

Transactional Lock-Free Objects for Real-time Java

F. Pizlo M. Prochazka S. Jagannathan J. Vitek
Purdue University

ABSTRACT

Priority inversion is an important concern in providing robust synchronization in real-time systems. When a high-priority task attempts to acquire a lock held by a low priority task, it is often necessary to momentarily resume the execution of the low priority task so as to allow it to leave the critical region safely, ensuring that shared resources are not in an inconsistent state. Once these resources are properly released, the high priority task can proceed. In pathological cases, the priority of several threads may have to be increased, and the high priority tasks can experience unbounded delays.

An alternative approach would record the original values of shared objects whenever they are modified, restoring them if the executing thread is interrupted by a higher-priority one. This approach thus treats the critical section as a lightweight *transaction*. This paper presents an extension to the Real-time Specification for Java with *transactional lock-free* (TLF) objects. Atomic methods of TLF-objects can be accessed concurrently without risking priority inversion. The semantics of our transactions are such that a high-priority thread will always succeed when trying to enter an atomic section. The time to enter is bounded by the number of locations updated within the atomic section. Experimental results undertaken in the context of Ovm, a virtual machine framework for Java that implements the Real-Time Specification for Java, indicates that transactional lock-free objects can improve the responsiveness of high priority threads compared to priority-inheritance based approaches at the cost of a reduction throughput.

1. INTRODUCTION

The Real-Time Specification for Java (RTSJ) [6] holds promise to play a key role in the construction of real-time systems in a type-safe, high-level, programming language. The RTSJ is being evaluated for use in mission-critical systems by the likes of Boeing [24] and JPL [18], and has one high-quality commercial implementation [14], as well as a number of open source incarnations [4, 8] and variants [19, 7, 30, 9, 27]. This paper focuses on the RTSJ programming model, and in particular, on how to write correct and efficient real-time code in the presence of priority preemptive threads and mutual exclusion.

Priority inversion is a well-studied problem in concurrent real-time programming. Avoiding priority inversion is especially important in mission critical or real-time applica-

tions [17, 10, 22, 23]. Priority inheritance and priority ceiling emulation are two well-known protocols that attempt to avoid priority inversion. The priority ceiling emulation (PCE) technique raises the priority of any locking thread to the highest priority of any thread that ever uses that lock (i.e., its priority ceiling). This requires the programmer to supply the priority ceiling for each lock. In contrast, the priority inheritance protocol (PIP) will raise the priority of a thread only when holding a lock causes it to block a higher priority thread. When this happens, the low priority thread inherits the priority of the higher priority thread it is blocking. Yet another alternative is to have privileged threads, for example those executing on behalf of the operating system. These threads can often disable interrupts or preemption that effectively locks lower-priority threads from acquiring critical resources. Regardless of the approach, once a thread enters a synchronized section its locks cannot be summarily relinquished without potentially violating synchronization invariants.

Even with a priority inversion avoidance protocol a high-level thread can experience an unbounded delay if a low priority thread's does not relinquish its lock, or, in less extreme cases, if it takes longer than planned to complete its work. While this situation may be described as a programming error, it is nevertheless a case that can occur, especially in large and complex embedded software such as those found in avionics systems.

The RTSJ supports priority preemptive threads with a minimum of 28 unique priorities. Furthermore, it provides, by default, an implementation of a priority inheritance algorithm and supports priority ceiling emulation. Priority inheritance may be transitive since a low-priority thread may try to acquire resources held by an even lower priority threads. In the RTSJ, application programmers are also faced with the additional problem that some low-priority whose priority is boosted may trigger a garbage collection, effectively blocking the high-priority thread from executing for the entire duration of the collection.

We propose to investigate an alternative concurrency control mechanism based on *transactional lock-free* (TLF) objects. In this scheme, concurrency control is implemented by critical sections which provide atomicity via a simple compiler-assisted transactional mechanism. Whenever a high priority thread tries to enter an atomic method of a TLF-object, the operation always succeeds in time bounded by the number of objects updated in the method. Any lower priority thread

executing within an atomic method will be evicted, and its changes undone. The lower priority thread is guaranteed to be reexecuted when the high priority thread completes its work. Since the overheads to perform rollbacks are charged only to low-priority threads, our scheme favors responsiveness over throughput.

This paper defines the semantics of TLF-objects and describes their implementation within a real-time Java virtual machine. The salient features of our proposal are as follows:

- *A simple programming model* in which priority inversion can not occur and higher priority threads are guaranteed to enter critical sections within a bounded number of steps.
- *Integration with the RTSJ* in a backwards compatible way. No changes to the Java language are required, we introduce annotations that are interpreted by the VM. Our proposal can coexist with traditional synchronization.
- *Efficient implementation* which required modest changes to an existing RTSJ virtual machine and its optimizing compiler.

Transactional extensions to programming languages have received renewed attention of late. This work is closely related to that of Harris [11] and Welc *et.al.* [32]. The main difference between our work and theirs, besides choice of implementation technique, is that we must ensure that space for a transaction is bounded.

