Efficient Wait-Free Queue Algorithms for Real-Time Synchronization*

Philippas Tsigas  Yi Zhang
Department of Computing Science
Chalmers University of Technology
\{tsigas, yzhang\}@cs.chalmers.se

Abstract

The Real-Time Specification for Java provides protected, non-blocking, shared access to objects used by both regular Java threads (\texttt{java.lang.Threads}) and the time-critical \texttt{NoHeapRealtimeThreads}. Such access is offered via a set of \textit{wait-free queue} classes. These classes are provided explicitly to enable communication between the real-time \texttt{NoHeapRealtimeThreads} and the regular Java threads; they have a unidirectional nature with one side of the queue for the real-time threads and the other one for the non-real-time ones. This set of \textit{wait-free queue} classes is of big importance not only to real time Java but also to any real-time synchronization system.

Efficient algorithmic implementations of these queue classes are presented in this paper. The algorithms are designed to exploit the unidirectional nature of these queues and they are more efficient, with respect to space complexity, compared to previous wait-free implementations, without losing in time complexity. The space complexity of our algorithms is $O(M + N)$ where $N$ is the maximum number of concurrent tasks that the \texttt{Queue} supports and $M$ is the size of the \texttt{Queue}. The space complexity of the previous best solution is $O(N \times M)$. The time complexity of our algorithm and the previous best one is $O(N)$ for each task. Experiments we’ve performed suggest that our algorithms are typically 9\% – 36\% faster than the previous best one.

*This work is partially funded by the national Swedish Real-Time Systems research initiative ARTES (www.artes.uu.se), supported by the Swedish Foundation for Strategic Research.
1 Introduction

In this work, we present algorithmic implementations of the wait-free queue classes of the Real-time Specification for Java. These implementations are designed to have the unidirectional nature of these queues in mind and they are more efficient, with respect to space, compared to previous wait-free implementations, without losing in time complexity. The wait-free queue classes proposed in the Real-time Specification for Java are of general interest to any real-time synchronization system where hard real-time tasks have to synchronize with soft or even non real-time tasks.

Java was originally designed by Sun to facilitate the development of embedded system software [8], it was designed more to simplify programming than to enable programmers to write software that compiles reliably with real-time constraints. In order to facilitate its major goal of operating system and hardware independence, in areas such as thread behavior, synchronization, interrupts, memory management, and input/output Java’s expressiveness was designed to be weak. However, these are among the critical areas needing explicit management for meeting application timeliness requirements. On the other hand because the simplicity, the object orientation appeal and most significantly the Java platform’s independence offer, greater cost-savings potential in the real-time domain than in the desktop and server domains: real-time computing systems use many different processor types and operating systems. The matching between Java and real-time software development led to the formation of the Real-Time for Java Experts Group (RTJEG) that began developing the Real-Time Specification for Java (RTSJ) in March 1999 under the Java Community Process [4]. The goal was to provide a platform that let programmers correctly reason about the temporal behavior of executing software. In their effort to do so, they enhanced Java in 7 areas [5]: 1) thread scheduling and dispatching, 2) memory management, 3) synchronization and resource sharing, 4) asynchronous event handling, 5) asynchronous transfer of control, 6) asynchronous thread termination, 7) physical memory access.

In the area of synchronization and resource allocation the RTSJ introduces a set of new synchronization mechanisms, a set of wait-free queue classes. This wait-free queue classes became obligatory for the support of protected, concurrent access of data by both regular threads (java.lang.Threads) and the time-critical NoHeapRealtimeThreads\(^1\) that were introduced by the RTSJ to have an implicit execution eligibility logically higher than the garbage collector. NHRTs can not access (allocate or even reference)

\(^1\)We are going to use the NHRT term to denote the NoHeapRealtimeThread for the rest of the paper.
any objects in the heap, which means that they should be able to run while the garbage collector is running. **NHRTs** are introduced for code with a very low tolerance of non-scheduled delays. The RTSJ does not mandate algorithms or specific time constants for such, but requires that the semantics of the implementation are met.

**Wait-Free Synchronization:** In the area of synchronization and resource allocation applications often need to share serializable resources. Java also provides the ability to introduce concurrency mechanisms into applications. Traditionally, concurrency mechanisms for synchronization and resource allocation use mutual exclusion to protect the consistency of the shared data by allowing only one process at a time to access the method/class. If one declares a method to be *synchronized*, Java will prevent more than one thread from executing that method at any time. The keyword *synchronized* is the only mechanism required by the specification that can enforce mutual exclusion in the traditional sense in RTSJ as in Java. Mutual exclusion i) causes large performance degradation; ii) leads to a complex scheduling analysis since tasks can be delayed, because they were either preempted by other more urgent tasks, or because they are blocked before a critical section by another process that can in turn be preempted by another more urgent task and so on (This is also known as the convoy effect); iii) and most significantly for the real-time systems it leads to priority inversion in which a high priority task can be blocked for an unbounded time by a lower priority task [16]. Several synchronization protocols have been introduced to solve the priority inversion problem for uniprocessor [16] and multiprocessor [14] systems. The solution presented in [16] solves the problem for the uniprocessor case with the cost of limiting the schedulability of task sets and also making the scheduling analysis of real-time systems hard. The situation is even worse in a multiprocessor real-time system, where a task may be blocked by another task running on a different processor [14]. For the RTSJ, it was decided that the least intrusive specification for allowing real-time safe synchronization is to require that implementations of the Java keyword *synchronized* includes one or more algorithms that prevent priority inversion among real-time Java threads that share the serialized resource. But still the use of the *synchronized* keyword implementing the required priority inversion algorithm was not sufficient to both prevent priority inversion and allow the special **NHRTs**, that were introduced in the RTSJ to give the means to time-critical tasks to get an execution eligibility logically higher than the garbage collector, to do so [4]. The RTJEG realized that a non-blocking, protected access to objects shared between **NHRT** and regular Java threads is the only solution
to the problem. The decision of the RTJEG to provide wait-free queues to enable communication between the regular Java threads and the real-time NHRT follows research in recent years, in which several researchers have investigated the use of wait-free shared-object algorithms as an alternative to lock-based mechanisms in object-based real-time systems [1, 2, 3, 7, 15, 17]. Moreover, research in real-time operating systems [6, 11, 9, 19] has also shown how to incorporate wait-free techniques in real-time kernels.

