268–274, New York, NY, USA, 2004. ACM.

[7] K. Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, Feb. 2004.

[8] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2):5, 2007.

[9] D. Gifford and A. Spector. Case study: IBM's system/360-370 architecture. *Commun. ACM*, 30(4):291–307, 1987.

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

[11] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2):146–196, 2005.

[12] M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *DISC '02: Proceedings of the 16th International Conference on Distributed Computing*, pages 339–353, London, UK, 2002. Springer-Verlag.

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

[14] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008.

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

[16] Intel. Ia-32 intel architecture software developer's manual, volume 3: System programming guide, 2004.

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

[18] M. R. Lowry. Software Construction and Analysis Tools for Future Space Missions. In J.-P. Katoen and P. Stevens, editors, *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2002.

[19] M. M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[20] RTCA. Software Considerations in Airborne Systems and Equipment Certification (DO-178B), 1992.

[21] M. F. Spear, A. Shriraman, L. Dalessandro, S. Dwarkadas, and M. L. Scott. Nonblocking transactions without indirection using alert-on-update, http://www.cs.rochester.edu/research/synchronization/rstm/v4api.shtml. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 210–220, New York, NY, USA, 2007. ACM.

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

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

[24] D. Wagner. Data Management in the Mission Data System. In *Proceedings of the IEEE System, Man, and Cybernetics Conference*, 2005.