The paper is organized as follows. We begin with an overview of relevant features of the Real-time Specification for Java. Section 3 introduces transactional lock-free objects. Section 4 describes the implementation of TLF-objects within Ovm. Section 5 reports on experimental results.

2. REAL-TIME SPECIFICATION FOR JAVA

We overview the salient features of the Real-time Specification for Java (RTSJ). The RTSJ extends Java with support for real-time programming in a backward compatible way. The RTSJ requires no changes to syntax of Java and allow real-time and plain Java codes to co-exist in one virtual machine. One design choice made in the RTSJ is to extend the Java programming model with two abstractions: (a) regions of memory, called scoped memory areas, which are not subject to garbage collection, and (b) real-time threads that never interact with the heap and thus can never interfere with, or be affected by, the garbage collector. Technically real-time threads come in two flavors, `RealtimeThread` and `NoHeapRealtimeThread`, and only the latter is guaranteed not to experience garbage collection pauses. Both kinds of threads can be created with any of the 28 priorities over and above the ten priorities defined in standard Java and be given scheduling parameters that are either periodic, aperiodic, or sporadic.

Priority inversion is avoided by defining *monitor control policies*. The RTSJ overloads the meaning of the `synchronized` keyword by allowing programmers to specify a monitor control policy for each Java language monitor. Two policy

classes are provided by default: `PriorityInheritance` and `PriorityCeilingEmulation` that implement the familiar notions of priority inheritance and priority ceiling respectively. The static method `MonitorControl.setMonitorControl(p)` can be used to set the default policy for the entire virtual machine, which `setMonitorControl(target, policy)` sets the policy for a single object, `target`.

3. TRANSACTIONAL LOCK-FREE OBJECTS

An alternative approach to the above mentioned priority inversion avoidance schemes is to use lightweight transactions instead of monitors to control access to critical sections. For example, one can apply optimistic concurrency semantics [31] to any number of concurrently executing tasks, allowing these tasks to simultaneously enter the same critical section (*i.e.* a sequence of operations protected by the same lock). If the operations performed by these tasks do not conflict they will be allowed to *commit* their changes, making their updates visible to other threads in the program; otherwise, one or more task will be aborted, and their changes discarded. An aborted task may retry execution.

Our approach can be viewed as a hybrid transactional model that combines features of both pessimistic and optimistic concurrency semantics in a manner suited to real-time systems. As in a pessimistic concurrency control model, only one thread is allowed to execute within a TLF-object at any given point. As in an optimistic concurrency control model, a thread can be aborted while executing within a TLF-object, if a higher-priority thread requests access to the same object. As in these other approaches, state information is logged by each thread to ensure that when an abort does occur, updates it has performed can be reverted to ensure that consistency invariants are not violated.

We propose a simple language extension inspired by the work of Harris and Fraser [11] and Welc *et.al.* [32] geared towards priority preemptive real-time systems. We introduce a single new keyword `@atomic` used to declare transactional critical sections:

```
@atomic void update( int i ) {
    if ( i > this.field )
        this.field = i;
}
```

All writes performed within the method, and the methods it invokes, are guaranteed to become visible when the current thread leaves the critical section. In the case a transaction is aborted all changes performed by the thread within the synchronized region are undone and the method is re-executed.

We refer to objects with `@atomic` methods as *transactional lock-free* (TLF) objects¹. A TLF-object acts as a monitor with respect to its atomic methods. There can be only one thread concurrently executing an atomic method of a given

¹Anderson *et.al.* have proposed lock-free shared objects in [2]. The main difference between our works is that we provide a language-based solution, so none the operations performed by an aborted thread can be witnessed by other threads. This is not the case in Anderson's work where only the transactional object is protected, shared variables may be modified by an aborted thread and these will not be undone.

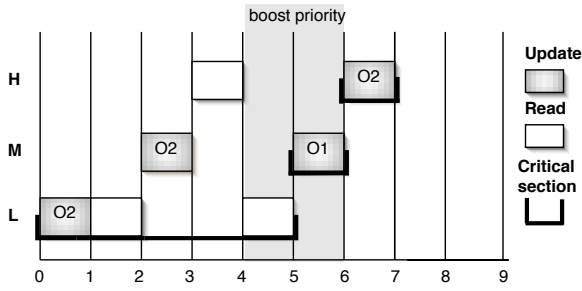


Fig. 1: A sequence of actions performed by different priority threads under priority inheritance trying to enter critical sections guarded by the same lock. Low priority thread L is released at time step 0, mid priority thread M is released at time step 2, and high priority thread is released at time step 3. L and M operate with boosted priorities at time steps 5 and 6, respectively. The execution of H's critical section is delayed until time step 6.

TLF-object. We ensure that a thread is allowed to enter a critical section, only if the TLF-object protecting it is not currently being accessed, or if the TLF object is currently owned by a lower priority thread. In the latter case the lower priority thread is evicted from the critical section, its changes are undone, and it is permitted to re-execute whenever the high priority thread completes its work.