Wait-free implementation of shared data objects is an alternative approach for the problem of inter-task communication and synchronization. Wait-free mechanisms allow multiple tasks to access a shared object at the same time, but without enforcing mutual exclusion to accomplish this. A wait-free implementation of a shared data object guarantees that every process accessing the object always completes its operation in a bounded number of its own steps, regardless of interleaving (process halts, failures, scheduler behavior). Wait-free inter-task communication does not allow one task to block another task and thus gives significant advantages over lock-based schemes because: 1) it does not give priority inversion and avoids lock convoys that make scheduling analysis hard and delays longer. 2) it provides high fault tolerance (processor failures will never corrupt shared data objects) and eliminates deadlock scenarios from two or more tasks both waiting for locks held by the other. 3) more significantly, it completely eliminates the interference between process schedule and synchronization; thus, giving a more compositional framework to argue about the ‘task’ behavior under the effect of the scheduler and the synchronization mechanism. This gives the ability to a task to keep its execution eligibility during communication and synchronization, and this was the feature that was incumbent in the RTSJ. All the above mentioned advantages come from the fact that wait-free solutions are not penalized from the negative effects of blocking.

**Related Work and Our Contribution:** Concurrent FIFO queue data structures are fundamental data structures used in many programs and algorithms and, as can be expected, many researchers have proposed implementations for them. Although there are many non-blocking implementations (see [18] for references), only few of them are wait-free. In a non-blocking algorithm, some operations are allowed to perform unbounded number of steps when they are concurrent with other operations; this, of course, is not allowed in a wait-free algorithm. All previously mentioned constructions (wait-free or not) were targeted towards asynchronous systems; such constructions require hardware support for
strong synchronization primitives such as Compare-and-Swap etc. These primitives are not available in the Real-Time Specification for Java. As a matter of fact in the RTSJ only read and write memory operations are supported. The reason is the hardware-independence property that the RTSJ wants to preserve.

Recent research at the University of North Carolina has shown that wait-free algorithms can be simplified considerably in real-time systems by exploiting the way that processes are scheduled for execution in such systems [1, 15]. In particular, if processes are scheduled by priority, then object calls by high-priority processes automatically appear to be atomic to lower-priority processes executing on the same processor. Consequently they show an implementation of the Compare-and-Swap from reads and writes in a priority-based uniprocessor system [15]. In a consequent paper [2], a wait-free implementation of a linked-list from compare-and-swap for priority-based systems is presented. These results combined can offer an efficient implementation, with respect to time complexity, that satisfies the specifications of the wait-free queue classes in RTSJ. The space complexity of this implementation is $O(N \times M)$ where $N$ and $M$ is the maximum number of concurrent tasks that the queue supports and the size of the queue respectively; the time complexity of this implementation is $O(N)$ for each task.

In this paper implementations of RTSJ queue classes with $O(M + N)$ space complexity and $O(N)$ time complexity are presented. Experiments we’ve performed suggest that our algorithms are typically 9% – 36% faster than the previous best one. The wait-free queue classes that are provided by RTSJ have been designed to enable communication between the real-time NoHeapRealtimeThreads and the regular Java threads; they have a unidirectional nature with one side of the queue (read or write) for the real-time threads and the other one (write or read, respectively) for the non-real-time ones. The implementations presented in this paper are designed having the unidirectional nature of these queues in mind in order to gain efficiency; to the best of our knowledge our implementations are the first unidirectional wait-free queue implementations in the literature.

The remainder of the paper is organized as follows: In Section 2 we give a brief introduction to the basic design features of the RTSJ and Section 3 presents our algorithms. In Section 4 we summarize our experimental results. We conclude in Section 5.
2 Synchronization and Resource Sharing in RTSJ

In this section, a short description of the basic design features of RTSJ, that we had to take into account in our implementation, are presented.

The RTSJ is designed for multithreading priority-based uniprocessor systems. The application program must see the minimum 28 priorities as unique; for example, it must know that a thread with a lower priority will never execute if a thread with a higher priority is ready. If threads with the same exact priority are eligible to run, they will execute in FIFO order. The RTSJ provides wait-free queue classes to provide protected, non-blocking, shared access to objects accessed by both regular Java threads and NHRT. These classes are provided explicitly to enable communication between the real-time execution of NHRT and regular Java threads. Basically, there exist two different new queue classes in RTSJ: the \texttt{WaitFreeWriteQueue} class and the \texttt{WaitFreeReadQueue} class.

