Multiprocessor-safe Wait-free Queue in RTSJ

Emad T. Aloqayli
Doctor of Philosophy

University of York
Computer Science

September 2014

Abstract

Currently, most computer systems are running on multiprocessors (or multi-cores). Moreover, the number of cores inside the processor are expected to increase. To be able to utilise the increased computational power in these systems, developers are enforced to expose more parallelism within their applications. Multi-threading is one of the common techniques that are used to introduce parallelism within computer applications.

Shared data structures are in the core of multi-threaded applications; these data structures facilitate the communication between the different threads to help in completing the designed tasks within the application. A control mechanism should be provided such that the access of any thread will not compromise the consistency and correctness of the data structure contents. The increased number of threads will result in an increased competition, this will lead to inevitable difficulties in understanding the interleaving scenarios at runtime, hence, the time analysis will be a very complex task.

The Real-Time Specification for Java (RTSJ) introduces different shared queues that can facilitate communication between different threads within the application. However, these queues are uni-directional enabling communication between standard Java threads and the realtime thread classes, which the RTSJ introduces.

The work presented in this thesis introduces a novel algorithm for concurrently accessing shared data structures in a shared memory multi-processor (or multi-core) systems. The proposed algorithm is implemented as an array-based First-In-First-Out (FIFO) queue, which improves the scalability and time predictability in multi-threaded applications. The algorithm utilises the different features that the RTSJ introduces to ensure the time predictability.

Contents

List of Figures

List of Tables

Listings

Acknowledgements

I am very much grateful to my supervisor, Professor Andy Wellings, for his support, patience, guidance and invaluable time throughout the period of my study. I like to thank the Realtime system research group for all of their support and help, especially Professor Neil Audsley and Dr. Rob Davis for their technical advices.

I would like to thank my friends, Dr. Mohammad AlRahmawy, Dr. Abdul Haseeb Malik, Hashem Ghazzawi, Dr. Shiyao Lin and David Griffin for their help and support during the progress of my study.

I would like to thank my mother, my brothers, and my family in-law for their continuous support and encouragement. Lastly, I would never completed my research without the endless support of my wife and my sons and daughters, I like to thank them for their time and effort that encouraged me all the time.

Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References.

Chapter 1

Introduction

Commercial shared memory multiprocessor computer systems have been widely used in real life applications [103]; with expectations that they will become more common as they tend to provide better performance by introducing low-cost chip multi-threading. Continuous improvements of these machines are needed in order to meet the increased demand on computational power required by real-time applications.

Generally, the performance boosts have been achieved by hardware manufacturers by improvements in cache memories, instructions optimization and clock speed [139]. Meanwhile, software developers concentrate on developing correct applications while relying on the performance enhancements achieved by the hardware improvements.

However, the increased demand for performance cannot be met by further improvements in hardware; due to mainly power consumption and heat dissipation issues [139]. Consequently, the multi-core architecture has been introduced to overcome these problems and promised better performance. In the multi-core architecture, the processor is equipped with multiple cores that share the same address space and local processor caches. The cores within the same processor will, theoretically, act like multiprocessors; however, there are some differences between the multi-cores and the multiprocessors; especially in the memory hierarchy [46]. Currently, hardware manufacturers provide servers that are equipped with 4 core processors and each core can execute 2 threads concurrently [75]; however, there is in the market specific purpose

processors with more than 100 cores [148].

Sequential applications, which were designed for single processors, will be able to run on multi-core processors. However, these applications are expected to run slower as the cores in a multi-core processor have less computation power compared to a single core processor. In general, cores are simpler and designed to run at lower speeds to overcome thermal issues [140].

Concurrency, for a long time, had been thought of as "the way to the future" [140]. The introduction of the multi-core architecture requires more parallelism within applications to exploit the processing power. Generally, concurrency is achieved by breaking down the application into small units of code, named threads; each performs a certain operation or set of operations. Threads can be executed concurrently on multiprocessor system or multiple cores within the same processor.

In general, concurrency is the execution of multiple tasks, simultaneously, and it might be required for these tasks to interact while executing. Concurrency can be achieved by multi-tasking, on a single processor, or parallel execution, on different cores, in the same processor, or different processors, in the system [155]. In the single processor case, the processor time is sliced among the tasks being executed, hence, there is only one active task at a time, which limits the complexity of interleaving the execution [91]. In the multiprocessor (or multi-core) case, threads are running in parallel, one or more thread on each processor, hence, the interleaving of execution of these threads (or tasks) is more complex and requires better understanding of the errors and exceptions that might arise due to this parallelism. In this thesis, the term "concurrent" is used to indicate potential or simultaneous execution of threads and the term "parallel" is used when there is actual simultaneous execution of threads.

Multi-threaded programs introduce additional errors to single-threaded programs, like deadlocks, livelocks, and data races; and these are more likely to appear when programs are executed in parallel rather than interleaved execution on a single processor. For example, consider two threads T1 and T2 concurrently accessing the class shown in listing 1.1. If T1 is executing the deposit method with an amount of £500 and T1 is executing the deposit

method with £400. If the execution of both threads is performed as follows:

- Balance=£0;
- **T1**: read the value of the Balance
- **T2**: read the value of the Balance
- **T1**: add £500 to Balance
- **T1**: update the Balance value
- **T2**: add £400 to Balance
- **T2**: update the Balance value

The Balance value after both threads finish execution is £400, although it should be £900. This case is called a race condition, and it arises because threads are allowed to concurrently access the shared resources. The concurrency control mechanism should ensure that concurrent accesses to the shared resources does not compromise the contents of the shared resources.

Listing 1.1: Simple multi-threaded application

```
1
2  public class Account {
3       double Balance;
4       public void withdraw(double amount){
5           Balance = Balance - amount;
6       }
7       public void deposit(double amount){
8           Balance = Balance + amount;
9       }
10   }
```

Implementing concurrent data structures and ensuring their correctness is a challenging task [103]. This is due to the fact that threads are executed asynchronously and they are subject to faults and interrupts. So, a synchronization mechanism is required to ensure the consistency and correctness of the data structure contents. The synchronisation mechanism should ensure coordinated access to the shared data structure, such that: (1) no partial update to the data structure contents can be seen by any other thread(s) and

(2) the failure of a thread will not impact on further accesses to the data structure. The synchronization mechanism should provide enough guarantees to ensure that the different operations are concurrently executed in order to fully utilise the available processing resources.

Lock primitives are the traditional, and common, mechanism to control threads' accesses to shared data structure [103]. By using locks, like semaphores and mutexes, certain sections of the thread code are executed in exclusion. This will guarantee the serialization of the threads' accesses to the same shared location(s) within the data structure, which ensures the integrity of its content. This integrity should hold even when code optimization is applied by the compiler or the Operating System (OS).

However, there are different problems associated with using locks. These problems might limit the scalability of the application or degrade the performance [103]. Different solutions have been proposed to overcome such problems, like priority inheritance [115], granularity (changing the size of the code protected by the lock) [53], and helpers (higher priority threads helping lower priority threads in completing their operations) [7]; However, these solutions do not prevent the occurrence of such problems they only provide recovery mechanism. At run time, the occurrence of such problems, and any further implications cannot be predicted, which might compromise the real-time properties of the threads being executed; which is not acceptable for real-time applications.

Concurrent programming is notoriously difficult and it is hard to design concurrent applications that are reliable and scalable [59]. So, concurrent programmers should have the necessary skills to build correct and efficient applications that can fully utilize the computation capabilities of the system.

In real-time applications, threads are executed based on their real-time properties, mainly priority [20]. When a thread requests a lock that is being held by another lower-priority thread, either it will yield the processor and be scheduled later, or it will hold the processor and keep spinning [120]. In this case, if the thread keeps spinning, the lock might be held for relatively long period, which will affect the performance. On the other hand, if the thread yields the processor to allow other threads to be executed this might also

negatively affect the performance; as the lock might be released soon after the higher-priority thread being re-scheduled. This, in turn, might result in valuable computation time being lost due to context switching.

Non-blocking access has been introduced as a concurrency control mechanism for shared-memory data structures; in which threads are concurrently executed, with no prior locking. Hence, non-blocking implementations are free from the problems associated with using locks [51]. The concurrent execution of the threads will produce arbitrary execution paths; the provided non-blocking implementation should guarantee the integrity of the data structure contents regardless of the threads execution interleaving [45].

However, this concurrent execution will only result in high performance if all accesses are read operations. In the case of one of the accessing threads performs a write operation, the provided implementation should perform the necessary actions to ensure the correctness of the threads' execution, such that the data structure integrity and consistency is not violated. The mechanism should invalidate all other current accesses to the same recently written shared location, and the threads that performed these accesses will need to be restarted.

Generally, non-blocking implementations outperform lock-based [127]. However, it is not an easy task to provide a practical and reliable non-blocking implementation. Moreover, it is not an easy task to prove the correctness of the implementation; in general, semi-formal proof and simulation are the common methods for testing the correctness.

The proposed non-blocking implementations are based on the presence of underlying primitives that allow atomic update of the data structure contents, like Compare And Swap (CAS) and Load-Linked/Store-Conditional (LL/SC) [45]. These primitives allow atomic updating of single memory locations, and are used to non-blocking operations on high level data structures such as queues or stacks.

1.1 Real-time Applications

Recently, computer systems prices had been noticeably decreasing, which encouraged the integration of these systems in most aspects of our daily life. This integration increased the demand for developing applications that are of high performance with a high degree of scalability [34]. In certain applications, these systems should provide reliable services with high availability. Different hardware architectures had been developed to support the increased demand on high availability applications like audio/video and high-speed networks.

Multiprocessor systems are considered an indispensable part of the infrastructure for mission-critical and life-critical applications, as they provide the required computational capabilities and maintains high availability by tolerating the failure of one or more processors [106]. Mission-critical applications are real-time applications in which the time predictability is as important as the correctness of the results [20]. In real-time applications performance is a very important issue, however, these applications also require predictability in their execution to ensure that timing constraints are met.

Real-time applications can be classified into: hard, firm, and soft systems; based on how strict the timing constraints are [20]. Hard real-time systems are those systems in which the miss of a deadline will result in a total system failure. In these systems, the miss of a deadline will result in a catastrophic results or endanger human life. Examples of these systems are airplanes, nuclear plants, and space stations.

In firm real-time systems, time constraints are less strict than hard real-time systems; so, occasional misses of deadlines can be tolerated. There are two consequences for the misses of the deadlines: (1) possible degradation of system performance and (2) the results of operations that miss the deadlines have no value. Video conferencing is an example of these systems. In soft real-time systems the occasional miss of deadlines will not result in a system failure, but will degrade the system performance. Soft real-time examples are banking application and airline online reservation system.

Currently, even single processor computers are becoming multi-core processors [32]. Accordingly, the analysis of real-time applications on multicore/-

multiprocessor systems is becoming a complex task due to the increase of complexity in both the real-time applications and the underlying hardware [44, 108].

The introduction of multi-core enforces reviewing the requirements of running real-time applications, as the execution of more than a thread on the a multi-core processor is different from running more than a thread on a single-core processor. In a multi-core environment the interleaving of threads execution might affect the time predictability; hence, it is possible to affect the ability of the concurrency access mechanism to provide wait-free guarantees.

The execution of more than a thread concurrently should be governed such that the integrity of the data structure contents is maintained all the time. It is possible that the interleaving of execution of concurrent threads might result in incorrect results if the concurrency mechanism did not consider all possible execution paths.

The concurrent access of the threads to the data structure contents might result on interference of the execution of one of the threads on the others, also it is possible for a thread to be required to repeat the same operation more than a time, in the case that more than a thread are trying to perform the same operation at the same time. For example, in the case that more than a thread are trying to reserve an empty node in the data structure to perform an enqueue operation; if they are trying to perform the operation on the same node, one of them will succeed in reserving the node while the other writers will try to reserve with another nodes. Accordingly, this is going to affect the time predictability of the threads execution. The concurrency mechanisms should provide certain guarantees on the maximum time required to perform a certain operation.

The multi-core processor is still in development, the number of cores are increasing; this indicates that the number of threads that can be executed concurrently are increasing, and the computational power of the multi-core processor is expected to be relatively higher. The concurrency mechanism should be able to scale as the number of cores scale to utilise the introduced computation power in the multi-core processor.

1.2 Real-time Specification for Java (RTSJ)

Java was introduced as a general purpose object-oriented programming language [111]. Since its introduction, Java has been used in many applications like enterprise software, web-based applications, home appliances, and mobile phones [77]. Java has been developed with many new concepts that encouraged most developers to consider it as the standard language for their applications. The most important features and concepts of Java are "write once, run anywhere" (WORA), object-oriented, automatic garbage collection and memory management.

The WORA enables the programmers to develop their applications once and compile it into bytecode; using the Java Virtual Machine (JVM) this bytecode can be executed on any computer architecture that provides a JVM, which will minimize the time and cost of developing the applications. Java supports automatic garbage collector. In Java, the memory allocation for the application is performed in the heap memory space. When this memory space is consumed, or the application triggers the requirement for free space, the garbage collector will reclaim the memory space allocated for any objects within the heap that are not reachable anymore. This will remove the burden of the memory management from the programmer.

Although it has many attractive features and it is being widely used in many different applications, Java was not considered within the real-time domain. Real-time applications require deterministic execution time, which standard Java cannot support. In a real-time application, threads have real-time properties, which will govern their scheduling and execution; however, thread management in Java does not provide any guarantees for threads execution.

Garbage collection in Java can occur at any moment during the application execution; during the garbage collection process the application may be halted. There is no guarantees on the time taken for the garbage collection process as it depends on many factors like heap size and number of objects within the heap. Although, there are additional techniques (like incremental algorithms) still the garbage collection can occur at anytime and with unbounded time.

Furthermore, in Java classes are not loaded until they are first referenced by the application. The time take to load a class depends on its size and the initialisation ; which requires an additional time when each class is loaded. This will result in additional delays during the execution time, and these delays can be unbounded.

The Real-Time Specification for Java (RTSJ) is an extension for standard Java to facilitate developing real-time applications [16]. It enhances Java in seven different areas, which are: thread scheduling, memory management, synchronisation and resource sharing, asynchronous event handling, asynchronous transfer of control, asynchronous thread termination, and physical memory access.

RTSJ introduced Scoped and Immortal memory areas in addition to heap memory. These new memory areas are not subject to garbage collection, hence, threads using these areas cannot suffer delays due to the garbage collection process. In addition to the standard Java threads, in RTSJ three new thread classes are introduced: `RealtimeThread`, `NoHeapRealtimeThread`, and `AsyncEventHandler` as schedulable objects. `NoHeapRealtimeThread`s are not allowed to reference any object in heap memory to avoid any delays if the referenced objects suffers halts because of the garbage collection process.

Explicit support for multiprocessor system is only provided in RTSJ 1.1 [153]. RTSJ 1.1 provides the support for multiprocessor systems based on the ability provided by the operating system. For different reasons, the processors in the system can be grouped in sets by the operating system, and the processor affinity is used to indicate on which processor's set a thread or schedulable object can be executed. RTSJ 1.1 provides the programmer with the access to the affinity, which enables the programmer the ability to distributed the different schedulable objects within the application across the available processor's sets.

In RTSJ, `NoHeapRealtimeThread` threads are not allowed to access or reference any object in the Java heap. This ensures that the execution of `NoHeapRealtimeThread` threads is free from garbage collection stalls. However, it might be required for the `NoHeapRealtimeThread` threads to communicate with schedulable objects accessing the Java heap. RTSJ provides

`WaitFreeReadQueue` and `WaitFreeWriteQueue`; in both queues, only one `NoHeapRealtimeThread` thread is allowed to access the queue, hence, the access of the queue from the `NoHeapRealtimeThread` thread is wait-free. From the other side of the queue's, multiple Java heap schedulable objects can access the queue. The access is synchronised, hence it is not wait-free. The wait-free queue in RTSJ is provided only for concurrent access between `NoHeapRealtimeThread` threads and those schedulable objects that access the Java heap [153].

Different enterprise application organisations have developed RTSJ implementations like Sun JRS, IBM Websphere, and AICAS JamaicaVM [119], which provides promises that RTSJ will be a reliable environment for real-time applications. RTSJ has been evaluated for mission-critical systems [113].

1.3 Thesis hypothesis

The RTSJ provides a versatile environment to develop real-time applications with predictable execution times. Unfortunately, the initial specification did not explicitly address multiprocessor concerns. As a result, the support provided for wait-free communication is inadequate. However, it is possible to utilise the RTSJ features to develop a wait-free non-blocking queue implementation that can provide the required time predictability and scalability at run time. This implementation can be used as a building block in real-time applications.

Over time, it is expected that the size of the application (i.e. the number of schedulable objects within the application) or the hardware and operating system that support the application will expand. This might affect the execution time of the application as the system is expanded.

Throughout the thesis the term "predictability" is defined to mean at run time the worst-case execution time of the schedulable objects within the application can be determined and the worst-case response time system behaviour can be determined. The term "scalability" is defined as the ability of the application to scale as more resources , like CPU and memory, are added to the system.

1.4 Thesis outline

The thesis is organised in six chapters, as follows:

`Chapter 2 Non-blocking synchronisation approaches`: this chapter reviews the blocking and non-blocking synchronisation approaches. Both approaches will be introduced and discussed concentrating on the main features and issues. A classification of the different approaches of non-blocking synchronisation will be provided; with an explantation of issues that should be considered when implementing a non-blocking synchronisation mechanism.

`Chapter 3 RTSJ and non-blocking wait-free queues`: this chapter introduces the main features of the RTSJ that enable it to be an adequate environment to develop real-time applications. A detailed evaluation of the wait-free queues provided by the RTSJ is undertaken and it is shown that they are inadequate for use in multiprocessor applications. The different non-blocking mechanisms that are implemented in RTSJ will also be discussed.

`Chapter 4: Multiprocessor Wait-Free Queue for RTSJ`: this chapter provides the assumptions, the structure of the algorithm that implements the queues, and other details including the size of the queue.

`Chapter 5 Multiprocessor-safe Wait-Free Queue for RTSJ`: in this chapter the algorithm pseudo code and the implementation of the proposed algorithm is presented. The algorithm is a non-blocking wait-free queue within the RTSJ environment that is targeting the multiprocessor environment.

`Chapter 6 Evaluation`: this chapter provides an argument of the correctness of the implementation with an analysis of its performance based on a developed case study.

`Chapter 7 Conclusion and future work`: this chapter concludes the thesis with a discussion of further work that is intended to be done.

Chapter 2

Non-blocking synchronisation approaches

Concurrency control mechanisms can be classified as either blocking or non-blocking. In blocking implementations, accessing threads should have exclusive access to the shared object(s) prior to performing any operation; which might include modification to the encapsulated data [92]. Non-blocking implementations, optimistically, allow the threads to access the shared objects, without exclusivity; and only interact whenever the accessing threads will compromise the integrity of the data structure [48].

Generally, if all the operations performed on the shared objects are reading, these operations will be serialised in the blocking implementations, unless a read/write lock is used, while in the non-blocking implementations these operations will run concurrently; which will increase the throughput of the application. The main advantage of the non-blocking implementations is that the application can progress in the case of the failure of any thread, which might not be the case in blocking implementations; for example, in blocking implementation the failure of any thread that is holding a lock might affect the progress of other threads that require access to that lock.

This chapter reviews the main non-blocking approaches. Section 2.1 discuss the blocking mechanism. In section 2.2 non-blocking approaches are discussed along with the issues that should be considered when designing

such algorithms. Examples of the non-blocking approaches are introduced in section 2.3. Section 2.4 provides a summary of the chapter.

2.1 Blocking Mechanisms

With blocking mechanisms, threads are required to gain exclusive access to the shared object(s), when a writing operation is required. When a thread tries to access a shared object that had been locked by another thread it will be blocked. The thread will stay blocked until the thread holding the lock releases it; then the thread can access that object, unless another scheduled thread with higher priority is waiting to access that same object.

A blocking mechanism, or lock-based mechanism, is the common concurrency control mechanism to control access to shared objects including shared data structures [92]. This is due to the fact that the use of locks is simple and easy to use by the programmers. Currently, there are different blocking mechanisms that can be used as building blocks in concurrent applications.

While having exclusive access to the shared object(s) all operations performed by the thread will appear as atomic to all other threads [5]. This will avoid any race conditions between the threads and guarantee the integrity of the shared object(s) contents. However, exclusive access enforces the serialization of concurrent accesses to the same shared object. In applications with a high number of threads, it is possible for certain objects to become hot-spots, if many threads require access to the same shared object(s). This will result in high contention on these objects [142], which will affect the execution time of all the threads accessing these objects.

Although the use of locks is simple and easy to program, there are different problems associated with using locks (e.g. deadlocks and priority inversion [28]). These problems might have negative impact on one or more threads or on the application as a whole. There have been different solutions proposed to overcome these problems (e.g. the Banker's algorithm for deadlock avoidance [90] and priority inheritance to limit priority inversions [125]), however, these solutions might compromise the real-time properties of the threads during the execution time or are difficult to implement in multiprocessor environments.

The remainder of this section summarises the main issues associated with lock-based concurrency control mechanisms.

2.1.1 Contention and Lock Granularity

In large multi-threaded applications resource contention is expected at any moment during execution, for example, the quantum chemistry application MADNESS has 65M distinct locks and an average of 30K acquisitions per second per thread [142]. In such applications, contention might limit the scalability of the application and degrade the performance due to long queues of waiting threads.

Lock granularity is used as a contention avoidance technique [140]. Instead of having one lock for a large shared object, the object can be partitioned into smaller objects each with its own lock; this will minimize the interference between threads that do not require access to the same area within the large object [117]. However, partitioning should be performed carefully, as increasing the number of locks will increase the burden on the programmer to ensure that the access to these locks is performed properly.

Generally, there are two granularity levels: coarse and fine. In coarse granularity, the large shared object will be partitioned into a small number of shared objects with a lock associated with each partition. In fine granularity, the large object will be partitioned into a large number of small shared objects, with a lock associated with each object. The developer should carefully design the granularity as a trade-off between the targeted performance and the complexity of the design.

2.1.2 Deadlocks

Irrespective of lock granularity, deadlocks will still occur. Deadlocks occur when multiple threads are requesting exclusive access to multiple resource. In the simplest case, one thread holds a resource whilst requesting access to another resource. That resource is held by another thread that is requesting access to the first resource. Neither of them can proceed [101]. Backoff-and-retry protocols are proposed as a solution for the deadlock problem [140];

threads involved in the deadlock will release all locks being held, and backoff for a certain amount of time, then retry to perform the locking. However, these protocols might suffer from live-lock, which is basically entering a deadlock every time the backoff is performed. Deadlocks become more complex if there are more than two threads involved in the deadlock.

At runtime, a thread might request a lock that is being held by another thread. In this case, the requesting thread either spins-waiting or sleeps-waiting [142]. In case of the spin-waiting, the thread will hold the processor waiting for the lock to be released; during the waiting time the processor will be performing no useful work. In sleep-waiting, the thread will yield the processor to other threads ready to execute, and will be scheduled to continue execution when the lock is free; this requires performing context switching. In both cases, there will be valuable computation time lost. Further problems might arise in either cases, for example if the thread performing sleep-waiting is preempted continuously by other higher priority threads, the sleep-waiting thread might starve.

2.1.3 Lock Convoying

Lock convoying occurs when multiple threads of the same priority repeatedly contend for the same lock [116] and round-robin scheduling occurs for threads with the same priority. Those threads that fail to acquire the lock relinquishes the remainder of their scheduling quantum and forces a context switch. The repeated context switching might introduce an overhead that may result on degrading the overall performance. In case of many threads, it is possible that one or more of the waiting thread will starve while waiting to acquire the lock. From real-time perspective, this might affect the predictability of the execution time, as the time required to finish executing might be difficult to bound.

2.1.4 Composability

Another major issue that faces the use of locks in large applications is composability [123]; it is not possible to take two lock-based correct pieces

of code and combine them into a larger piece. Logically, this increases the developing and testing time to prove an error-free application. For example, in a bank account there is deposit and withdraw operations, shown in listing 2.1; performing such operations on the account requires locking the account to ensure the consistency of the account contents. The withdraw and deposit operations can be considered as abstract and their execution does not interfere with the execution of any other thread. Composing the transfer operation from one account to another requires performing the lock on both accounts, as shown in listing 2.2. The transfer operation can be though of (lock(A); lock(B)) sequence. If for some reason two transfer operations are occurring at the same instance of time; the first is (lock(A); lock(B)) and the other is (lock(B); lock(A)) and both operation succeed in performing the first step; then both operations enter a deadlock status; as each of them is waiting the other indefinitely. Hence, composing a larger lock operation from a smaller correct lock operations might not produce a correct large operation.

Listing 2.1: "Account class"

```
1 class Account {
2     private double balance;
3     synchronized void deposit(double amount) {
4         balance = balance + amount;
5     }
6     synchronized void withdraw(double amount) {
7         balance = balance - amount;
8     }
9 }
```

Listing 2.2: "Transfer method"

```
1 void transfer(Account accountA, Account accountB, int
    amount) {
2     synchronized (acc1) {
3         synchronized (accountB) {
4             accountA.withdraw(amount);
5             accountB.deposit(amount);
6         }
7     }
8 }
```

2.2 Non-blocking Mechanism

Non-blocking concurrency mechanisms were introduced as an alternative concurrency control mechanism to blocking or lock-based mechanisms. With non-blocking mechanisms, threads do not acquire locks for the shared objects being accessed, hence, threads accessing the same shared object will not block each other [48]. If any thread dies whilst executing, it will not block the access to the shared object(s) being accessed by other threads running concurrently and requires access to the shared resources being accessed or affect the progress of any other threads within the application. Thus, non-blocking mechanisms are free from problems associated with the blocking mechanisms; like deadlocks [140]. However, non-blocking mechanisms are hard to implement and prove correct [140][37].

An important issue that should be considered when designing a non-blocking implementation is to guarantee the data coherency; regardless of the object data size it should be manipulated as one set, either in reading or writing [131]. Control variables are used to co-ordinate access to the objects being accessed; so the writer thread will not perform any action with an objects being accessed by any reader(s), and to prevent access from any reader thread to the object being written.

Non-blocking mechanisms should provide enough guarantees such that the threads' execution will not compromise the integrity of the shared data structure contents. Generally, this occurs when there are multiple accesses to the same shared object and at least one of the accesses is a write operation. In such case, the mechanism should ensure that the write operation should appear as an atomic operation for all other threads and all threads currently accessing the updated object should retry based on the new value. The solution should be compliant with the application requirements (e.g. it might be allowed for the reader threads to process the most recent value, as long as that value is consistent).

Generally, a shared data structure is composed of multiple shared objects, like nodes in a queue, and control variables that can be used by operations that access the shared objects to achieve the non-blocking guarantees. Writer

and reader threads perform write and read operations on the shared objects within the shared data structure. Write and read operations can be thought of performing the operation in three stages:

- **Entry protocol**: different threads compete for reading/writing to the shared data structure; the read/write operation is performed on a single shared object with the data structure. One of the threads will succeed in reserving a shared object and the other threads try with the other empty shared objects in the data structure [1]. During the entry protocol, the failure of any thread will not prevent the progress of all other threads performing the operation on the shared data structure.

- **Data access**: the thread, succeeded in reserving a shared object is going to perform the operation that had been designed by the programmer. During this stage, the designed operation is executed in isolation of the other threads on the shared object. This guarantees that the updates to the shared objects will appear as atomic to all other threads.

- **Exit protocol**: upon finishing the designed operation, the thread should ensure that the necessary updates to the data structure are performed in order to reflect on the operation performed on the accessed shared object.

The designed operation to perform the data access stage is application dependent, hence, the computation time required to finish this stage will be known in advance. For the entry and exit stages, different threads are competing while trying to reserve a free shared object within the shared data structure, hence, the time required to perform these stages will differ from one thread to another. This time will depend on different factors, like the number of threads currently competing and their real-time properties like priority.

Generally, the time required to perform the entry and exit protocols are much smaller than the time between two consecutive accesses by the same thread. The threads that just succeeded in the entry protocol are going to

[1] This is different from the thread blocking, as threads which are blocked are doing nothing and are waiting for other operations to finish to be able to continue execution.

perform the data access and exit protocol. These threads might also face delays during their execution like preemption. Hence, the thread(s) that just succeeded in the entry protocol will not affect the current progress of the other threads performing the entry protocol.

2.2.1 Classification of approaches

There are different non-blocking implementations proposed in the literature. These implementations can be classified based on different features, like real-time properties guaranteed, the threads coordination approach, and the supported data structure.

2.2.1.1 Based on the real-time properties guarantee

Non-blocking implementations should be designed with a certain level of guarantee on the amount of time required for a thread to execute the entry and exit stages. As during these stages the thread's execution might be affected by other threads concurrently accessing the data structure. Non-blocking implementations can be classified into three groups based on the level of guarantee they provide [143]:

1. **Wait-free**: for all threads accessing the same data structure, each thread is guaranteed to finish its access within a maximum specific number of steps; regardless of the actions performed by other threads concurrently accessing the same data structure. Generally, this requires careful design for the entry and exit stages to ensure that the interference of the threads during these stages is totally predicted. Both Simpson's Four Slot mechanism in [131] and Hendler and Shavit Work Dealing algorithm in [63] guarantee that every read or write operation will be performed within a specific maximum number of steps.

2. **Lock-free**: during execution, there will be application-wide progress, thus, a thread accessing the shared data structure might be affected by any other thread currently accessing the same shared data structure. However, at least one thread is guaranteed to finish its access within

a maximum specific number of steps. It is possible that one or more threads might starve, as they continuously will be affected by other higher priority threads. In [21] a lock-free approach is introduced based on universal construction.

3. **Obstruction-free**, there will be application-wide progress at any specific time, there is no guarantee on the number of steps any thread will finish its access. Threads might starve during execution as it is possible for one or more threads to be delayed indefinitely. The Transactional Memory (TM) implementation in [66] provides obstruction-free guarantees; the obstruction free can provide non-blocking implementations that are free of lock-based problems, however, it is not efficient enough to meet the predictability requirements of real-time applications.

Of the above approaches, only wait-free provides the predictability needed for real-time applications.

2.2.1.2 Thread coordination

Generally, threads might need to communicate with each other to coordinate their access to the shared data structure. This communication can be asynchronous communication, or communication that requires synchronising the operations currently performed by the different threads accessing the shard data structure.

Asynchronous communication

In asynchronous communication, threads performing the data production and consumption communicate by setting flags to indicate that there is data ready for consumption or the data has been consumed. Thus, it is assumed, theoretically, there is no interference between the threads execution. So, the execution of any thread will not be blocked by any other thread [131].

Generally, if the shared data structure is based on a single word (which can be stored in a single register) the underlying hardware primitives can be used to guarantee the atomicity of the shared contents. If the shared data structure is based on objects, additional operations are required to ensure

that the object's sequential specifications are held [85]. In real time systems, the execution of these operations as part of the concurrent application should be analyzable and predictable [23].

To ensure the shared object consistency, asynchronous implementations [131] [81] used different techniques to guarantee that the writers and the readers are executing in isolation, which is in most cases using flags.

The four-slot algorithm introduced by Simpson [131] represents a simple non-blocking mechanism with a one writer and one reader restriction, where multiple copies of the shared data is stored, as shown in figure 2.1. The 4 slots are divided into two pairs left and right. In each pair there is a top and bottom slots. The writer writes the data into one of the slots and set the corresponding flags; `latest`, which indicates which pair the writer most recently wrote to and `index`, which indicates which slot in the pair the writer most recently wrote to. This indicates that there is a new data ready to be consumed. The reader controls only the `reading` flag, to indicate which pair is being read. The reader has no control over the controls that are set by the writer.

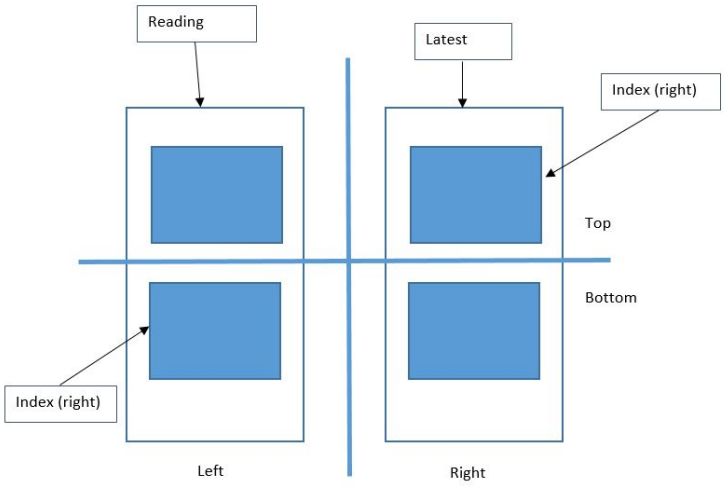


Figure 2.1: Simpson's four-slot algorithm

There is no direct communication between the threads and none of the threads are blocking the execution of the other. However, in Simpson's algorithm there is only four data slots. The writer performs the write

operation on one of the four slots. The reader is following the writer's footsteps. In case, where the writer is faster than the reader, the writer might perform one or more write operations on the same data store, and the reader will only read the most recent copy. In the case where the reader is faster than the writer, it might be possible for the reader to perform one or more read operations from the same slot. Hence, Simpson's algorithm might not be suitable for all applications, like computer networks, where each written network packet should be processed once.

Synchronised communication

In synchronous communication, the progress of any thread might be affected by current operations performed by other threads [26]. Hence, the implementation should provide certain guarantees on the limit of the impact any thread can suffer from all other threads concurrently accessing the same data structure. For example, in Transactional Memory (TM) implementations, the thread committing its changes to the shared object, or node within the data structure, will interfere with the progress of all other threads currently accessing the same shared location; in TM this is achieved by implementing a collision manager that is responsible for resolving the collision results from different transactions accessing the same location in the shared area.

Generally, in multi-threading applications, threads require communication with each other; this communication will affect the concurrency within the application [79]. For example, in a producer-consumer application, the producer(s) will allocate the data in a buffer and the consumer(s) will access that buffer to consume the produced data [110]. The buffer size will be controlled by the implementation based on the application requirements. In certain applications the buffer size will be 0; in such case, if either of the threads, the consumer or the producer is ready to produce or consume, it will block until the other end is available and ready to communicate.

2.2.1.3 Type of supported shared data structures

Non-blocking implementations can be classified into three main categories according to how the data is being stored: universal construction, array based,

and link based [127].

Universal construction

Universal construction is a process that converts data that can be safely sequentially accessed into a concurrent nonblocking data structure [127]. This technique was first proposed by Herlihy in [65]. Herlihy's appraoch requires the data structure to be copied on every update [101], this will affect the performance, and requires large memory space, especially if the data structure is large. Further enhancement performed on Herlihy's proposal [4, 84] mainly reduce the part of the data structure being copied.

Different implementations have been proposed to provide wait-free guarantees by implementing shared data structures using the universal construction [25, 21, 43]. Generally, these implementations are memory consuming and enforce limitations on the maximum number of threads that can access the shared data structure concurrently [127] and are not scalable [144].