Fig. 2 illustrates the actions undertaken by different threads when they share access to a critical section defined as a transactional object. At time step 0, a low-priority thread transactionally executes regions of code that updates object O2 (which is not necessarily the transactional object but can be any object visible to that thread). Before it completes execution of its transaction, the thread is interrupted by a medium-priority thread causing the modification to O2 to be undone (time step 2). Before the medium priority thread is allowed to execute its transaction, it is interrupted by a high priority thread which transactionally updates O2. Once the high priority completes its work the medium priority thread is allowed to proceed (time step 5). Finally the low-priority thread regains control and completes its work (time step 7 to 9). Note that the low-priority thread must reexecute its actions. Thus, when a lower-priority thread is evicted, it must resume execution from the beginning of the critical section; no intermediate results from its previous aborted execution are preserved.

In contrast, Fig. 1 shows a comparable scenario using a priority-inversion avoidance protocol. In this case, the interruption of the low-priority thread L by medium-priority thread M at time step 2 is allowed because M does not immediately enter the critical section. Thread H preempts M at time step 3, in order to get to execute. The priority of L is boosted at time step 4 and M at time step 5. With a protocol such as priority inheritance, the maximum time that H must wait before it can begin execution within the critical section is only bounded if the length of individual critical sections can be bounded. In contrast, TLF-objects bound the time that a higher-priority thread must wait to enter a critical

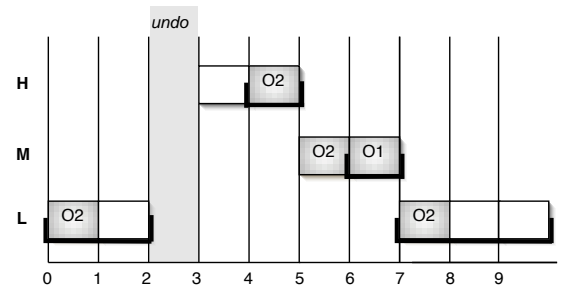


Fig. 2: A sequence of actions performed by different priority threads sharing under transactional semantics. Threads M is released at time step 2, causing thread L to abort and undo its write. Thread H is released at time step 3, preempting M. Thread L is reexecuted at time step 7. Notice that in this example transaction reduce overall throughput but increase responsiveness of the high priority thread.

section as a function of the number of updates that must be undone. Since undoing the effects of an update involves simply restoring the original value of the object at the time the transaction was entered, this cost is significantly easier to calculate and predict than the worst case execution time (WCET) of critical sections². The cost of transaction is that overall throughput is reduced due to the cost of undoing and reexecuting threads.

3.1 Example

Consider the doubly linked list data type, `DLList`, defined in Fig. 3. The class has two atomic methods, `insert()` which performs ordered insertion and `getMax()` which returns the list's largest value. An auxiliary method, `splice()`, inserts a new cell in a list. A program with two threads, a high priority thread H and a low priority L, which interact via a shared `DLList` object is given next (the actions performed by each thread are shown, the indentation hints at the relative start times):

```
L:   shared.insert( obj)
H:   shared.getMax()
```

Assume L is scheduled first, it acquires the shared TLF-object, enters the critical section and start executing the insertion code. If H is scheduled before L completes, the VM will notice that `shared` has been acquired by another thread. By definition of a priority preemptive system, the priority of L *must* be less than the H. This leaves no ambiguity as to what must happen next. L is aborted; if it has already performed some of its updates, they are undone and H is permitted to enter the critical section. Once H completes its work, L will be rescheduled and will re-execute the call to `insert()`.

In this example the cost aborting a call of `insert()` is bounded

²Practitioners still estimate WCETs by empirical means, running the program on different inputs and measuring execution time.

```

class DList {
    Comparable value;
    DList prev, next;

    @atomic DList insert( Comparable v ) {
        DList cur = this, pre = null;
        while ( cur != null && v.gte( cur.value ) )
            { pre = cur; cur = cur.next; }
        if ( pre == null )
            return new DList( value ).splice( this );
        else {
            pre.splice( new DList( value ) );
            return this;
        }
    }

    DList splice( DList after ) {
        DList tmp = this.next;
        this.next = after;
        after.next = tmp;
        after.prev = this;
        return this;
    }

    @atomic Comparable getMax() {
        Comparable max = this.value;
        DList cur = this;
        while ( cur != null ) {
            if ( max.lte( cur.value ) ) max = cur.value;
            cur = cur.next;
        }
        return max;
    }
}

```

Fig. 3: TLF-object example. `insert()` performs an ordered insertion in the list. Calls to `insert` start a new transaction which protects the internal call to `splice()` ensuring that all updates are logged. If the transaction completes changes become visible, otherwise they are undone. The latency of calls to `insert()` are bounded by the cost of restoring the three cells updated within `splice()`.

by the cost of undoing writes to three `DList` objects, while aborting `getMax()` incurs no extra cost.