![Figure 1: The WaitFreeReadQueue class](image1)

![Figure 2: The WaitFreeWriteQueue class](image2)

Both these queue classes are unidirectional. The information flow for the \texttt{WaitFreeWriteQueue} is from the real-time side to the non-real-time one, as shown in Figure 1. The information flow for the \texttt{WaitFreeReadQueue} is from the non-real-time side to the real-time one, as shown in Figure 2. When a NHRT wants to send data to a regular Java thread, it uses the \texttt{write} (real-time) operation of \texttt{WaitFreeWriteQueue} class. Regular threads use the \texttt{read} (non-real-time) operation of the same class to read information. The \texttt{write} side is non-blocking and wait-free, so that NHRT will not experience delays from the garbage collection. The \texttt{read} operation, on the other hand, is blocking. Since the \texttt{write} is wait-free and the arrival dynamics are incompatible, data can be lost. To avoid delays of allocating memory elements, class constructors statically allocate all memory used for queue elements, giving the queue a finite limit. The \texttt{WaitFreeReadQueue} class, which is unidirectional from non-real-time to real-time, works in the converse manner.

The third queue class that is described in the RTSJ is the \texttt{WaitFreeDeQueue} class and is implemented
by putting back-to-back a \texttt{WaitFreeWriteQueue} and a \texttt{WaitFreeReadQueue}. The formal specification for the \texttt{WaitFreeWriteQueue}, \texttt{WaitFreeReadQueue} and \texttt{WaitFreeDeQueue} classes, can be found in [5].

3 The Algorithms

Algorithmically the implementations of the two wait-free queue classes (\texttt{WaitFreeWriteQueue} and \texttt{WaitFreeReadQueue}) are quite similar. In this section, we present the implementation of the \texttt{WaitFreeWriteQueue} class to illustrate the ideas behind the constructions. In Appendix D, we present the pseudocode for the \texttt{WaitFreeReadQueue} algorithm.

```java
public class SeqQueue {
    RTQueueCell head, tail;
    RTQueueCell dumbcell;

    void SeqQueue() {
        head = dumbcell;
        tail = dumbcell;
        dumbcell.data = null;
        dumbcell.next = null;
    }

    public java.lang.Object read() {
        RTQueueCell temp;
        temp = (RTQueueCell)head.next;
        if (temp != null)
            head = temp;
        return temp;
    }

    public boolean write(java.lang.Object object) {
        RTQueueCell temp;
        temp = new RTQueueCell();
        temp.data = object;
        temp.next = null;
        //tail and tail.next will be shared
        //read/write in concurrent implementation
        tail.next = temp;
        tail = temp;
        return TRUE;
    }
}
```

Figure 3: The Sequential implementation of the queue

3.1 Informal Description

To simplify the presentation of our algorithm, we start with a simple sequential queue implementation. We will then discuss how to extend this sequential algorithm to a concurrent queue implementation with the specifications that we are looking for. The Java-pseudo-code for this sequential queue is shown in Figure 3.

As it can be seen in Figure 3 we implemented algorithmically the queue using a singly linked list. For efficiency reasons, we choose the front of the queue, where we only delete nodes, to be the head of the list, and the rear of the queue, where we only insert, to be the tail of the list. In this way we only use the operations of the linked list that modify the head and the tail of the list. In order to minimize the interference between the \texttt{write (enqueue)} and \texttt{read (dequeue)}\footnote{Throughout the paper, the terms queue \texttt{write} and \texttt{enqueue} are used interchangeably. The same also holds for the \texttt{read} and \texttt{dequeue} operations.} operations, we introduce a \texttt{dumbcell}
in the empty list. In this way, when executing a dequeue operation, only the head needs to be checked in order to see whether the queue is empty or not. If the next field of the head is null, the queue is empty. Therefore, the dequeue operation needs only to check the head variable and only the tail needs to be checked for the enqueue operation. For the initialization for the simple sequential queue we define a dumcell, with null in its next field, and let the head and the tail of the queue point to it, statements 5 to 8 in Figure 3.

Figure 4 shows the structure of the cells of the linked list that we are using. The class RTQueueCell has two public members: one is for the data entry, the other is the next pointer that singly links the elements of the list.

To extend the sequential version to a concurrent wait-free queue implementation, first we will use a simple announce-and-help scheme for the enqueue operations. The announce-and-help scheme uses the priority based scheduler to achieve wait-freedom. This scheme is based on the task priorities to guarantee that an operation will finish in a bounded number of steps regardless of the status of the other operations, as follows: First, each enqueue-task announces the data (writes a pointer to the memory where the data are) that it wants to enqueue in a special Announcement array. The enqueue-task with priority $i$ will use the $i$th position of the array. After the announcement step, the enqueue-task reads and helps the data that have been announced in the array one by one, starting from the lowest priority up to its own priority. During this helping phase, if an enqueue-task $A$ is not going to be preempted by a higher priority task, then all current enqueue operations, with lower priority than the priority of $A$, that are announced will be helped/enqueued by $A$. If the enqueue task $A$ is preempted by a higher priority task $B$ during its helping phase, then there are two cases:

```java
class RTQueueCell {
    public Object data = null;
    public Object next = null;
}
```

Figure 4: Definition of the queue cell

- $B$ is not an enqueue-task on the same queue: then the task $A$ will continue its program steps after $B$ finishes, from the same queue-state from which it was pre-empted. Dequeue operations on the same terms queue read and dequeue. To distinguish between the queue read/write and normal read/write memory operation, we are using typewriter type style for the queue operations and serif type style for the memory ones.
queue are executed by tasks that have lower priority and therefore they can not preempt enqueue operations on the same queue.

- \( B \) performs an enqueue operation on the same queue: in this case \( B \) is going to announce its task and help all lower priority tasks that are announced and its own task that has just been announced. Therefore task \( A \) will be helped by \( B \). Because the priorities are bounded, there always exists a task which will not be preempted by another queue task. Therefore, all tasks that announced their operations will be helped (either by themselves or by higher priority tasks).