Queue

A queue is one of the most common data structures in concurrent applications [146]. Generally, with queue data structures, the enqueue operation is performed at one end of the queue and the dequeue operation on the other end, to support applications that require queues with First-In-First-Out (FIFO) semantics.

Queue based algorithms manipulate the data as an infinite array [70] or a finite array [49]. The programmer should ensure proper selection of the queue size according to the application requirements to ensure best performance and least memory overhead. In the case of a bounded size, the distance between the writer and reader threads is limited by the queue size. In the case of faster reading, the reader threads might try to perform a read operation from an empty list. In the case of faster writing, the writer threads might perform a write operation on a full list [98]. In the case of an unbounded queue size, it is possible for fast writing operations to result in high memory overhead.

There have been different non-blocking queue implementations that are designed for Operating System (OS) or applications [96, 147, 41, 100], each implementation has different level of guarantees and limitation. In [104] a queue based lock-free algorithm introduced to enhance the scalability of the

shared data structure by implementing the elimination technique. When concurrently performing enqueue and dequeue operations on the same queue, the enqueue operations are going to be serialised. In such case, using the elimination technique an enqueue operation to the queue can be eliminated using a dequeue operation.

Generally, lock-free algorithms are based on the CAS operation to support atomic update of the data structure contents; however, in [105] a lock-free queues introduced based on the fetch-and-add (F&A) instruction available in the x86 processors. The authors argued that, although the F&A instruction is a less powerful that the CAS, the implementation of their algorithm provides better performance in both single- and multi-processor environments.

Linked list

Linked lists are used in data structure [69] that might be bounded or unbounded. Unlike queues, linked list based data structures allow the enqueue and dequeue operations to be performed anywhere in the data structure. This requires careful handling of the operations performed on the data structure to avoid any race conditions situations that might occur during burst writing or burst reading.

Concurrent linked lists are attractive for applications that require a high degree of concurrency, as the enqueue/dequeue operations can be performed anywhere in the list [144], which increases the throughput of the application. However, this flexibility in the design requires high memory management overhead. For example, to free any node in the list, so it is memory can be reclaimed, should be performed such that there is no more referencing to that node from any of the processing threads. It might be possible that a node is being accessed by a preempted thread, the reclamation of the memory of that node might affect the progress of that thread when resuming [126]. Hence, the concurrency mechanism should provide support to avoid any such case, which might include booking of all references to the nodes and checking the status of the nodes on every update of every thread accessing the shared data structure.

In linked-list data structures, the dequeue operation should free the node and reclaim its memory location or return the node to a free pool, if the

implementation provides such an option. This operation requires that the dequeuing thread is the last thread to access the node. According to [127], in some implementations this can be achieved by associating a counter with every node, however, if a thread accessed a certain node and is preempted during execution, other threads might create any number of nodes that are successors to the node being accessed by the preempted thread. These nodes cannot be deleted unless that thread is resumed and finishes its execution. Shann et al. argued that all linked list implementations are subject to memory management problems (e.g. ensure a node is not referenced before its memory reclamation is reclaimed), hence these implementation might not produce the claimed level of efficiency [127].

Stacks

Unlike queues, stacks are a special case of linked list based data structures that satisfy the requirement of Last-In-First-Out (LIFO) semantics. The first non-blocking stack implementation was proposed in [145] as a single-linked list with a pointer to the top of the stack. The implementation has been compared to many well-known lock-based implementations, and gives better performance under high parallelism [101]. However, the **top** pointer represents a bottleneck to the implementation, especially in high contention, which limits the scalability of the data structure [103].

In [129] a technique is proposed to limit this contention, by matching a currently running **pop** operation with a concurrent **push** in progress. Thus, the operations will eliminate each other, and the data structure will not be affected. Although this technique can enhance the performance of the implementation, it might only be practical for certain application. Applications that enforce certain order on reading and writing operations might not be able to benefit from this technique.

2.2.2 Non-blocking design issues

Non-blocking mechanisms are free from lock-based problems and can provide required scalability. However, there are different issues that should be considered and carefully handled to enable fully utilising their potential.

2.2.2.1 Multi-processor and multi-core processors

Generally, non-blocking mechanisms were introduced for single-core processors [146, 94] and depend on the concept that only a single thread is running at any specific time. This limits the complexity of the implementation. The introduction of multi-core processors enables multiple thread to run concurrently on a single processor machine. Hence, the threads' execution will be arbitrarily interleaved, which increases the implementation complexity of any algorithm. A non-blocking mechanism should guarantee that updating any field in the shared data structure should appear as an atomic operation to all other thread(s) concurrently accessing the same shared data structure [69]. Thus, the interleaved accesses to the shared object(s) should be properly handled to ensure that it will not compromise the shared object integrity [85].

Figure 2.2 shows the execution of two threads T1 and T2 on a single-core processor with multi-tasking, and a multi-core processor with parallelism. On the single core processor machine, T1 starts executing and waits after some time because of a memory access operation, after the time slice for T1 is finished, a context switching time is required in order to make safe all the current states of T1 and the loading of T2. One further context switching is required to load T1 to finish execution. In the multi-core case, each of the threads is loaded into a different core, hence, no context switching is required. If there is no dependency between the threads execution each thread can proceed with no interference from the other thread.

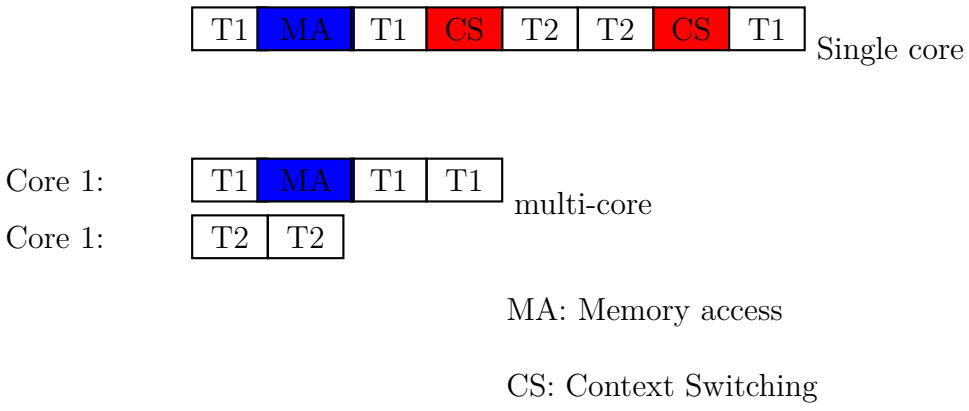


Figure 2.2: Threads execution on single and multi-core processors

The state of all currently running threads should be synchronised with all other threads, so that the synchronisation mechanism can take the required decision accordingly. The "challenge is to minimize the cost of inter-core communication, such that threads residing in different cores can efficiently exchange state information" [87, p. 1].

In multiprocessor and multi-core processor environments, concurrent accesses to the same shared object may result in data races [8], if any of the threads update the shared object. Generally, the exit protocol will result in either a write operation or a check on the accessed shared object(s) contents to ensure that no changes occurred since the last read, as the execution should be completed with consistent data.

To prevent data races, the wait free mechanism should be able to roll-back any updates performed by threads that cause the race [154]. To prevent consecutive data race, the mechanism should be able to roll-back the effects and back-off the violating thread, allowing enough time for the winner thread to finish execution [154]. Generally, roll-backs have negative influence on the performance; as it is not just a waste of computational time, there will be an additional overhead to undo all the changes performed. The STM implementation introduced in [114] addresses the cost of roll-back, the implementation uses an additional overhead of book-keeping in order to perform partial roll-back of the changes, instead of full roll-back; which minimises the overhead of roll-back.

As the number of processors increase in the system, or the number of cores in a multi-core processors, it is logical to expect more contended data updates to occur. In such cases, only one thread will be able to commit its updates and all other threads should retry. In large systems (where it is possible to have 100's of processors), the number of retries could be very high, and the back-off will result in high inefficiency within the system [26].

In certain applications, threads perform complex operations like write-after-read and read-after-write operations [40], which increases the complexity of the non-blocking algorithm's design and the possibility of compromising the shared data structure contents. The memory consistency and cache coherency could be used to overcome these problems, such that all threads will progress

with consistent data [133]. Mainly, this occurs in cases subject to data and memory hazards, where more than a consecutive operation requires read or writing of the same variable. In other cases, like read-within-write and write-within-read, the execution of one of the threads will be delayed until the other one finishes, this will affect the overall performance of the system as one or more of the processors will be idle for a certain period. This will also affect the system's predictability. Allowing multi-location storing of the data in the shared area, arrays or linked lists [131], partially solves these problems by eliminating the interference between the producers and consumers, but with an added overhead due to synchronization and control operations.

In [133] different examples of data and memory hazards are introduced. For example, it is possible for a read operation for a variable to appear directly in the code after a write operation; in this case, a delay between the two operations should be inserted to allow enough time for the write operation to finish, otherwise, the read operation might read inconsistence data. The more common these operations appear in the code, the more delay will occur, which might affect the performance.

2.2.2.2 Memory Management

A non-blocking implementation should ensure that all accesses to the shared data structure are monitored, such that the consistency of all data items within the data structure are maintained [127]. Also, a data item within the data structure can only be deleted when there are no further references to it.

Programming languages that support garbage collection, like Java, might remove the burden from the programmer to perform such monitoring, however, it is possible for an object to reside in the memory for long period if it is referenced by a thread that is halted or being preempted continuously.

Valois in [149] proposed a solution that depends on freeing (or deleting) the node from the data structure based on a counter that represent the current number of references to that node. This solution has different issues, as a node being accessed by a preempted thread cannot be deleted until that thread finishes execution, this also includes all successor nodes added while

the thread is preempted. In general, the proposed solution have an additional overhead in both processing time and memory space requirements. This solution might compromise the real-time properties of the threads within the application.

The hazard pointers solution proposed in [99] is a more practical solution, as it requires less computation time and memory requirements in performing the reclamation of the memory locations occupied by the de-queued objects. In the hazard pointer solution, each thread logs in a list all nodes accessed during the node execution; this list can only be written by the owning thread and read by all other threads. When a node should be deleted by the thread, it is removed to a private list; hidden from all other threads, and the thread performs regular check on the read lists of all other threads. When there are no more references to that object by the other threads it can be safely deleted.

2.2.2.3 Real-time characteristics

In multi-processor system (and multi-core), when multiple threads compete on the same shared location, only one thread will succeed and the others will either spin-waiting or sleep-waiting. The thread with the lowest priority (or an unlucky thread) will lose the competition to all other threads. In real-time applications, threads should meet their real-time properties, thus, the implementation should ensure that delays in execution caused by other threads will not affect meeting these properties.

Hence, from a real-time perspective, proposed wait-free algorithms should be analysed to determine the worst case execution time of its operations. The operations should include: addition and deletion cost of object contents, memory management and the interaction with other parts of the system like I/O. It is even possible certain algorithms can be application specific rather than general purpose.

The analysis of the thread's execution should include the costly CAS operation [80], which is part of any interaction with the data structure. Moreover, the CAS operations will become more costly in contention periods, where the CAS will be performed each time the thread is retried.

As the number of threads grow, the interleaving in threads execution will increase the complexity of the application. Different sources can cause further delays in threads execution like a cache miss [26]. This will affect the predictability of not only the thread's execution but the execution of all other threads currently in progress.

2.2.2.4 ABA problem

In general, non-blocking implementations that at the end of the thread (or transaction) execution the thread is able to update its changes to the shared area if the initial values of the accessed locations are not changed. However, it is possible for a thread T1 to access a shared variable V1 and its value is A to be preempted during execution. While T1 is preempted, it is possible that a thread T2 updates the shared location to B, and another thread T3 updates V1 again to A. When T1 resumes execution, it is going to updates its changes to the shared area based on that V1 value is A, while it should not as the value of V1 is changed to B then to A; this is well-known as the ABA problem.

Generally, non-blocking algorithms are based on primitives that support atomic update of the memory locations, like CAS and Load-Link/Store-Conditional (LL/SC). In LL/SC operation, a memory location is read by the LL operation. This memory location will be updated by a new value using the SC operation, if this location has the same value that is read by the Load-Link operation [141].

The LL/SC is more complex to implement and might suffer false fails. In a multiprocessor system, if another processor writes to a memory location that is near to the location being read by an LL operation the SC operation will fail, unless the implementation is able to monitor the memory translation page [141].

Unlike the CAS operation, LL/SC is not subject to the ABA problem, as the memory location read by the LL operation will be cached in a special register, and this location will be invalidated if any access occurs to that memory location; even if the thread is being preempted.

Although non-blocking implementations are free from lock-based associated problems, like deadlocks and priority inversion, CAS based non-blocking implementations suffer the "ABA" problem [150]. “The ABA problems occurrence is due to the intricate and complex interactions of the applications concurrent operations and, if not remedied, ABA can significantly corrupt the semantics of a nonblocking algorithm.” [31, p. 1].

During the threads execution, a thread might read a current value of a shared object, say "A"; the thread might be preempted after reading the value of the shared location long enough such that the same shared location will be updated to "B" then to "A" again. When the preempted thread resumed, it will finish execution successfully, although it should fail and retry.

To avoid the ABA problem in non-blocking implementations three techniques had been used in the literature [24, 99, 33].

The first, is associating a counter with each object in the shared data structure first proposed by IBM [38]. The counter will be incremented each time a thread accesses the object and decremented each time a thread releases it. The object will be deleted by the thread that last releases the object, allowing the reclamation of the memory space reserved by the object. Generally, this solution is suitable for applications in which the data stored in the same node within the data structure might be consumed by multiple reader threads. This solution can be applied to cases where applications require multiple write or read on the same shared location, like array, hence, it cannot be considered as a general solution for all implementations; in the case of the stack, nodes are added to or deleted from the stack once only.

The second, uses reference counting [33, 61], where a counter is associated to each shared object and this counter will be incremented by each thread accessing that object. If the thread is preempted while execution, the counter will be used to ensure that the object had not been updated by any other thread since the thread last access it. If updated, the thread will abort and retry. In contrast to the counter in [38]; the counter in this solution is only incremented and never decremented.

Both solutions are based on the availability of the double-word CAS operation, which is viable in 32-bit processors that support double-word

CAS (64-bit CAS operation). The current 64-bit processors support only single CAS operation. However, this solution is applicable in certain cases; like the stack case, as the reference counter can be assigned to the `top` pointer; and each thread can monitor any changes to the counter during its execution. If any change has occurred to the counter, this indicates that there is another `pop` or `push` operation completed which requires aborting the current operation and retry. It is possible also for this solution to be used in non-blocking implementations in which the data stored in the nodes are allowed to consumed only once. In this case, a reference counter is assigned to each node, and the accessing threads can use this counter to ensure that there is no changes occurred, if the thread being preempted.

The third solution is a generic solution that can be used for any non-blocking implementation, the hazard pointers which is "a memory management methodology that allows memory reclamation for arbitrary reuse" [99]; the solution is supposed to remove the burden of reclaiming the memory locations of the deleted objects from the implementation.

In the hazard pointers, each writer thread maintains a list of all shared objects it accesses. Besides that, each thread maintains a retired list; or the shared object(s) that the thread finished working on and want to delete. When the retired list of any thread reached a certain predefined size, it scans the hazards pointer of all other threads; to free those objects that are not referenced any more by any other thread. The use of hazards pointers solution ensures that no memory allocated to a shared object will be reclaimed unless there is no more references to that object, which ensure safe deletion of shared objects.

Although the hazards pointers solution guarantees safe reclamation of memory allocated to shared objects, it suffers different issues. There is an increased amount of memory that will be consumed to maintain the different lists; this amount increases proportionally with the number of threads and the number of shared objects being created. In addition, the amount of time required to perform the scanning of the lists might affect the threads' execution time, as the time required to perform the scanning is also proportional to the number of threads and the number of live objects.

Moreover, it might create further drawbacks, like the case of multiple objects being referenced by a thread that is preempted for long period of time; these objects will reside in the list of all other threads that accessed them earlier until the preempted thread is allowed to finish execution.

The ABA problem occurrence might result in any arbitrary situation, thus is important that a CAS based non-blocking implementation adapts a proper mechanism to avoid the ABA problem. To further understand the consequence of the ABA problem, the following examples discusses the scenario of an ABA problem occurrence.

Assume a Last-In-First-Out (LIFO) linked-list stack shared data structure. The node is composed of a pointer to a reusable data store and a pointer to the next node in the stack, as shown in listing 2.3. Threads accessing the stack either add a new node, `push`, or remove an existing node, `pop`. The `pop` operation, listing 2.4 removes a node from the top of the stack, figure 2.3. The `push` operation, listing 2.5, adds a new node at the top of the stack, figure 2.4.

Listing 2.3: Stack node

```
1 struct Node
2 {
3     PTR* data;
4     Node* next;
5 };
```

Listing 2.4: Pop operation

```
1 Node pop(){
2   Node* newTop, oldTop;
3   if (Top == NULL) return false;
4   oldTop = Top;
5   do {
6       newTop = oldTop->Next;
7   } while(CAS(Top, oldTop, NewTop));
8 }
```

Listing 2.5: Push operation

```
1 Node pop(){
2   Node* newTop, oldTop;
3   if (Top == NULL) return false;
```

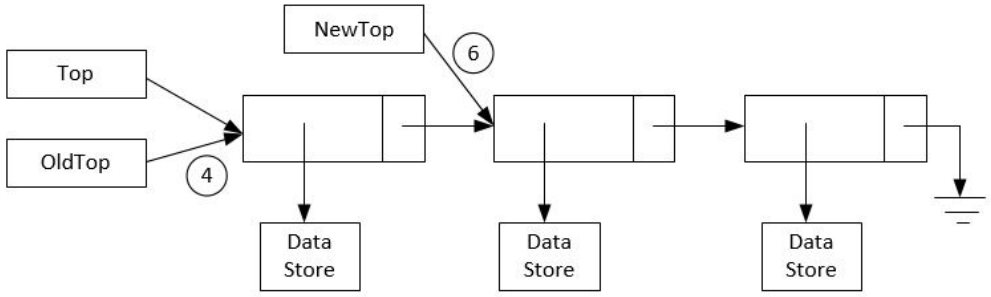


Figure 2.3: Pop operation

```
4     oldTop = Top;
5     do {
6         newTop = oldTop->Next;
7     } while(CAS(Top, oldTop, NewTop));
8  }
```

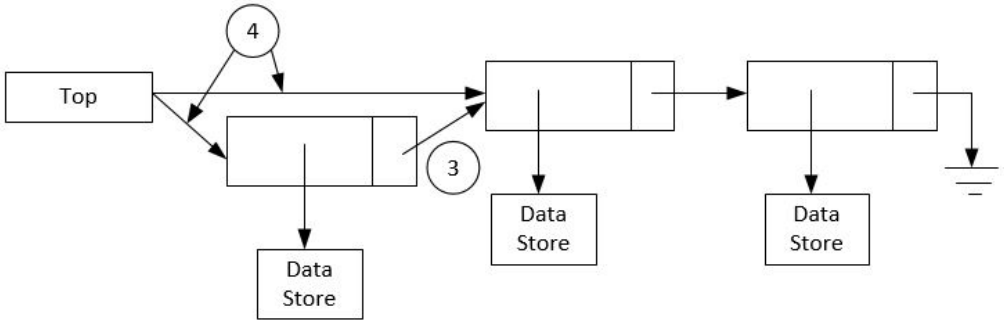


Figure 2.4: Push operation

If a thread T1 performing a pop operation, is preempted after reading the top value and before performing the CAS operation to update the top of the stack, listing 2.4; with stack status as shown in figure 2.5. During T1's preemption, another thread T2, performed two pop operations (nodes A and B) and a push operation (node A); the status of the stack will be as shown in figure 2.6. When T1 resumed, it will finish it's pop operation and the CAS

operation will succeed. The top will be updated with reference to node C.

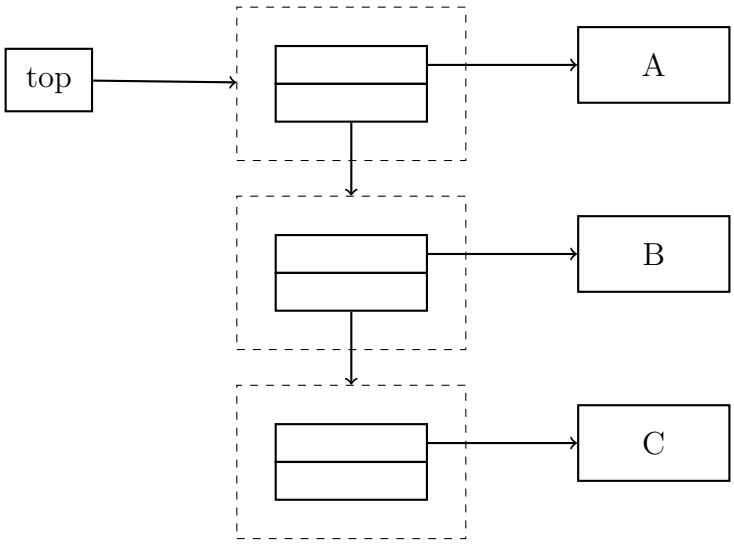


Figure 2.5: Stack before T1 preemption

Thread T1 should be aborted and retried based on the current case of the stack, rather than removing node A and updating the top of the stack to node C. Accordingly the pop operation should be reviewed to ensure that CAS only will succeed when there is no update to the top pointer occurred during thread T1's preemption.

Among the three solutions, the ABA problem in the stack case is best solved using a version number, which is associated with the `top` pointer [126]. The version number is used to ensure that no changes occurred to the stack during the thread preemption.

The `top` pointer of the stack includes a reference counter as shown in 2.6. The reference counter should be set to an initial value when the top pointer first initiated. The pop and push operations, should be revised, as shown in listings 2.7 and 2.8, respectively; if any thread preempted while performing

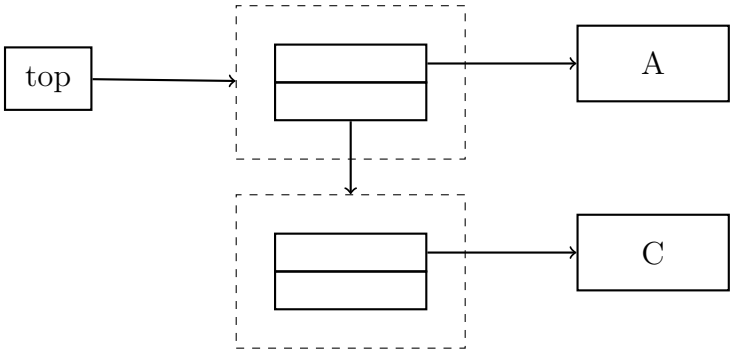


Figure 2.6: Stack After T1 resumed

the pop operation after reading the top pointer value it is not going to suffer the ABA problem, as the reference counter is incremented before performing the CAS. A preempted thread is not going to suffer the ABA problem, as the CAS will be invalidated if any update occured to the top pointer during the preemption.

Listing 2.6: "Top pointer"

```
1 struct stackTop{
2    int refcounter;
3    Node* FirstNode;
4 }
```

Listing 2.7: "Revised pop operation"

```
1 Node pop(){
2   stackTop* newTop, oldTop;
3   if (Top->FirstNode == NULL) return false;
4   oldTop = Top;
5   do {
6       newTop = oldTop->FirstNode->next;
7       newTop.refcounter = oldTop.refcounter + 1;
8   } while(CAS(Top, oldTop, NewTop));
```

```
9 }
```

Listing 2.8: "Revised push operation"

```
1 void push(Node* NewNode){
2    stackTop oldTop, newTop;
3    oldTop = Top;
4    do {
5         NewNode->next = TOP->FirstNode;
6         newTop->FirstNode = NewNode;
7         newTop.refcounter = oldTop.refcounter + 1;
8    } while (CAS(TOP, oldTop, newTop));
9 }
```

2.3 Examples of Non-blocking Approaches

Non-blocking implementations have been developed to provide a reliable and scalable environment to access shared data structures. These implementations are designed to meet the requirements of certain applications or as a general concurrency mechanism that can be used as a building block on any application.

2.3.1 General and specific implementations

In early non-blocking implementations, there have been limitations placed on the concurrency level of the shared data structures; in [82] only one process (or thread) is allowed to access the shared data structure, and in [131, 63] the number of threads is limited to only one writer and one reader threads. In Simpson's implementation in [131] there is a limitation on the application

usage, as the implementation designed such that the writer can overwrite a previously written data, which had not been yet consumed; hence, it is suitable for applications that require that the reader can read the most recent written consistent data.

The first non-blocking stack implementation was proposed by Treiber in [145] as a single linked list with top pointer [126]. The implementation supports the reuse of the nodes, thus, a free pool is provided to manage the allocation, use, and reuse of the nodes. However, the implementation requires high computation overhead. Although Treiber's implementation outperformed many other stack-based implementations, either lock-based or non-blocking [101], stack based implementations have limited scalability as the number of threads increase [64]. Certain stack based implementations limit the number of threads that concurrently access the shared data structure to avoid contention issues [15].

To overcome the problem of limited scalability, a non-blocking stack implementation is proposed by Hendler and Shavit in [64] that supports the elimination process; in which a `pop` operation in progress can be paired with a concurrent `push` operations. Logically, this will increase the parallelism of the implementation. In [64] a thread T1 attempts to perform an operation on the stack and fails due to another thread currently in progress on the stack. T1 announces its intention to perform the operation; T1 will read a random location from the collision array (which is designed to hold all failing operations). T1 keeps trying until it pairs with another announcing thread (finds a pop if wants to perform a push, or vice versa); if a certain time elapsed before being able to pair, T1 tries to perform the operation directly on the

stack again.

T1 can only collide with (or eliminate) T2 if both threads performing opposite operations (a pop can not eliminate another pop). The elimination process is a two consecutive CAS operations; the first in which the colliding thread, T1, will mark its location in the collision array as in progress of elimination and the second is to mark the collided thread location, T2, as reserved. This will raise the possibility of a race condition as it might be possible after succeeding in the first CAS and before performing the second CAS for another thread, T3, to succeed in reserving the collided thread, T2. Moreover, it might be possible for a thread T4 trying to collide with T1, while at the same time T1 is trying to collide with T3. If both T1 and T4 succeeded in their first CAS operation at the same time, T1 only can finish the elimination process, while T4 is required to retry with another thread.

The elimination process helps in minimizing the contention on the stack, however, from a real-time perspective, the time required for a thread to finish its pop or push operation is not bounded. In high loads, there is high probability that the thread will finish fast, as the collision array will contain high number of announcements; but a thread might be unlucky enough to all the time read from locations that announce the same operation. In low loads, a thread might be unlucky if it reads all the time from empty locations. As the read operation from the collision array is performed randomly.

Colvin and Groves improved the Hendler and Shavit implementation in [27] using a simpler data structure and better symmetry within the elimination process; Colving and Groves implementation is better in performance and scalability. Shavit and Touitou in [129] improved Hendler and Shavit

implementation by designing the elimination technique using trees, in which colliding pop and push operations will not affect the stack, hence, the implementation is more scalable that the original implementation in [64].

In [96] Massalin and Pu introduced a non-blocking stack implementation that is simple to use, however, the implementation requires the availability of the CAS and DCAS hardware operations, which makes the implementation limited to specific hardware architectures. Afek et al. presented a wait-free stack in [2]. Although the implementation is simple, it is based on an infinite array, which is not practical as it requires high memory overhead.

Queue based implementations can be based on either linked-lists or arrays [103]. An array-based implementation proposed in [82] and a linked-list based implementation proposed in [63]; both implementations provide a wait-free guarantee, however, they enforce constraint on the number of threads concurrently accessing the shared data structure to one writer and one reader.

An array based implementation proposed by Herlihy in [70] based on an infinite array; which may suffer from memory overhead. A modular concurrent data structure (MCRingBuffer) is proposed in [87], which enables storing any data type within the data structure. The implementation is designed to utilise the computation capability offered by the multi-core processors.

Stone in [137] introduced a non-blocking linked-list circular queue implementation, which suffers data race between the enqueue and dequeue operations [100].

Michael and Scott implementation in [100] is non-blocking and based on a linked-list data structure, however, the implementation suffers memory management issues, as the implementation allows the re-use of the removed

nodes. Two CAS operations are required to add/remove a node to the data structure. If a thread performs the first CAS, then is preempted long enough for any other thread to complete its add/remove operation, the thread when resumed will break the data structure. Thus, additional overhead should be implemented to perform the required memory management.

Ladan and Sahvit [83] introduced a new dynamic memory lock-free queue algorithm for a multiprocessor architecture, which outperforms the most effective dynamic memory concurrent algorithm introduced by Michael and Scott [101]. For multicore processors it is expected that it is best to utilize the reduced cost CAS operation in the multicore processors, "This question remains to be answered." [83, p. 325]. Dechev et al. in [31] introduced the first lock-free design of a dynamic resizable array; that runs on multi-core processors with higher efficiency compared to previous solutions, but even with the high performance its not ABA free [30].

2.3.2 Transactional Memory (TM)

The success of the transactions in exploiting a high degree of concurrency within the database domain encouraged the adaption of transactional execution as a general concurrency model in the parallel programming domain [60]. In TM, the concurrent part of the code is marked as a transaction, that should be executed as an atomic block. There will be no locking for any of the shared objects accessed during the transaction execution, hence, TM is a non-blocking concurrent mechanism. The implementation should monitor all shared memory accesses during the transaction execution, and only interact

when necessary [128]. Generally, the transaction might require access to different shared objects, or memory locations during the execution; at the end of the transaction execution, it might be required to update some of these memory locations with new values. This is only allowed, if none of the accessed locations have been changed during the transaction's execution [3]. Otherwise, all updates will be discarded and the transaction will be retried.

Transactions are executed without locking, optimistically, assuming that the transaction will finish execution before any of the accessed memory locations have been changed [58]. This is based on the idea that transactions are light code portions, which does not require long execution periods [112]. Accordingly, this will minimize the chances of transactions being retried.

TM implementations can be classified into: hardware (HTM), software (STM) and hybrid (HyTM) implementations [60]. Hardware implementations rely on underlying hardware support to provide atomic update to the accessed memory locations [68] but may not be available on all processor architectures. Software implementations do not require any specific hardware requirements, the atomicity is guaranteed using software; which makes them architecture neutral [59]; however, STM implementation requires high execution overhead, especially in high loads. Hybrid implementations are a mix of hardware and software requirements, at runtime, the best execution scenario will be selected to guarantee the highest possible performance [9, 102].

The first HTM implementation was proposed by Herlihy and Moss in [68]; the implementation requires changes in the underlying cache coherency protocols, which hardware manufacturers did not provide at that time. Shavit and Touitou proposed the first Software TM in [128], the implementation is

hardware independent, thus, it is architecture neutral. In [128] the implementation enforces a limitation on the number of transactions that can be executed concurrently and the amount of memory that can be allocated to each transaction.

Transactions can be executed in any arbitrary order. During the transaction execution, it is possible to update any accessed shared memory location. To guarantee the integrity of the shared memory contents, each transaction (or the implementation) should maintain a list of all accessed memory locations, with their initial values [55]. Generally, the transactions either perform direct update to the memory location(s) or will buffer all updates to be perform at the end of the transaction execution [60].

An update manager should be implemented to facilitate monitoring the accesses and updates to the shared memory. With each update, the update manager will interact according to the update mechanism performed by the transactions; **direct** [3] or **buffer** [67]. Accordingly, when the number of concurrent executed transactions are high, it is expected to have high spatial (due to log lists) and computational (due to retries of execution) overhead.

The update manager should maintain a read-list and a write-list of all accessed memory locations [58]. Whenever, a transaction performs an update to a shared location, the update manager should invalidate all other transaction(s) currently accessing that location. Accordingly, a conflict manager should be used to scrutinize the lists to detect any conflict. The conflict manager will take a decision of which conflicting transactions will be aborted and resumed and which are allowed to proceed. The decision process should be carefully designed to avoid any problems like starvation,

a transaction is being aborted indefinitely or performance degradation, a single transaction allowed to proceed at the cost of many transactions being aborted.

Generally, if the update manager adopted a direct memory update mechanism, the conflict manager should adopt an **eager** technique, in which the aborting decision will be taken immediately after every update. On the other hand, the conflict manager can adopt a **lazy** technique if the update manager uses a buffered mechanism, in which the aborting decision will be delayed until the transaction finishes execution [60].

The conflict manager is a very important component within the TM implementation, as it might highly affect the performance of the implementation if wrong decisions are taken. In case of conflicting transactions, it is required to clear the logs of the conflicting transactions and resuming their execution. The computational cost will scale as the number of transactions increase, which might negatively affect the performance.

In high contention periods, transaction abortions are expected to increase. In multiprocessor system, if large size transactions are executing, in a lazy update mechanism mode, high computation time will be lost; as the abortion of the conflicting transactions will be delayed until the commit stage [123]. On the other hand, if small size transactions are executing, in eager update mode, the abort percentage is expected to be very high and the transactions might hold the processors without performing any progress. As part of the conflict manager, a contention manager should be implemented to detect the contention cases [69].

The contention manager is supposed to defer the execution of the con-

tending transactions to avoid high percentage of aborts. There are different contention manager implementations proposed to avoid contentions, like backoff, aggressive, priority, randomised, greedy, and Karma [69]. A good contention manager can solve the contention by delaying the execution of the contending transactions, such that allowing more threads to finish execution successfully; eventually, this will minimize the wasted processors' time and increase the performance of the system.

In Hardware TM, memory updates are based on the underlying cache coherency protocol, which is limited to word size per update [67]; the transaction can perform multiple updates. In Software TM implementations, the transaction can update a memory location, an object, or multiple objects [128]. In Software TM implementations, the data structure should be designed such that it will allow maximum concurrency. At runtime, it is possible for different transactions to access different locations within the object, if the data structure is object based, one of these transactions will finish execution successfully and all the others will retry.

The granularity within the data structure is an important feature that can be used to tolerate the level of conflict, consequently this will have direct effect on the contention. In Hardware TM, the granularity is one word, hence, the parallelism can be very high. In Software TM careful granularity level should be considered, as this can minimize the computations overhead and spatial requirements of the TM implementation.

Since the first introduction for a hardware TM in 1993 [67] and software TM in 1995 [128], different TM implementations had been introduced [3, 9, 52, 55]. Each of these implementations addressed one or more of the TM advantages,

and/or try to overcome some of the TM obstacles.

Harris and Fraser introduced a lightweight atomic transactions implementation in [59]. Instead of using lock-based access to shared objects or data structures, atomic transactions are used. The introduced implementation does not suffer from any lock-based problems, however, it cannot provide real-time guarantees, as there is high overhead in atomic transactions execution.

Although TM implementations had been under development for nearly two decades, there are different issues that limit the use of TM as a general concurrent mechanism within real time applications. Among the most important issues are input/outut operations, support for legacy applications and contention [60].

At run time, transactions might require input or output operations [57, 72]; which are irreversible operations; accordingly, transactions that perform input/output operations must succeed when committing their updates and other conflicting transactions will be retried. However, it is possible that two transactions that have already performed input or output operations conflict, in this case, allowing both operations to commit might compromise the integrity of the data structure contents.