The semantics of `@atomic` methods are such that the original values of all objects updated within the method, or methods called from it, are logged including objects of other types. Thus the call to `splice()` is also protected by the transaction initiated in `insert()`. Just as with Java synchronization, calling an unprotected method (such as `splice()`) directly from another thread may reveal inconsistent state. For instance, consider the following program:

```

L:   shared.insert( obj )
M:   shared.splice( cell )
H:   shared.insert( obj )

```

If a medium priority thread is released while L is within its critical section, it may observe values modified by L. If H is called before L completes the transaction may be aborted and the values observed by M become stale.

3.2 Semantics

The semantics of TLF have been designed to suit the requirements of RTSJ; in particular we have tried to bal-

ance expressiveness of the programming model with space and time efficiency concerns. TLF-objects are based on a flat non-nested transactional model. Whenever a real-time thread T invokes a method annotated with the `@atomic` keyword on an object `tobj`, an implementation is required to checked if the `tobj` is currently owned by another thread. If it is the case the other thread is aborted and the ownership field of `tobj` is cleared. Thread T then *acquires* ownership of `tobj` and begins executing. When T *exits* from the critical section, all changes it made are finalized and ownership of `tobj` is reset. A thread may exit a critical section *normally*, when the method returns, or *exceptionally*, by throwing an exception from the method. In both cases all changes made by the thread will be finalized.

The original values of all objects updated by a thread between the time a TLF-object is acquired and the critical section is exited are recorded in a log to ensure state consistency if the thread is interrupted prior to its exit of the section. An implementation is required to allocate space for logs at virtual machine start up before any real-time thread are started. To support both plain real-time threads and `NoHeapRealtimeThreads`, logs are allocated in immortal memory. The logging policy is implementation dependent – the only requirement is that space required by the log be bounded. Since we log the original value of an object the first time it is updated, each modified object must be logged at most once. It is permissible to log partial objects. Locations in an array may be logged at most once per write to the location.

An implementation is only required to ensure that objects are in a consistent state when a thread exits its critical section. Attempts to access objects being modified are unchecked programming errors as they may yield stale value of these objects.

Aborted transactions are automatically re-executed. The process is transparent to the application. A thread is neither able observe that a transaction was re-executed nor allowed to explicitly trigger an abort.

Code Constraints. An implementation is allowed to reject any `@atomic` method that may perform blocking operations. This can be validated by an off-line static analysis of real-time code. The analysis only needs to classify methods as, either, safe or unsafe. A method is *safe* if it does not contain synchronized statements. Moreover, safe methods cannot include calls to native methods since such methods can perform blocking operations. In practice, native methods are inspected manually, and declared safe on a case-by-case basis. For the same reason, safe methods cannot make reflective calls, or refer to classes that cannot be statically guaranteed to be available and initialized³. Finally, a method is safe if all methods called in its body can be proved safe.

³Class initialization is an issue for RTSJ programs. Java semantics mandate classes to be loaded and initialized lazily, but to obtain any measure of predictability a RTSJ-virtual machine will likely load all classes aggressively and initialize them ahead of time. Reflection can potentially refer to classes that have not yet been loaded and must be used carefully.

Space Bounds. The upper bound of the log size is user-specified. As the layout of objects and the logging policy are implementation dependent, space requirements are estimated based on the maximum number of objects that can be modified in a critical section. For arrays the number used is that of writes to individual locations. The RTSJ provide a class for this purpose called `SizeEstimator`⁴; we extend this notion to bound transactional object memory requirements `TLFSizeEstimator`.

```
class TLFSizeEstimator extends SizeEstimator {
    TLFSizeEstimator( Method m);
    void reserve( Class c, int count);
    void reserveArray( Class c, int count);
}
```

Each atomic method has an associated estimator. The space required to log the changes performed in the method is declared by calling the `reserve()` and `reserveArray()` methods. `reserve(C, i)` sets aside space to log i instances of class C . `reserveArray(A, i)` sets aside space to log i writes to an array of type A . Once the estimator is complete, it must be registered with the VM using `registerAtomic()` method. The following code fragment register estimators for the methods of Fig. 3.

```
tins = new TLFSizeEstimator( dllist.insert_method);
tins.reserve( DLList.class, 3);
OVM.registerAtomic( tins);

tmax = new TLFSizeEstimator( dllist.getMax_method);
OVM.registerAtomic( tmax);
```

The `insert()` method may update three objects of class `DLList`, while the `getMax()` method performs no updates.

3.3 Discussion

The design of TLF-objects is fully backwards compatible with the RTSJ. Plain RTSJ code can execute on virtual machine supporting TLF-objects. Indeed, programs can use a mixture of TLF and traditional synchronization. The `@atomic` keyword does not require change to Java syntax; it is an annotation consistent with the JSR-175 Metadata extensions to Java. The keyword can also be replaced by a marker exception⁵.

The transactional model adopted here could be relaxed at some cost in complexity and performance. For example, providing support for nested transactions would mean that aborting the execution of one atomic method may lead to a cascade of aborts to undo the effects of other atomic methods. Logging operations become more complex as a transaction must maintain one log per nested transaction to support nested aborts. This implies that the same object may appear in the log of several transactions. Furthermore, the cost of write barriers increases as it is necessary to check in which transaction's log the object has been stored.