The RTSJ has been designed for uniprocessor systems. The RTSJ as well as Java support only plain memory synchronization primitives like atomic reads and writes to memory locations, as opposed to other advanced synchronization primitives like \texttt{Compare-and-Swap}. The weak vocabulary of Java in memory synchronization comes again from the fact that different processors support different memory synchronization primitives and Java was designed to be hardware independent. Although reads and writes are very weak synchronization primitives in the context of general asynchronous systems, by exploiting the fact that the tasks are executed by priority, it has been shown that they are universal primitives for priority-based uniprocessor systems [15]. In RTSJ the application program must see the minimum 28 priorities as unique; for example, it must know that a thread with a lower priority will never execute if a thread with a higher priority is ready. If threads with the same exact priority are eligible to run, they will execute in FIFO order.

During the design phase of any shared data object, a problem that arises from the use of memory read/write operations is the “enabled late-write” problem [15]. The “enabled late-write” problem arises when a low priority task \( A \) is preempted while it is about to write to a memory position, and is preempted by other tasks that access and modify the same memory position. When task \( A \) resumes running, it overwrites the previous “fresh” value with an “old” one. Anderson et al. [15] proposed a majority voting scheme to overcome the problem. Their scheme requires \( 2N - 1 \) memory words to solve “the enabled late-write” problem for \( 1 \) word.

In this paper we propose a new more efficient scheme to face the “enabled late-write”. The new scheme tries to avoid the problem from the beginning by:

1. Making sure that, when a task \( A \) is preempted before writing to position \( p \), all other tasks that write on to \( p \), (while \( A \) is preempted) write the same value that \( A \) wanted to write. In order to
establish this, we guide the tasks to go through the same computational steps as A when they have to decide about the value that they want to write on the same memory location.

2. When the above is possible, we organize the shared variables that might suffer from the “enabled late-write” problem as arrays that carry information that can be used algorithmically to determine the correct/new value of the variable.

We believe that, the same idea can be used when algorithmically designing other shared objects for the RTSJ.

```java
public class WaitFreeWriteQueue {
    ...
    private MemoryArea MemPool;
    private java.lang.Object[] Announcement;
    private RTQueueCell[] tail;
    private RTQueueCell head;
    // get the minPriority from the scheduler
    private int minPriority;
    // get the maxPriority from the scheduler
    private int maxPriority;
    ...
}
```

Figure 5: Shared private variables for WaitFreeWriteQueue

```
RTQueueCell dumbcell = new RTQueueCell();
...
Announcement = new java.lang.Object[maxPriority + 1];
tail = new RTQueueCell[maxPriority + 1];
for (i=minPriority;i<=maxPriority;i++) {
    Announcement[i] = null;
    tail[i] = null;
}
tail[minPriority] = dumbcell;
head = dumbcell;
dumbcell.data = null;
dumbcell.next = null;
```

Figure 6: Initialization for WaitFreeWriteQueue

The wait-free part in this class is the part that implements the enqueue operations. The wait-free write operations share also the private variable MemPool that hold references to a MemoryArea. The shared private variables for our WaitFreeWriteQueue are as shown in Figure 5. All RTQueueCells should be allocated from the MemoryArea. The Announcement array is used to hold the the different enqueue operations. The tail and Announcement arrays are of equal length, equal to the real-time priority level supported by the scheduler. For the head of the queue we use the simple variable head.

The minPriority and maxPriority are the minimum and maximum priorities that real-time threads can be assigned, respectively. This information can be obtained from the scheduler. All shared variables will be initialized when constructing the queue. The initialization is similar to that of the sequential version. Because we now use an array to represent the tail, we need to initialize this array in a way that makes it easy for the algorithm to find the correct tail (the dumbcell), when a task accesses the

---

The RTSJ introduces the memory area concept, which is a region of memory outside the garbage-collected heap that you can use to allocate objects. The RTSJ uses the abstract class MemoryArea for this.

---

3The RTSJ introduces the memory area concept, which is a region of memory outside the garbage-collected heap that you can use to allocate objects. The RTSJ uses the abstract class MemoryArea for this.
queue for the first time. When a task accesses the tail array, it checks from the cell of the lowest priority task to the highest to find a non-null cell. Henceforth, we initialize the cell corresponding to the lowest priority point to the dumbcell. During the initialization part, we also need to initialize the Annoucement array with the value null, that means that there are no announced operations. The pseudo-code for the initialization is described below in Figure 6. The initialization of the local variables is part of the pseudo-code description of the algorithm described in Figure 7.

Now, in order to extend the sequential version that we presented at the beginning of this section to the concurrent one that we are aiming for, we first need to make sure that the shared read/write operations to the tail and the tail.next variables (the shared variables of our implementation where overwriting might take place) do not suffer from the “enabled late-write” problem. The wait-free enqueue operation is presented in the write function below. The announce-and-help scheme, that is used in our implementation, uses the priority based scheduler to achieve wait-freedom. Each priority is mapped to the respective entry of the array Announcement. An enqueue operation first gets the priority of its thread, then it allocates a free cell from the memory area assigned to the queue. The memory area is where the queue and its internal elements are allocated. After writing the data in the free cell, the task announces this cell in the Announcement array at the index that is associated to its priority. This constitutes the last part of the announcement phase. This is, as we will see, the “linearizability point” of the enqueue operation at the linearizability history. After it announces the object that it wants to enqueue in the Announcement, a task will enter the helping phase that was described at the beginning of this subsection. The helping phase is described in relation to the implementation pseudo-code in Figure 7. During the helping phase, an enqueue operation with priority i helps the tasks with priority $j \leq i$ that have been announced in the array Announcement, one at a time, starting from the operation with the smallest priority that it can find (statement 26 in the implementation). For each such operation, it finds the tail of the queue (statements 32-42 on the protocol); then puts the data announced at the end of the tail of the queue; then changes the tail variable to point to the new position; and finally cleans the Announcement[$j$].