There had been different solutions offered for this problem [109, 14, 134, 135, 56, 11, 97]. The provided solutions depend on identifying if the transactions requires input or output operations before or during their execution. Transactions that are identified to have input/output operation in advance of their execution, will be guaranteed to succeed, hence, only one transaction will be allowed to guarantee at a time, which will limit the degree of parallelism. Transactions that are identified to have input or output operation during

execution, will commit in two stages, the first one is to ensure that there are no conflict and the second stage is a real commit stage.

In large scale applications, it is possible to have different components, which might be lock based or non-blocking [11, 118]. Based on that, transactions might suffer conflicting with lock-based threads. TM implementation should be aware of lock-based accesses to the shared contents and ensure proper execution of the transactions within such environment [39].

In multiprocessor systems, and multi- and many-core processors, it is expected to have high contention periods [54], where thousands of objects should be updated tens of times each second [36]. The contention can be minimized by the backoff of the contended transactions to allow other transactions to finish execution. However, the backoff decisions might lead to further problems, like starvation, if the same transaction(s) being preempted continuously or performance degradation, if large sized transactions are retried by small sized transactions [69]. Accordingly, TM implementations should ensure that the contention management avoid such cases and provide results that meet the real-time application requirements.

The first TM implementation to address real-time application requirements was introduced by Manson et al. in [95]. The implementation introduce the concept of Preemptible Atomic Regions (PAR). If a higher priority thread is ready for execution, the current atomic region will be preempted, even if the higher priority thread is not executing an atomic region. The implementation has two limitations, the first it is designed for uni-processor systems and the second in case of interrupts the effects of the atomic region will be lost and cannot be undone.

Schoeberl et al. introduced in [124] a hardware TM that is targeting hard real-time applications. The different components of the TM is designed to ensure that the maximum number of retries is bounded; which results in a predictable execution time for the atomic regions. Although the results of the experiment is promising, the implementation is not practical is it requires fully associative cache; which is very expensive and enforce limitation on the atomic region size.

A software TM implementation is introduced by Sarni et al. in [122] that targets soft real-time applications. The implementation integrates a real-time scheduler. This was the first TM implementation to consider the deadline as a factor when scheduling the transactions. However, in cases of high contention, the increased number of retries might compromise the predictability of the transactions' execution. Moreover, in solving the conflict, the older transaction is allowed to proceed, which might affect the real-time properties, as the retried transaction might have higher priority or earlier deadline.

The deadline issue had been addressed in the Software TM implementation proposed by Fahmy and Ravindran in [42]. The implementation contributes the real-time characteristics of the transaction within the scheduling decision as the contention manager is designed to achieve the Earliest-Deadline-First (EDF) semantic. Although the implementation is promising and competitive to lock-based and other non-blocking implementations, it only provides obstruction free guarantees.

In [6] a simplified implementation of lock-free transactional solution for real-time systems was introduced; however, it is designed for uni-processor

environment with the requirement of the multi-word CAS. The implementation bounds the number of retries by modifying the scheduler. The time required for any thread can be bounded as long as no higher priority thread interfered the threads in progress.

2.4 Summary

Although existing non-blocking synchronisation approaches promise performance boosts and implementations that are free from lock-based associated problems, the proposed algorithms have their own problems, like having substantial storage overheads, possible serialisation of non-conflicting operations, or requiring additional hardware support not available so far in current CPUs [45].

In general, TM implementations are not adequate to support a deterministic concurrent application [86], as the proposed implementations require high spatial and computational overhead, and there is no limitation on the maximum number of retries.

From real time perspective, only wait-free implementations are suitable for hard real-time applications; as they guarantee a bounded execution time of the threads accessing the shared data structure. With lock-free and obstruction-free implementations, low-priority threads might starve, as the time required to interact with the data structure cannot be bounded due to the interference of other threads concurrently accessing the data structure. which make it more suitable for firm and soft real-time applications.

Chapter 3

RTSJ and non-blocking wait-free queues

Java has proved that it is a premier environment for real-time embedded systems [12]; currently, Java has presence in different manufactured real-time applications and dominance in certain applications, like Blu-ray Disks. Real-time applications can vary from simple applications, tens of lines of code supported using a single processor machine, to large scale applications that are complex and require high computation power [19]. Standard Java might be an appropriate environment for simple real-time applications, but it is not a reliable environment to implement large scale and hard real-time applications; as it provide adequate support for threads with hard and soft time constraints.

RTSJ is an extension for standard Java to facilitate developing large scale real-time applications [35]. The first version of the RTSJ Specification does not provide explicit support for multi-processor environments, however,

implementing applications within the multi-processor environment was not ruled out. The increased adoption of multiprocessor systems in real-time applications, has resulted in explicit support for multiprocessor environments in the latest RTSJ version, although it is only limited to Symmetric Multi-Processor systems (SMP) [71].

This chapter introduces the RTSJ features that facilitate the development of real time applications. This chapter is organised as follows. Section 3.1 discuss the RTSJ features. In section 3.2 a discussion for the wait free queues provided in the RTSJ specifications. Algorithms that are developed for RTSJ environment is introduced in section 3.3. Section 3.4 provides a summary of the chapter.

3.1 Core RTSJ Feautres

RTSJ enhances the Java environment in seven different areas; which are: thread scheduling, memory management, synchronization and resource sharing, asynchronous event handlers, asynchronous transfer of control, asynchronous thread termination, and physical memory access [35].

In this section we introduce the main RTSJ features that are used by this thesis: real-time threads and memory areas.

3.1.1 Real-time threads

The standard Java `Thread` class provides no support for real-time properties. Accordingly, RTSJ introduces two new thread classes, `RealtimeThread` and `NoHeapRealtimeThread`. The `RealtimeThread` class extends the standard

Java thread class (java.lang.Thread) and the `NoHeapRealtimeThread` class extends `RealtimeThread` class.

The `ReleaseParameters`, `SchedulingParameters` and `MemoryParameters` objects are passed to the constructor of the RealtimeThread, during initialisation, to indicate the real-time properties of the thread. These properties are used by the scheduler to guarantee that these threads will be executed based on their real-time properties.

A `NoHeapRealtimeThread` is not allowed to reference the standard Java heap memory, hence, NoHeapRealtimeThreads are free from any garbage collection interrupts. Accordingly, NoHeapRealtimeThreads facilitate developing applications that are time critical as these threads are guaranteed not to be preempted by any garbage collection activities [151].

RTSJ 1.1 support application execution in multiprocessor environments by depending on the support provided from the underlying operating system [153]. The execution of RealtimeThreads in a multiprocessor environment is supported using affinity sets; "processor affinity is common across operating systems and has become the accepted way to specify the constraints on which processor a thread can execute." [153, p.12].

A processor affinity set is the set of processors that a Java thread or RTSJ schedulable object can be executed on. The representation of the AffinitySet class can be communicated with using a BitSet, where each bit refers to a logical processor ID. An RTSJ implementation should define the relation between the logical processors and the physical processors in the underlying hardware architecture.

The RTSJ supports priority ceiling emulation and priority inheritance;

setting the ceiling of a shared object to the highest priority in the system results in synchronised code executing non-preemptively.

3.1.2 Memory Areas and Memory management

In general, real-time systems have limited memory resources due to different constraints like size and power requirements [151]; which requires effective management of the memory. Standard Java provides heap memory, which is a single space for all memory allocations.

From real-time perspectives the heap memory has a major limitation. All objects created by the threads will be allocated within the heap; at a certain point, the heap will become full. In such case, an allocation failure will trigger the garbage collection process, which, when started, will halt all currently in progress threads with no expectation of the time required to finish the garbage collection process.

To avoid halting the application during the garbage collection process, incremental garbage, and concurrent collection techniques have been introduced [130]. In incremental garbage collector an incremental step is performed at every memory allocation operation. Based on these increments the garbage collection will be done at different phases and the application will only halt during these phases.

The concurrent garbage collector will perform the garbage collection concurrently with the programs being executed, however, small halts might occur from time to time; “Concurrent garbage collectors run concurrently with the application and only stop it for a short synchronization phase in the

beginning or end of the collection" [136, p. 1]. The garbage collection process will reclaim any memory space allocated for object(s) that are not accessible any more; this should provide free memory space to allocate new objects to be created. In certain cases, the garbage collector fails to free enough memory space; in such cases the heap will be expanded. During the expansion process all threads will be suspended. The garbage collection process introduces pauses in the program execution, which affects the time predictability of all running threads. This pauses might compromise the real-time properties of the threads. In addition, the memory space is allocated and freed randomly, hence, the time required to allocate any object in the memory may not be fixed.

RTSJ introduces new memory areas to facilitate developing applications that are not dependent on the standard Java heap. The new `MemoryArea` class enables better management of the available memory space. Except for Heap, instances of the `MemoryArea` class are not subject to garbage collection. `MemoryArea` is an abstract base class, there are four classes that extend the `MemoryArea` class: `ImmortalMemory`, `ScopedMemory`, `ImmortalPhysicalMemory` and `Heap`. RTSJ provides strict rules regarding the allocation and referencing between these memory areas, and each memory area has it's own characteristics.

`ImmortalMemory` is a memory area that can be used by all the schedulable objects "The schedulable objects required by this specification are defined by the classes RealtimeThread, NoHeapRealtimeThread, and AsyncEventHandler. Each of these is assigned processor resources according to their release characteristics, execution eligibility, and processing group values. Any

subclass of these objects or any class implementing the Schedulable interface are schedulable objects and behave as these required classes" [16, p. 37]. in the application, the lifetime of any object created in the `ImmortalMemory` is the same as the application. `ImmortalMemory` is a memory space allocated outside the standard Java heap, which makes it safe for NoHeapRealtimeThreads accesses.

`ScopedMemory` is an abstract class and is used when it is required to perform operations on objects that have limited lifetimes. All objects within the `ScopedMemory` instances will be freed when the reference count of the memory area is zero. Like `ImmortalMemory`, `ScopedMemory` exists outside the standard Java heap, hence, it is available for NoHeapRealtimeThreads, as it is not subject to garbage collection.

`LTMemory`, `LTPhysicalMemory`, `VTMemory` and `VTPhysicalMemory` extends the `ScopedMemory` class. In `LTMemory` the memory allocation time is guaranteed to be linear as long as the consumed space is less than the *initial* size of the memory area, while in `VTMemory` the memory allocation time is not linear. For `LTMemory` and `VTMemory` the system does not guarantee the availability of the memory space if the memory area to be expanded beyond the *initial* size.

A schedulable object can use the `enter` or `executeInArea` methods to access a memory area. For each schedulable object there is a *scope stack* to keep track of which memory areas had been entered. RTSJ enforces strict rules regarding `enter` or `executeInArea` and referencing objects across memory areas. These rules ensures proper execution of the threads in a safe environment.

The class definitions for the MemoryArea and and ScopedMemory are shown in listings 3.1 and 3.2, respectively [152]. Listing 3.3 shows a simple example of a RealtimeThread entering a ScopedMemory area and executing a runnable object inside the ScopedMemory area.

Listing 3.1: MemoryArea class definition

```
1  public abstract class MemoryArea {
2    // constructors
3    protected MemoryArea(long sizeInBytes);
4    protected MemoryArea(long sizeInBytes, Runnable logic);
5
6    // methods
7    public void enter();
8      // Associate this memory area to the current schedulable
          object for the duration of the logic.run method
          passed as a parameter to the constructor
9    public void enter(Runnable logic);
10     // Associate this memory area to the current
          schedulable object for the duration of the logic.run
          method passed as a parameter
11 public static MemoryArea getMemoryArea(Object object);
12     // Get the memory area associated with the object
13   public long memoryConsumed();
14     // Returns the number of bytes consumed
15   public long memoryRemaining();
16     // Returns the number of bytes remaining
17   public long size();
18     // Returns the current size of the memory are
19 }
```

Listing 3.2: ScopedMemory class definition

```
1  public abstract class ScopedMemory extends MemoryArea  {
2    // constructors
3    public ScopedMemory(long size);
4    public ScopedMemory(long size, Runnable logic);
5
6    // methods
7    public void enter();
8    public void enter(Runnable logic);
9    public long getMaximumSize()
10 }
```

Listing 3.3: Example of a realtime thread accessing a ScopedMemory area

```
1  public class rtThread extends RealtimeThread {
2          public rtThread(PriorityParameters pri,
               PeriodicParameters per) {
3                  super(pri, per);
4          }
5          public void run(){
6            while(true){
7                  boolean throwAway = waitForNextPeriod();
8                  final LTMemory smArea = new
                      LTMemory(32*1024);
9                  smArea.enter(new Runnable() {
10                         public void run(){
11                                 LTMemory myMem = (LTMemory);
12                                 RealtimeThread.getCurrentMemoryArea();
13                                 String st = "In␣a␣SM␣area";
14                                 myMem.setPortal(st);
15                         }
16                 });
17           }
18         }
19 }
```

3.2 RTSJ Wait-Free queues

NoHeapRealtimeThread instances are never allowed to allocate or reference any object allocated in the heap and standard Java threads cannot enter a ScopedMemory area. The constructors of NoHeapRealtimeThread require a reference to ScopedMemory or ImmortalMemory, hence, all memory allocation performed in the given area. If NoHeapRealtimeThread is required to synchronise on a shared object with RealtimeThread or a standard Java thread, it is possible that the NoHeapRealtimeThread might be blocked. Even with the presence of priority inheritance or priority inversion avoidance algorithms, it might be possible for the NoHeapRealtimeThread to be blocked by the garbage collector, as the RealtimeThread and standard Java thread reference the heap, which is subject to garbage collection. Hence, "Instances of the NoHeapRealtimeThread class have an implicit execution eligibility that must be logically higher than any garbage collector" [16, p. 22].

In RTSJ, `WaitFreeWriteQueue` and `WaitFreeReadQueue` facilitate the communication between a NoHeapRealtimeThread from one side and a RealtimeThread or a standard Java threads from the other side. This eliminates any possibility that the NoHeapRealtimeThread is preempted by the garbage collector. These queues are bounded in size and support First-In-First-Out (FIFO) semantics.

The `WaitFreeWriteQueue` is a single writer multiple reader queue. The writer is an instance of a NoHeapRealtimeThread and the readers are standard Java threads or heap-using schedulable objects. The NoHeapRealtimeThread will, using the `write` method, add new data items to the queue; if the queue

is full, the `write` method will return `false`. The `read` method will try to remove the oldest data item in the queue, if any, otherwise the `read` method blocks until a data item is available for reading.

The `WaitFreeReadQueue` is a single reader multiple writer queue. The reader is an instance of a NoHeapRealtimeThread and the writers are standard Java threads or heap-using schedulable objects. Using the `write` method new data items can be added to queue. If the queue is full, the `write` method blocks and the thread synchronises with other writer threads (that are currently blocked) until an empty location in the queue is available for writing. The `write` method is synchronised. The `read` method is used to remove the oldest data item from the queue. As there is only one reader, the method is not synchronised. The `read` method does not block, it returns `null` when the queue is empty.

Both queues are fixed in size, determined at the initialisation of the queue [16]. From the NHRT thread side, the queues guarantee wait-free access. Schedulable objects, standard Java and heap-using RT threads accessing these queues might block while reading or writing, due to different reasons like preemption by the garbage collector. In the case that more than NHRT thread is required to access the queue, "it has to supply its own synchronization protocol between the readers to ensure only one of them attempts access at any point in time" [153].

The programmer should ensure such blocking will not affect the application requirements by allowing enough space in the queue, as a write operation, could return a false to the calling NHRT thread, indicating a full queue. In such case, the programmer should design a mechanism to avoid discarding the

data item if this violates the application requirements; for example applications that require every produced data item must be consumed.

In JamaicaVM, aicas RTSJ implementation [47], the `WaitFreeReadQueue` and `WaitFreeWriteQueue` are implemented as described in the RTSJ specifications only for compatibility. JamaicaVM argues that its real-time garbage collector provides the necessary determinism to execute standard Java threads, thus programmers do not need these queues for communication between the standard Java threads and the RTSJ real-time threads. IBM RTSJ implementation, WebSphere Real Time [74] and Sun RTSJ implementation, Sun Java Real-Time System [138] both implement real-time garbage collectors. In both implementation nothing is mentioned on how the `WaitFreeReadQueue` and `WaitFreeWriteQueue` are implemented.

Wait free queues in RTSJ is not intended to support general communication between threads; "It is important to stress that this motivation is slightly different from the usual motivation for "waitfree" queues given in the real-time literature" [153, p.17]. On the other side, the requirements of the wait free queue that is being developed here is to support general concurrent access for shared data structure with wait free guarantees.

3.3 Non-blocking implementations for RTSJ environments

Generally, non-blocking implementations introduced for the RTSJ environment are seeking priority-inversion avoidance, performance advantages, and imple-

mentations that are free from lock-based problems like deadlocks [59, 146, 94]. There are different algorithms introduced to support non-blocking communication within the RTSJ environment. Each of these algorithms addresses certain requirements of the concurrent access to shared data structure.

A wait-free algorithm is proposed for RTSJ in [146]; the algorithm implemented offers wait free guarantees while allowing multiple real-time threads to access the shared queue. The algorithm implements the "enabled late-write" scenario, where higher priority threads can help lower priority preempted threads to finish their enqueue or dequeue operation; which ensures that the interference resulted from higher priority threads will have minimal effect on the lower priority threads.

In [94] a non-blocking implementation of atomic regions, preemptible atomic regions is proposed, which depends on using lightweight transactions to facilitate concurrent access to shared data structures. Accordingly, the overhead due to unsuccessful attempts will be minimum. The implementation provide stronger correctness guarantees compared to lock-based.

Both implementations in [94, 146] are designed for a single processor environment, where a single thread is running at any specific time. In a multiprocessor environment (or multi-core processors) where it is possible for the execution of multiple threads accessing the shared structure concurrently to interleave, both implementations provide no guarantees for the data structure contents consistency or atomic regions execution correctness [18]. Currently, the trend is toward the multi-core and multiprocessor environments and in the next RTSJ version support for such environments are provided; hence, the requirements is to introduce an algorithm that is able to utilise the features

that RTSJ provides and safely execute a multi-threaded application in a multi-processor environment.

In [13] a wait-free communication implementation is proposed, Wait-Free Pair Transactions (WFPT). The implementation offers multiple communication channels between the reader and writer threads. A writer/reader thread can only write to/read from one WFPT, which is the same for the reader thread. Basically, each node in the data structure is a Simpson's four-slot node [131].

In WFPT, the writer thread can write to or read from a specific node in the data structure. The writes will be visible to the associated reader only when committed. The reader thread associated with the same WFPT can also perform read and write operations; however, none of the reader thread writes will be visible for the writer thread.

The implementation is not a general communication data structure between reader and writer threads; as, each writer and reader threads is associated with only one WFPT. Besides that, a faster writer might result in writes that have not been processed by the reader, which is not acceptable in certain applications, like packets in a network router. Moreover, the reader might perform a read operation for an outdated data, if the writer had not performed the commit operation yet.

3.4 Summary

RTSJ introduces different features, building over the standard Java features, which enables developing realtime applications and guarantees the application

safety in the execution environment.

RTSJ introduces wait free queues that facilitate the communication between NoHeapRealtimeThread from one side and RealtimeThread and standard Java threads from the other side; in order to avoid garbage collection influence on NoHeapRealtimeThread. In the RTSJ wait free queues from the NoHeapRealtimeThread side of the queue only one thread is allowed to access the queue, and from the other side more than a standard Java and RealtimeThread threads are allowed to access the queue. This is for time predictability concerns, as the NoHeapRealtimeThread all the time will be able to have direct access to the queue without any possible collision. However, these queues are designed such that it is not guaranteed for the NoHeapRealtimeThread to have an empty queue in the case of enqueueing operation, and the programmer should provide the mechanism to handle such case, like discard the current data item or overwrite the most recent data item; according to the application requirements. In the case of more than a NoHeapRealtimeThread are required to access the queue, the programmer should design the required mechanism to ensure that the operations of the NoHeapRealtimeThread threads are not interfering with each other.

Hence, those queues are not wait free by definition, and they are not intended for general purpose communication among the threads in a multi-threaded application.

There had been different algorithms proposed for threads communications in a multi-threaded application within the RTSJ environment, however, to the best of our knowledge, none of these algorithms are designed for the multiprocessor environment.

Our algorithm is designed to support safe execution of a multi-threaded application in the multiprocessor environment with wait free guarantees.

Chapter 4

Multiprocessor Wait-Free Queues for RTSJ

In general, shared data structure implementations are either designed for general purpose communication among the application threads or to meet certain application requirements, which makes them for specific purposes only. From a real-time perspective, in any shared data structure implementation the worst-case execution time of its access and its memory requirements are the key factors of consideration.

This chapter introduces a new queue based wait-free algorithm designed for multiprocessor systems within the RTSJ environment. Section 4.2 introduces the pseudo code of the algorithm. Section 4.1 provides a discussion of the computational model and assumptions of the algorithm. In section 4.3 the overall approach is described and in 4.4 a discussion of the different issues related to the design of the algorithm is given. The timing analysis of the algorithm is discussed in 4.5. The implemented algorithm is presented in the

next Chapter.

4.1 Computational model and assumptions

Generally, shared data structures are designed to facilitate communication between threads in a multi-threaded application; each implementation makes some assumption and is designed to satisfy certain applications' requirements. For example, Simpson's four slot algorithm provides wait-free guarantees but limits the number of reader and writer threads to one each. The algorithm presented in this thesis assumes the following:

1. all processors are homogeneous (that is the platform supports SMP)
2. all applications thread are either periodic or sporadic (with a minimum inter-arrival rate)
3. global scheduling is supported between all cores hosting the RTSJ application
4. communication between threads is either via atomic shared variables or through wait-free queues
5. with respect to wait-free queues, a thread is either a producer of data (writer) or a consumer (reader) for the same wait-free queue; threads, however, can access multiple wait-free queues
6. for each wait-free queue, a fixed number of data items is produced or consumed by each thread during a single release; without loss of generality, we assume this to be 1 in this thesis

7. all data produced must be consumed; consumers may find a queue empty but producers must never find a queue full

8. the act of producing and consuming data is application dependent and can be time consuming

9. the `java.util.concurrent.atomic` package is available and multi-processor safe.

10. the number of concurrent threads accessing the data structure should be pre-known and fixed at run time.

In order to meet assumption 7, it is necessary to impose some constraints of the application. The algorithm assumes that the average production rate of the producer threads is equal to the average consumption rate of the consumer threads, as shown in equation 4.1. To guarantee this, the algorithm assumes that the reader and writer threads concurrently accessing the data structure are periodic or sporadic executing at their minimum inter arrival rates. There is no limitation on the maximum number of threads allowed to access a wait-free queue, as long as the production and consumptions rates are equal; hence, the algorithm is a general purpose communication channel in a multi-threaded application, however, the number of these threads should be known in advance, as the queue is sized based on these numbers (see section 4.4.1).

$$\sum_{\forall i \epsilon readers} \frac{1}{T_i} = \sum_{\forall j \epsilon writers} \frac{1}{T_j} \tag{4.1}$$

In order to avoid garbage collection overheads, data shared between real-time threads must be stored in scoped or immortal memory. As immortal memory is never cleared, non persistent data must be stored in scoped memory.

The scoped memory areas are embedded in the queue structure to ensure that the number of scoped memory areas are initialised during the data structure initialisation stage and the access to these areas complies with the RTSJ memory assignment rules.

The queue size, discussed in detail later, should be known in advance for predictability purposes; hence, the last assumption. The threads concurrently accessing the queue, for either enqueue or dequeue, should find an empty node from the first time while searching the queue. This ensures that the overall time required to perform the enqueue or dequeue operations is predictable; this is discussed in detail later in this chapter.

4.2 Algorithm pseudo code

The algorithm is an array based queue shared data structure. The data structure supports two operations: enqueue and dequeue. The writer thread performing an enqueue operation: reserves a node in the `scopedMemoryPool`, produces the data, and add the index of the node to an integer array. The reader thread performing the dequeue operation: reads a node index from the integer array, consumes the data from that node, and updates its status to empty. The algorithm pseudo code is provided in listing 4.1.

Listing 4.1: "Algorithm pseudo code"

```
public WaitFreeQueue (int numberOfReaders , int numberOfWriters , long scopeSize ){
    INTEGER queueSize = 2 * ( numberOfReaders + numberOfWriters ) + 1;
    scopedMemoryPool = new QueueNode [queueSize];
    AtomicInteger readList = new AtomicInteger [queueSize];
    readListInvalid = new int[queueSize];
    for int i=0 to queueSize
        scopedMemoryPool [i] = new QueueNode (scopeSize);
        readList [i] = new AtomicInteger (-1);
        readListInvalid [i] = -1;
}

public boolean enqueue ( Runnable rObject) {
INTEGER reservedNode = -1;
INTEGER enqueueLocation ;
INTEGER currentLocation ;
BOOLEAN dataStrucUpdated = false ;
INTEGER inV;

    set priority to maximum
        do {
            reservedNode = reservedNode + 1
            if (reservedNode == queueSize)
```

```
        return "full queue"
    } until (update reserved of scopedMemoryPool [reservedNode] to true );
set priority to default
    produce the data in the corresponding scoped memory backing store
set priority to maximum
do {
    currentLocation = readListHead
    enqueueLocation = (currentLocation + 1) (MOD) queueSize
    inV = readListInvalid [ enqueueLocation ]
    if ( succeed to update readListHead with enqueueLocation )
        readList [ enqueueLocation ] <- reservedNode
} until (succeed to update the readListHead)

set priority to default
return true ;
}

public boolean dequeue ( Runnable rObject) {
INTEGER dequeueLocation ;
INTEGER currentTail ;
INTEGER index ;
INTEGER value ;
BOOLEAN test = false ;
```

```
do {
    currentTail <- get readListTail value
    deqLocation <- current tail + 1 (MOD) queueSize
    if  readList(deqLocation) == -1
        return  "empty queue, no data ready for consumption."

} until (succeed to update readListTail with deqLocation)
    get the index of the node in readList(deqLocation) to consume.
    set the value of  readList(deqLocation) to -1
return to normal priority
    consume the data in the corresponding scoped Memory backing store
set priority to maximum
    set the reserved boolean of the corresponding node to false
set priority to default
return true
}
```

4.3 Overall Approach

The algorithm uses two arrays and several atomic variables needed to ensure that threads' accesses to the data structures are safe, as shown in figure 4.1. The first array holds the actual data that is produced and consumed; it is called the `scopedMemoryPool` as effectively it contains a pool of scoped memory areas. Data can be inserted and removed from any position in this pool by the algorithm. The second array is a more traditional FIFO integer queue. It is called the `readList`. Each node in the `scopedMemoryPool` contains a boolean variable and a `scopedMemory` reference. The boolean variable indicates that the node is free to be accessed for writing or holds data to be consumed. The `scopedMemory` reference is to a `scopedMemory` object where the actual data is produced and consumed. The `readList` holds the list of nodes that contain data to be consumed. Atomic head and tail variables are used to enable the reader and writer threads to access the queue. The writer threads enqueue the position of produced data in the `scopedMemoryPool` at the head of the `readList` and the reader threads dequeue the data's position from the tail of the queue.

A key property of the algorithm is that writers always search for an empty node in the `scopedMemoryPool` starting from the first element. It is guaranteed to find an unreserved node by the time it gets to the end of the array (see Section 4.4.1).

The writer and reader threads accessing the data structure perform enqueue (or write) and dequeue (or read) operations, respectively. The write and read operations are divided into two parts: the part that requires interaction with

the data structure and the part that does not require interaction with the data structure. During the part that does not require interaction with the data structure (i.e. actual data production and consumption) the progress of a thread is not affected by the progress of any other threads concurrently accessing the data structure.

On the other hand, the part that requires interaction with the data structure, like reserving a node for writing or reading and updating the data structure, the progress of the thread might be affected by the progress of other threads concurrently accessing the data structure. For example, a thread performing a write operation might fail to reserve a node in the queue due to another thread concurrently running on another processor (or core) that has succeeded in reserving the same node. Hence, during node reservation in the `scopedMemoryPool` and the updating of the `readList`, the threads run non-preemptively. This ensures that there is no interference of threads concurrently accessing the data structure that are scheduled on the same processor in the system. This allows a simple bounding of the worst-case access times, as for a writer thread to reserve a node, in the worst case all processors (or cores) are occupied by writer threads competing together to reserve an empty node. This is a race condition that cannot be avoided. The unlucky writer is going to retry a number of times equal to the number of processors (or cores) - 1; assuming it looses the race to all other writers concurrently scheduled on the other processors. In the case that the number of writer threads is greater than the number of processors (or cores) in the system, if any writer thread is preempted just after finishing the node reservation step, the new writer is not going to affect the current competition

between the writers, as our assumption is that all processors executes at the same speed. Hence, the new writer is going to search the `scopedMemoryPool` starting from the first node, while the competing writers are ahead in the `scopedMemoryPool`, as shown in figure 4.2.

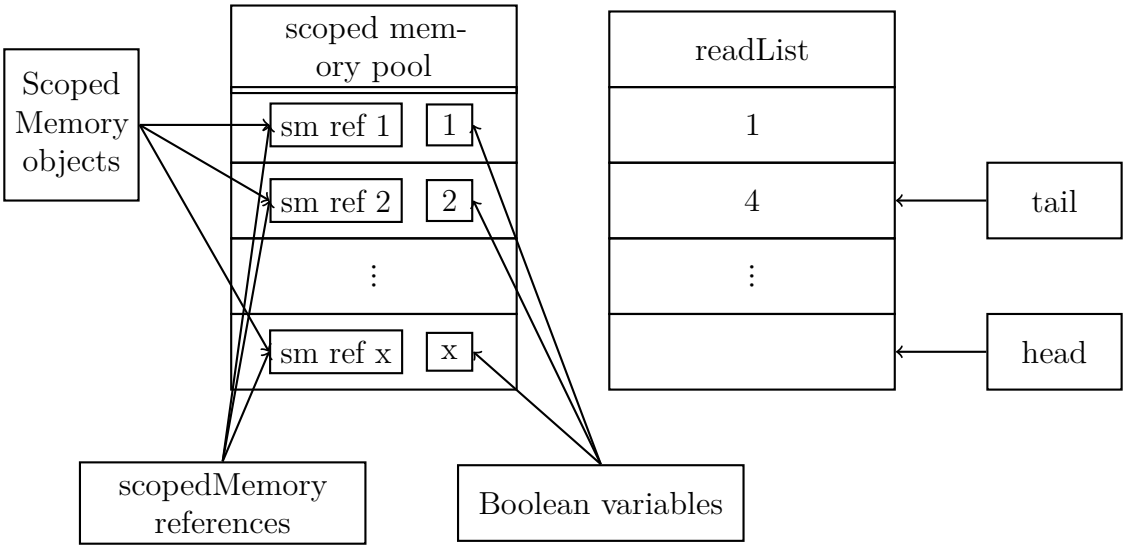


Figure 4.1: Block diagram of the data structure

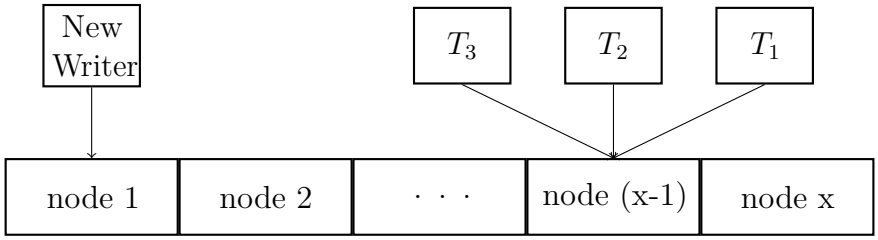


Figure 4.2: New writer case

4.4 Design issues

It is a complex task to design an algorithm that can provide wait-free guarantees; especially if the algorithm is supposed to provide general purpose

communication for a multi-threaded application on a multi-processor (or multi-core) system. Thus, the algorithm is utilizing all the features of the RTSJ to ensure that it meets the design requirements.

4.4.1 Data structure size

In general, in applications where the average consumption rate of the reader threads is equal to the average production rate of writer threads [1] the data structure size can be bounded. This ensures that all produced data items are enqueued and processed. The algorithm consequently targets applications that have production rate of the writer threads that is equal to the production rate of the reader threads, equation 4.2.

Assuming that the number of readers is given by ($n_{readers}$) with different release parameters, and similarly for the number of writer threads ($m_{writers}$). For each thread τ_i there is a period T_i and a deadline D_i.

$$\sum_{\forall i \epsilon readers} \frac{1}{T_i} = \sum_{\forall j \epsilon writers} \frac{1}{T_j} \tag{4.2}$$

The worst case occurs, when the reader threads are scheduled to run at the start of their periods at a time when there is no data in the queue to consume. In the subsequent periods, the readers are scheduled to complete just before their deadlines. At the same time, the writer threads are scheduled to complete just before their deadlines in their first periods, and early at their

[1] If the average consumption rate for the reader threads is larger there will be no problem. If the average production rate for the writer threads is larger, then at a certain moment, the writer threads will not be able to enqueue any produced data, as the queue will be full. In this case, the programmer should provide a proper mechanism to handle un-enqueued data items if the application requires that all produced data items should be consumed.

following periods, as shown in figure 4.3.

We assume the time required to execute the queuing and dequeuing protocols are negligible compared to the time required by the thread to produce or consume the data in any of the data structure nodes. Consequently, we are interested in finding the largest distance that the writers can be ahead of the readers, which can be represented as follows, where t is some point in time:

$$\sum_{\forall i \epsilon writers} \left(\lfloor \frac{t + D_i}{T_i} \rfloor + 1 \right) - \sum_{\forall j \epsilon readers} max \left(\lfloor \frac{t - D_j}{T_j} \rfloor, 0 \right) \tag{4.3}$$

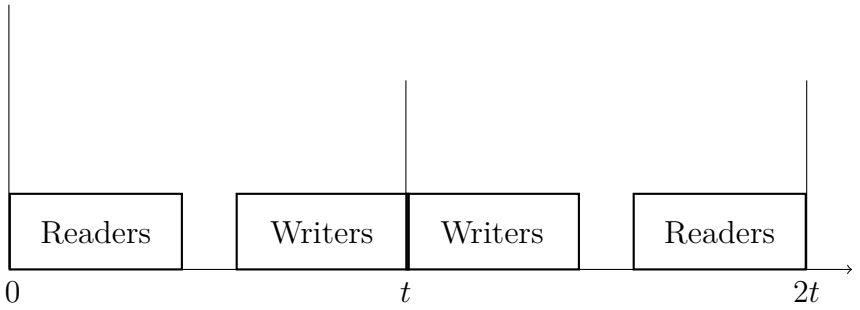


Figure 4.3: Worst case scenario of writers and readers execution

Equation 4.3 is composed of two parts, the first is how much the writers can be ahead of the average rate of production, 4.4, and the second is how much the readers are behind the average rate of consumption, 4.5. The first

part can be rewritten as:

$$\begin{aligned} & \sum_{\forall i \epsilon writers} \left(\lfloor \frac{t + D_i}{T_i} \rfloor + 1 \right) - \sum_{\forall i \epsilon writers} \frac{t}{T_i} \\ & \leq \sum_{\forall i \epsilon writers} \left(\frac{t}{T_i} + \frac{D_i}{T_i} + 1 \right) - \sum_{\forall i \epsilon writers} \frac{t}{T_i} \\ & = n_{writers} + \sum_{\forall i \epsilon writers} \frac{D_i}{T_i} \end{aligned} \tag{4.4}$$

We assume that for the writers the worst-case situation is all writers produced just before their deadline and then immediately on their next release. Hence to represent this scenario we must add D_i to t to ensure that we get the correct number of releases. The +1 is because in an interval of any arbitrary small length, we can have at least one release.