A more modest change would entail recursive atomic sections, i.e. allowing a thread to call atomic methods on an object if it already has acquired that object. In the example of Fig. 3, this would allow method `splice()` to be declared

⁴In the RTSJ size estimators are used to set bounds on the size of memory regions.

⁵Marker exceptions are well known idiom, the declaration `@atomic void f() {...}` becomes `void f() throws Atomic {...}`.

atomic. The drawback of this change to the semantics is that some extra checks have to be performed when a thread enters and leaves a critical section, and that it is more difficult to check statically if a method is safe. Furthermore, there would be additional runtime exceptions if a transaction tried to acquire a different object.

Extending our semantics to support a full optimistic transactional model would increase throughput as more than one thread may execute in the same critical section, but would also increase the cost of exiting the region as it would be necessary to check that no conflict on shared state has occurred, and would also increase the program's space requirements as multiple logs for the same object have to be maintained at the same time.

4. OVM IMPLEMENTATION

TLF-objects have been implemented in the RTSJ configuration of the Ovm virtual machine framework. Ovm is an open source framework for building language runtimes. The framework contains more than 150K lines of code and over 2000 classes, including an interpreter, a just-in-time compiler and an ahead-of-time compiler. These components can be specialized and assembled into an *Ovm configuration* customized for a particular problem domain. The RTSJ configuration yields a virtual machine implementing the Real-time Specification for Java. This configuration compiles real-time Java code ahead of time and has a fast user-level threading system based on compiler inserted polling code. The platforms currently supported are x86/RTLinux and PPC/MacOSX. The ahead-of-time compiler produces code competitive with Sun's HotSpot virtual machine.

The main design decision taken in the Ovm implementation of TLF-objects is to limit number of concurrent transactions in order to reduce space requirements and permit a more efficient implementation. Our implementation restricts applications to have *at most one* executing transaction at any given instant. We believe this restriction to be acceptable in practice as we are targeting system with relatively small numbers of threads and where transactional sections are short.

The layout of object, described by their *object header*, is modified with an extra bit that indicates whether the object has been logged by the current transaction.

The implementation of the basic transactional primitives is described in Fig. 4. When a thread attempts to enter an atomic section, it must check that no other transaction is executing. The VM maintains a reference to the current thread within a critical section in `VM.currentTransaction`. If there is an ongoing transaction executing on behalf of another thread, it is immediately aborted. Note that the thread executing within the section must have lower-priority in order for the thread performing the **acquire** operation to be executing. In any case, the transaction ownership field is set to the new thread and the acquisition succeeds. Exiting a critical section, requires clearing all the stamp fields of all logged objects as well as the `currentTransaction` field. Every write to an object is protected by a barrier. We extend the existing barrier code (which enforces scoped memory semantics) to log the object if a transaction is active and the

```

acquire ( thread )
    owner ← VM.currentTransaction
    if owner ≠ null
        abort( owner )
    VM.currentTransaction ← thread

exit ( )
    foreach o in VM.currentLog
        o.stamp ← false
    clearLog()
    VM.currentTransaction ← null

write ( o )
    if VM.currentTransaction ≠ null ∧ o.stamp = false
        log( o )

log ( o )
    log ← VM.currentLog
    if log.length + sizeof( object ) + 2 ≤ log.size
        throw logOverflowError
    addToLog( log, object, sizeof( object ) )
    o.stamp ← true

abort( thread )
    foreach ( addr, i, size ) in VM.currentLog
        copy( addr, 0, log, i, size )
        clearLog()
    thread.postAbortedTransactionException()

```

Fig. 4: Pseudo code for the basic transactional primitives in the Ovm implementation.

object's stamp is not set. The log records triples that consist of the object's address, size, and contents. An object is first added into the log before setting its stamp field to true. In case the size of the log was misspredicted an exception is thrown. As this signals a programming error, the transaction will not be re-executed automatically. The error propagates into user code, and if not caught will terminate the current task. A transaction is aborted by rolling back the changes it performed and posting an aborted exception so that whenever the thread is sheduled next it recieve the exception which has the effect of unrolling the stack out of the critical section.

Finally, we show the translation of an atomic method, a method such as:

```

@atomic void method() {
    ...body ...
}

```

is transformed by the ahead-of-time compiler into

```

void method() throw PragmaNoPollcheck {
    while ( true ) {
        try {
            acquire( currentThread );
            raw_method(); }
        catch (AbortedTransactionException _) { continue; }
        catch (Throwable t) { exit(); throw t; }
        exit();
        break; }
}

```

In Ovm the `PragmaNoPollcheck` exception is a marker that instructs the compiler not to emit pollchecking code within the body of the translated method, thus ensuring that all transactional operation are executed atomically⁶.

Note that in our implementation this translation is performed transparently within the virtual machine, in particular we do not change the signature of the method. Furthermore operations such as `abort()` are Ovm internal and not part of any Java level API.