The wait-free queue class we designed are used to provide communication between NHRTs and regular Java threads. To untangle the effect of garbage collection, statica memory management is needed for nodes of the queue class. Statical memory management is not the subject of this paper; but,
a simple scheme is presented here, in the Appendix A, to show the feasibility of such a scheme. Other better and more efficient schemes are possible. In the Appendix B, we describe implementations of the other methods supported by the WaitFreeWriteQueue class.

```java
public boolean write(java.lang.Object object) {
    boolean find = false;
    int i, j, mypriority;
    RTQueueCell tempcell;
    tempcell = null;
    java.lang.Object tempAnnounce;
    java.lang.Thread currentone;
    // Find your priority
    mypriority = 0;
    currentone = java.lang.Thread.currentThread();
    // Allocate a cell in the MemoryArea
    try {
        tempcell = MemPool.newInstance(RTQueueCell);
        tempcell.data = object;
        tempcell.next = null;
    } catch (OutOfMemoryError x) {
        return false;
    }
    // Announce your operation
    Announcement[mypriority] = tempcell;
    // Enter helping phase and help
    // lower priorities and yourself
    for (i = minPriority; i <= mypriority; i++) {
        tempAnnounce = Announcement[i];
        if (tempAnnounce == null)
            continue;
        // Try to find the actual tail
        find = false;
        for (j = minPriority; j <= maxPriority; j++) {
            if (j == tempcell) {
                // Enqueue the announcement
                tempcell.next = tempAnnounce;
                tempcell = tempAnnounce;
                Announcement[i] = null;
            } break;
        }
    }
    // No preemption detected, the actual tail has been found.
    tempcell = tail[j];
    else
        // Preemption detected!
        // there are 2 possibilities
        if (Announcement[i] == null) {
            // Preempted, helped but not completely.
            // Help the task with priority $i$ to update tail and Announcement array
            tail[i] = (RTQueueCell)tempAnnounce;
            Announcement[i] = null;
            continue;
        }
        else
            // Preempted and helped by a higher
            // priority task that helped the task that
            // you were helping and you. Return!
            return true;
}
```

Figure 7: Wait-free enqueue operation for the WaitFreeWriteQueue

Figure 8 shows the lock-based read operation of the WaitFreeWriteQueue class. It's a straightforward implementation that uses mutual exclusion to serialize concurrent dequeue operations.

3.2 Correctness Proof

In the helping phase two sets of variables are used, the tail array (tasks help to enqueue data at the tail of the queue) and the Announcement array; all of them are shared variables and can be read and
```java
public synchronized java.lang.Object read() {
    RTQueueCell tempcell;
    tempcell = head.next;
    if (tempcell != null)
        head = (RTQueueCell) tempcell;
    return tempcell;
}
```

Figure 8: Lock-based dequeue operation for \texttt{WaitFreeWriteQueue}

written by different tasks. In the implementation, the value of a variable \texttt{Announcement[i]} changes from \texttt{null} to a non-null value, when a task with priority \texttt{i} announces its \texttt{enqueue} operation. The value of the same variable changes back to \texttt{null} when the item that the \texttt{enqueue} operation wanted to enqueue was enqueued by the same operation or by another higher priority enqueue operation. If there are \texttt{e} enabled writes that are ready to write to \texttt{Announcement[i]} then at least \texttt{e - 1} of them are helping operations and have priority higher than \texttt{i} and want to change the value of the \texttt{Announcement[i]} from \texttt{non-null} to \texttt{null}. The one “enabled late-write”, that might exist, is the write with priority \texttt{i} that wants to announce a new enqueue operation. This write will not be scheduled before the other pending writes, with higher priority, take place, and thus, its write will not be overwritten by them. The above proves the following lemma:

\textbf{Lemma 1} \( \forall i, \min\text{Priority} \leq i \leq \max\text{Priority}, \text{Announcement}[i] \) \text{ will not suffer from the “enabled late-write” problem.}

\textbf{Lemma 2} \text{When a task } A \text{ is preempted just before it writes to the tail array, a higher priority task will write the same content to the same position in the tail array.}

\textbf{Proof:} The decision of what to write on the tail is based on the contents of the \texttt{Announcement} and \texttt{tail} arrays. If a higher priority task preempted task \texttt{A} just before \texttt{A} was to write the \texttt{tail} array, then, since, nothing changed on the \texttt{Announcement} and \texttt{tail} of the object from the time that \texttt{A} read them, the higher priority task, that preempted \texttt{A}, will compute the same value to write to the \texttt{tail} array. \( \square \)

If we would have used a simple \texttt{tail} variable for the queue, as it is used in the sequential implementation of the queue, the “enabled late-write” problem could have happened in the \texttt{tail} variable. To solve that problem, we organize the \texttt{tail} of the queue as an array. Each location in the array corresponds to the respective priority. All tasks with the same priority will be executed in a FIFO order and use
the same location in the array. Each enqueued item from a task with priority $i$ will become the tail of the queue once, and the $i$th index of the tail will point to it. In our construction, all tasks that try to help a task with priority $i$ that has been announced, are going to write to the tail array at the index that corresponds to priority $i$. For example, when a task $A$ is helping with task $C$, it is preempted by another high priority task $B$ and has an enabled write on the tail array. Then the new task $B$ will help the same task $C$ also and will go through the same computational steps and will update the same entry of the tail array with the same value as the preempted enabled write of task $A$. This is guaranteed from Lemma 2. In this way, the “enabled late-write” problem can not take place in any tail[$i$] variable. This sketches a proof of the following lemma.