For readers, we assume that the worst-case situation is all readers consume immediate at their release time and then at their deadline on their next release. Hence to represent this scenario we must subtract D_j to t to ensure that we get the correct number of releases. The "-1" appears because we are making the second summation term as small as possible, and so we substitute

$x - 1$ for $\lfloor x \rfloor$ since $\lfloor x \rfloor >= x$ -1. Hence the second part becomes

$$\begin{aligned} & \sum_{\forall j \epsilon readers} \frac{t}{T_j} - \sum_{\forall j \epsilon readers} max\left(\lfloor \frac{t + D_j}{T_j} \rfloor, 0\right) \\ & \leq \sum_{\forall j \epsilon readers} \frac{t}{T_j} - \sum_{\forall j \epsilon readers} \left(\frac{t}{T_j} + \frac{D_j}{T_j} - 1\right) \\ & = n_{readers} + \sum_{\forall j \epsilon readers} \frac{D_j}{T_j} \end{aligned} \tag{4.5}$$

Inserting into Equation 4.3 and simplifying gives the max distance between the writers and the readers is upper bounded by

$$n_{writers} + \sum_{\forall i \epsilon writers} \frac{D_i}{T_i} + n_{readers} + \sum_{\forall j \epsilon readers} \frac{D_j}{T_j} \tag{4.6}$$

Given the threads' deadlines and periods the buffer size can be calculated, which might not be an integer value. But as it is a strict upper bound, the maximum buffer size is given by:

$$n_{writers} + n_{readers} + \lceil \sum_{\forall i \epsilon writers} \frac{D_i}{T_i} + \sum_{\forall j \epsilon readers} \frac{D_j}{T_j} \rceil \tag{4.7}$$

assuming implicit deadlines (i.e. $D_i = T_i$) then the required buffer size is given by:

$$2\left(n_{readers} + n_{writers}\right) \tag{4.8}$$

According to equation 4.8, it is guaranteed that at any moment of time t, there is any empty node in the queue for a writer thread performing an enqueue operation.

However, a key property of the algorithm is that writers always search for an empty node in the `scopedMemoryPool` starting from the first element. It must therefore be guaranteed to find an unreserved node by the time it gets to the end of the array it is possible for the empty node to become available for writing after the writer thread passed the location of that node, as shown in figure 4.4.

We can apply a similar argument to that above to determine what impact this has on the required queue size. Consider an arbitrary point in time, t when a writer starts searching the `scopedMemoryPool`. The maximum number of items in the buffer must be less than:

$$\sum_{\forall i \epsilon writers} \left(\lfloor \frac{t + D_i}{T_i} \rfloor + 1 \right) - \sum_{\forall j \epsilon readers} max \left(\lfloor \frac{t - D_j}{T_j} \rfloor, 0 \right) \tag{4.9}$$

Hence, the number of items in the buffer must be

$$\begin{aligned} &\leq \sum_{\forall i \epsilon writers} \left(\frac{t}{T_i} + \frac{D_i}{T_i} + 1 \right) - \\ &\quad \sum_{\forall j \epsilon readers} \left(\frac{t}{T_j} - \frac{D_j}{T_j} - 1 \right) \end{aligned} \tag{4.10}$$

If $D = T$

$$2n_{writers} + 2n_{readers} + \sum_{\forall i \epsilon writers} \frac{t}{T_i} + \sum_{\forall j \epsilon readers} \frac{t}{T_j} \tag{4.11}$$

Now as the average production and consumption rates are equal:

$$\sum_{\forall i \epsilon writers} \frac{t}{T_i} = \sum_{\forall j \epsilon readers} \frac{t}{T_j} \tag{4.12}$$

Therefore, the number of items in the buffer must be

$$> 2\left(n_{readers} + n_{writers}\right) \tag{4.13}$$

In the worst case, there could be $n_{writers}$ trying to write, therefore to ensure that a writer finds a slot on one pass we must add an extra $n_{writers}$ spaces. Hence, the buffer size must be

$$2n_{readers} + 3n_{writers} \tag{4.14}$$

Given this queue size, the algorithm guarantees that

- The time required for the writer or reader thread to perform the enqueue or dequeue operation is bounded; which is proportional to the number of writer and reader threads concurrently accessing the data structure.
- All produced data is enqueued in the data structure, which ensures that every produced data item is consumed.

The algorithm imposes no limitations on the maximum number of reader or writer threads concurrently accessing the data structure, however, the number of reader and writer threads should be known prior to initializing the data structure. If it is required to change the number of reader or writer

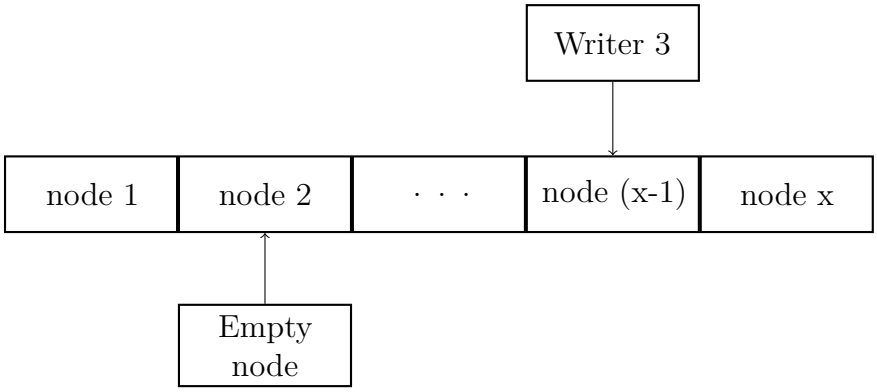


Figure 4.4: Special case scenario

threads accessing the data structure at runtime the data structure should be re-initialized.

It is possible to set the number of reader and writer threads to a certain number that is higher than the initial number of threads accessing the data structure, if it is expected that the number of threads during execution can increase. However, this comes at the expense of the spatial concern. The programmer could set the number of threads accessing the data structure to ensure that the required memory space is reserved during the initialsation of the data structure, during execution time the number of threads can vary without being larger that that number. However, in the case of setting large number of threads, and the real number during the execution did not reach that number there will be a wasted reserved space, that could be used for other purposes in the system.

The number of threads can be set to a number higher than the initial number of threads accessing the data structure, if it is expected that during execution the number of threads could increase; however, this should be governed by many factors like the availability of memory space in the system,

the size of the data item to be written/read, and the difference between the expected number of threads and the initial number is logical.

For example, if the initial number of threads accessing the data structure is 10 and it is expected during run time that it is possible to have 30 threads concurrently accessing the data structure, to set the number of threads at the initialisations of the data structure to 30 and not to 100, for example.

4.4.2 Data structure design

The data structure can be initiated in any memory area, assuming that the free space is sufficient for the application requirements. If the data structure life is expected to be the duration of the application, it is recommended to initiate the data structure in immortal memory. However, objects created in immortal memory cannot be deleted; hence, the immortal memory size is going to be consumed with time. To utilize the new memory areas that RTSJ introduces, the data structure can be divided into two parts; the booking area (the part that facilitate the communication between the threads) and the actual storage units of the produced data. The data structure cannot be initiated in the standard Java heap; as the structure supports no-heap real-time threads.

The booking area, explained in detail below, maintains a record of the current status of each node in the data structure with the necessary variables (pointers) that maintain safe access to the data structure contents. This part can be initiated in Immortal memory, as the amount of memory required for this part is fixed during the application initialization. Figure 4.5 presents the

suggested distribution of the data structure. The booking area is stored in Immortal memory, and a Scoped Memory backing store is associated with each node in the data structure. The produced data is stored in scoped memory area(s).

Associating a scoped memory backing store for each node in the data structure enables better utilization of the available memory and less effort for the programmer to perform the required memory management; as the allocated memory in the scoped memory backing store is reclaimed when the number of references to the backing store is zero.

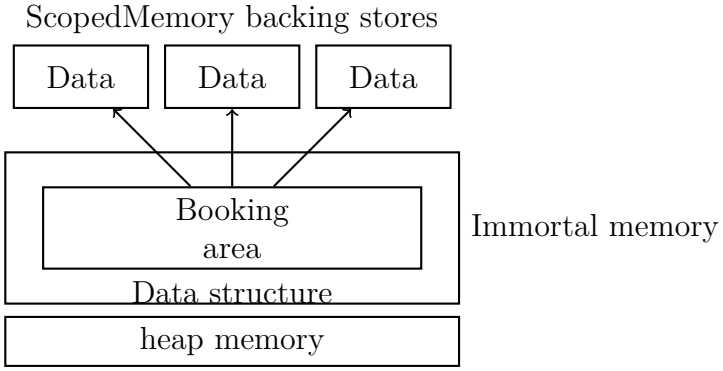


Figure 4.5: Suggested memory allocation of the data structure

To avoid any interference between the reader and writer threads, the booking area is composed of a `scopedMemoryPool` and a Data Position (an integer) Queue; shown in figure 4.6. The writer threads enqueue the data in the data content array; the actual data is stored in the scoped memory backing stores. The `readList` maintains a list of all nodes in the `scopedMemoryPool` that are holding data available for consumption. The reader threads read the indexes from this queue and consume the data in the corresponding nodes of the data content array.

A writer thread searches the data content array starting from the first node trying to find the first available node for writing; accordingly, there is no pointers (head or tail) associated with this array. If more than a writer thread compete on reserving the same node, only one is going to succeed in reserving the node and all the others keep searching forward in the array for an available node. The loosing writer threads do not restart the search from the beginning, every time the writer thread fails in reserving a node (due to another thread competing to reserve the node at the same time and succeed) the writer thread continues with the next node until it succeed in reserving a node.

To provide wait free guarantees, the number of retries that a writer or a reader thread performs when trying to enqueue or dequeue a node should be bounded. Thus, the data structure is sized such that a writer or a reader thread only needs to search once to find a node to produce data in or consume the data it contains.

As discussed earlier, RTSJ reclaims the allocated memory space of a scoped memory backing store when the number of active schedulable objects in that backing store is zero. There is going to be certain amount of time after the writer thread finishes the data production and leaves the memory area and before a reader thread accesses the node for consumption, hence, the memory space allocated for the data is reclaimed and the backing store of the `ScopedMemory` is going to be empty when the reader tries to consume that data. RTSJ 1.1 introduces the pinnable `ScopedMemory` to address this race condition (which is unavoidable in producer consumer systems); according to RTSJ a pinned `ScopedMemory` backing store contents will not be deleted

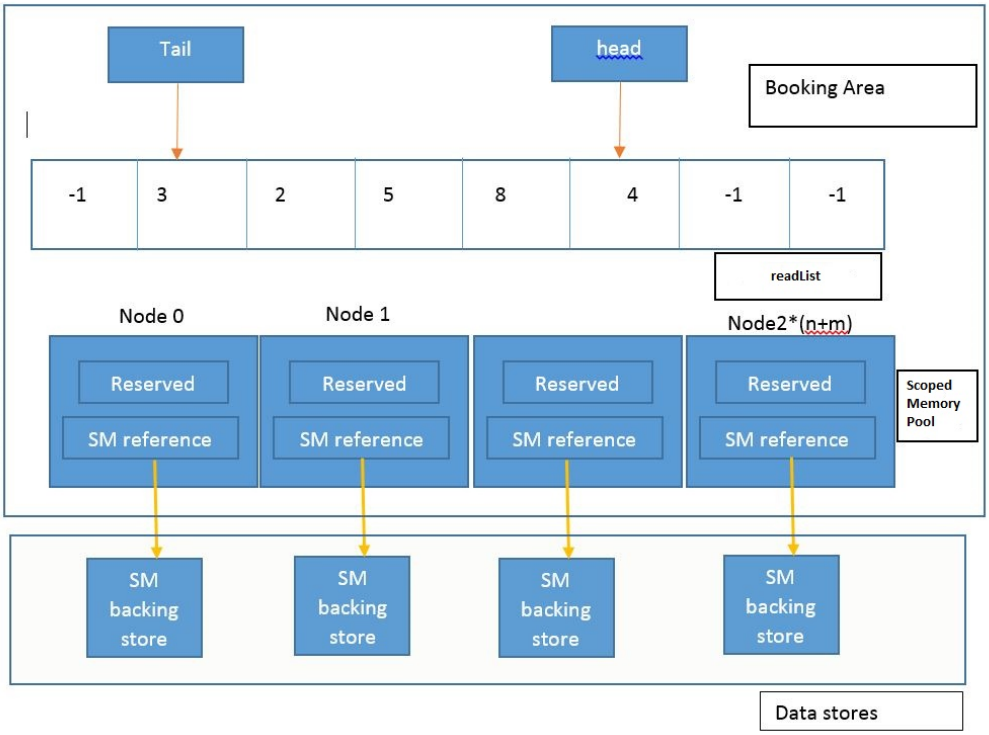


Figure 4.6: Data structure components and suggested memory allocation

until it is unpinned and the active schedulable objects in that memory area is zero. The algorithm presented in this thesis is implemented on JamaicaVM [47]; which is currently based on RTSJ 1.0.2, therefore, until the RTSJ 1.1 implementation is released the pinnable feature is not available. Thus, a simulation of the pinning feature is adopted, in which an additional thread is created in the `ScopedMemory` backing store as part of the data production and the reader thread consuming the data stops that thread (which then terminates); accordingly, the data in the `ScopedMemory` backing store is only deleted after it is consumed.

The memory requirements of the algorithm can be determined by the number of the reader and writer threads concurrently accessing the data

structure, and the amount of memory required to store the data produced by the writer threads, which is application dependent; thus the programmer can determine the overall amount of memory required to implement the algorithm and ensure that the system is able to provide such an amount of memory.

4.5 Time analysis

The algorithm is designed to provide wait free guarantees for a multi-threaded application in a multiprocessor (or multi-core) environment. Threads accessing the data structure performs write and read operation. Accordingly, it is important to ensure that a writer or reader thread execution time is bounded by defining the maximum time that a thread suffers interference from other threads running concurrently. In a multi-processor or a multi-core environment, a thread execution can be interfered by any other thread concurrently accessing the same data structure; either running on the same processor (with higher priority) or in a different processor (regardless of the priority).

In the worst case, a writer thread T_w performing a write operation fails to reserve a node from the `scopedMemoryPool` a number of times equal to the number of writer threads in the application except for itself; all other writer threads are competing at the same time and succeed in reserving nodes before T_w. Moreover, if T_w is unlucky enough T_w fails also in adding the index of the node to the `readList` a number of times equal to the number of writer threads in the application except for itself.

In a multiprocessor system, the writer threads are scheduled in the same processor (or core) or different processors (or cores). To avoid the interference

of the execution of a writer thread T_w from other writer threads scheduled on the same processor (or core) it is required that the execution of T_w is not preempted during the node reservation or adding the index of the node to the `readList`. Accordingly, the parts of the code in the writer thread that performs the node reservation and adding the node index to the `readList` is executed while the priority of the writer thread is set to maximum priority. Consequently, a writer thread only fails to reserve a node if there is another writer thread concurrently competing to reserve the same node and scheduled on another processor or core.

If the writer threads within the application are scheduled to run concurrently, such that each writer thread is scheduled to run on a different processor (or core) in the system, a writer thread T_w at a maximum is going to fail in reserving the node a number of times equal to the number of processors (or cores) in the system.

In the case of the number of writer threads is more than the number of processors (or cores) in the system and all writer threads are scheduled to run concurrently (i.e. there is more than one writer thread scheduled to run on the same processor) the number of retries to reserve a node in the queue is limited to the number of processors (or cores) because searching for an empty node starts from the beginning of the queue; any writer thread searches for an empty node in the pool starting from the first node. For example, if there are three writer threads T_{w1}, T_{w2}, and T_{w3} (and the size of the queue is 8 nodes) and two processors in the system. Writer thread T_{w1} is scheduled on processor P_1 and writer threads T_{w2} and T_{w3} are scheduled on P_2. If T_{w1} and T_{w2} are competing to reserve the first node and T_{w2} succeeds and T_{w1} fails.

T_{w1} is going to try to reserve node 2, while T_{w3}, if it is able to arrive before T_{w1} succeeds is going to start the reservation from node 1.

Also note, context switching is required before a thread actually starts execution; the processor performs a context switching between the thread being executed and the thread to be executed. Relatively, context switching requires longer time compared to the time required to perform different operations on the processer [89]. In the previous example, if T_{w3} is scheduled directly after T_{w2} succeeds, P_2 is required to perform context switching and T_{w3} should be executed to the point that is starts the enqueue method (and starts the competition on the node reservation). Hence, T_{w1} progress is not affected by the arrival of T_{w3}. Hence, the maximum number a writer thread fails is limited to the number of processors.

If we assume that the number of writer threads and the number of reader threads concurrently accessing the data structure are m and n, respectively. Accordingly, the size of the data structure is more than $(3m + 2n)$. The time required for a writer thread to finish an enqueue operation is:

$$T_{en} = T_r + T_p + T_u \tag{4.15}$$

where:

T_r : the time required for the writer thread to reserve a node in the data structure.

T_p : the time required for the writer thread to perform the production of the data in the node.

T_u : the time required for the writer thread to update the data structure.

In the worst case, the writer thread succeeds in reserving the last node in the pool (i.e. all other nodes except the last node are reserved by a writer, a reader is consuming the data, or contains data for consumption), accordingly:

$$T_r = \sum_{i=1}^{(3m+2n)} T_{Rretry} \tag{4.16}$$

where

T_{Rretry}: is the time required to try to reserve a single node in the `scopedMemoryPool`

If all the writer threads are trying to add the index of the node to the `readList` at the same time, then the last writer thread to succeed should try m times, accordingly, T_u is:

$$T_u = \sum_{i=1}^{m} T_{Uretry} \tag{4.17}$$

where

T_{Uretry}: is the time required to try to add the index of the node to the `readList` once.

Accordingly, the time required to perform an enqueue operation can be written as:

$$T_{en} = \sum_{i=1}^{(3m+2n)} T_{Rretry} + T_p + T_u \tag{4.18}$$

The time required to perform the production of the data is application dependent, which should be determined by the application programmer. The time required to reserve a node in the queue is proportional to the number of reader and writer threads concurrently accessing the data structure, while the time required to add the node index to the `readList` is proportional to the number of the writer threads only.

On the other hand, a reader thread accessing the dequeue method only competes with other reader thread when trying to read a node index from the `readList`. The time required for a reader thread to finish the dequeue method is:

$$T_{de} = T_{res} + T_c + T_{up} \tag{4.19}$$

where:

T_{res} : the time required for the reader thread to read a node index from the `readList`

T_c : the time required for the reader thread to consume the data in the node.

T_{up} : the time required for the reader thread to update the data structure.

A reader thread trying to read a node index from the `readList`, in the worst case, will retry a number of times that is equal to the number of reader threads concurrently accessing the data structure (i.e. all other reader threads are trying to read a node index from the `readList` at the same time),

accordingly:

$$T_{res} = \sum_{i=1}^{n} T_{Resretry} \tag{4.20}$$

where

$T_{Resretry}$: is the time required to try to reserve a node index from the `readList` once.

The time required to perform the data consumption is application dependent, which should be determined by the application programmer. The reader thread is not going to face any competition while updating the data structure (setting the `reserved` variable of the corresponding node to false). accordingly, the time required to finish the dequeue operation is:

$$T_{de} = \sum_{i=1}^{n} T_{Resretry} + T_c + T_{up} \tag{4.21}$$

4.6 Summary

The WaitFreeQueue algorithm presented in this thesis is designed to enable communication between threads within a real-time multi-threaded application in a multiprocessor environment. The algorithm provides wait-free guarantees assuming that all threads meet their real-time properties and the average production rate of the writer threads is equal to that of the reader threads.

The algorithm supports periodic and sporadic real-time threads. Allowing aperiodic real-time threads to access the data structure has implications that

compromise the guarantees that the algorithm provides. In particular, it becomes impossible to bound the size of the queue.

The computational and memory requirements of the algorithm can be determined ahead of runtime, the programmer must ensure that there is enough resources in the system for the algorithm. The changes in the application requirements (i.e. the size of the data or the number of reader or writer threads concurrently accessing the data structure) requires re-initializing the data structure to reflect on these changes.

The data structure can be loaded in any memory area, except the standard Java heap memory. Currently, each data item produced is allocated in a separate scoped memory backing store. This ensures that the progress of any writer or reader thread during the data production or consumption is not affected by the progress of any other writer or reader thread in the application. This also helps in better memory management, as the programmer does not need to perform memory management, as the allocated space in the scoped memory backing store is going to be reclaimed automatically after the consumption of the data is finished.

Chapter 5

Wait Free queue algorithm implementation

In this chapter the algorithm implementation in JamaicaVM RTSJ is presented. The algorithm is a non-blocking wait-free queue within the RTSJ environment that is targeting the multiprocessor environment.

5.1 Using the algorithm

The algorithm uses the scoped memory backing store as the memory space to produce and consume the data. Accordingly, the algorithm supports all types of applications that require a shared data structure, as it is possible to store any data, regardless of the type or size.

Using the algorithm requires developing writer and reader threads. The data production and consumption processes are performed using a runnable object, which is passed as a parameter when accessing the enqueue and

dequeue methods, respectively. The runnable objects is designed according to the application requirements.

As discussed earlier, the current version of the RTSJ does not implement the pinnable scoped memory area. Thus, the PinThread, presented in listing D.8, simulates the pinnable scoped memory area. The algorithm attaches the PinThread instantiated in the scoped memory backing store to the scoped memory portal. PinData is a real-time thread that is put to sleep long enough such that the data produced and stored in the scoped memory backing store is consumed. The produceData and consumeData methods are used to store and retrieve the data.

Listing B.2 presents a writer thread sample. The presented writer thread accesses the enqueue method to perform the data production. The programmer should develop the writer thread such that it reflects the application requirements. The listing in B.1 presents a reader thread sample. The reader thread access the dequeue method to perform the data consumption.

The ruannable objects to perform the data production and data consumption should be developed by the programmer according to the application requirements. The writer and reader threads access the enqueue and dequeue methods once every period, this enables the programmer to better estimate the data production and consumption rates. However, if the application requires that the writer or reader thread should perform the data production or consumption more than once every period, the programmer should be able to estimate the production rates accordingly.

In a multi-threaded application, the shared data structure is used to enable the threads within the application to communicate with each other. However,

the shared data structure is only one building block of the multi-threaded application, the application might have many other operations to perform.

5.2 Wait-free queue implementation

This section introduces the implementation of the algorithm on aicas RTSJ, JamaicaVM [47]. Figure 5.1 presents a class diagram of the wait free algorithm, a full listing of the algorithm implementation is provided in appendix D.2. The class diagram shows four classes: QueueNode, WriterThread, ReaderThread and WaitFreeQueue. The writer thread performs the enqueue operation and the reader thread performs the dequeue operation; the Runnable object is passed to the corresponding method to perform the designed operation.

The QueueNode contains an atomicBoolean variable, `reserved` and a Scoped Memory object. The `reserved` flag used to indicate that the node is empty, `reserved`=false, or the node have data to be consumed, reserved=`true`, it is `false` by default. The Scoped Memory objet, `sMemory`, holds a reference to the Scoped Memory backing store.

The WaitFreeQueue class contains the enqueue and dequeue methods, `scopeMemoryPool` and `readList` arrays, and `readListHead` and `readListTail` AtomicVariables. `scopeMemoryPool` is an array of the QueueNode class, and the `readList` is an integer array.

5.2.1 The queue node

The QueueNode class is the basic component of the data structure, shown in listing 5.1. The `reserved` AtomicInteger is `false` by default; indicating

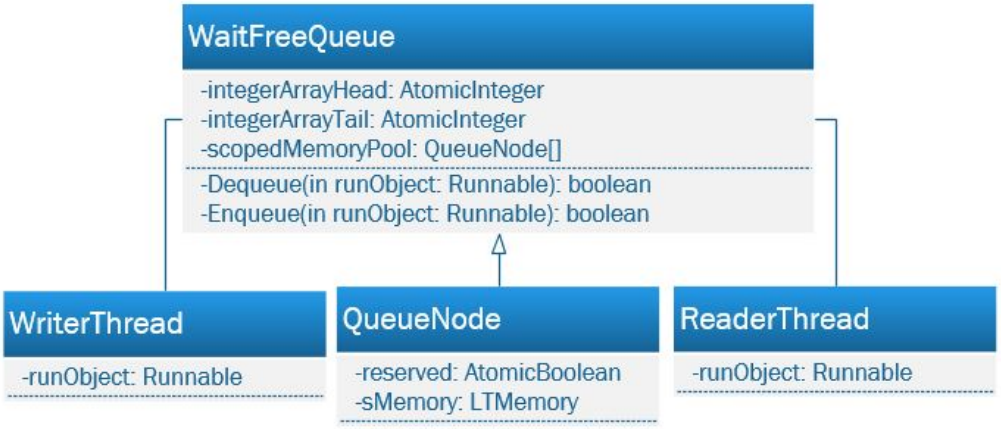


Figure 5.1: Class diagram of the wait free algorithm

that the node is available for writing. The writer threads trying to perform an enqueue operation might compete on reserving the same node. The first to set `reserved` to `true` reserves the node, and all the other writer threads will try with the next node, and so on. When a reader thread consumes data from a node in the queue, it sets the associated *reserved* AtomicInteger to `false` to indicate that the data has been consumed and the node is available for the writer threads. The writer and reader threads produce and consume the data on/from the backing store associated with that node of the queue.

The `sMemory` is a `ScopedMemory` object; the object holds a reference to the scoped memory backing store, where the data is produced and consumed. The size of the scoped memory backing store is application dependent, thus, the programmer should set the required amount of memory during the initialisation process of the data structure.

At runtime, any node in the data structure might be in one of four states, as shown in figure 5.2:

- **Empty**: initially, all nodes in the data structure ar empty and available.
- **Data being processed**: a writer thread reserved the node and currently is producing the data in the node.
- **Data available**: The writer thread finished the data production and the node is available for the reader threads.
- **Data being consumed**: a reader thread reserved the node and currently consuming the data, when the reader thread finishes the data consumption process, the node is set back to empty.

Listing 5.1: QueueNode class

```
1  import java.util.concurrent.atomic.*;
2  import javax.realtime.*;
3  import javax.realtime.ScopedMemory.*;
4  public class QueueNode {
5          public final AtomicBoolean reserved;
6          public final LTMemory SMemory;
7
8          public QueueNode(long size) {
9                  SMemory = new LTMemory(size);
10                 reserved = new AtomicBoolean(false);
11         }
12 }
```

5.2.2 Data structure initialisation

The data structure should be initialised prior to its usage. The programmer should identify the number of writer and reader threads concurrently accessing

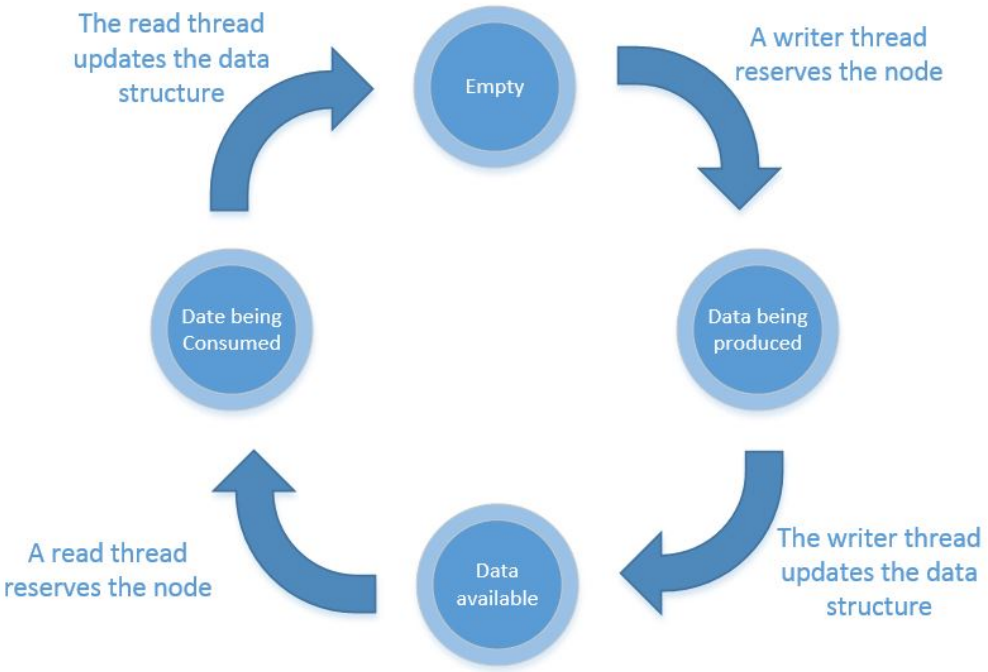


Figure 5.2: The node life cycle

the data structure and the size of the `ScopedMemory` backing store, a full listing of the WaitFreeQueue class is provided in D.2.

As discussed earlier, CAS based non-blocking implementations are subject to the ABA problem. During the node reservation it is possible that more than a writer (or reader) threads are competing to reserve the same node. It is important to ensure that if the writer (or reader) thread is preempted while performing the node reservation and another writer (or reader) thread succeed in reserving the subject node while that thread is preempted, the preempted thread fails to finish the reservation process when resumed. In a multiprocessor or multi-core environment, to avoid the occurrence of the ABA problem, if multiple writer (or reader) threads are competing to reserve the same node and these threads are scheduled on different processors or cores,

the variables that facilitate the access to the shared data structure are defined as atomic variables. Accordingly, if multiple writer (or reader) threads are competing to reserve the same node and scheduled on different processors (or cores) only one succeeds in updating the associated variable, and all the other threads fail.

The algorithm implements the version number as the avoidance solution to the ABA problem; accordingly, a writer (or reader) thread performing an enqueue (or dequeue) operation uses a version number; the thread ensures that the version number is the same before trying to update the associated variable; otherwise the thread is considered to be preempted and another thread succeeded in reserving the subject node. The *readListInvalid* array is used to store the version numbers.

5.2.3 Enqueue Method

The enqueue operation is divided into three steps:

- **Reserve a node**: the writer thread searches the data structure starting from the beginning trying to reserve the first empty node. The writer thread to succeed in reserving the node sets the `reserved` atomic boolean of that node to true; shown in figure 5.3.

- **Produce the data**: the designed runnable object is then executed in the scoped memory backing store; which is supposed to perform the operation required by the application; shown in figure 5.4.

- **Update the data structure**: the writer thread then add the index of

the node to the `readList` head, to enable the reader threads to access that node to consume the data; shown in figure 5.5.

The writer thread that succeeds in reserving the node sets the *reserved* atomic boolean to `true`. If more than a writer thread competing to reserve the same empty node, the writer thread that succeed in setting the *reserved* atomic boolean to `true` reserves the node and all other writer threads are going to retry the reservation with the next empty node.

Once the `reserved` atomic boolean of the node is set to true the node is considered as unavailable for all other writer threads except for the writer thread succeed in reserving it; during the production of the data, the writer thread suffers no interference from any other writer thread concurrently accessing the data structure. As part of the data production, the writer thread creates a real-time thread and assign it to the portal of the ScopedMemory area, this ensures that after the writer thread finished execution and exits the Scopedmemory backing store, the number of references to the backing store will not be zero; this is the simulation of the pinning of the Scopedmemory area.

The writer thread after finishing the data production updates the data structure by adding the index of the node at the `readList` head. It is possible that multiple writer threads are trying to add the index of the corresponding nodes they reserved at the head of the `readList` concurrently. The `readListHead` is an atomic integer variable. Accordingly, if there are more than a writer thread that are trying to update the value of the `readListHead` only one is going to succeed at a time, and all other writer threads is going to

retry, and so on.

Hence, a writer thread accessing the enqueue method is going to suffer the competition with the other writer threads, concurrently accessing the data structure, during the node reservation and the data structure update. During the data production the writer thread execution is not affected by the progress of any other thread (writer or reader) concurrently accessing the data structure. Thus, to bound the time required to perform the enqueue method it is required to bound the number of retries a writer thread performs when reserving a node or updating the data structure.

Figure 5.3 shows an example of a data structure of six nodes. In the figure, nodes one and six holds data that is available for consumption and the data in node three is being consumed. A writer thread is performing a node reservation by searching the data structure from the beginning of the queue. Assuming that there are no other writer threads competing, the writer thread should succeed in reserving the first empty node, in such case, node two.

The writer thread updates the `reserved` variable and access the corresponding scoped memory backing store, as shown in figure 5.4. Upon finishing the data production, the writer thread adds the index of the node to the head of the `readList` and updates the value of the `readListHead`, as shown in figure 5.5.