⁶Poll checks are normally inserted at all potentially recursive method entry, and in loops. Their role is to check if a thread should yield.

5. EXPERIMENTAL RESULTS

To quantify the performance of TLF compared to priority inheritance, Figs. 5 and 6 shows the maximum execution time for high and low priority threads executing the list insertion microbenchmark described in Section 3.1⁷. The benchmark was executed on a 1.66 GHz Athlon, with 1 GB memory in single-user mode. The results were gathered by running the benchmark 7 times, discarding the first run.

The x-axis indicates the size of the list, and the y-axis mesures the maximum time necessary (in microseconds) to insert an object at each position in the list 100 times; each insertion causes the tail of the list following the insertion point to be extended to hold the new item. The lower priority thread inserts repeatedly, while the high priority thread is a periodic thread running under a 10ms period, that inserts a single value 100 times.

As the figure shows, the maximum execution time for higher priority threads under a TLF scheme is roughly a factor of 3 faster than a scheme using PIP, and is effectively the same as a scheme in which no synchronization is used to mediate access to the list. This is because the a relatively small amount of data is logged – when a thread enters the section, it needs to only record the current list element being modified. This is a three word object that contains the current value, a reference to the previous element, and a reference to the next element. If the lower priority thread is aborted, we need only revert the list by restoring this object. Note that maximum execution time for TLF does not increase linearly as the size of the list grows, unlikely the priority inheritance scheme, since the lower priority thread is evicted immediately; using priority inheritance, the wait time for a high priority thread is related to the amount of work remaining in the critical section that must be completed by the lower priority thread.

As expected, the maximum execution time for lower priority threads under TLF is worse than under a PIP scheme since such threads must reexecute their critical section in its entirety once they are aborted. However, the cost of reex-

⁷To quantify Ovm's code generation quality, the non-synchronized version of this benchmark runs roughly 10% slower than the same program in Sun's Hotspot VM.

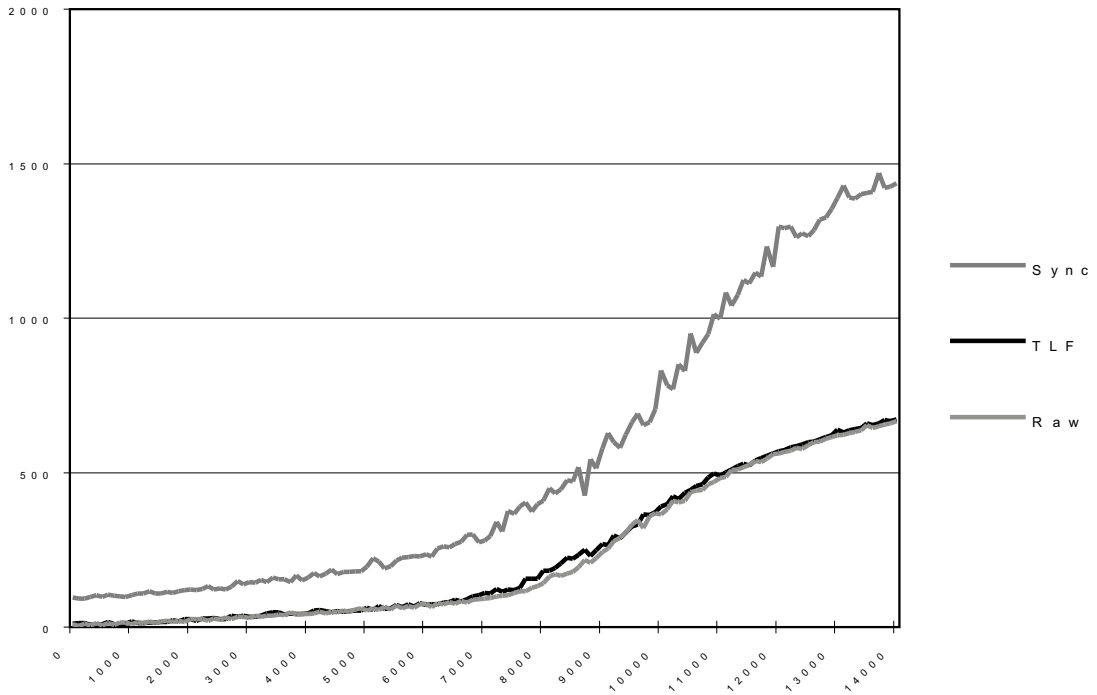


Fig. 5: Responsiveness. Maximum execution times for a high-priority thread with synchronization (Sync), TLF-objects (TLF), and no concurrency control (Raw).