**Lemma 3** $\forall i, \text{minPriority} \leq i \leq \text{maxPriority}, \text{tail}[i]$ will not suffer from the “enabled late-write” problem.

Now, we need to give a way for the tasks to read the tail array and compute the real tail of the queue. Each item in the tail array has been the real tail of the queue at some point in time but only one of them is the current tail of the queue. In our implementation, there is at most one tail entry that has the value null on its next field. As we are designing a concurrent queue, an enqueue operation can be preempted anywhere; a task $A$ can be preempted between statement 73 and 74 by a task $B$. The tail array then will have no element with the value null on its next field. The actual tail in this case should point to the object enqueued by a task $C$, which is being helped by task $A$ ($A$ executes statement 73 and 74 i only when it is helping another task). During the helping phase of its enqueue operation, task $B$ need to find the tail of the queue and uses the local variable temptail to store it. In the pseudo-code, when task $B$ executes statements 32-38, it goes through the tail array from the lowest priority to the highest priority and tries to find the one index in the array with null in the ‘next’ field, if there is one. If there is no overlapping with enqueue operations, task $B$ will find the index with null in the next field. It will store the value in temptail (statement 42).

**Lemma 4** temptail.next will not suffer from the “enabled late-write” problem.

**Proof:** When a task $A$ with priority $j$ helps a task with priority $i$ that has been announced, where $i < j$, all items in the announcement array from minPriority to $i-1$ should have the value null because
task $A$ starts its helping phase from $\text{minPriority}$, and, tasks with priority less than $j$ can not preempt task $A$ and make changes in the announcement array. Before task $A$ updates the next field of the tail of the queue (statement 73), nothing changes in the tail array and the announcement array. If a task $B$ with priority $k$, where $k > j$, preempted task $A$, task $B$ will add its own announcement in position $k$ and nothing between $\text{minPriority}$ to $j$ in the announcement array will change. Therefore task $B$ will help the announcement of the task with priority $i$ and will find the same tail and make the same decision with task $A$ and finally put the same value as $A$ on $\text{temptail.next}$.

If task $B$ overlapped with other $\text{enqueue}$ tasks, then task $B$ might not find an index on the array with $null$ in the next field. If this happens, task $B$ has already enough information to find the actual tail of the queue and help the preempted task to update the tail of the queue. To see this, let us look at the possible ways that the above could have happened; there are two cases:

- Task $B$ preempts the lower priority task $A$, when $A$ was between statements 73 and 74; e.g. $A$ had just finished enqueueing the data before updating the tail of the queue. The actual tail of the queue at this point is the task which is being helped by both task $A$ and task $B$. Task $B$ will help task $C$ to update the tail when $B$ runs statements 50-51.

- Task $B$ is preempted by a higher priority task $D$ and $D$ updates the $\text{tail}$ array in such a way that task $B$ misses the actual tail of the queue when $B$ is scheduled back. In this case, $D$ will help all lower priority tasks. So, task $B$ just needs to stop its helping phase and return. $B$ will detect that it has been helped and return in statement 58.

The above sketches a proof that items are going to be put on the singly link-list one after the other.

Since different tasks are going to try to help the same task, we need to show that an item is not going to be enqueued more than one time. That is the reason that statement 62 is used from task $B$ to detect that it has preempted a task $A$ when $A$ was between statements 74 and 75 of its pseudo-code. When the preemption happens, the announcement has been added to the queue as a tail but not been cleaned, which has been read by task $B$ in $\text{temptail}$. If such a preemption is detected, the task $B$ will help task $A$ to clean the announcement array, when task $B$ executes statement 65. As both of them want to write $null$ at the same position, no “enabled late-write” problem exist. Statement 70-71 is used from task $B$ to detect that it had been preempted by a higher priority tasks $D$ and to conclude that task $D$ has helped the task that $B$ was helping when preempted.
The following lemma also proves that it is necessary and sufficient for a task to help other tasks with priority up to its own priority.

**Lemma 5** When a task $A$ with priority $i$ announces an enqueue data in the Announcement array, all elements of the array from $i + 1$ to $\text{maxPriority}$ have the value null.

**Proof:** Assume towards a contradiction that $\text{Announcement}[j]$ is not null, where $j > i$. Then there must exist a task $B$ with priority $j$ thus announced its enqueue object in $\text{Announcement}$ array and the announcement by task $B$ hasn’t been “cleaned”. $\text{Announcement}[j]$ is cleaned as the last step of the enqueue operation. The task $A$ must preempt task $B$ to announce its enqueue object in $\text{Announcement}$, in order to preempt task $B$, $i > j$ must hold. This is a contradiction.

As the contents of the $\text{Announcement}$ array from index $i + 1$ to index $\text{maxPriority}$ are null when task $A$ announce its operation, there is no need to help them. It is sufficient to help tasks with priority up to $i$. As task $A$ can preempt any lower priority task after it has announced, it is necessary to help them.  