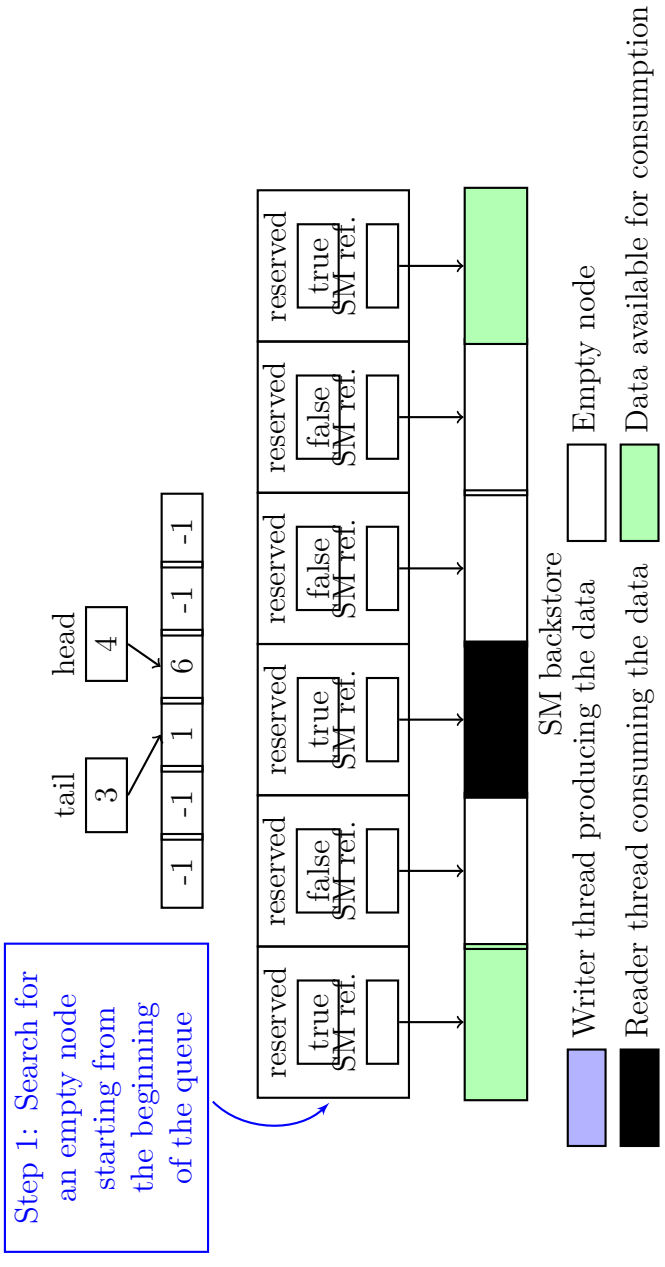


Figure 5.3: Enqueue method, step 1: reserving a node

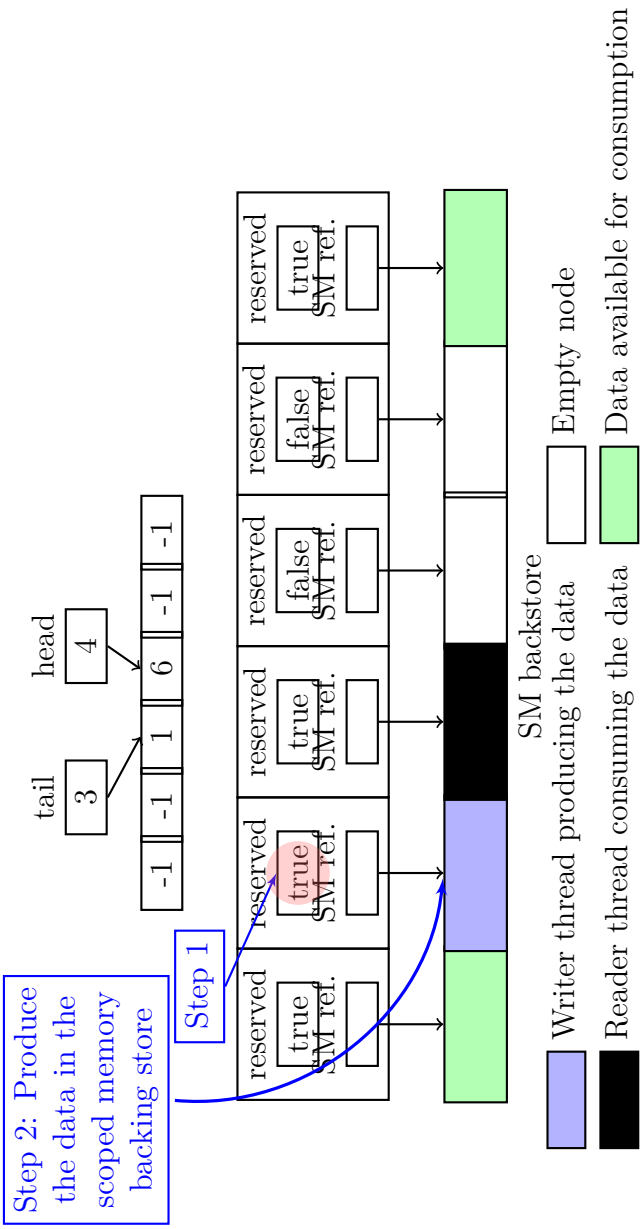


Figure 5.4: Enqueue method, step 2: produce the data

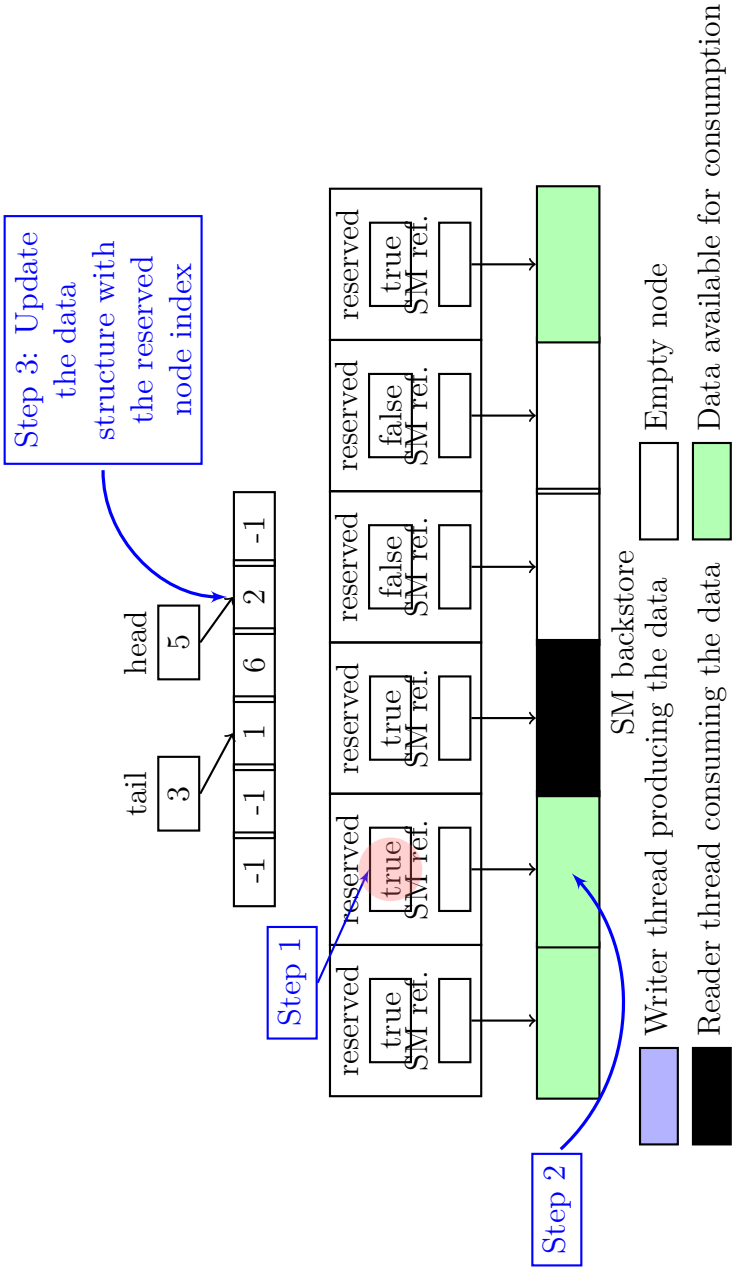


Figure 5.5: Enqueue method, step 3: update the data structure

5.2.4 Dequeue Method

A reader thread accessing the dequeue method is going to perform the following steps:

- **Reads a node index**: the reader thread reads a node index from the `readList` tail and move the `readListTail` to point to the next location in the `readList`.
- **Consumes the data**: the reader thread access the corresponding scoped memory backing store and consumes the data stored.
- **Updates the data structure**: the reader thread then updates the data structure to indicate that the node is empty and it is available for writing.

It is possible that a reader thread that is accessing the dequeue method to find the data structure empty and there is not any node that holds data available for consumption; as this dependent on the distribution of the reader and writer threads' execution concurrently accessing the data structure and their scheduling parameters.

To ensure that the `readListHead` does not lag behind the `readListTail`, all the locations of the `readList` are initialised to -1. A reader thread accessing the dequeue method is expecting the location of the `readListTail` to have a value between 0 and `queueSize -1`. The reader thread that succeeds in reading the index of the node from the `readList` (updates the `readListTail` value to point to the next location in the `readList`) resets the value of the location (that holds the index of the node it just read) to -1. Accordingly, the dequeue method returns false (indicating empty queue) if the value of the location that the `readListTail` points to is -1.

If there are more than a reader thread trying concurrently to read an index from the `readList`, only one succeeds in reading the index that the `readListTail` points to; this is because the `readListTail` is an AtomicInteger variable. Thus, only one reader thread is able to update its value, and all other reader threads concurrently trying to read an index of a node are going to fail. The failing reader threads retries to read a node index from the `readList`, if any.

The reader thread that succeeds in reading the node index consumes the data from the corresponding scoped memory backing store, when the reader thread finishes consuming the data and before exiting the Scopedmemory backing store, it stops the real-time thread assigned to the portal of the Scopedmemory backing store, hence, when the reader thread exist the Scopedmemory backing store, the number of references to the backing store is zero and a memory reclamation is performed. Then the reader thread updates the `reserved` variable of the consumed node to false. This indicates that the node is empty and available for the writer threads to perform the production of the data.

During the consumption of the data and updating the node status, the reader thread does not compete with any other reader thread, thus the reader threads are only competing during reading the index of the node from the `readList`.

Figure 5.6 shows the first step of a reader thread accessing the dequeue method; the thread reads the index of the node, node 1 in this case, and updates the value of the `readListTail` pointer, to 4 in this case; as shown in figure 5.7. The reader thread consumes the data from the scoped memory

backing store, and updates the status of the node, as shown in figure 5.8.

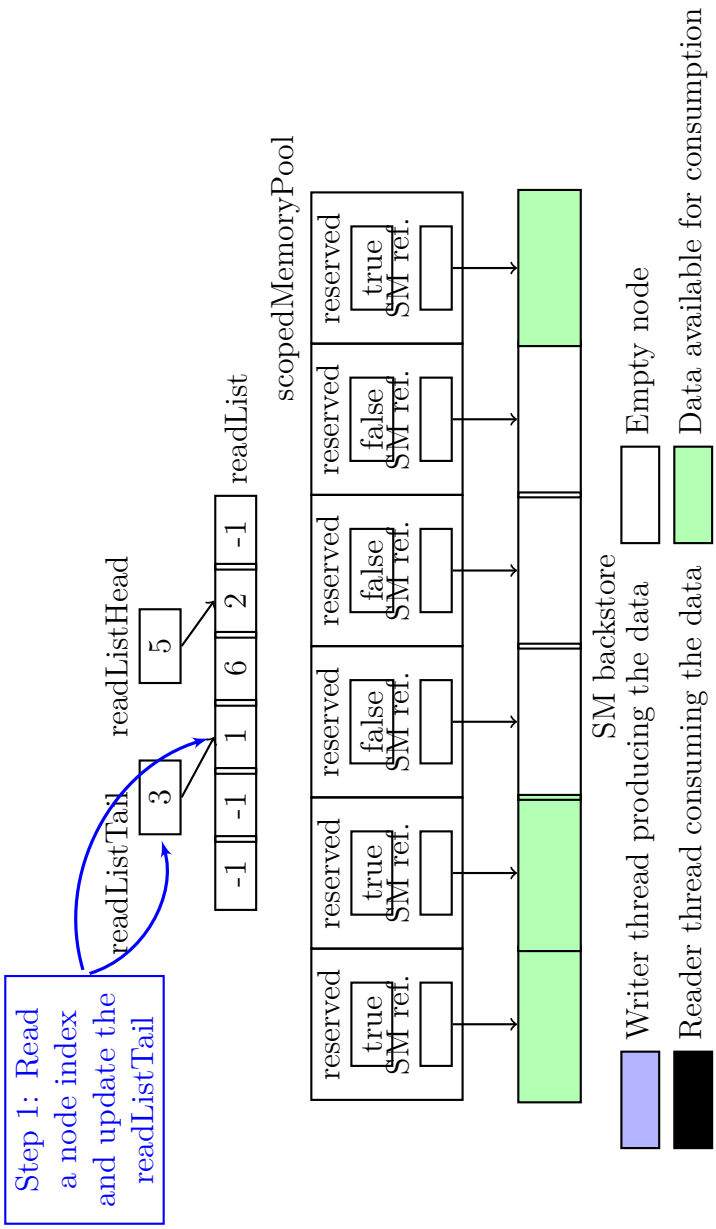


Figure 5.6: Dequeue method, step 1: read a node index

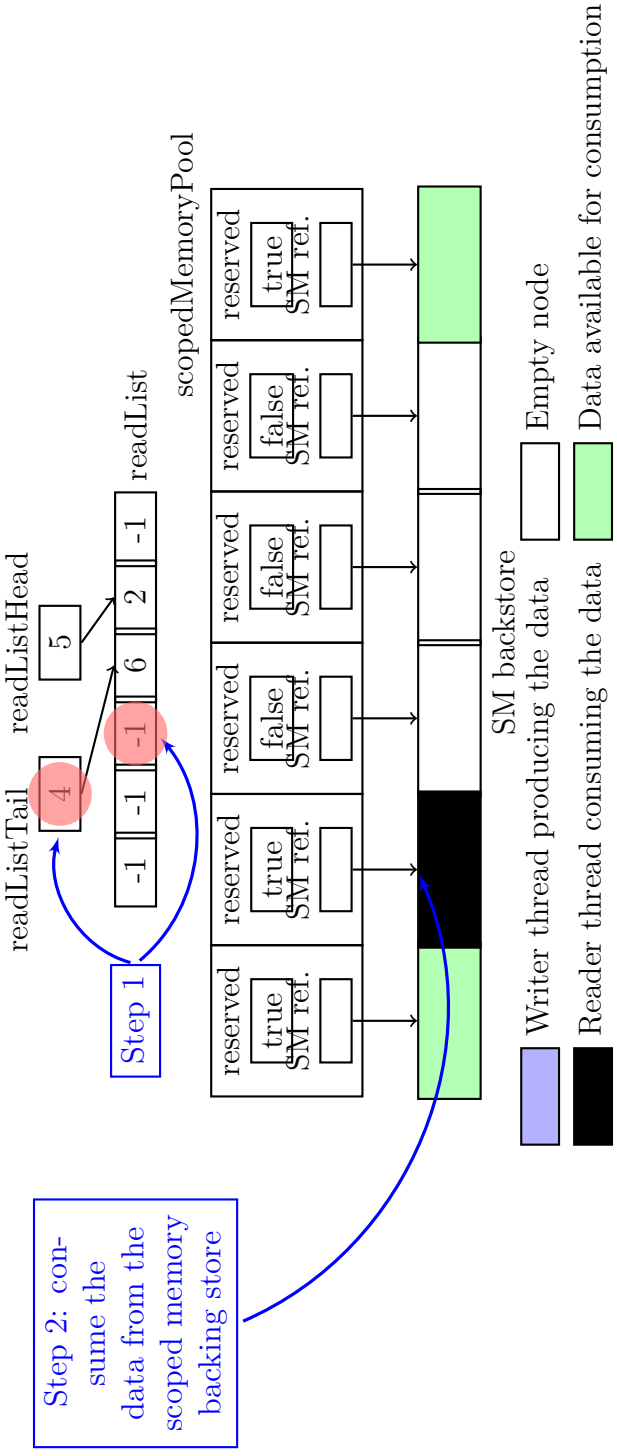


Figure 5.7: Dequeue operation, step 2: consuming the data

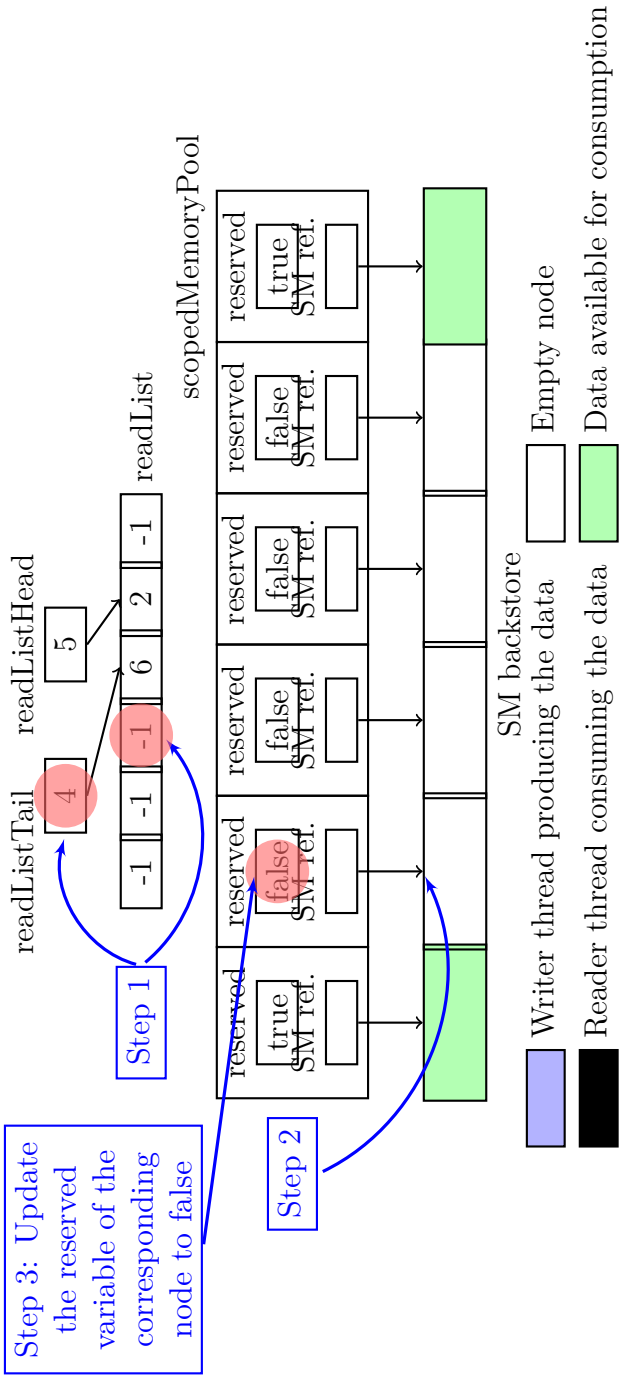


Figure 5.8: Dequeue method, step 3: updating the data structure

Chapter 6

Evaluation

This chapter introduces the evaluation of the algorithm; verifying its correctness, measuring its performance, and comparing its performance against a lock-based, and an RTSJ implementation versions.

The evaluation of the WaitFreeQueue algorithm is performed for two reasons. The first is to measure the time response of the algorithm for different real life scenarios, and estimate the computational cost required for the algorithm. The second is to ensure the correctness of the algorithm when evaluated using application that requires intensive computational power and memory requirements.

To verify the correctness of the algorithm, Java PathFinder is used to perform model checking of the algorithm and to execute different test cases; the test cases are designed in order to verify that the reader and writer threads concurrently accessing the threads are performing the dequeue and enqueue operation correctly based on the queue's initial state.

In section 6.1 the algorithm's correctness is verified using Java PathFinder

model checker. The different test cases is shown in section 6.3.

6.1 Software verification tools and Java PathFinder (JPF)

There is an increasing dependency on computer systems in all aspects in modern life, hence, it is essential to ensure that software is highly reliable; especially if the cost of failure might be expensive in both human safety and monetary terms. Thus, it is important to ensure that computer programs are bug free.

In a multi-threaded application, based on the current status of each concurrently running thread it is possible that the application might be in any arbitrary state at runtime. The threads' execution interleaving make it possible to have countless number of execution paths, which makes it hard to manually prove the correctness of the algorithm. Testing a concurrent algorithm programmatically is consequently a very hard task as the number of paths explodes as the number of concurrently running threads increases and the size of the concurrent code increases [62]. Moreover, it might be difficult to reproduce any error that occur as the execution environment is dynamic.

Among the common software correctness validation tools are formal methods and model checking [88]. Formal methods are mathematical techniques that are used at different stages of the software development: design, implementation and testing. Basically, the goal of formal methods is to mathematically model the behaviour of the system in a precise way [10]. On the other hand,

model checking is the automatic analysis of a computer program in order to identify an errors in the design.

For formal methods to provide reliable results rigorous description of the software model should be provided, however, this comes at the cost of increased learning curve. Besides that, formal methods are complex, expensive and labor intensive [88]. Moreover, it cannot fix wrong assumptions in the design.

Model checking has proven that it is cost-effective and can be integrated easily with conventional design methods, and it can be used to analyse a certain set of the software program properties [10]. Model checkers confirm that these properties hold or provide a report that they are violated. Furthermore, model checkers can provide feedback on the errors found and may help in discovering design errors. Bowen et al. in [17] argued that the use of formal methods does not eliminate the need for model checkers and software testing.

To verify the correctness of the algorithm model checking is used to ensure that all possible execution paths are free from deadlocks. In addition, testing scenarios are used to ensure that when the queue is in certain states the algorithm performs as required and the enqueue and dequeue operations are performed correctly.

Model checking has been developed to provide the necessary tool that can be used to reason about the correctness behaviour of software programs [78]. Java PathFinder (JPF) is a Java Virtual Machine (JVM) that is designed to provide state model checking for Java applications [29]. By testing all execution paths of the program, JPF identifies those paths that might result in problems. JPF stores the state of each tested execution path to ensure

that each path is tested once; if any problem detected in an execution path the stored state of the path is used for back tracing.

Java PathFinder (JPF) is an open source model checker that has a highly customizable environment for the verification of Java programs. JPF was developed at the Ames Research Center in NASA in 2005 and it is available for free.

As the number of execution paths can be very large, this is called the state explosion problem. JPF uses the state matching; “each time JPF reaches a choice point, it checks if it has already seen a similar program state, in which case it can safely abandon this path, backtrack to a previous choice point that has still unexplored choices, and proceed from there. That’s right, JPF can restore program states, which is like telling a debugger "go back 100 instructions" [107].

JPF is able to provide a report on any defect found in the program using the trace stack it creates while executing. “first it automatically executes your program in all possible ways to find defects you don’t even know about yet, then it explains you what caused these defects." [107].

In addition, JPF uses the partial order reduction in order to reduce the state space; this limits the analysis to only context switches at operations that can have effects across thread boundaries. JPF achieve this using Java bytecodes and reachability information obtained from the garbage collector.

JPF is not aware of atomic variables within Java programs, hence for test purposes only, the atomic variables used in our wait-free algorithm are implemented using synchronized methods. The implementation of the boolean and integer atomic variables are shown in appendix A.

Assertions in Java are used to confirm certain assumptions about the behaviour of the program [73]. An assertion statement has two forms: assert expr1; and assert expr1 : expr2. In both cases expr1 is a boolean expression. In the first form if the expr1 evaluates to false an AssertionError is thrown with no details. In the second form expr2 has a value which is used as part of the AssertionError. However, according to the documentation expr2 cannot invoke a method that is declared as void.

The use of the assertions in all the testing cases is meant to ensure that all the enqueue and dequeue operations are perform correctly; for example, if two dequeue operation is initiated at the same time, and there are enough nodes in the queue to consume, the first two nodes in the queue will be consumed, or if there is only one node in the queue, then one of the operations will succeed and the other will return false.

The algorithm is implemented in the aicas RTSJ implementation, JamaicaVM, and use atomic variables to facilitate the access to the data structure contents. The current version of JPF is not aware of atomic variables and the RTSJ extension. Accordingly, JPF is not able to model check the algorithm as it is. The parts of the algorithm that relate to the RTSJ, like executing the runnable objects in a scopedMemory area, are performed in isolation; hence, the execution of these parts will not be affected by the progress of execution of any other thread. Therefore, these parts of the algorithm can be removed when performing the model checking of the algorithm.

6.2 Model checking the algorithm

It is important to ensure that the algorithm performs the different operations correctly and at run time the progress of any operation is not compromising the integrity of the data structure contents. Model checking is used to ensure that the execution of the different operations at run time do not suffer any problems like data races or deadlocks.

The algorithm is model checked on a dedicated server that has four processors (with four cores each) and 32GB of memory. The Java code used to perform the model checking is listed in appendix B. Figure 6.1 shows the results of testing the algorithm with two reader and two writer threads, and it shows that there is no error detected and the algorithm is error free. Figure 6.2 shows the results of model checking the algorithm with three reader and three writer threads, and is shows that there is no errors detected while model checking the algorithm.

By comparing the results of figure 6.1 and figure 6.2 it is clear that the number of tested paths significantly increased. The number of backtracked paths in figure 6.1 is 1639091 compared to 796855400 in figure 6.2.

It is clear that as the number of threads concurrently accessing the data structure increase the number of back traced execution paths explodes, hence, it is not possible to perform model checking for more than 3 threads. However, the threads accessing the data structure are performing the same operations (writers perform enqueue and readers perform dequeue), during these operations the main concern is when the threads are interacting with the data structure; performing the reservation and updating the data structure.

```
etao500@csresearch1: /JPFTesting$ jpf +classpath=. -Xmx24024m Wfq
JavaPathfinder v6.0 (rev 778) - (C) RIACS/NASA Ames Research Center
====================================================== system
under test
application: Wfq.java
====================================================== search
started: 26/03/14 06:42
====================================================== results
no errors detected
====================================================== statis-
tics
  elapsed time: 00:05:01
  states: new=422620, visited=1216471, backtracked=1639091, end=1523
  search: maxDepth=67, constraints hit=0
  choice generators: thread=422620 (signal=0, lock=270159, shared ref=24050), data=0
  heap: new=515059, released=3953445, max live=518, gc-cycles=1609801
  instructions: 37647524
  max memory: 429MB
  loaded code: classes=95, methods=1395
====================================================== search
finished: 26/03/14 06:47
```

Figure 6.1: JPF model checking, 2 reader and 2 writer threads

As these operations are performed correctly for 3 readers or 3 writers, it is expected that the same operation will be performed correctly as the number of threads increase. However, it is expected that the time required to perform this operation is going to increase. For example, if 6 writers are concurrently trying to perform the enqueue operation, it is expected that the time required for the last writer to succeed is longer than the case when there is only 3 writers.

6.3 Model-Checked test cases

The above tests just illustrate how the state expands significantly with just three readers and writers. Any attempt to increase this number results in the JPF running out of memory. Hence, the scenarios we have developed are restricted to a maximum of three readers and three writers.

At runtime, due to the interleaving of execution of the threads concurrently

```
etao500@csresearch1: /JPFTesting$ jpf +classpath=. WaitFreeQueue
JavaPathfinder v6.0 (rev 1038) - (C) RIACS/NASA Ames Research Center
========================================================== system
under test
application: WaitFreeQueue.java
========================================================== search
started: 29/08/15 18:30
========================================================== results
no errors detected
========================================================== statis-
tics

  elapsed time: 17:09:42
  states: new=138441862, visited=658413538, backtracked=796855400, end=3770
  search: maxDepth=92, constraints hit=0
  choice generators: thread=138441862 (signal=0, lock=93359031, shared ref=3624898), data=0
  heap: new=1427, released=1557306251, max live=531, gc-cycles=783950497
  instructions: 13777607387
  max memory: 6028MB
  loaded code: classes=87, methods=1373
========================================================== search
finished: 30/08/15 11:40
```

Figure 6.2: JPF model checking, 3 reader and 3 writer threads

accessing the queue, the status of the queue could be in any of the following:

- **Empty**: for example, at the beginning of execution.

- **Partially empty**: there are some nodes that contain data to be consumed and there is enough space for all current writers, however, the number of readers trying to access the queue is more than the nodes containing data.

- **Partially full**: the queue is almost full. There is enough data for the current number of readers, however, the number of writers trying to enqueue new data items in the queue is more than the available nodes in the queue.

- **Full**: the queue is full. In this case, it is expected that there are readers to consume the available data items.

To ensure the correctness of the algorithm, all of these cases should be challenged with different test cases. The test cases are designed such that:

- **Multiple writer threads**: are trying to enqueue new data items to the queue.
- **Multiple reader threads**: are trying to consume the available data items in the queue.
- **Two producers and two consumers**: are trying to access the queue concurrently.

For each of the test cases, the queue is going to be initialised in the suitable state for the test; a pre-condition state of the test. Then the producers and consumers are run to perform the designed enqueuing and dequeuing operations; the test case is performed. After that, the state of the queue is checked to ensure that the performed operations are correctly executed; the post condition of the test case. In each test case, the writer or reader thread is going to perform one enqueue or dequeue operation.

As the test cases are intended to test for the correctness of the algorithm, there will be no real production or consumption of the data during these tests.

The result of any test case will be either passed or failed. When using the JamiacaVM to perform the testing the number of messages that appeared in the results is equal to those expected; for example, when three different readers are performing dequeue operations it is expected to have three messages indicating the results of each dequeue operation. However, when performing the testing using the JPF, the number of messages that appear in the results

is much larger than the expected number, this is due to the fact that JPF performs back-tracing for all execution paths, hence, there might be enormous execution paths and the messages will be shown for each execution path. However, it is important to confirm that all the result messages are reviewed and it is ensured that none of them appear to be failed.

For all the test cases, the writer thread and the reader thread is shown in appendix B. Both the writer and the reader thread is set to perform one enqueue or dequeue operation, respectively. The number of writers and readers is set as the test case requires and the main class is shown in WaitFreeQueue.java shown in appendix C.

6.3.1 Empty queue

The expected results of the test cases in the case of empty queue are:

- **For the three writers**: it is expected that the first three nodes in the queue are going to contain data for consumption after the enqueue operations are finished.

- **For the three readers**: it is expected that the queue is going to be empty, as the queue does not contain any data for consumption and the three dequeue operations return false.

- **For two readers and two writers**: the result depends on the order of the threads execution, however, at most the queue is going to have two nodes containing data for consumption, it is possible to have one or no nodes, as well. Any of these cases is considered as pass for the test.

6.3.1.1 Test case 1: multiple writers are trying to enqueue new data items

During this test case, three writer threads are trying to perform enqueue operations on the queue, initially the queue is empty. TheQueue code shown in TheQueue.java in appendix C. The queue is initialised with all nodes empty. The three writers are expected to enqueue the data items in the first three nodes: 0, 1, and 2. Hence, when a writer thread is trying to enqueue in the node number 2, the postcheck method is started to test for the presence of data items in the first three nodes of the queue and the rest of the nodes in the queue are empty.

The test case is performed using JPF to check for the correctness of the test case and algorithm correctness. The result of the test is shown in figure 6.3. The results show that there are no errors detected during the test case.

```
JavaPathfinder v6.0 (rev 1038) - (C) RIACS/NASA Ames Research Center
============================================================ system
under test
application: WaitFreeQueue.java
============================================================ search
started: 09/09/15 22:50
Test Passed
– repeated 132 times
Test Passed
Test Passed
============================================================ results
no errors detected
============================================================ statis-
tics
elapsed time: 00:00:46
states: new=64766, visited=158120, backtracked=222886, end=578
search: maxDepth=96, constraints hit=0
choice generators: thread=64766 (signal=0, lock=35769, shared ref=936), data=0
heap: new=7617, released=559557, max live=451, gc-cycles=221121
instructions: 6000129
max memory: 244MB
loaded code: classes=87, methods=1374
============================================================ search
finished: 09/09/15 22:51
```

Figure 6.3: Test case 1: 3 writer threads with empty queue

6.3.1.2 Test case 2: multiple readers are trying to dequeue data items from the queue

TheQueue is shown in TheQueue.java in appendix C. The queue is initialised with all nodes empty. The three readers are expected to perform no operations and return false, as there is no data in the queue. Hence, the test passes if the dequeue operation returns false, and to fail if the dequeue operation succeeds.

The test case is performed using JPF to check for the correctness of the test case and algorithm correctness. The result of the test is shown in figure 6.4. The results indicate that there are no errors detected during the test case.

```
JavaPathfinder v6.0 (rev 1038) - (C) RIACS/NASA Ames Research Center
========================================================== system
under test
application: WaitFreeQueue.java
========================================================== search
started: 09/09/15 23:00
Test Passed
– repeated 118 times
Test Passed
========================================================== results
no errors detected
========================================================== statis-
tics
elapsed time: 00:00:01
states: new=1108, visited=2465, backtracked=3573, end=22
search: maxDepth=35, constraints hit=0
choice generators: thread=1108 (signal=0, lock=476, shared ref=124), data=0
heap: new=2593, released=11838, max live=486, gc-cycles=3314
instructions: 67869
max memory: 244MB
loaded code: classes=87, methods=1374
========================================================== search
finished: 09/09/15 23:00
```

Figure 6.4: Test case 2: 3 reader threads with empty queue

6.3.1.3 Test case 3: two writers and two readers are accessing the queue

TheQueue is shown in TheQueue.java in appendix C. The queue is initialised with all nodes empty. The two writers are expected to perform two enqueue operations, one each, and the two readers are expected to perform two dequeue operations, one each. The queue after the writer and readers threads finish execution, might contain none, one, or two nodes containing data to be consumed; this is dependent on the order of execution of the reader and writer threads. The reader and writer threads are expected to perform their operations on nodes 0 and 1; hence, any operation performed on any other nodes is considered as a failure to the test.

The test case is performed using JPF to check for the correctness of the test case and algorithm correctness. The result of the test is shown in figure 6.5. The results indicate that there are no errors detected during the test case.

6.3.2 Partial empty queue

The expected results of the test cases in the case of partial empty queue are:

- **For the three writers**: it is expected that the three writers are going to succeed in performing the enqueue operations. The status of the queue after that might be partial full or full.

- **For the three readers**: it is expected that the queue is going to be empty or partial empty, depending on the initial number of data items

```
JavaPathfinder v6.0 (rev 1038) - (C) RIACS/NASA Ames Research Center
========================================================== system
under test
application: WaitFreeQueue.java
========================================================== search
started: 09/09/15 23:16
Test Passed
– repeated 85 times
Test Passed

========================================================== results
no errors detected
========================================================== statis-
tics
elapsed time: 00:01:02
states: new=132686, visited=426680, backtracked=559366, end=351
search: maxDepth=74, constraints hit=0
choice generators: thread=132686 (signal=0, lock=77810, shared ref=5736), data=0
heap: new=139836, released=1279995, max live=483, gc-cycles=544933
instructions: 10877471
max memory: 244MB
loaded code: classes=87, methods=1374
========================================================== search
finished: 09/09/15 23:17
```

Figure 6.5: Test case 3: 2 writer and reader threads with empty queue

in the queue; the dequeue operation that does not succeed in consuming data item, false is returned.

- **For two readers and two writers**: the result depends on the order of the threads execution, however, at most the queue is going to have two new nodes containing data for consumption.

6.3.2.1 Test case 1: multiple writers are trying to enqueue new data items

During this test case, three writer threads are trying to perform enqueue operations on the queue, initially the queue is partially empty; as certain nodes are set to contain data. TheQueue code shown in TheQueue.java in appendix C. The queue is initialised with the first two nodes containing data. The three writers are expected to enqueue the data items in nodes: 2, 3, and

4. The postcheck method is started to test for the presence of data items in the first five nodes in the queue and the rest of the nodes in the queue are empty.

The test case is performed using JPF to check for the correctness of the test case and algorithm correctness. The result of the test is shown in figure 6.6. The results show that there are no errors detected during the test case.

```
JavaPathfinder v6.0 (rev 1038) - (C) RIACS/NASA Ames Research Center
========================================================== system
under test
application: WaitFreeQueue.java
============================================================ search
started: 10/09/15 21:13
Test Passed
– repeated 78 times
Test Passed
============================================================ results
no errors detected
============================================================ statis-
tics
elapsed time: 00:00:33
states: new=98812, visited=253507, backtracked=352319, end=571
search: maxDepth=99, constraints hit=0
choice generators: thread=98812 (signal=0, lock=60366, shared ref=1038), data=0
heap: new=17979, released=809353, max live=480, gc-cycles=350180
instructions: 8660209
max memory: 244MB
loaded code: classes=87, methods=1374
============================================================ search
finished: 10/09/15 21:14
```

Figure 6.6: Test case 1: 3 writer threads with partial empty queue

6.3.2.2 Test case 2: multiple readers are trying to dequeue data items from the queue

TheQueue shown in TheQueue.java in appendix C. The queue is initialised with the first two nodes containing data items. The three readers are expected to perform dequeue operations, one each. Hence, two of the dequeue operations is expected to succeed and the third to return false. The postcheck method is initiated when the third dequeue operation failed to reserve a node for

consumption and tries to return a false; the test case is considered as pass if the queue is empty and contains no data for consumption.