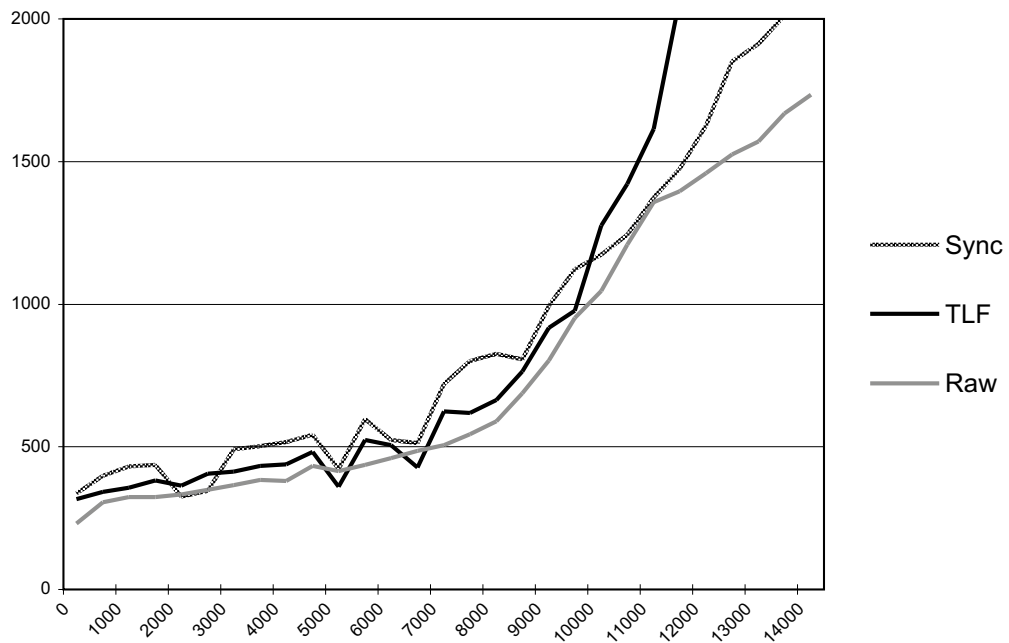


Fig. 6: Responsiveness. Maximum execution times for a low-priority thread with synchronization (Sync), TLF-objects (TLF), and no concurrency control (Raw).

ecution is small compared to the improved throughput for higher priority threads. Of course, changing the periodicity rate of higher priority threads may influence the shape of this graph; a higher-rate would imply low priority threads do less useful work and reexecute more frequently.

6. RELATED WORK

Our use of rollbacks to redo computation inside synchronized sections as a result of an undesirable scheduling is reminiscent of optimistic concurrency protocols first introduced in the 1980's [16] to improve database performance. Given a collection of transactions, the goal in an optimistic concurrency implementation is to ensure that only a serializable schedule results [1, 12, 5]. An analysis of the applicability of lock-free objects in hard realtime systems is described by Anderson *et.al.* [3]. Their focus is on defining schedules that guarantee tight bounds on the number of retries a thread may execute a critical section in the presence of higher-priority threads competing for the same resource. Transactional extensions of the priority ceiling protocol are studied in [26, 29], where a task with its priority higher than a given *abort ceiling* can abort the currently scheduled task.

Transactional lock-free objects have been presented in [2]. The emphasis of that work is to provide language independent support for transactional operations. This means that atomicity and isolation guarantees are limited to the transactional objects. In this paper we give a stronger guarantee, actions of a thread within a transactional section are only observable if the thread commits. The difference is that transaction do not only protect the transactional object but also all other objects that can be reached by following chains of references accessible to thread performing a transaction. This means that we ensure that the programmer will not be able to observe that a thread was reexecuted. A formal semantics for language-level transactions appeared in [31], interested readers are referred to that paper for a discussion of correctness.

Applying these techniques to a broader setting, researchers have also investigated lock-free objects [15, 28, 25]. Our use of logging writes inside synchronized sections distinguishes our approach from lock-free structures because synchronized sections serve as a protection mechanism for multiple (distinct) reads and writes. Conceptually, within its dynamic context, the original values of shared objects are logged and can be reverted if the section aborts. More recently, Herlihy *et.al.* describe a software transactional memory abstraction [13] for Java that allow transactional objects to be dynamically created. Harris and Fraser [11] also describe a lightweight transactional model for Java. Both these efforts share similar goals to ours, but differ in the semantics and implementations of the primitives chosen, and in the application domains addressed. Related efforts at providing hardware support for lock-free execution has been described by Rajwar and Goodman [20] and Rajwar looked at hardware assisted transactional memory [20, 21]

7. CONCLUSION

We have presented a mechanism that addresses the issue of effective scheduling of high-priority threads in priority preemptive realtime systems by introducing lightweight transactional regions that log objects accessed by a thread, and restores the contents of these logs when a thread are interrupted by a higher-priority ones. We show that the time to revert control to higher-priority threads can be bounded to be a function of the size of the transaction-maintained log, and the overheads to maintain transactional consistency is small both in space and time. The model is simple, requiring no major change in programming style or methodology, and is general, applicable to any RTSJ program. Performance results indicate that this scheme significantly outperforms lock-based and priority-inheritance based approaches in terms of responsiveness for high-priority threads.

Our design deliberately trades-off flexibility and generality for simplicity. Effectively dealing with nested transactions, allowing multiple transactional regions to execute concurrently, and reducing the amount of programmer intervention necessary to specify log sizes are several important extensions we hope to pursue.