The access of the queue is modeled by a history $h$. A history $h$ is a finite (or not) sequence of operation invocation and response events. Any response event is preceded by the corresponding invocation event. For our case there are two different operations that can be invoked, a write operation or a read operation. An operation is called complete if there is a response event in the same history $h$; otherwise, it is said to be pending. A history is called complete if all its operations are complete. In a global time model each operation $q$ “occupies” a time interval $[s_q, f_q]$ on a linear time axis ($s_q < f_q$); we can think of $s_q$ and $f_q$ as the starting and finishing time instants of $q$. During this time interval the operation is said to be pending. There exists a precedence relation on operations in a history denoted by $<_h$, which is a strict partial order: $q_1 <_h q_2$ means that $q_1$ ends before $q_2$ starts; Operations incomparable under $<_h$ are called overlapping. A complete history $h$ is linearizable if the partial order $<_h$ on its operations can be extended to a total order $\rightarrow_h$ that respects the specification of the object [10]. In Appendix C we prove that our implementation is a concurrent linearizable FIFO queue implementation.

**Theorem 1** Our algorithm for the $\text{WaitFreeWriteQueue}$ is a linearizable FIFO concurrent queue without the “enabled late-write” problem.
4 Experimental Results

To evaluate the performance of our algorithm, we compare it with the best previously known solution that was proposed in [2]. We performed our experiments on a real-time environment simulator based on the Real-Time Threads (RTT) Package [12]. The RTT provides priority-based preemptive scheduling for real-time threads that is similar to the real time java virtual machine specification. To the best of our knowledge there is no implementation of a real time java virtual machine available yet.

We performed experiments with 1, 2, 4, 8 and 16 concurrent enqueue tasks. The parameters of the task sets were selected so that the tasks are schedulable and are based on the following formulas:

\[ T_i = (n - i) \times T, \quad i = 0, \ldots, (n - 1) \]
\[ C_i = \begin{cases} (n - i) \times C, & i = 1, \ldots, (n - 1) \\ (18 - (n - 1)), & i = 0 \end{cases} \]

\( T \) and \( C \) are the period and computation time of the highest priority task respectively and \( \frac{T}{C} = 20 \). \( T_i \) and \( C_i \) are the period and computation time of the task \( i \) respectively. \( N \) is the number of concurrent enqueue tasks. The task sets are scheduled with the rate-monotonic scheduling algorithm. With the task parameters described above, the processor utilization is \( U = \sum \frac{C_i}{T_i} = 90\% \) for all task sets and all task sets are schedulable under rate-monotonic scheduling [13].

The results of our experiments show that our algorithm does not decrease only dramatically the space requirements but is also from 9\% to 36\% faster than best previously known. The average response times of the highest, middle and lowest priority task of each task set are shown in Figure 9.

![Figure 9: Response times of the lowest priority, middle priority and highest priority task](image-url)
5 Conclusion

Efficient implementations of the RTSJ queue classes are presented in this paper. The wait-free queue classes proposed in the Real-time Specification for Java are of general interest to any real-time synchronization system where hard real-time tasks have to synchronize with soft or even non real-time tasks. The implementations presented here are designed with the unidirectional nature of these queues in mind and they are more efficient, with respect to space, compared to previous wait-free implementations without losing in time complexity. The space complexity of our algorithms is $O(M + N)$ where $N$ is the maximum number of concurrent tasks that the Queue supports and $M$ is the size of the Queue. The space complexity of the previous best solution is $O(N \cdot M)$. The time complexity of our algorithm is $O(N)$. Experiments we’ve performed suggest that our algorithms are typically 9% – 36% faster than the previous best one.

There are several ways that future research in wait-free synchronizations can contribute to real-time Java. A very promising, we believe, is the investigation of practical wait-free implementations of garbage collection in the RTSJ model. The garbage collector is a central component of the Java environment. Wait-free implementation will improve the programmers ability to correctly reason about the temporal behavior of their Java programs.

References


**Appendix A: Memory Management for Queue Node**

A simple statical memory management scheme for the wait-free queue class is provided in this section. Other statical memory management schemes can be also used for the queue class presented here.

Each task will be statically associated an array of queue nodes during initialization. The length of the array depends on the expected length of the enqueue. A flag is associated to each item in the array; this flag is used to tell whether the node is free or not. When an enqueue task want to allocate a node, it will go through the array and return the first free node if such a node exists. The enqueue task will mark the node as ‘not free’. A dequeue task will free the node by setting the flag to be ‘free’ after it finish its operation on the node.
No mutual exclusion is required for allocating and freeing a node with the above scheme. For each array, there is only one enqueue task associated with the array and the dequeue task which got this node will only write on the flag of this node. At any time, only one task can write the flag: if the node is free, only the enqueue task will update the flag to make the node occupied; if the node is not free, only a dequeue task will update the flag to make the node free. Therefore, the above scheme is wait-free. As mentioned before, memory management is not the subject of this paper, but we provide a simple and working statatical memory management for our scheme to show the feasibility of such a scheme. Other better and more efficient schemes are possible.

Appendix B: Other Methods Supported by the Class

Here we describe how to implement the other methods supported by the WaitFreeWriteQueue class.

For wait-free queue classes in RTSJ, programmers need to manage nodes themselves. Functions isFull and size are related to memory management and their implementation can be varied according to the memory management scheme. We show how these two functions can be implement under the memory scheme presented in Appendix A. As each task has its own associated array, isFull will return True when there is no free node in the associated array; otherwise it will return False. The size of the wait-free queue is equal to the number of allocated nodes by all tasks. The force function requires a task to overwrite an old value when the queue is full. We can implement the force function with the helping of the memory management scheme. When allocating a node, the last allocated node will be recorded. When all nodes are occupied, a task can not enqueue a new data. If it calls the force function, it will enqueue the value into the last allocated node.