The test case is performed using JPF to check for the correctness of the test case and algorithm correctness. The result of the test is shown in figure 6.7. The results indicate that there are no errors detected during the test case.

```
JavaPathfinder v6.0 (rev 1038) - (C) RIACS/NASA Ames Research Center
========================================================== system
under test
application: WaitFreeQueue.java
========================================================== search
started: 09/09/15 23:00
Test Passed
– repeated 91 times
Test Passed
========================================================== results
no errors detected
========================================================== statis-
tics
elapsed time: 00:00:01
states: new=1108, visited=2465, backtracked=3573, end=22
search: maxDepth=35, constraints hit=0
choice generators: thread=1108 (signal=0, lock=476, shared ref=124), data=0
heap: new=2593, released=11838, max live=486, gc-cycles=3314
instructions: 67869
max memory: 244MB
loaded code: classes=87, methods=1374
========================================================== search
finished: 09/09/15 23:00
```

Figure 6.7: Test case 2: 3 reader threads with empty queue

6.3.2.3 Test case 3: two writers and two readers are accessing the queue

TheQueue shown in TheQueue.java in appendix C. The queue is initialised with all nodes empty. The two writers are expected to perform two enqueue operations, one each, and the two readers are expected to perform two dequeue operations, one each. The queue after the writer and readers threads finish execution, might contain none, one, or two nodes containing data to be

consumed; this is dependent on the order of execution of the reader and writer threads. The reader and writer threads are expected to perform their operations on nodes 0 and 1; hence, any operation performed on any other nodes is considered as a failure to the test.

The test case is performed using JPF to check for the correctness of the test case and algorithm correctness. The result of the test is shown in figure 6.8. The results indicate that there are no errors detected during the test case.

```
JavaPathfinder v6.0 (rev 1038) - (C) RIACS/NASA Ames Research Center
========================================================== system
under test
application: WaitFreeQueue.java
============================================================ search
started: 09/09/15 23:16
Test Passed
– repeated 88 times
Test Passed
============================================================ results
no errors detected
============================================================ statis-
tics
elapsed time: 00:01:02
states: new=132686, visited=426680, backtracked=559366, end=351
search: maxDepth=74, constraints hit=0
choice generators: thread=132686 (signal=0, lock=77810, shared ref=5736), data=0
heap: new=139836, released=1279995, max live=483, gc-cycles=544933
instructions: 10877471
max memory: 244MB
loaded code: classes=87, methods=1374
============================================================ search
finished: 09/09/15 23:17
```

Figure 6.8: Test case 3: 2 writer and reader threads with empty queue

6.3.3 Partial Full queue

The expected results of the test cases in the case of partial full queue are:

- **For the three writers**: it is not guaranteed that the three writers are going to succeed in performing the enqueue operations, as it is possible

for the queue to have less than three empty nodes. The status of the queue after that might be full.

- **For the three readers**: the three dequeue operations are going to succeed from the first three nodes containing data.

- **For two readers and two writers**: as the dequeue operations is going to be performed from the begining of the queue and the enqueue is going to be performed at the end of the queue, then the number of nodes that are containing data for consumption is going to be the same, however, the order of these nodes is going to differ.

6.3.3.1 Test case 1: multiple writers are trying to enqueue new data items

During this test case, three writer threads are trying to perform enqueue operations on the queue, initially the queue is partially full; as certain nodes are set to contain data. TheQueue code shown in TheQueue.java in appendix C. The queue is initialised with the first six nodes containing data. The three writers are expected to enqueue the data items in nodes: 6, 7, and 8. The postcheck method is started to test for the presence of data items, the test is passed if all the nodes in the queue are containing data.

The test case is performed using JPF to check for the correctness of the test case and algorithm correctness. The result of the test is shown in figure 6.9. The results show that there are no errors detected during the test case.

```
JavaPathfinder v6.0 (rev 1038) - (C) RIACS/NASA Ames Research Center
========================================================== system
under test
application: WaitFreeQueue.java
========================================================== search
started: 11/09/15 00:53
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
========================================================== results
no errors detected
========================================================== statis-
tics
elapsed time: 00:00:30
states: new=84100, visited=215143, backtracked=299243, end=514
search: maxDepth=93, constraints hit=0
choice generators: thread=84100 (signal=0, lock=52758, shared ref=1008), data=0
heap: new=11884, released=679366, max live=471, gc-cycles=297134
instructions: 6764315
max memory: 244MB
loaded code: classes=87, methods=1374
========================================================== search
finished: 11/09/15 00:54
```

Figure 6.9: Test case 1: 3 writer threads with partial empty queue

6.3.3.2 Test case 2: multiple readers are trying to dequeue data items from the queue

TheQueue shown in TheQueue.java in appendix C. The queue is initialised with the first four nodes containing data items. The three readers are expected to perform dequeue operations, one each. The dequeue operations is expected to succeed in consuming the nodes from the first three nodes. The postcheck method is initiated when the third dequeue operation succeed in reserving node number two for consumption. In such case, the test case is considered as pass if all the nodes in the queue is empty except for node three.

The test case is performed using JPF to check for the correctness of the test case and algorithm correctness. The result of the test is shown in figure 6.10. The results indicate that there are no errors detected during the test

case.

```
JavaPathfinder v6.0 (rev 1038) - (C) RIACS/NASA Ames Research Center
========================================================== system
under test
application: WaitFreeQueue.java
========================================================== search
started: 09/09/15 23:00
Test Passed
– repeated 64 times
Test Passed
========================================================== results
no errors detected
========================================================== statis-
tics
elapsed time: 00:00:11
states: new=27886, visited=73616, backtracked=101502, end=197
search: maxDepth=60, constraints hit=0
choice generators: thread=27886 (signal=0, lock=16776, shared ref=508), data=0
heap: new=6560, released=245114, max live=471, gc-cycles=100329
instructions: 1662239
max memory: 244MB
loaded code: classes=87, methods=1374
========================================================== search
finished: 11/09/15 00:18
```

Figure 6.10: Test case 2: 3 reader threads with empty queue

6.3.3.3 Test case 3: two writers and two readers are accessing the queue

TheQueue shown in TheQueue.java in appendix C. The queue is initialised with partially full queue (7 nodes out of 10 are not empty). The two writers are expected to perform two enqueue operations, one each, and the two readers are expected to perform two dequeue operations, one each. The queue after the writer and readers threads finish execution should contain 7 non-empty nodes, however, it is possible to have any combination of empty and non-empty nodes dependent on the order of execution of the reader and writer threads.

The reader threads are expected to perform the dequeue operations from nodes 0 and 1. The writer threads might perform the enqueue operations at

nodes 0 and 1, or later in the queue. The test is considered as passed, if the number of non-empty nodes in the queue are 7.

The test case is performed using JPF to check for the correctness of the test case and algorithm correctness. The result of the test is shown in figure 6.11. The results indicate that there are no errors detected during the test case.

```
JavaPathfinder v6.0 (rev 1038) - (C) RIACS/NASA Ames Research Center
========================================================== system
under test
application: WaitFreeQueue.java
========================================================== search
started: 09/09/15 23:16
Test Passed
– repeated 73 times
Test Passed
========================================================== results
no errors detected
========================================================== statis-
tics
elapsed time: 00:01:02
states: new=132686, visited=426680, backtracked=559366, end=351
search: maxDepth=74, constraints hit=0
choice generators: thread=132686 (signal=0, lock=77810, shared ref=5736), data=0
heap: new=139836, released=1279995, max live=483, gc-cycles=544933
instructions: 10877471
max memory: 244MB
loaded code: classes=87, methods=1374
========================================================== search
finished: 09/09/15 23:17
```

Figure 6.11: Test case 3: 2 writer and reader threads with empty queue

6.3.4 Full queue

The expected results of the test cases in the case of full queue are:

- **For the three writers**: the three writers are going to fail in performing the enqueue operations, as the queue is full. The status of the queue after the test is full.

- **For the three readers**: the three dequeue operations are going to succeed from the first three nodes in the queue.

- **For two readers and two writers**: according to the order of the execution of the threads it might be possible to have any scenario. However, the dequeue operations is expected to succeed from the first two nodes in the queue, it is expected that the two enqueue operations is going to fail as the progress of the enqueue operation ahead in the queue is going to fail in finding any empty nodes.

6.3.4.1 Test case 1: multiple writers are trying to enqueue data items in full queue

TheQueue shown in TheQueue.java in appendix C. In this test case, three writers are trying to enqueue three new data items to the queue, one each. The queue is set such that all the nodes in the queue contain data. In this test, the enqueue method is expected to return false, as there are no space for enqueueing any new data items. The test case passes if the three enqueue operations return false, and fails if any of the operations tries to finish the enqueue operation.

The test case is performed using JPF to check for the correctness of the test case and algorithm correctness. The result of the test is shown in figure 6.12. The results indicate that there are no errors detected during the test case.

```
JavaPathfinder v6.0 (rev 1038) - (C) RIACS/NASA Ames Research Center
========================================================== system
under test
application: WaitFreeQueue.java
========================================================== search
started: 11/09/15 01:03
Test Passed
– repeated 97 times
Test Passed
========================================================== results
no errors detected
========================================================== statis-
tics
elapsed time: 00:00:08
states: new=17700, visited=47441, backtracked=65141, end=64
search: maxDepth=59, constraints hit=0
choice generators: thread=17700 (signal=0, lock=11020, shared ref=634), data=0
heap: new=19863, released=131082, max live=480, gc-cycles=63646
instructions: 1603905
max memory: 244MB
loaded code: classes=87, methods=1373
========================================================== search
finished: 11/09/15 01:03
```

Figure 6.12: Test case 4: 3 writer threads with partial empty queue

6.3.4.2 Test case 2: multiple readers are trying to dequeue data items from the queue

TheQueue shown in TheQueue.java in appendix C. The queue is initialised with all nodes containing data items. The three readers are expected to perform dequeue operations, one each. The dequeue operations are expected to succeed in consuming the nodes from the first three nodes. The postcheck method is initiated when the third dequeue operation succeed in reserving node number two for consumption. In such case, the test case is considered as pass if the first three nodes are empty and all other nodes are containing data.

The test case is performed using JPF to check for the correctness of the test case and algorithm correctness. The result of the test is shown in figure 6.13. The results indicate that there are no errors detected during the test

case.

```
JavaPathfinder v6.0 (rev 1038) - (C) RIACS/NASA Ames Research Center
========================================================== system
under test
application: WaitFreeQueue.java
========================================================== search
started: 11/09/15 00:11
Test Passed
– repeated 63 times
Test Passed
========================================================== results
no errors detected
========================================================== statis-
tics
elapsed time: 00:00:17
states: new=30030, visited=79782, backtracked=109812, end=195
search: maxDepth=63, constraints hit=0
choice generators: thread=30030 (signal=0, lock=17896, shared ref=542), data=0
heap: new=7516, released=265626, max live=471, gc-cycles=108573
instructions: 1867991
max memory: 244MB
loaded code: classes=87, methods=1374
========================================================== search
finished: 11/09/15 00:11
```

Figure 6.13: Test case 2: 3 reader threads with empty queue

6.3.4.3 Test case 3: two writers and two readers are trying to concurrently access the queue

TheQueue shown in TheQueue.java in appendix C. The queue is initialised with all nodes containing data items. The two writers are expected to perform enqueue operations, one each, and the two readers are expected to perform dequeue operations, one each. The dequeue operations are expected to succeed in consuming the nodes from the first two nodes in the queue, while the enqueue operations are expected to fail, as the queue by the time the writer threads are trying to enqueue contain no space for new data items.

The test case is performed using JPF to check for the correctness of the test case and algorithm correctness. The result of the test is shown in figure 6.14. The results indicate that there are no errors detected during the test

case.

```
JavaPathfinder v6.0 (rev 1038) - (C) RIACS/NASA Ames Research Center
========================================================== system
under test
application: WaitFreeQueue.java
========================================================== search
started: 28/10/15 03:47
Test Passed
– repeated 58 times
Test Passed
========================================================== results
no errors detected
========================================================== statis-
tics
elapsed time: 00:02:14
states: new=465856, visited=1618409, backtracked=2084265, end=97
search: maxDepth=57, constraints hit=0
choice generators: thread=465856 (signal=0, lock=262347, shared ref=29634), data=0
heap: new=1126, released=1176035, max live=423, gc-cycles=2047768
instructions: 42720815
max memory: 244MB
loaded code: classes=87, methods=1378
========================================================== search
finished: 28/10/15 03:49
```

Figure 6.14: Test case 2: 2 writer and 2 reader threads with full queue

6.3.5 Test cases summary

The results of the test cases are shown in table 6.1; the results indicate that the algorithm is correct; as long as, the Java Pathfinder is correct and the use of the RTSJ introduces no errors. The Sections 6.4 and 6.5, beside giving an insight on the performance figures, provide an evidence that the use of the RTSJ is correct and introduce no errors in the algorithm.

6.4 Performance Evaluation

The enqueue method is divided into three parts: reserve a node in the data structure, perform the data production, and update the data structure. The dequeue method is divided into three parts: read a node index from the

Test case		Result
Empty	3 readers	Test passed
	3 writers	Test passed
	2 readers and 2 writers	Test passed
Partial empty	3 readers	Test passed
	3 writers	Test passed
	2 readers and 2 writers	Test passed
Partial full	3 readers	Test passed
	3 writers	Test passed
	2 readers and 2 writers	Test passed
Full	3 readers	Test passed
	3 writers	Test passed
	2 readers and 2 writers	Test passed

Table 6.1: Test cases results summary

`readList`, perform the data consumption, and update the data structure. The time required to perform the data production and consumption is application dependent. The programmer should estimate the time required to execute this part of the code. The time required to execute the other parts of the enqueue and dequeue methods is dependent on different variables, like the number of reader and writer threads and the current status of the data structure.

In this section, the response time of the algorithm is analysed. The progress of the threads during the production and consumption of the data is not affecting the progress of any other thread concurrently accessing the data structure; hence, the performance of the algorithm is measured for those parts of the enqueue and dequeue methods that require interaction with the data structure, as the progress of the thread at these parts might be affected by the progress of any other thread performing the same operation.

For a writer thread to perform the reservation process, it is required to compete with all other concurrently running writer threads. In the worst

case, the writer thread retries to reserve a node in the data structure and succeeds in reserving the last node (i.e. all other nodes are reserved by a writer, reserved by a reader, or holds data to be consumed). The writer thread is guaranteed to find an empty node in the data structure from the first iteration, it does not require searching the data structure for a second time.

After the data production, the writer thread needs to add the node index to the `readList`; which might require the writer thread to compete with the other writer threads concurrently accessing the data structure. However, this time the writer thread might retry a number of times equal to the number of writer threads concurrently performing the update of the data structure.

The reader thread is required to compete with the other reader threads concurrently accessing the data structure while reading a node index from the `readlist`; the update of the data structure does not require any competition.

For time analysis purposes, two tests are performed. The first test intends to measure the average time required to perform the enqueue and dequeue methods (except for the data production and consumption, which is application dependent). The second test intends to measure the worst case time, where the writer threads every time are reserving the last node in the data structure, and the reader thread is retrying number of times equal to the number of reader threads.

The two test are performed on a server that has four Intel Xeon E5345 processors (Quad core, 2.3GHz, 32KB L1 cache, 4MB L2 cache). During the tests the server are not shared with any other users and all processes that might affect the progress of the test are terminated. The server is

running Ubuntu 12.04 LTS operating system and JamaicaVM multicore RTJS implementation version 6.2-5-8142.

From a real-time perspective, all the threads are set for the same properties. The release time is the same as the period start time and the deadline is at the end of the period. All threads are executed with the same priority, which is the minimum priority of the system plus 4.

The WaitFreeQueue is initiated in the immortal memory, and all the data items are stored in scoped memory backing stores. Accordingly, the algorithm is safe for real-time and NHRT threads.

6.4.1 Response time test analysis

During the average response time test, the enqueue and dequeue method is performed on the queue from more than a writer and reader threads. The average time is measured for high number of retries to ensure that the average represent reliable values. The number of writer and reader threads concurrently accessing the data structure varies between 2 and 8.

During the test, each writer and reader thread is accessing the enqueue and dequeue methods 50K times. This is to ensure that the average response time is performed for a large number of enqueue and dequeue operations and the measured time represents reliable results.

Figures 6.15 and 6.16 presents the average response time for the enqueue and dequeue methods.

The results of the average response time indicate that the response of the algorithm is relatively stable as the number of the threads increase. This

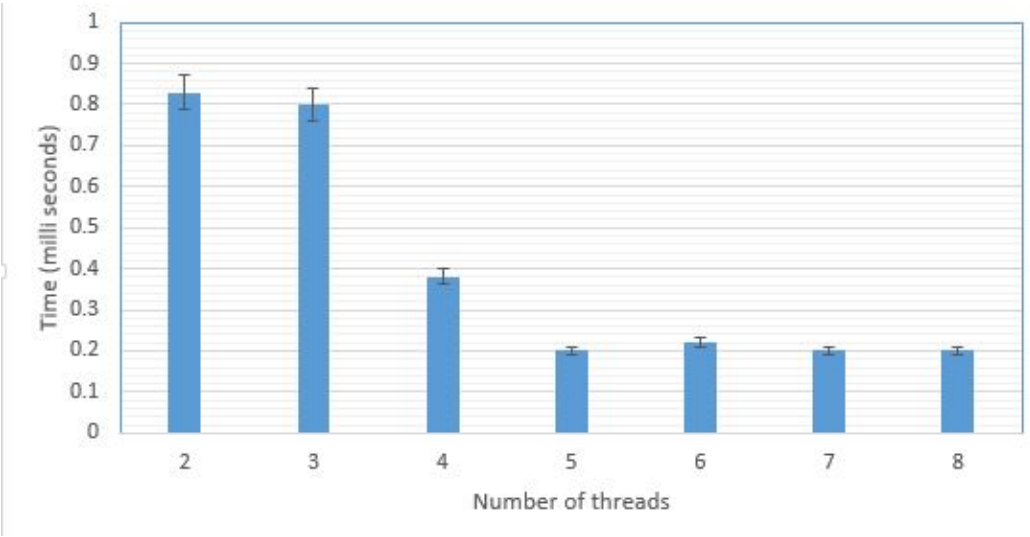


Figure 6.15: Average response time for the enqueue operation

is due to the fact that the cache hit ratio is highly increasing as the same operation is retried frequently overtime. When the number of threads is 8, the overall number of times the threads is accessing the enqueue method is 400K times, which ensures that the average response time reflects the actual time required to perform the operation. The standard deviation of the enqueue and dequeue method indicates that the average times are normally distributed and there is no statistical difference when the number of threads changes.

6.4.2 Worst case response time

In the worst case, a writer thread trying to perform an enqueue operation is going to succeed in reserving the last node in the queue, which is the longest time to perform the reservation process. The purpose of this test is to measure the time required for a writer thread to reserve a node in the WaitFreeQueue in the worst case. Accordingly, the WaitFreeQueue is initiated such that all

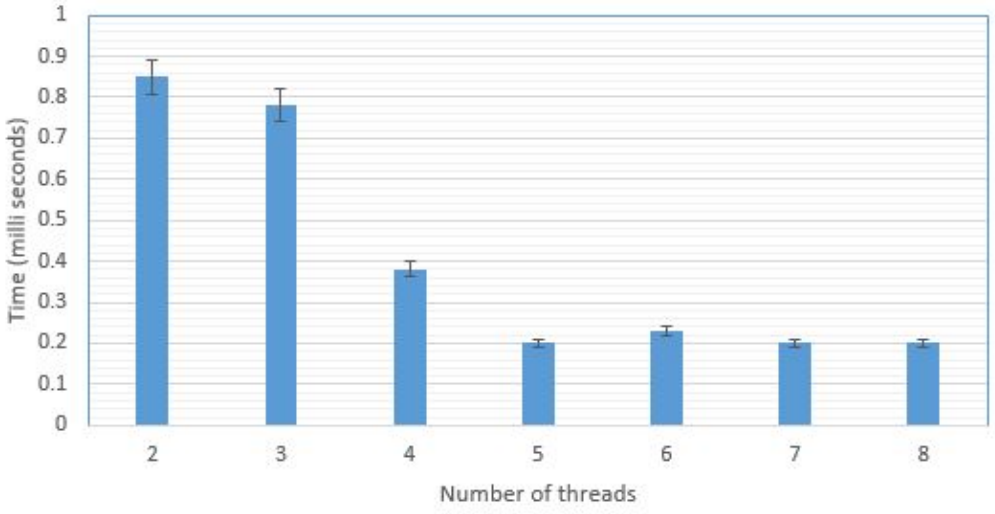


Figure 6.16: Average response time for the dequeue operation

the nodes in the queue is initialised to a reserved state, except for the last node. Thus, the writer threads are supposed to reserve the last node in the queue. However, the writer threads are going to falsely reserve the node, so all the writer threads appear to reserve the last node, no change to the state of that node is performed. The reader threads, from the other side, read the index of the node and perform no real dequeue operation; the state of the node is left as is.

During the test, each writer and reader thread is accessing the enqueue and dequeue methods 50K times. This is to ensure that the average response time is performed for a large number of enqueue and dequeue operations and the measured time represents reliable results; for 2 thread, there are 100K enqueue or dequeue operations performed, and for 8 threads the number of enqueue or dequeue operations performed is 400K times.

Figures 6.17 and 6.18 shows the response time to reserving the last node

in the queue for the enqueue and dequeue methods. The results show that, the average response time for reserving a node in the queue is less that the time required to reserve the last node in the queue. Besides that, the time required to perform the dequeue operation is less than the time required to perform the enqueue operation; as the writer threads are competing twice while accessing the enqueue method, the first to reserve a node and the second to update the queue, while the reader threads are competing only once, when reading a node index from the queue.

The results also indicate that both the average time to reserve a node and the average time to reserve the last node in the queue has an inverse relation with the number of threads.

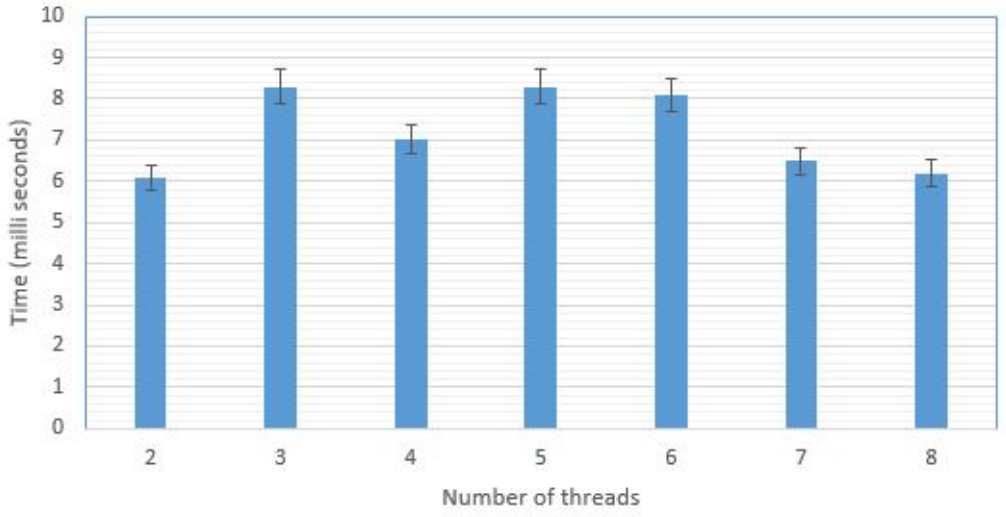


Figure 6.17: Worst case response time for the enqueue operation

In general, the average response time is relatively shorter than the worst case response time. This is due to that, in the average response time, the measured time is the time required to perform the operation, enqueue or

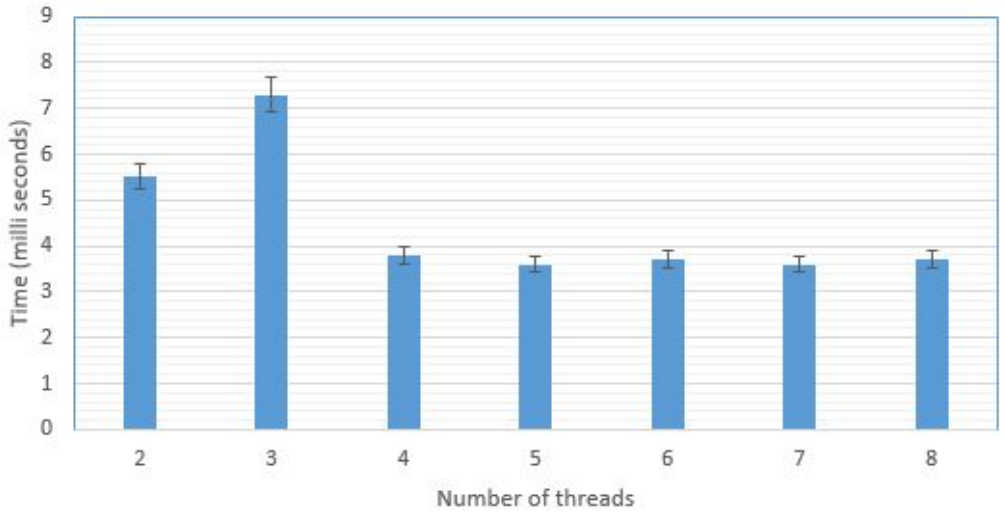


Figure 6.18: Worst case response time for the dequeue operation

dequeue, while in the worst case response time, the measured time is the time required to reach the end of the queue for all of the threads.

In general, at run time the actual response time is in between the average response time and the worst case response time; as the execution of the reader and writer threads are going to interleave and there will be empty nodes across the queue, and the writer threads do not need to reach the end of the queue in most of the enqueue methods.

In the case of larger number of threads, the size of the queue is going to be larger; hence, it is expected that the actual time required to find an empty node in the queue is going to increase. The time required to find an empty node in the queue will be dependent on many factors, like the number of other threads concurrently trying to perform the same operation, the time required to perform the data consumption/production, and the interleaving of execution between the reader and writer threads.

The actual time required to perform the enqueue or dequeue operation is expected to be relative to the size of the queue, although at run time the status of the queue could be changing frequently. Expect of the data production/consumption time, the time required to perform the enqueue or dequeue operation requires nearly 50μs. This time is expected to increase as the number of threads increase, as the queue size is going to increase.

However, it is expected that the time required to perform the enqueue and dequeue operations is not going to highly increase as the interleaving of execution of reader and writer threads are going to leave empty nodes across the queue, hence, the time required to perform the enqueue operation is going to be relatively close to the current measurements. This is due to the fact that all the enqueue operations are going to be performed ahead in the queue. Regarding the dequeue operations, are going to be performed directly on the enqueued nodes, hence, it is expected that the time required to perform this operation is not going to relatively increase.

6.5 Case study - Image Processing

Digital image processing is an area that is of high interest in computer technology [50]. Currently, real-time image processing can be found in many applications, like digital cell-phone, intelligent robots, video surveillance systems, high-definition television, medical imaging devices, spectral imaging systems and defense applications [93]; these application can be soft, firm, or hard real-time applications.

The concurrency control mechanism for shared data structures within

these applications should be designed such that it fully exploits the inherent concurrency within the application, adopt the best load-balancing technique and can be easily implemented [22].

Generally, image/video applications require processing huge amounts of data to conduct a certain decision; this requires performing certain procedures developed by the programmer. In image processing applications that require real-time characteristics the response time of these procedures should be bounded [132].

The amount of data to be processed depends on different factors like image size, the color depth, and the frame rate (in case of video) [121]. For example, in [76] a real-time image processing application for face detection is investigated. The application supports the processing of 500 frames per second for 8-bit color images that have 512X512 pixel size; nearly 128MB of image data should be processed every second.

There are many methods and techniques within the image processing domain that are applied on images to achieve different objectives [1]; like histogram, convolution, filters, correlation, smoothing, sharpening, and intensity. In certain cases, during the image capturing there might be sources of noise that affect the quality of the image. In such cases, the captured image is a degraded version of the original scene. In this case, the image restoration techniques are used to restore the original version of the image from the degraded version. Image enhancement can be used to increase the image dynamic features such that it is better to be viewed and qualify for further processing.

Based on that, according to the application requirements, an image or

set of images are processed to identify certain characteristics of the image. Based on the findings of this processing further techniques are applied such that the final image qualifies to certain conditions that are designed by the application.

6.5.1 Image Histogram

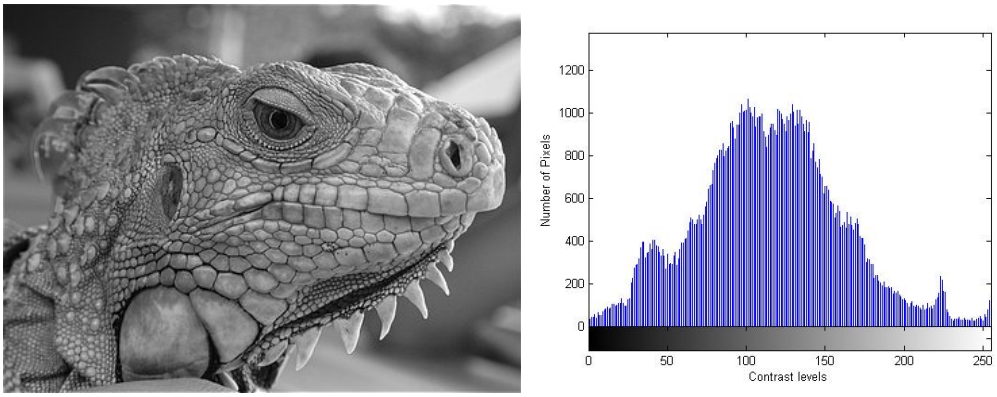
The image histogram is one of the most common techniques that is used in digital image processing; it is an essential process that is applied on the image in order to perform most of the image processing methods and techniques. Basically, the image histogram is a representation of the image contrast [50]; which is represented by equation 6.1. Figure 6.19 shows an image with its histogram representation.

The image histogram is used to better understand the image such that further operations can be performed on that image; for example, based on the image histogram it is possible to produce an image with better quality, a certain compression technique can be applied, a segmentation process can be performed, or a threshold level can be identified.

$$h\left(r_k\right) = n_k \tag{6.1}$$

where:

r_k is the Kth intensity level and n_k is the number of pixels in the image that have that level of intensity. For example, in an 8bit image there is 256 intensity levels.



(a) Original image (b) Image histogram

Figure 6.19: Original image and its histogram

In multi-threaded applications, the different threads collaborate to accomplish the designed task. In the case of image histogram, different threads can collaborate to concurrently process the image in order to produce the histogram. For example, instead of using one thread to produce the image histogram, the image can be divided to different segments and multiple threads can concurrently process these segments, by using n threads to produce the image histogram instead of one image, the time required to perform the process might be =< 1/n.

The histogram is an example in which the inherited concurrency of the application can be exposed to the shared data structure implementation. A data structure implementation that provides a high degree of parallelism is able to provide the required scalability as the number of processors (or cores) in the system scale.

Image	Pixel size	number of pixels	image size (KB)	color depth (bit)
Small	384 X 288	110,592	33.2	8
Medium	1920 X 1200	2,304,000	181	8
Large	2817 X 3181	8,960,877	444	8

Table 6.2: The selected images' details

6.5.2 Experimentation

For evaluation purposes, three images are considered to achieve diversity in the time required to perform the data production and consumption. The selected images, shown in figures 6.20, 6.21 and 6.22 are grey scale images; hence, each pixel is represented by an 8bit. Accordingly, the number of levels are 256.

Table 6.2 shows the details of the three selected images. The three images selected such that the time required to perform the enqueue or dequeue operation significantly varies from one image to another. For example, in the case of the small image if the image data is divided between four thread to calculate the image histogram each image is supposed to process 22,118 pixels compared to nearly 2,240,219 pixels in the case of the large image.

From the evaluation perspective, producing the histogram is a process that is divided into three steps:

Pre-processing: a thread prepares the image for processing by dividing the image into segments, based on the number of threads that are going to produce the histogram. The thread then enqueues the details of each segment in a node in the data structure.

Processing: at this step, the processing threads collaborates together con-

Figure 6.20: Small image

currently to produce the histogram; each thread reads one or more segment details from the WaitFreeQueue, the histogram corresponding to that segment is produced and enqueued in another WaitFreeQueue.

Post-processing: at this step, a writer thread reads all the partial histograms (produced for the segments) to produce a complete histogram.

Figure 6.23 shows a block diagram of the image histogram application. The imageDistributionThread process the image and divide it into segments, the details of each segment is enqueued in the imageDataPool. The number of segments is equal to the number of pixelCountingThreads; such that each thread produces the histogram for one of the segments. The pixelCountingThreads enqueue the produced histograms of the segments into imageCountPool. Finally, the countCollection thread collects all partial histograms and produce the complete histogram of the image. The code listings of the threads are provided in the Appendix D.2.

Figure 6.21: Medium image

Accordingly, the enqueue of the segment details in the imageDataPool and the enqueue of the histogram of the segment in the imageCountPool required relatively short time compared to the the production of the histogram of the segment and the collection of all partial histograms (both performed during the dequeue method). The time required to perform the production of the histogram of the segment and the collection of all partial histograms to produce the complete histogram is expected to increase as the size of the image increase.

For example, when there are four threads processing the partial histograms of the segments the number of pixels each thread is processing for small, medium, and large images are 27,648, 576,000, and 2,240,219, respectively. Practically, in the large image case, each of the four threads is processing a segment that is 20 time the size of the small image.

Accordingly, the real-time properties of the threads is reviewed for every image to ensure that these properties are adequate for the time required to perform the designed task.

Figure 6.22: Large image

Generally, in image processing applications the image histogram is produced for the images (or a movie, in which there is a certain frames, or pictures in each second) to perform further operations on the image. The evaluation is only concerned with testing the correctness of the algorithm and measuring its response time, and not the actual operation of the image histogram. Thus, the evaluation of the WaitFreeQueue is performed as follows:

- Step 1: the evaluation is performed by an implementation of the WaitFreeQueue, Jamaica RTSJ WFQ (given in appendix D.1), and lock-based versions.