8. REFERENCES

- [1] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *sigmod*, pages 23–34, 1995.
- [2] James Anderson, Srikanth Ramamurthy, Mark Moir, and Kevin Jeffay. Lock-free transactions for real-time systems. In *Real-Time Database Systems: Issues and Applications*. Kluwer Academic Publishers, Norwell, Massachusetts, 1997.
- [3] James H. Anderson, Srikanth Ramamurthy, and Kevin Jeffay. Real-time computing with lock-free shared objects. *ACM Transactions on Computer Systems*, 15(2):134–165, May 1997.
- [4] William S. Beebe, Jr. and Martin Rinard. An implementation of scoped memory for real-time Java. *Emsoft - LNCS*, 2211, 2001.
- [5] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
- [6] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000. www.javaseries.com/rtj.pdf.
- [7] Dries Buytaert, Frans Arickx, and Johan Vos. A profiler and compiler for the Wonka Virtual Machine. In *USENIX JVM'02 Work in Progress*, San Francisco, CA, August 2002.
- [8] Angelo Corsaro and Doug Schmidt. The design and performance of the jRate Real-Time Java implementation. In *The 4th International Symposium on Distributed Objects and Applications (DOA'02)*, 2002.
- [9] Urs Gleim. JaRTS: A portable implementation of real-time core extensions for Java. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '02): August 1–2, 2002, San Francisco, California, US*, Berkeley, CA, USA, 2002. USENIX.
- [10] John B. Goodenough and Lui Sha. The priority ceiling protocol: A method for minimizing the blocking of high priority Ada tasks. *ACM SIGADA Ada Letters*, 8(7):20–31, Fall 1988.
- [11] Tim Harris and Keir Fraser. Language support for lightweight transactions. *ACM SIGPLAN Notices*, 38(11):388–402, November 2003.

- [12] Maurice Herlihy. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Transactions on Database Systems*, 15(1):96–124, 1990.
- [13] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *ACM Conference on Principles of Distributed Computing*, pages 92–101, 2003.
- [14] Timesys Inc. jTime, 2003.
<http://www.timesys.com>.
- [15] E. H. Jensen, G. W. Hagensen, and J. M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical report, Lawrence Livermore National Laboratories, 1987.
- [16] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 9(4):213–226, June 1981.
- [17] Douglass Locke, Lui Sha, Ragunathan Rajkumar, John Lehoczky, and Greg Burns. Priority inversion and its control: An experimental investigation. *ACM SIGADA Ada Letters*, 8(7):39–42, Fall 1988.
- [18] NASA/JPL and Sun. Golden gate, 2003.
<http://research.sun.com/projects/goldengate>.
- [19] Kelvin Nilson. Adding real-time capabilities to Java. *Communications of the ACM*, 41(6):49–56, June 1998.
- [20] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 294–305, Austin, Texas, December 1–5, 2001. IEEE Computer Society TC-MICRO and ACM SIGMICRO.
- [21] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, volume 37, 10 of *ACM SIGPLAN notices*, pages 5–17, New York, October 5–9 2002. ACM Press.
- [22] Douglas C. Schmidt, Sumedh Mungee, Sergio Flores-Gaitan, and Aniruddha S. Gokhale. Alleviating priority inversion and non-determinism in real-time CORBA ORB core architectures. In *IEEE Real Time Technology and Applications Symposium*, pages 92–101, 1998.
- [23] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 29(9):1175–1185, September 1990.
- [24] David Sharp. Real-time distributed object computing: Ready for mission-critical embedded system applications. In *Proceeding of the Third International Symposium on Distributed-Objects and Applications (DOA'01)*, 2001.
- [25] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*, pages 204–213, New York, August 1995. ACM.
- [26] LihChyun Shu, Michal young, and Ragunathan Rajkumar. An abort ceiling protocol for controlling priority inversion. In *Proceedings of the First Workshop on Real-Time Computing and Applications (RTCSA)*, pages 202–206, 1994.
- [27] Fridtjof Siebert. Hard real-time garbage collection in the Jamaica Virtual Machine. In *Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, Hong Kong, 1999.
- [28] R.L. Sites. *Alpha Architecture Reference Manual*. Digital, 1992.
- [29] Hiroaki Takada and Ken Sakamura. Real-time synchronization protocols with abortable critical sections. In *Proceedings of the First Workshop on Real-Time Computing and Applications (RTCSA)*, pages 48–52, 1994.
- [30] Jörgen Tryggvesson, Torbjörn Mattsson, and Hansruedi Heeb. Jbed: Java for real-time systems. *Dr. Dobb's Journal of Software Tools*, 24(11):78, 80, 82–84, 86, November 1999.
- [31] Jan Vitek, Suresh Jagannathan, Adam Welc, and Antony L. Hosking. A Semantic Framework for Designer Transactions. In *Proceedings of the European Symposium on Programming*, March 2004.
- [32] Adam Welc, Antony L. Hosking, and Suresh Jagannathan. Transactional Monitors for Concurrent Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'04)*, June 2004.