The implementations of clear and isEmpty functions are shown in Figure and 10 do not depend on memory management scheme.

Appendix C: Correctness Proof

In this section, we prove that our implementation is a concurrent linearizable queue implementation. In order to do so, we will show that any possible history ($<_{A}$), produced by our implementation, can be extended to a total order ($\rightarrow_{H}$) by using a “linearization point” for each operation. The “linearization
public void clear()
{
    int i;
    for (i=minPriority;i<=maxPriority;i++)
    {
        Announcement[i] = null;
        tail[i] = null;
    }
    tail[minPriority] = DumbNull;
    head = DumbNull;
    DumbNull.next = null;
}
public boolean isEmpty()
{
    if (head.next == null)
        return true;
    else
        return false;
}

Figure 10: Implementation of clear and isEmpty functions

point” of an operation is an atomic point on its execution, during which the operation takes effect.

Lemma 6 The write to the announcement array is the “linearization point” for the write operations.

Proof: By Lemma 5, when a task A with priority i executes statement 23, all items of the announcement array from i + 1 to maxPriority have the value null. Task A will help all operations announced in Announcement from the lowest to its own priority. Enqueue operations with lower priority than i that have been announced by executing statement 23, will be enqueued before A’s announcement on the announcement array. If the current task A is preempted by a higher priority task after executing statement 23, the announcement will be enqueued before the announcement of the task with higher priority. So, the execution order of statement 23 in the write operation extends the precedence partial order to a total order that respects the FIFO specifications of the WaitFreeWriteQueue class.

Lemma 7 The read of the next field of the head of the queue is the “linearization point” for the reads of the queue.

Proof: Since, mutual exclusion is used between read operations on the queue, the order in which they get access to the critical section totally orders them. But as the read operations of the queue have lower priority than all the write operations of the queue, they can be preempted and run concurrently with write operations. As all high priority tasks will appear atomic to a low priority task, a write operation

22
will only be observed if it starts executing before statement 3 of the read operation. By selecting then the execution of the statement 3 of a read operation as its “linearizability point”, all operations are totally ordered with a relation that extends the precedence relation and respects the specification of the WaitFreeWriteQueue class.

The lemma below proves that our queue implementation is a FIFO one and that no enqueued element gets lost. For simplicity we introduce write(empty) operations in the history when the queue is empty.

**Lemma 8** In a complete history such that write(x) →ₕ write(y), then read(x) →ₕ read(y).

**Proof:** From the assumption, we have that write(x) →ₕ write(y) which means x is announced before y. If there is no overlapping, x will be put in the list before y as in the sequential version. If overlapping exists, by lemma 5, the task A who announces y has higher priority than the task who announces x. As task A will help from the task with lowest priority to itself, it will put x in the list before y. As read uses mutual exclusion, only one read operation processes the list from the head to the tail. So read operations will find x first.

The lemma below proves that dequeue operations dequeue items that have really been enqueued.

**Lemma 9** In a complete history, if x is read, then it has been written, and write(x) →ₕ read(x)

**Proof:** The linearizability point of the read(x) is the point where the read operation reads the next field of the head. Because a write operation announces its operation and the announcement takes place before the helping phase, and in the helping phase the announcement will be put in the next field of a tail[i]. If x is read, then some task must write in the field during its helping phase. Helping an announcement can only happen after it has been announced by some task in the announcement array. So, the x read by a task must have been written before the read operation.

**Appendix D: The WaitFreeReadQueue algorithm**

The same idea can be used to implement WaitFreeReadQueue. When implementing WaitFreeWriteQueue, we need to find a way to announce one thread’s intention. For WaitFreeWriteQueue, it can be done simply by put the data the thread want to enqueue into the Announcement array. For WaitFreeReadQueue, we can introduce a common cell, dummycell. The Announcement array is initialized to
be all null as in the implementation of WaitFreeWriteQueue. Whenever, a thread want to read/dequeue from the queue, it put the dumbcell into the Announcement array. The code of the read function for the WaitFreeReadQueue class is shown in Figure 11.
public java.lang.Object read()
{
    int find = 0, i, j;
    int myPriority;
    RTQueueCell tempAnnounce;
    java.lang.Thread currentThread;
    RTQueueCell tempnext, temphead;

    currentThread = java.lang.Thread.currentThread();
    myPriority = currentThread.getPriority();
    Announcement[myPriority] = DumbNull;
    for (i = minPriority; i <= maxPriority; i++)
    {
        tempAnnounce = Announcement[i];
        if (tempAnnounce == null)
            continue;
        tempnext = tempAnnounce.next;
        if (tempAnnounce != DumbNull && tempnext != null)
        {
            if (tempnext != head[i])
                head[i] = (RTQueueCell) tempnext;
            Announcement[i].next = null;
            continue;
        }
        if (tempAnnounce != Announcement[i]) continue;
        find = -1;
        for (j = minPriority; j <= maxPriority; j++)
        {
            temphead = head[j];
            if (temphead == null)
            {
                find = j;
                break;
            }
        }
        if (find < 0) break;
        temphead = head[find];
        if (Announcement[i] == DumbNull)
        {
            Announcement[i] = temphead;
            tempnext = (RTQueueCell) Announcement[i].next;
            if (tempnext != null)
            {
                Announcement[i].next = null;
                head[i] = tempnext;
            }
        }
    }
    tempnext = Announcement[myPriority];
    Announcement[myPriority] = null;
    if (tempnext == DumbNull) tempnext = null;
    return tempnext;
}

Figure 11: Wait-free dequeue operation for WaitFreeReadQueue