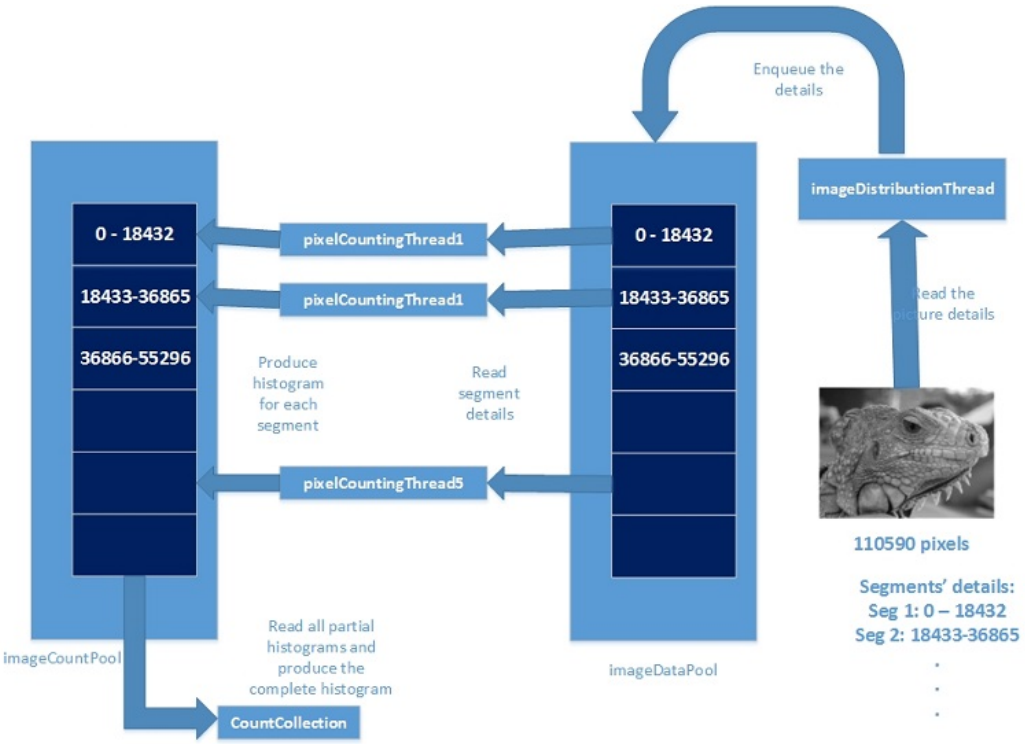


Figure 6.23: Block diagram of the image histogram application

- Step 2: the image is divided into number of segments equal to the number of processing threads. in each iteration each processing thread is performing the operation once. For example, if there are four pixelCountingThreads, the imageDistributionThread divides the image into four segments, and each of the pixelCountingThread perform the designed operation on one of the segments.

- Step 3: the evaluation is performed for the selected images only. For each image, the same process is performed 50K times. For example, if there are four pixelCountingThreads and the small image is being processed. In each iteration each thread of the pixelCountingThreads processes one of the segments. This is repeated 50K times. In each iteration pixelCountingThread can process any of the segments.

- Step 4: When performing the dequeue operation, it is possible that the dequeue operation return a false value; indicating that at that instance of time there is no data for consumption in the queue. Such cases are not included in the time analysis. It is worth mentioning that during the evaluation, the number of dequeue operations that return false is limited to the number of threads performing the dequeue operation; which is the case where the dequeue operation is accessed before any enqueue operation. Besides that, the dequeue operation is taking longer time to finish, which enables the threads performing the enqueue operation enough time to enqueue enough data that keeps the processing thread busy all the time.

- Step 5: the produced image histogram in each iteration is stored and

verified later with the original histogram produced for the image. This is to ensure that all operations performed during the evaluation produce correct results.

- step 6: the number of retries that are required to finish the enqueue operation is not monitored during the evaluation. However, in more than 90% of the cases, the reservation of the node in the queue is achieved before the thread reaches the middle of the queue. This indicates that the worst case scenario is not occurring.

- step 7: the experiment is performed on a machine that has four Intel Xeon E5345 processors (each processor is a Quad core, 2.3GHz, 32KB L1 cache, 4MB L2 cache), these cores are real cores, no hyper-threading is used, hence a maximum of 16 threads can actually run concurrently on the system. During the evaluation, the machine is completely dedicated for the application and all other applications that might affect the progress of the threads execution is terminated.

- step 8: global scheduling is used, so any thread can be scheduled at any processor, and all threads are running on the same priority.

Figures 6.24 and 6.25 show the average response time for the enqueue and dequeue methods for the small image, respectively. In the figure, the values represent the average time required to perform the enqueue or dequeue operations as the number of threads change. It is clear that the average time required to perform the enqueue or dequeue operation is inversely proportional with the number of threads, this is because that as the number of threads increase the workload for each thread becomes smaller.

The figures show that the WaitFreeQueue algorithm outperforms both the lock-based and the JamaicaVM RTSJ WFQ, especially, for higher work load (dequeue operation), where the thread is expected to have more time processing the data and less time interacting with the queue. In the dequeue operation case, the WaitFreeQueue shows high stability and the average of the response time is highly proportional to the number of threads.

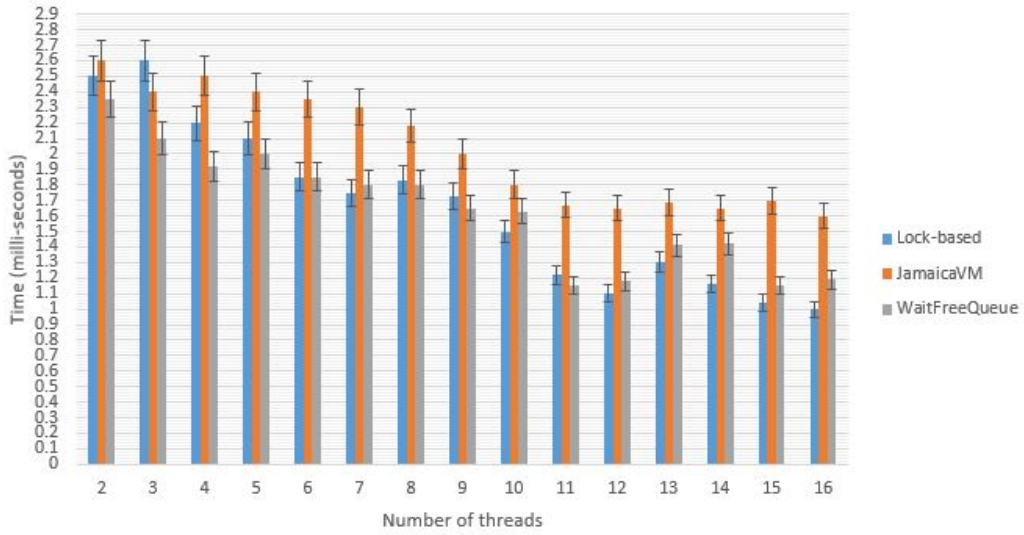


Figure 6.24: Average of response time for the enqueue operation, small image

Figures 6.26 and 6.27 show the response time for performing the enqueue and dequeue operation of the medium image. The WaitFreeQueue outperforms the lock-based and JamaicaVM RTSJ WFQ in both the enqueue and dequeue operations. For the enqueue operation the results show that as the number of threads increases the algorithm become more stable. The dequeue results show that the response of the algorithm is stable as the number of threads increases. Besides that, it is clear that the time required to perform the dequeue operation is significantly increased compared to the time required

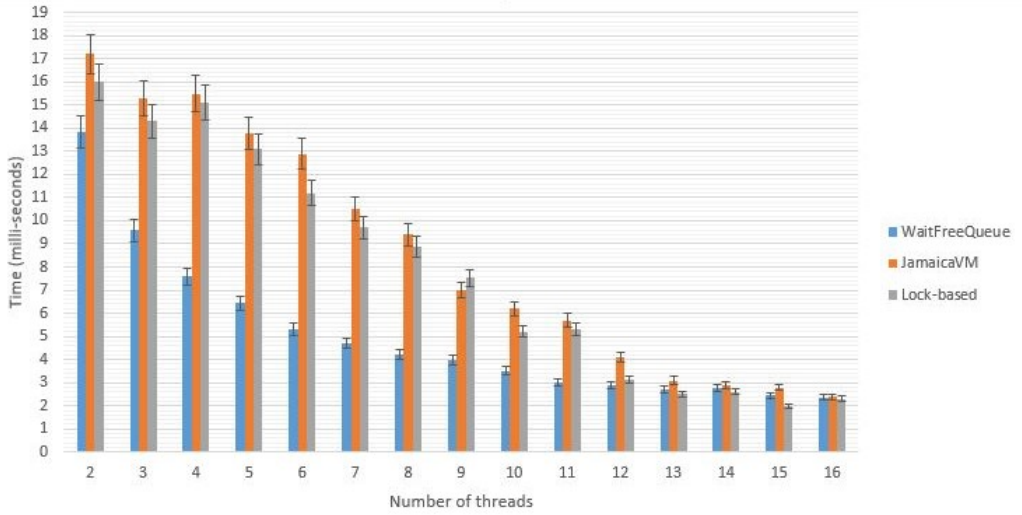


Figure 6.25: Average of response time for the dequeue operation, small image

for processing the small image. However, for higher number of threads the results of the WaitFreeQueue and the lock-based become more closer and the the lock-based appear to perfrom better; this is mainly due the use of the costly CAS operation.

In the case of the dequeue operation, the time required to perform the method is relatively high, hence, the possibility of having different threads interacting with the queue at the same time becomes lower.

Figures 6.28 and 6.29 show the response time for performing the enqueue and dequeue operation of the large image. The results show that the Wait-FreeAlgorithm outperforms the lock based and JamaicaVM RTSJ WFQ. The main influence on the WaitFreeQueue results is the time required to perform the CAS operation, which increases as the number of threads increases. In addition, the average time of the dequeue operation is significantly increases compared to the small and medium images, for example, for 8 thread the average time is nearly 230milli second, compared to 70 milli second in the

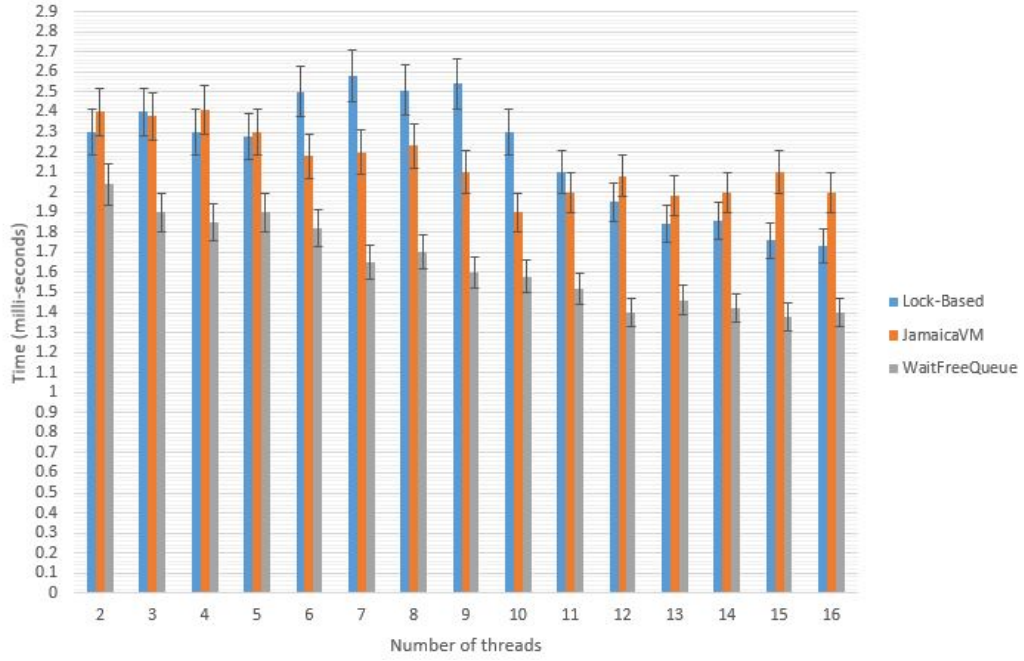


Figure 6.26: Average of response time for the enqueue operation, medium image

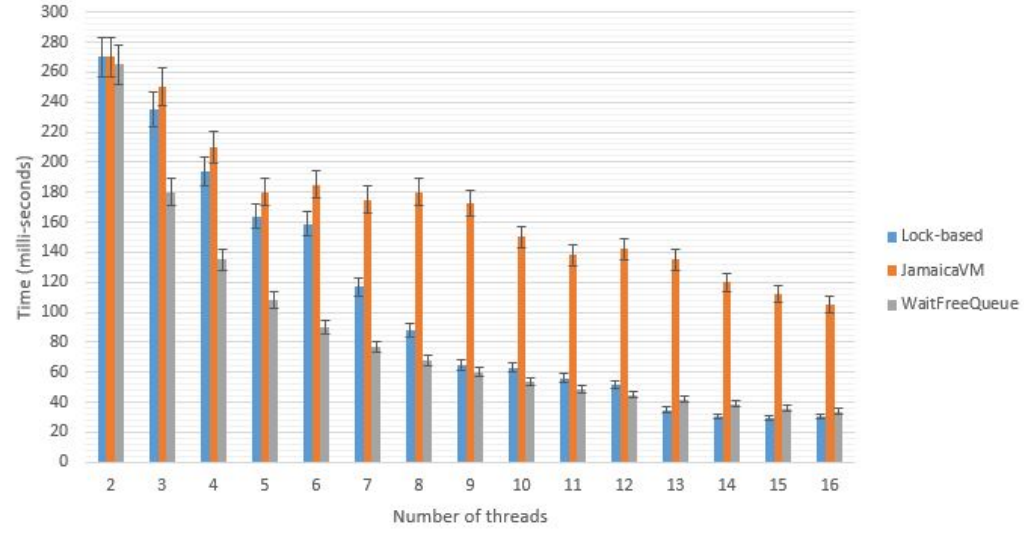


Figure 6.27: Average of response time for the dequeue operation, medium image

medium image case.

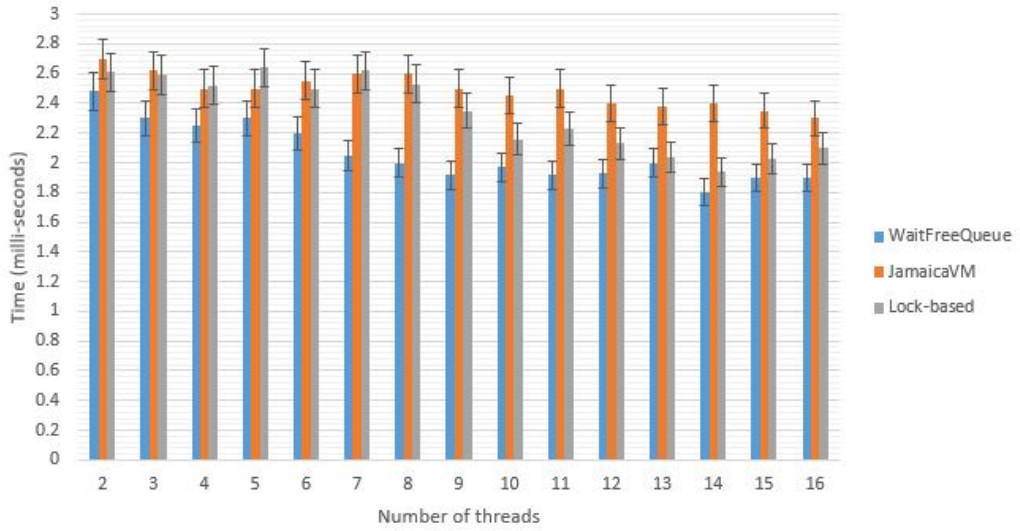


Figure 6.28: Response time for the enqueue operation, large image

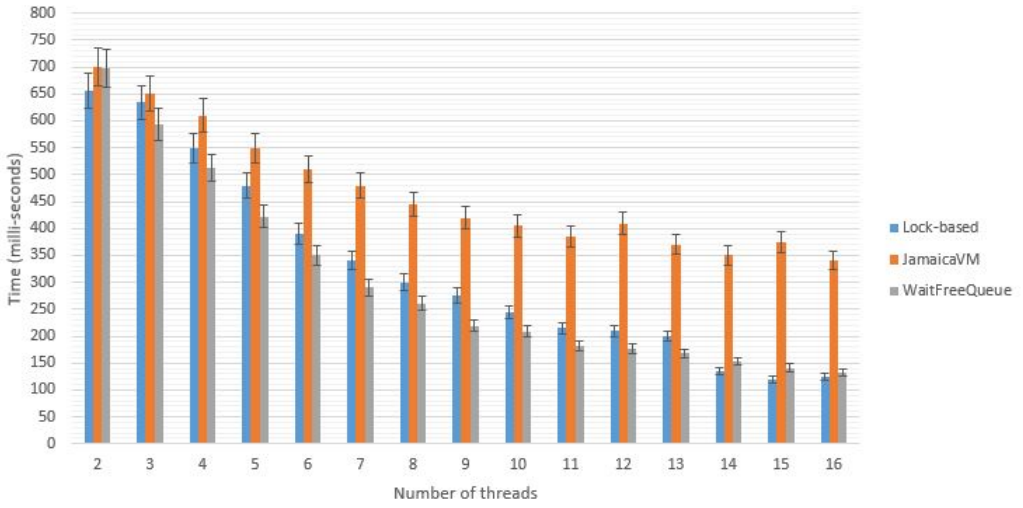


Figure 6.29: Response time for the dequeue operation, large image

6.6 Summary

In this chapter the proposed algorithm correctness and performance is evaluated using model checking, test cases, and a case study.

The model checking of the algorithm is used to ensure that the algorithm is correct from errors like deadlocks and data races. Java Pathfinder is used to model check the algorithm; JPF backtraces all possible execution paths and verifies that the algorithm is error free. The model checking is performed for 2 writer and 2 reader threads, and for 3 writer and 3 reader threads concurrently accessing the WaitFreeQueue. The results show that the algorithm is error free.

The test cases are used to test the behaviour of the algorithm under different conditions. For each of the test conditions, the behaviour of the algorithm is monitored to ensure that it is working correctly as designed. In all the test cases, the code is executed using JPF to further check for error free code when performing different enqueue and dequeue operations. In all the test cases, the WaitFreeQueue is initialised according to the test case, and a post check is used to ensure that the state of the queue is modified as expected during the execution.

The performance of the algorithm is evaluated to measure the time required to perform the enqueue and dequeue operations in average and in the worst case. In the average response time, the enqueue and dequeue operations are performed extensively on the queue, and the average time required to performed these operations are measured. In the worst case, the enqueue and dequeue operations are performed to reserve the last node in the queue,

this measures the worst case execution time when the enqueue and dequeue operations are required to perform the maximum number of CAS operations.

In the results of the performance evaluation, the average response time for the enqueue operation is less than 1 milli-seconds, while in the worst case the time required is nearly 7 milli-seconds. For the dequeue operation, in the average response time the time requires is less than 1 milli-seconds and in the worst case is nearly 4 milli seconds. This can be referred to that, in the dequeue operations a single CAS operation is required, while in the enqueue operation two CAS operations are required for a success operation.

To test the algorithm in real environment, an image processing application is developed and implemented using JamaicaVM RTSJ implementation. The algorithm is designed to build the histogram for a set of images, the images is selected such that the enqueue and dequeue operations is going to have significant difference in the work load for each image. For all the executions, the results of the histogram is compared to the standard histogram of the image been working on, to test for the correctness of the operations.

In the image processing cases study, the enqueue and dequeue operations is executed large number of times (nearly 50K times) to produce a reliable response time of the algorithm. The results of the algorithm are compared to a lock-based and JamaicaVM RTSJ WFQ implementations. In all cases, the WaitFreeQueue outperforms its lock-based and JamaicaVM RTSJ WFQ counterparts.

Chapter 7

Conclusion and future work

Multi-processor and multi-core systems are the inevitable future of computer systems. Within the foreseen future, a multi-core processor on an end user machine is expected to has 10 cores and the number of cores for high end servers is expected to reach 100. Accordingly, it is required that the applications are developed while fully exploiting the parallelism in order to utilise the computational power these systems introduce. Shared data structures are a key component in multi-threaded applications. Developing a shared data structure that can provide the required scalability is a challenging task.

The RTSJ introduces new features that enable developing multiprocessor and multi-core safe real-time applications. The features that the RTSJ introduce enable the programmers to develop shared data structures that are safe in multiprocessor environment and can be designed such that the multi-threaded application can scale as the number of threads scale.

7.1 Conclusions

This thesis introduces an algorithm that provides wait free guarantees for multi-threaded applications in multi-core and multi-processor environments. The algorithm is implemented as a FIFO queue data structure. The different features that the RTSJ introduces are utilised in order to satisfy the required spatial and time predictability constraints of real-time applications. The algorithm is designed for general purpose use, thus, it can be used to provide the communication between any writer and reader threads within any multi-threaded application.

To ensure the correctness behaviour of the algorithm, it is model checked using JPF. The model checking ensures that the interleaving of the threads execution does not results in any errors; like deadlock and data races. The time response of the algorithm is analysed to estimate the time required to perform the enqueue and dequeue methods.

The algorithm response time for the average time required to perform an enqueue and dequeue operation and the worst case scenario are performed. The results indicate that the algorithm is stable and time required to perform the enqueue and dequeue methods are bounded. A time analysis of the worst case time required to perform the operations relate to the shared data structure is performed. The analysis shows that the algorithm outperforms the lock-based and the JamaicaVM RTSJ WFQ. Also, the algorithm promises stable performance as the number of threads increases.

After the evaluation of the WaitFreeQueue algorithm, it was notices the following:

- **Enqueue operation always succeeds**: the WaitFreeQueue algorithm designed such that a writer thread accessing the queue succeeds in finishing the enqueue operation by searching the queue only once. In all the tests performed the enqueue operation succeeds in reserving a node in the queue from by searching only once. This enable using the algorithm in any multi-threaded application.

- **Number of retries**: for every enqueue operation the writer thread needs to reserve a node in the queue. The writer threads searches the queue for an empty node starting with the beginning of the queue. The writer thread keeps retrying with the next node until it succeeds in reserving an empty node. Although the number of retries depends on different factors and it might be possible for the same set of factors to yield different results when tested more than once, as it depends on the runtime environment. But it was noticed from the results and testing that in 90% times of the cases the writer threads are able to succeed in reserving a node before reaching the middle of the queue.

- **WaitFreeQueue vs. Lock-based and JamaicaVM RTSJ WFQ**: the WaitFreeQueue outperforms the lock-based and the JamaicaVM RTSJ WFQ. As the number of threads increases, the number of competing threads increase and the number of retries increase, hence, the WaitFreeQueue response time become closer to the lock-based, as each retry requires executing two costly CAS operation. This was the main reason for the decline of the average execution time in the WaiFreeQueue. Currently, the WaitFreeQueue algorithm code is being refined in order

to enhance the response time and decrease the need for performing the CAS operation.

7.2 Future work

The WaitFreeQueue algorithm represents a basic building block in any multi-threaded application. Further improvements are planned in order to increase the algorithm performance and enhance its response time. In addition, different variations of the algorithm and other basic data structure is being designed to provide a set of algorithms that satisfy the demand of the different multi-threaded applications. The planned future work as follows:

- **Memory requirements**: the memory requirements of the algorithm varies according to the number of writer and reader threads and the operations being performed by the application. However, the basic memory requirements of the algorithm can be analysed and estimated to ensure that the system is able to provide the required amount of memory ahead of applying the algorithm. The programmer should identify the memory requirement that is application dependent, for example, in the image histogram of the grey scale image it is required to have an integer array of 256 locations, if the image histogram is required for a colored image, the array size will be dependent on the color depth.

- **support standard Java, sporadic and aperiodic threads**: if it is required for standard Java, sporadic and aperiodic threads to access the WaitFreeQueue; the implication of their access must be carefully

investigated; as these threads might stall during the execution, which might affect the progress of other threads concurrently accessing the queue. The algorithm design should be reviewed, such as the size of the queue, to ensure that the execution of the real-time and NHRT threads is not affected. Furthermore, the usage of the memory areas should be reviewed such that it adheres to the RTSJ rules regarding the memory referencing.

- **variations of the WaitFreeQueue algorithm**: the current implementation of the WaitFreeQueue isolates the writer threads execution from the reader threads execution, and the data storage from the implementation. The implementation is loaded in immortal or scoped memory while each data item is stored in a different scoped memory area. However, in certain applications the use of the scoped memory areas to store the data item might be costly, as it is required for each enqueue and dequeue operation to enter a scoped memory area to produce and consume the data. If the data size is low, like an integer or a boolean, it is possible to boost the performance of the application by storing the data item locally in the same scoped memory area and avoid the time required to enter and exit the scoped memory area. The applications requirements can be used to derive the introduction of different variations of the WaitFreeQueue algorithm.

- **WaitFreeStack and WaitFreePool**: the most common shared data structure is the queue. Based on the design semantics of the WaitFreeQueue it is being investigated to introduce an algorithm implementation

that is based on the stack semantic, Last-In-First-Out (LIFO), and pool, where the enqueue and dequeue operations can be performed randomly without any order. However, both data structures intend to provide wait free guarantees.

- **Real-time properties**: during the evaluation, the real-time properties of the threads is adjusted to be suitable for the tasks being executed. For example, the time required for two threads to produce the histogram of the large image is relatively much higher than the time required for producing the histogram for the small image. Further evaluation is being designed such that the real-time threads fails to meet their real-time properties, such as deadline, are identified and handled. Further to that, evaluation for the enqueue and dequeue methods to be performed for the cases when the real-time properties of the threads might changed during run time; like priority and deadline.

Appendices

Appendix A

Atomic variables

Listing A.1: Boolean Atomic variable

```
1  public class AtomicBoolean {
2
3    public AtomicBoolean(boolean init) {
4      value = init;
5    }
6
7    public synchronized void set(boolean val) {
8      value = val;
9      if(value) becameTrue = true;
10   }
11
12   public synchronized boolean get() { return value; };
13
14
15   public synchronized boolean compareAndSet(boolean expect, boolean update) {
16     if (value == expect) {
17       value = update;
18       if(value) becameTrue = true;
19       return true;
20     } else return false;
21   }
22
```

```
23   public synchronized boolean wasTrue() {
24     return becameTrue;
25   }
26
27   private boolean value;
28   private boolean becameTrue = false;
29 }
```

Listing A.2: Integer Atomic variable

```
1  public class AtomicInteger {
2
3    public AtomicInteger(int init) {
4      value = init;
5      max = init;
6    }
7
8    public synchronized void set(int val) {
9      value = val;
10     if(value > max) max = value;
11   }
12
13   public synchronized int get() { return value; };
14
15   public synchronized int getMax() { return max; };
16
17   public synchronized int getAndSet(int newValue) {
18     int tmp = value;
19     value = newValue;
20     if(value > max) max = value;
21     return tmp;
22   }
```

```
23    public synchronized boolean compareAndSet(int expect, int update) {
24      if (value == expect) {
25        value = update;
26        if(value > max) max = value;
27        return true;
28      } else return false;
29    }
30
31    public synchronized int incrementAndGet() {
32      value++;
33      if(value > max) max = value;
34      return value;
35    }
36
37    public synchronized int decrementAndGet() {
38      value--;
39      return value;
40    }
41
42    private int value;
43    private int max;
44  }
```

Appendix B

JPF Model checking

Listing B.1: ReaderThread for JPF testing

```
1  public class ReaderThread extends Thread {
2          TheQueue localDataPool;
3                  public ReaderThread(TheQueue dPool) {
4                  this.localDataPool = dPool;
5          }
6          public void run() {
7                  localDataPool.dequeue();
8          }
9  }
```


Listing B.2: WriterThread for JPF testing

```
1  public class WriterThread extends Thread {
2          TheQueue localPool;
3
4          public WriterThread(TheQueue lPool) {
5                  this.localPool = lPool;
6          }
7          public void run(){
8            localPool.enqueue()
9          }
10 }
```

Listing B.3: Queue Node

```
1 public class QueueNode {
2   public final AtomicBoolean reserved;
3
4   public QueueNode(long size) {
5     reserved = new AtomicBoolean(false);
6   }
7 }
```

Listing B.4: The algorithm code

```
1  public class WaitFreeQueue extends Thread {
2          public void run(){
3                                  int numberOfReaders = 3;
4                                  int numberOfWriters = 3;
5                                  TheQueue imgDataPool = new TheQueue(numberOfReaders,
                                       numberOfWriters, 900000);
6                                  WriterThread wt1 = new WriterThread(imgDataPool);
7                                  WriterThread wt2 = new WriterThread(imgDataPool);
8                                  ReaderThread rt1 = new ReaderThread(imgDataPool);
9                                  ReaderThread rt2 = new ReaderThread(imgDataPool);
10                                 wt1.start();
11                                 wt2.start();
12                                 rt1.start();
13                                 rt2.start();
14         }
15         public static void main(String args[]){
16                 WaitFreeQueue wfq = new WaitFreeQueue();
17                 wfq.start();
18         }
19 }
```

Listing B.5: The algorithm code

```
1  public class TheQueue {
2          private final int queueSize;
3          private final QueueNode  scopedMemoryPool[];
4          private final AtomicInteger readList[];
5          private final AtomicInteger readListTail  = new  AtomicInteger(-1);
6          private final AtomicInteger readListHead  = new  AtomicInteger(-1);
7          private final AtomicInteger readListCount = new  AtomicInteger(0);
8
9          public TheQueue(int numberOfReaders, int numberOfWriters, long scopeSize){
10                 this.queueSize = 2 * (numberOfReaders + numberOfWriters);
11                 scopedMemoryPool = new QueueNode[queueSize];
12                 readList = new AtomicInteger[queueSize];
13                 readListInvalid = new int[queueSize];
14                 for(int i=0;i<queueSize;i++){
15                         scopedMemoryPool[i] = new QueueNode(scopeSize);
16                         readList[i] = new AtomicInteger(-1);
17                         readListInvalid[i] = -1;
18                 }
19         }
20
21         public boolean enqueue() {
22                 int reservedNode = -1;
```

```
23                int enqueueLocation;
24                int currentLocation;
25                boolean okTail=false;
26                int inV;
27                do {
28                        reservedNode++;
29                        if (reservedNode == queueSize-1)
30                                return false; // QUEUE IS FULL
31                } while(!scopedMemoryPool[reservedNode].reserved.compareAndSet(false, true));
32                do {
33                        currentLocation = readListHead.get();
34                        enqueueLocation = (currentLocation + 1) % queueSize;
35                        inV = readListInvalid[enqueueLocation]; //for ABA avoidance
36                        if (readListHead.compareAndSet(currentLocation, enqueueLocation)){
37                                readList[enqueueLocation].set(reservedNode);
38                                okTail = true;
39                        }
40                } while (!okTail);
41                int throwAway = readListCount.incrementAndGet();
42                assert throwAway > 0 : "atomic integer failed";
43                return true;
44        }
45
```

```
46          public boolean dequeue() {
47                  int dequeueLocation;
48                  int currentTail;
49                  int index;
50                  do {
51                          currentTail = readListTail.get();
52                          dequeueLocation = (currentTail+ 1) % queueSize;
53                          if (readList[dequeueLocation].get() == -1) {
54                                  return false;
55                          }
56                  } while (!readListTail.compareAndSet(currentTail, dequeueLocation));
57                  index = readList[dequeueLocation].get();
58                  scopedMemoryPool[index].reserved.set(false);
59                  readList[dequeueLocation].set(-1);
60
61                  return true;
62          }
63  }
```

Appendix C

JPF test cases

Listing C.1: The writer thread

```
1  public class WriterThread extends Thread {
2          TheQueue localPool;
3
4          public WriterThread(TheQueue lPool) {
5                  this.localPool = lPool;
6          }
7          public void run(){
8            localPool.enqueue()
9          }
10 }
```

Listing C.2: The reader thread

```
1  public class ReaderThread extends Thread {
2          TheQueue localDataPool;
3                  public ReaderThread(TheQueue dPool) {
4                  this.localDataPool = dPool;
5          }
6          public void run() {
7                  localDataPool.dequeue();
8          }
9  }
```

Listing C.3: The main class

```
1  public class WaitFreeQueue extends Thread {
2          public void run(){
3                                  int numberOfReaders = 3;
4                                  int numberOfWriters = 3;
5                                  TheQueue imgDataPool = new TheQueue(numberOfReaders,
                                       numberOfWriters, 900000);
6                                  WriterThread wt1 = new WriterThread(imgDataPool);
7                                  WriterThread wt2 = new WriterThread(imgDataPool);
8                                  ReaderThread rt1 = new ReaderThread(imgDataPool);
9                                  ReaderThread rt2 = new ReaderThread(imgDataPool);
10                                 wt1.start();
11                                 wt2.start();
12                                 rt1.start();
13                                 rt2.start();
14         }
15         public static void main(String args[]){
16                 WaitFreeQueue wfq = new WaitFreeQueue();
17                 wfq.start();
18         }
19 }
```

Listing C.4: The algorithm code

```
1  public class TheQueue {
2          private final int queueSize;
3          private final QueueNode  scopedMemoryPool[];
4          private final AtomicInteger readList[];
5          private final int readListInvalid[];
6          private final AtomicInteger readListTail  = new  AtomicInteger(-1);
7          private final AtomicInteger readListHead  = new  AtomicInteger(-1);
8          private final AtomicInteger readListCount = new  AtomicInteger(0);
9
10         //for testing purposes
11         int numberofdequeuefails = 0;
12         int numberofenqueuefails = 0;
13
14         public TheQueue(int numberOfReaders, int numberOfWriters, long scopeSize){
15                 this.queueSize = (2 * (numberOfReaders + numberOfWriters)) + numberOfWriters;
16                 scopedMemoryPool = new QueueNode[queueSize];
17                 readList = new AtomicInteger[queueSize];
18                 readListInvalid = new int[queueSize];
19                 for(int i=0;i<queueSize;i++){
20                         scopedMemoryPool[i] = new QueueNode(scopeSize);
21                         readList[i] = new AtomicInteger(-1);
22                         readListInvalid[i] = -1;
```

```
23                  }
24          }
25          public boolean enqueue() {
26                  int reservedNode = -1;
27                  int enqueueLocation;
28                  int currentLocation;
29                  boolean okTail=false;
30                  int inV;
31                  do {
32                          reservedNode++;
33                          if (reservedNode == queueSize-1) {
34                                  System.out.println("Test Failed, Full queue");
35                                  numberofenqueuefails++;
36                                  return false; // QUEUE IS FULL
37                          }
38                  } while(!scopedMemoryPool[reservedNode].reserved.compareAndSet(false, true));
39                  do {
40                          currentLocation = readListHead.get();
41                          enqueueLocation = (currentLocation + 1) % queueSize;
42                          inV = readListInvalid[enqueueLocation]; //for ABA avoidance
43                          if (readListHead.compareAndSet(currentLocation, enqueueLocation)){
44                                  readList[enqueueLocation].set(reservedNode);
45                                  okTail = true;
```

```
46                      }
47              } while (!okTail);
48              int throwAway = readListCount.incrementAndGet();
49              assert throwAway > 0 : "atomic integer failed";
50              return true;
51      }
52
53      public boolean dequeue() {
54              int dequeueLocation;
55              int currentTail;
56              int index;
57              do {
58                      currentTail = readListTail.get();
59                      dequeueLocation = (currentTail + 1) % queueSize;
60                      if (readList[dequeueLocation].get() == -1 && currentTail ==
                           readListTail.get()) {
61                              numberofdequeuefails++;
62                              System.out.println("Test failed, Returning False to dequeue
                                   operation");
63                              numberofdequeuefails++;
64                              return false;
65                      }
66              } while (!readListTail.compareAndSet(currentTail, dequeueLocation));
```

```
67              // perform some operation for some time, for testing reasons.
68              index = readList[dequeueLocation].get();
69          scopedMemoryPool[index].reserved.set(false);
70              readList[dequeueLocation].set(-1);
71              return true;
72          }
73          public boolean getnodestate(int i) {
74              return scopedMemoryPool[i].reserved.get();
75          }
76          public void printqueue(){
77              for(int i =0; i<queueSize;i++)
78                  System.out.print(scopedMemoryPool[i].reserved.get() + "␣␣");
79              for(int i =0; i<queueSize;i++)
80                  System.out.print(readList[i].get() + "␣␣");
81              System.out.println();
82          }
83          public void setnodestate(int i, int j) {
84              scopedMemoryPool[i].reserved.set(true);
85              readList[j].set(i);
86          }
87          public int getnofdeqfails(){
88              return numberofdequeuefails;
89          }
```

```
90          public int getnofenqfails(){
91                  return numberofdequeuefails;
92          }
93          public int getq(int i){
94                  return q[i];
95          }
96  }
```

C.1 Empty queue - three writers case

Listing C.5: The main class code

```
1  public class WaitFreeQueue extends Thread {
2          public void run(){
3                  int numberOfReaders = 0;
4                  int numberOfWriters = 3;
5                  int qsize = 2 * (numberOfReaders + numberOfWriters) + numberOfWriters;
6                  TheQueue waitFreeQueue = new TheQueue(numberOfReaders, numberOfWriters,
                       900000);
7
8                  // Pre-conditions for an empty queue, nothing is done.
9
10                 // starting and joining the threads
11                 WriterThread wt1 = new WriterThread(waitFreeQueue);
12                 WriterThread wt2 = new WriterThread(waitFreeQueue);
13                 WriterThread wt3 = new WriterThread(waitFreeQueue);
14                 wt1.start();
15                 wt2.start();
16                 wt3.start();
17                 try {
18                         wt1.join();
19                         wt2.join();
```

```
20                      wt3.join();
21              } catch (InterruptedException e) {
22                      // TODO Auto-generated catch block
23                      e.printStackTrace();
24              }
25
26              // Postcheck conditions
27              // test pass if the first three nodes contain data and all
28              // other nodes are empty
29              if (waitFreeQueue.getnodestate(0) == true &&
30                      waitFreeQueue.getnodestate(1) == true &&
31                      waitFreeQueue.getnodestate(2) == true)
32                      System.out.println("Test␣Passed");
33              else
34                      System.out.println("Test␣failed");
35              for(int i=3;i<qsize-1;i++)
36                      if (waitFreeQueue.getnodestate(i) == false)
37                              System.out.println("Test␣Passed");
38                      else
39                              System.out.println("Test␣failed");
40
41      }
42      public static void main(String args[]){
```

```
43                  WaitFreeQueue wfq = new WaitFreeQueue();
44                  wfq.start();
45          }
46  }
```

C.2 Empty queue - three readers case

Listing C.6: The main class code

```
1  public class WaitFreeQueue extends Thread {
2          public void run(){
3                  int numberOfReaders = 3;
4                  int numberOfWriters = 0;
5                  int qsize = 2 * (numberOfReaders + numberOfWriters) + numberOfWriters;
6                  TheQueue waitFreeQueue = new TheQueue(numberOfReaders, numberOfWriters,
                       900000);
7
8                  // Pre-conditions for an empty queue, all nodes are empty.
9
10                 // starting and joining the threads
11                 ReaderThread rt1 = new ReaderThread(waitFreeQueue);
12                 ReaderThread rt2 = new ReaderThread(waitFreeQueue);
```

```
13              ReaderThread rt3 = new ReaderThread(waitFreeQueue);
14              rt1.start();
15              rt2.start();
16              rt3.start();
17
18              //ensure that the threads finish execution before check the
19              //post conditions.
20              try {
21                      rt1.join();
22                      rt2.join();
23                      rt3.join();
24              } catch (InterruptedException e) {
25                      e.printStackTrace();
26              }
27
28              // Postcheck conditions
29              // test pass if all nodes are empty, one of the dequeue operations is
                   expected to fail
30              if(waitFreeQueue.getnodestate(0) == false &&
31                              waitFreeQueue.getnodestate(1) == false &&
32                              waitFreeQueue.getnodestate(2) == false &&
33                              waitFreeQueue.getnodestate(3) == false &&
34                              waitFreeQueue.getnodestate(4) == false &&
```

```
35                                  waitFreeQueue.getnodestate(5) == false &&
36                                  waitFreeQueue.getnofdeqfails()==3)
37                          System.out.println("Test Passed");
38                  else
39                          System.out.println("Test failed");
40          }
41          public static void main(String args[]){
42                  WaitFreeQueue wfq = new WaitFreeQueue();
43                  wfq.start();
44          }
45  }
```


C.3 Empty queue - tow readers and two writers case

Listing C.7: The algorithm code

```
1  public class WaitFreeQueue extends Thread {
2          public void run(){
3                  int numberOfReaders = 2;
4                  int numberOfWriters = 2;
5                  int qsize = 2 * (numberOfReaders + numberOfWriters) + numberOfWriters;
```

```
6               TheQueue waitFreeQueue = new TheQueue(numberOfReaders, numberOfWriters,
                    900000);
7
8               // Pre-conditions for an empty queue, all nodes are empty.
9
10              // starting and joining the threads
11              WriterThread wt1 = new WriterThread(waitFreeQueue);
12              WriterThread wt2 = new WriterThread(waitFreeQueue);
13              ReaderThread rt1 = new ReaderThread(waitFreeQueue);
14              ReaderThread rt2 = new ReaderThread(waitFreeQueue);
15              wt1.start();
16              wt2.start();
17              rt1.start();
18              rt2.start();
19
20              //ensure that the threads finish execution before check the
21              //post conditions.
22              try {
23                      wt1.join();
24                      wt2.join();
25                      rt1.join();
26                      rt2.join();
27              } catch (InterruptedException e) {
```

```
28                          e.printStackTrace();
29                  }
30
31                  // Postcheck conditions
32                  // test pass if all nodes are empty, one of the dequeue operations is
                        expected to fail
33                  if(waitFreeQueue.getnodestate(0) == false &&
34                                  waitFreeQueue.getnodestate(1) == false &&
35                                  waitFreeQueue.getnodestate(2) == false &&
36                                  waitFreeQueue.getnodestate(3) == false &&
37                                  waitFreeQueue.getnodestate(4) == false &&
38                                  waitFreeQueue.getnodestate(5) == false &&
39                                  waitFreeQueue.getnofdeqfails()==0)
40                          System.out.println("Test␣Passed");
41                  else
42                          System.out.println("Test␣failed");
43          }
44          public static void main(String args[]){
45                  WaitFreeQueue wfq = new WaitFreeQueue();
46                  wfq.start();
47          }
48  }
```

C.4 Partial empty queue - three writers case

Listing C.8: The main class code

```
1  public class WaitFreeQueue extends Thread {
2          public void run(){
3                  int numberOfReaders = 0;
4                  int numberOfWriters = 3;
5                  int qsize = 2 * (numberOfReaders + numberOfWriters) + numberOfWriters;
6                  TheQueue waitFreeQueue = new TheQueue(numberOfReaders, numberOfWriters,
                       900000);
7
8                  // Pre-conditions for a partial empty queue, the first three nodes
9                  // are assumed to have data, and their indexes added to readList.
10                 waitFreeQueue.enqueue();
11                 waitFreeQueue.enqueue();
12                 waitFreeQueue.enqueue();
13
14                 // starting and joining the threads
15                 WriterThread wt1 = new WriterThread(waitFreeQueue);
16                 WriterThread wt2 = new WriterThread(waitFreeQueue);
17                 WriterThread wt3 = new WriterThread(waitFreeQueue);
18                 wt1.start();
19                 wt2.start();
```

```
20                  wt3.start();
21                  try {
22                          wt1.join();
23                          wt2.join();
24                          wt3.join();
25                  } catch (InterruptedException e) {
26                          // TODO Auto-generated catch block
27                          e.printStackTrace();
28                  }
29
30                  // Postcheck conditions
31                  // test pass if the first six nodes contain data and all
32                  // other nodes are empty.
33                  if (waitFreeQueue.getnodestate(0) == true &&
34                          waitFreeQueue.getnodestate(1) == true &&
35                          waitFreeQueue.getnodestate(2) == true &&
36                          waitFreeQueue.getnodestate(3) == true &&
37                          waitFreeQueue.getnodestate(4) == true &&
38                          waitFreeQueue.getnodestate(5) == true)
39                          System.out.println("Test␣Passed");
40                  else
41                          System.out.println("Test␣failed");
42                  for(int i=6;i<qsize-1;i++)
```

```
43                          if (waitFreeQueue.getnodestate(i) == false)
44                                  System.out.println("Test Passed");
45                          else
46                                  System.out.println("Test failed");
47
48          }
49          public static void main(String args[]){
50                  WaitFreeQueue wfq = new WaitFreeQueue();
51                  wfq.start();
52          }
53  }
```


C.5 Partial empty queue - three readers case

Listing C.9: The main class code

```
1  public class WaitFreeQueue extends Thread {
2          public void run(){
3                  int numberOfReaders = 3;
4                  int numberOfWriters = 0;
5                  int qsize = 2 * (numberOfReaders + numberOfWriters) + numberOfWriters;
```

```
6           TheQueue waitFreeQueue = new TheQueue(numberOfReaders, numberOfWriters,
                900000);
7
8           // Pre-conditions for a partial empty queue.
9           waitFreeQueue.enqueue();
10          waitFreeQueue.enqueue();
11
12          // starting and joining the threads
13          ReaderThread rt1 = new ReaderThread(waitFreeQueue);
14          ReaderThread rt2 = new ReaderThread(waitFreeQueue);
15          ReaderThread rt3 = new ReaderThread(waitFreeQueue);
16          rt1.start();
17          rt2.start();
18          rt3.start();
19
20          //ensure that the threads finish execution before check the
21          //post conditions.
22          try {
23                  rt1.join();
24                  rt2.join();
25                  rt3.join();
26          } catch (InterruptedException e) {
27                  e.printStackTrace();
```

```
28      }
29
30      // Postcheck conditions
31      // test pass if all nodes are empty, one of the dequeue operations is
            expected to fail
32      if(waitFreeQueue.getnodestate(0) == false &&
33                  waitFreeQueue.getnodestate(1) == false &&
34                  waitFreeQueue.getnodestate(2) == false &&
35                  waitFreeQueue.getnodestate(3) == false &&
36                  waitFreeQueue.getnodestate(4) == false &&
37                  waitFreeQueue.getnodestate(5) == false &&
38                  waitFreeQueue.getnofdeqfails()==1)
39              System.out.println("Test Passed");
40          else
41              System.out.println("Test failed");
42      }
43      public static void main(String args[]){
44          WaitFreeQueue wfq = new WaitFreeQueue();
45          wfq.start();
46      }
47  }
```

C.6 Partial empty queue - tow readers and two writers case

Listing C.10: The algorithm code

```
1  public class WaitFreeQueue extends Thread {
2          public void run(){
3                  int numberOfReaders = 2;
4                  int numberOfWriters = 2;
5                  int qsize = 2 * (numberOfReaders + numberOfWriters) + numberOfWriters;
6                  TheQueue waitFreeQueue = new TheQueue(numberOfReaders, numberOfWriters,
                       900000);
7
8                  // Pre-conditions for a partial empty queue, all nodes are empty, nothing
                       done.
9                  waitFreeQueue.enqueue();
10                 waitFreeQueue.enqueue();
11                 waitFreeQueue.enqueue();
12                 waitFreeQueue.enqueue();
13
14                 // starting and joining the threads
15                 WriterThread wt1 = new WriterThread(waitFreeQueue);
16                 WriterThread wt2 = new WriterThread(waitFreeQueue);
17                 ReaderThread rt1 = new ReaderThread(waitFreeQueue);
18                 ReaderThread rt2 = new ReaderThread(waitFreeQueue);
```

```
19  wt1.start();
20  wt2.start();
21  rt1.start();
22  rt2.start();
23
24  //ensure that the threads finish execution before check the
25  //post conditions.
26  try {
27      wt1.join();
28      wt2.join();
29      rt1.join();
30      rt2.join();
31  } catch (InterruptedException e) {
32      e.printStackTrace();
33  }
34  // Postcheck conditions
35  // test pass if all nodes are empty, one of the dequeue operations is
        expected to fail
36  if(waitFreeQueue.getnodestate(0) == false &&
37          waitFreeQueue.getnodestate(1) == false &&
38          waitFreeQueue.getnodestate(2) == true &&
39          waitFreeQueue.getnodestate(3) == true &&
40          waitFreeQueue.getnodestate(4) == true &&
```

```
41                                  waitFreeQueue.getnodestate(5) == true &&
42                                  waitFreeQueue.getnodestate(6) == false &&
43                                  waitFreeQueue.getnodestate(7) == false &&
44                                  waitFreeQueue.getnodestate(8) == false &&
45                                  waitFreeQueue.getnodestate(9) == false &&
46                                  waitFreeQueue.getnofdeqfails()==0)
47                          System.out.println("Test Passed");
48                  else
49                          System.out.println("Test failed");
50          }
51          public static void main(String args[]){
52                  WaitFreeQueue wfq = new WaitFreeQueue();
53                  wfq.start();
54          }
55  }
```


C.7 Partial full queue - three writers case

Listing C.11: The main class code

```
1  public class WaitFreeQueue extends Thread {
2          public void run(){
```

```
3   int numberOfReaders = 0;
4   int numberOfWriters = 3;
5   int qsize = 2 * (numberOfReaders + numberOfWriters) + numberOfWriters;
6   TheQueue waitFreeQueue = new TheQueue(numberOfReaders, numberOfWriters,
        900000);
7
8   // Pre-conditions for a partial full queue, the queue is 9 nodes
9   // out of which 7 nodes are containing data, hence, one of the three writers
10  // is going to fail the enqueue operation.
11  waitFreeQueue.enqueue();
12  waitFreeQueue.enqueue();
13  waitFreeQueue.enqueue();
14  waitFreeQueue.enqueue();
15  waitFreeQueue.enqueue();
16  waitFreeQueue.enqueue();
17  waitFreeQueue.enqueue();
18
19  // starting and joining the threads
20  WriterThread wt1 = new WriterThread(waitFreeQueue);
21  WriterThread wt2 = new WriterThread(waitFreeQueue);
22  WriterThread wt3 = new WriterThread(waitFreeQueue);
23  wt1.start();
24  wt2.start();
```

```
25                  wt3.start();
26                  try {
27                          wt1.join();
28                          wt2.join();
29                          wt3.join();
30                  } catch (InterruptedException e) {
31                          // TODO Auto-generated catch block
32                          e.printStackTrace();
33                  }
34
35                  // Postcheck conditions
36                  // test pass if the all nodes are containing data.
37                  for(int i=0;i<qsize-1;i++)
38                          if (waitFreeQueue.getnodestate(i) == true)
39                                  System.out.println("Test␣Passed");
40                          else
41                                  System.out.println("Test␣failed");
42
43          }
44          public static void main(String args[]){
45                  WaitFreeQueue wfq = new WaitFreeQueue();
46                  wfq.start();
47          }
```

```
48 }
```

C.8 Partial full queue - three readers case

Listing C.12: The main class code

```
1  public class WaitFreeQueue extends Thread {
2          public void run(){
3                  int numberOfReaders = 3;
4                  int numberOfWriters = 0;
5                  int qsize = 2 * (numberOfReaders + numberOfWriters) + numberOfWriters;
6                  TheQueue waitFreeQueue = new TheQueue(numberOfReaders, numberOfWriters,
                       900000);
7
8                  // Pre-conditions for a partial full queue, most of the nodes contain data.
9                  waitFreeQueue.enqueue();
10                 waitFreeQueue.enqueue();
11                 waitFreeQueue.enqueue();
12                 waitFreeQueue.enqueue();
13
14                 // starting and joining the threads
15                 ReaderThread rt1 = new ReaderThread(waitFreeQueue);
```

```
16              ReaderThread rt2 = new ReaderThread(waitFreeQueue);
17              ReaderThread rt3 = new ReaderThread(waitFreeQueue);
18              rt1.start();
19              rt2.start();
20              rt3.start();
21
22              //ensure that the threads finish execution before check the
23              //post conditions.
24              try {
25                      rt1.join();
26                      rt2.join();
27                      rt3.join();
28              } catch (InterruptedException e) {
29                      e.printStackTrace();
30              }
31
32              // Postcheck conditions
33              // test pass if all nodes are empty except for node 3
34              if(waitFreeQueue.getnodestate(0) == false &&
35                              waitFreeQueue.getnodestate(1) == false &&
36                              waitFreeQueue.getnodestate(2) == false &&
37                              waitFreeQueue.getnodestate(3) == true &&
38                              waitFreeQueue.getnodestate(4) == false &&
```

```
39                                  waitFreeQueue.getnodestate(5) == false)
40                          System.out.println("Test Passed");
41                  else
42                          System.out.println("Test failed");
43          }
44          public static void main(String args[]){
45                  WaitFreeQueue wfq = new WaitFreeQueue();
46                  wfq.start();
47          }
48  }
```


C.9 Partial full queue - tow readers and two writers case

Listing C.13: The algorithm code

```
1  public class WaitFreeQueue extends Thread {
2          public void run(){
3                  int numberOfReaders = 2;
4                  int numberOfWriters = 2;
5                  int qsize = 2 * (numberOfReaders + numberOfWriters) + numberOfWriters;
6                  TheQueue waitFreeQueue = new TheQueue(numberOfReaders, numberOfWriters,
                       900000);
```

```
7
8                   // Pre-conditions for a partial empty queue, all nodes are empty, nothing
                        done.
9                   waitFreeQueue.enqueue();
10                  waitFreeQueue.enqueue();
11                  waitFreeQueue.enqueue();
12                  waitFreeQueue.enqueue();
13                  waitFreeQueue.enqueue();
14                  waitFreeQueue.enqueue();
15                  waitFreeQueue.enqueue();
16                  // starting and joining the threads
17                  WriterThread wt1 = new WriterThread(waitFreeQueue);
18                  WriterThread wt2 = new WriterThread(waitFreeQueue);
19                  ReaderThread rt1 = new ReaderThread(waitFreeQueue);
20                  ReaderThread rt2 = new ReaderThread(waitFreeQueue);
21                  wt1.start();
22                  wt2.start();
23                  rt1.start();
24                  rt2.start();
25
26                  //ensure that the threads finish execution before check the
27                  //post conditions.
28                  try {
```

```
29          wt1.join();
30          wt2.join();
31          rt1.join();
32          rt2.join();
33      } catch (InterruptedException e) {
34          e.printStackTrace();
35      }
36      // Postcheck conditions
37      // test pass if all nodes are empty, one of the dequeue operations is
            expected to fail
38      if(waitFreeQueue.getnodestate(0) == false &&
39              waitFreeQueue.getnodestate(1) == false &&
40              waitFreeQueue.getnodestate(2) == true &&
41              waitFreeQueue.getnodestate(3) == true &&
42              waitFreeQueue.getnodestate(4) == true &&
43              waitFreeQueue.getnodestate(5) == true &&
44              waitFreeQueue.getnodestate(6) == true &&
45              waitFreeQueue.getnodestate(7) == true &&
46              waitFreeQueue.getnodestate(8) == true &&
47              waitFreeQueue.getnodestate(9) == false &&
48              waitFreeQueue.getnofdeqfails()==0)
49          System.out.println("Test Passed");
50      else
```

```
51                          System.out.println("Test failed");
52          }
53          public static void main(String args[]){
54                  WaitFreeQueue wfq = new WaitFreeQueue();
55                  wfq.start();
56          }
57  }
```

C.10 Full queue - three writers case

Listing C.14: The main class code

```
1   public class WaitFreeQueue extends Thread {
2           public void run(){
3                   int numberOfReaders = 0;
4                   int numberOfWriters = 3;
5                   int qsize = 2 * (numberOfReaders + numberOfWriters) + numberOfWriters;
6                   TheQueue waitFreeQueue = new TheQueue(numberOfReaders, numberOfWriters,
                        900000);
7
8                   // Pre-conditions for a full queue,  7 nodes containing data items to be
                        consumed.
```

```
9           waitFreeQueue.enqueue();
10          waitFreeQueue.enqueue();
11          waitFreeQueue.enqueue();
12          waitFreeQueue.enqueue();
13          waitFreeQueue.enqueue();
14          waitFreeQueue.enqueue();
15          waitFreeQueue.enqueue();
16          waitFreeQueue.enqueue();
17          waitFreeQueue.enqueue();
18
19          // starting and joining the threads
20          WriterThread wt1 = new WriterThread(waitFreeQueue);
21          WriterThread wt2 = new WriterThread(waitFreeQueue);
22          WriterThread wt3 = new WriterThread(waitFreeQueue);
23          wt1.start();
24          wt2.start();
25          wt3.start();
26
27          //ensure that the threads finish execution before check the
28          //post conditions.
29          try {
30                  wt1.join();
31                  wt2.join();
```

```
32                      wt3.join();
33              } catch (InterruptedException e) {
34                      e.printStackTrace();
35              }
36
37              // Postcheck conditions
38              // test pass if the all nodes are containing data and the enqueue
39              // method fails three times.
40              if (waitFreeQueue.getnofenqueuefilas()==3)
41                      System.out.println("Test Passed");
42              for(int i=0;i<qsize-1;i++)
43                      if (waitFreeQueue.getnodestate(i) == true)
44                              System.out.println("Test Passed");
45                      else
46                              System.out.println("Test failed");
47
48      }
49      public static void main(String args[]){
50              WaitFreeQueue wfq = new WaitFreeQueue();
51              wfq.start();
52      }
53  }
```

C.11 Full queue - three readers case

Listing C.15: The main class code

```
1  public class WaitFreeQueue extends Thread {
2          public void run(){
3                  int numberOfReaders = 3;
4                  int numberOfWriters = 0;
5                  int qsize = 2 * (numberOfReaders + numberOfWriters) + numberOfWriters;
6                  TheQueue waitFreeQueue = new TheQueue(numberOfReaders, numberOfWriters,
                       900000);
7
8                  // Pre-conditions for a full queue, 7 nodes out of 9 contains data items to
                       be consumed
9                  waitFreeQueue.enqueue();
10                 waitFreeQueue.enqueue();
11                 waitFreeQueue.enqueue();
12                 waitFreeQueue.enqueue();
13                 waitFreeQueue.enqueue();
14                 waitFreeQueue.enqueue();
15
16                 // starting and joining the threads
17                 ReaderThread rt1 = new ReaderThread(waitFreeQueue);
18                 ReaderThread rt2 = new ReaderThread(waitFreeQueue);
```

```
19              ReaderThread rt3 = new ReaderThread(waitFreeQueue);
20              rt1.start();
21              rt2.start();
22              rt3.start();
23
24              //ensure that the threads finish execution before check the
25              //post conditions.
26              try {
27                      rt1.join();
28                      rt2.join();
29                      rt3.join();
30              } catch (InterruptedException e) {
31                      e.printStackTrace();
32              }
33
34              // Postcheck conditions
35              // test pass if  all nodes are containing data except for the first three
                   nodes
36              if(waitFreeQueue.getnodestate(0) == false &&
37                              waitFreeQueue.getnodestate(1) == false &&
38                              waitFreeQueue.getnodestate(2) == false)
39                      System.out.println("Test␣Passed");
40              else
```

```
41                          System.out.println("Test failed");
42                  for(int i=3;i<qsize-1;i++)
43                          if (waitFreeQueue.getnodestate(i) == true)
44                                  System.out.println("Test Passed");
45                          else
46                                  System.out.println("Test failed");
47          }
48          public static void main(String args[]){
49                  WaitFreeQueue wfq = new WaitFreeQueue();
50                  wfq.start();
51          }
52  }
```


C.12 Full queue - tow readers and two writers case

Listing C.16: The algorithm code

```
1  public class WaitFreeQueue extends Thread {
2          public void run(){
3                  int numberOfReaders = 2;
4                  int numberOfWriters = 2;
5                  int qsize = 2 * (numberOfReaders + numberOfWriters) + numberOfWriters;
```

```
6           TheQueue waitFreeQueue = new TheQueue(numberOfReaders, numberOfWriters,
                900000);
7
8           // Pre-conditions for a full queue, all nodes contain data items to be
                consumed.
9           waitFreeQueue.enqueue();
10          waitFreeQueue.enqueue();
11          waitFreeQueue.enqueue();
12          waitFreeQueue.enqueue();
13          waitFreeQueue.enqueue();
14          waitFreeQueue.enqueue();
15          waitFreeQueue.enqueue();
16          waitFreeQueue.enqueue();
17          waitFreeQueue.enqueue();
18          waitFreeQueue.enqueue();
19
20          // starting and joining the threads
21          WriterThread wt1 = new WriterThread(waitFreeQueue);
22          WriterThread wt2 = new WriterThread(waitFreeQueue);
23          ReaderThread rt1 = new ReaderThread(waitFreeQueue);
24          ReaderThread rt2 = new ReaderThread(waitFreeQueue);
25          wt1.start();
26          wt2.start();
```

```
27                  rt1.start();
28                  rt2.start();
29
30                  //ensure that the threads finish execution before check the
31                  //post conditions.
32                  try {
33                          wt1.join();
34                          wt2.join();
35                          rt1.join();
36                          rt2.join();
37                  } catch (InterruptedException e) {
38                          e.printStackTrace();
39                  }
40                  // Postcheck conditions
41                  // test pass if all nodes are empty, one of the dequeue operations is
                       expected to fail
42                  if(waitFreeQueue.getnodestate(0) == false &&
43                                  waitFreeQueue.getnodestate(1) == false &&
44                                  waitFreeQueue.getnodestate(2) == true &&
45                                  waitFreeQueue.getnodestate(3) == true &&
46                                  waitFreeQueue.getnodestate(4) == true &&
47                                  waitFreeQueue.getnodestate(5) == true &&
48                                  waitFreeQueue.getnodestate(6) == true &&
```

```
49                                  waitFreeQueue.getnodestate(7) == true &&
50                                  waitFreeQueue.getnodestate(8) == true &&
51                                  waitFreeQueue.getnodestate(9) == true &&
52                                  waitFreeQueue.getnofdeqfails()==0 &&
53                                  waitFreeQueue.getnofenqfails()==2)
54                          System.out.println("Test Passed");
55                  else
56                          System.out.println("Test failed");
57          }
58          public static void main(String args[]){
59                  WaitFreeQueue wfq = new WaitFreeQueue();
60                  wfq.start();
61          }
62  }
```

Appendix D

Case study code

D.1 JamaicaVM wait free queues

Listing D.1: JamaicaVM WaitFreeReadQueue

```
1 package javax.realtime;
2
3 public class WaitFreeReadQueue
4 {
5   final Object[] data;
```

```
6   final boolean notify;
7   volatile int next_write_at = 0;
8   volatile int next_read_at = 0;
9
10  int inc(int i)
11  {
12
13    if (i == this.data.length) {
14      i = 0;
15    }
16    return i;
17  }
18
19  static MemoryArea getBestMemArea(Runnable writer, Runnable reader, MemoryArea memory)
20    throws MemoryScopeException
21  {
22    if ((writer != null) && (!(writer instanceof Thread)) && (!(writer instanceof
          AbstractAsyncEventHandler))) {
23      throw new IllegalArgumentException("writer must be null, a Thread or an
            AbstractAsyncEventHandler");
24    }
25    if ((reader != null) && (!(reader instanceof Thread)) && (!(reader instanceof
          AbstractAsyncEventHandler))) {
```

```
26        throw new IllegalArgumentException("reader must be null, a Thread or an
              AbstractAsyncEventHandler");
27      }
28      if (memory != null)
29      {
30        if ((memory instanceof ScopedMemory))
31        {
32          if ((writer != null) && (!isAncestor(memory, getMemArea(writer)))) {
33            throw new MemoryScopeException("memory argument is incompatible with writer
                  thread");
34          }
35          if ((reader != null) && (!isAncestor(memory, getMemArea(reader)))) {
36            throw new MemoryScopeException("memory argument is incompatible with reader
                  thread");
37          }
38        }
39        return memory;
40      }
41      if ((reader != null) || (writer != null))
42      {
43        MemoryArea m1 = getMemArea(writer);
44        MemoryArea m2 = getMemArea(reader);
45        if (m1 == m2) {
```

```
46          return m1;
47        }
48        if ((m1 instanceof ImmortalMemory)) {
49          return m1;
50        }
51        if ((m2 instanceof ImmortalMemory)) {
52          return m2;
53        }
54        if (isAncestor(m1, m2)) {
55          return m1;
56        }
57        if (isAncestor(m2, m1)) {
58          return m2;
59        }
60        return ImmortalMemory.instance();
61      }
62      return ImmortalMemory.instance();
63    }
64
65    static MemoryArea getMemArea(Runnable t)
66    {
67      MemoryArea Result = null;
68      if ((t instanceof RealtimeThread)) {
```

```
69        Result = ((RealtimeThread)t).parameters().originalArea_;
70      }
71      if ((t instanceof AbstractAsyncEventHandler)) {
72        Result = ((AbstractAsyncEventHandler)t).parameters.originalArea_;
73      }
74      if (Result == null) {
75        Result = HeapMemory.instance();
76      }
77      return Result;
78    }
79
80    static boolean isAncestor(MemoryArea m1, MemoryArea m2)
81    {
82      MemoryArea p = m2;
83      while (p != null)
84      {
85        if (p == m1) {
86          return true;
87        }
88        if ((p instanceof ScopedMemory)) {
89          p = ((ScopedMemory)p).getParent();
90        } else {
91          p = null;
```

```
92        }
93      }
94      return false;
95    }
96
97    public WaitFreeReadQueue(int maximum, boolean notify)
98      throws IllegalArgumentException
99    {
100     this(null, null, maximum, ImmortalMemory.instance(), notify);
101   }
102
103   public WaitFreeReadQueue(int maximum, MemoryArea memory, boolean notify)
104     throws IllegalArgumentException
105   {
106     this(null, null, maximum, memory, notify);
107     if (memory == null) {
108       throw new IllegalArgumentException("Memory argument is null.");
109     }
110   }
111
112   public WaitFreeReadQueue(Runnable writer, Runnable reader, int maximum, MemoryArea memory)
113     throws IllegalArgumentException, MemoryScopeException
114   {
```

```
115       this(writer, reader, maximum, memory, false);
116     }
117
118     public WaitFreeReadQueue(Runnable writer, Runnable reader, int maximum, MemoryArea memory,
            boolean notify)
119       throws IllegalArgumentException, MemoryScopeException
120     {
121       if (maximum < 0) {
122         throw new IllegalArgumentException("maximum argument must be larger than 0: " +
              maximum);
123       }
124       memory = getBestMemArea(writer, reader, memory);
125       this.data = ((Object[])memory.newArray(Object.class, maximum + 1));
126       this.notify = notify;
127       this.next_read_at = 0;
128       this.next_write_at = 0;
129     }
130
131     public void clear()
132     {
133       this.next_read_at = this.next_write_at;
134     }
135
```

```
136   public boolean isEmpty()
137   {
138     return this.next_read_at == this.next_write_at;
139   }
140
141   public boolean isFull()
142   {
143     return inc(this.next_write_at) == this.next_read_at;
144   }
145
146   public Object read()
147   {
148     boolean needToNotify = isFull();
149     Object Result = null;
150     if (!isEmpty())
151     {
152       Result = this.data[this.next_read_at];
153       this.next_read_at = inc(this.next_read_at);
154     }
155     if (needToNotify) {
156       synchronized (this.data)
157       {
158         this.data.notify();
```

```
159         }
160       }
161       return Result;
162     }
163
164     public int size()
165     {
166       int n = this.next_write_at - this.next_read_at;
167       if (n < 0) {
168         n += this.data.length;
169       }
170       return n;
171     }
172
173     public void waitForData()
174       throws InterruptedException, UnsupportedOperationException
175     {
176       if (!this.notify) {
177         throw new UnsupportedOperationException();
178       }
179       synchronized (this.data)
180       {
181         while (isEmpty()) {
```

```
182           this.data.wait();
183         }
184       }
185     }
186
187     public void write(Object object)
188       throws InterruptedException
189     {
190       synchronized (this.data)
191       {
192         while (isFull()) {
193           this.data.wait();
194         }
195         this.data[this.next_write_at] = object;
196         this.next_write_at = inc(this.next_write_at);
197         if (this.notify) {
198           this.data.notify();
199         }
200       }
201     }
202 }
```

Listing D.2: JamaicaVM WaitFreeWriteQueue

```
1  package javax.realtime;
2
3  public class WaitFreeWriteQueue
4  {
5    final Object[] data;
6    volatile int next_write_at = 0;
7    volatile int next_read_at = 0;
8
9    int inc(int i)
10   {
11
12     if (i == this.data.length) {
13       i = 0;
14     }
15     return i;
16   }
17
18   int dec(int i)
19   {
20
21     if (i < 0) {
22       i += this.data.length;
```

```
23      }
24      return i;
25    }
26
27    public WaitFreeWriteQueue(int maximum)
28      throws IllegalArgumentException
29    {
30      this(null, null, maximum, ImmortalMemory.instance());
31    }
32
33    public WaitFreeWriteQueue(int maximum, MemoryArea memory)
34      throws IllegalArgumentException
35    {
36      this(null, null, maximum, memory);
37      if (memory == null) {
38        throw new IllegalArgumentException("Memory argument is null.");
39      }
40    }
41
42    public WaitFreeWriteQueue(Runnable writer, Runnable reader, int maximum, MemoryArea memory)
43      throws IllegalArgumentException, MemoryScopeException
44    {
45      if (maximum < 0) {
```

```
46        throw new IllegalArgumentException("maximum argument must be larger than 0: " +
              maximum);
47      }
48      memory = WaitFreeReadQueue.getBestMemArea(writer, reader, memory);
49      this.data = ((Object[])memory.newArray(Object.class, maximum + 1));
50      this.next_read_at = 0;
51      this.next_write_at = 0;
52    }
53
54    public void clear()
55    {
56      this.next_write_at = this.next_read_at;
57    }
58
59    public boolean isEmpty()
60    {
61      return this.next_read_at == this.next_write_at;
62    }
63
64    public boolean isFull()
65    {
66      return inc(this.next_write_at) == this.next_read_at;
67    }
```

```
68
69  public Object read()
70    throws InterruptedException
71  {
72    Object Result = null;
73    synchronized (this.data)
74    {
75      while (isEmpty()) {
76        this.data.wait();
77      }
78      Result = this.data[this.next_read_at];
79      this.next_read_at = inc(this.next_read_at);
80    }
81    return Result;
82  }
83
84  public int size()
85  {
86    int n = this.next_write_at - this.next_read_at;
87    if (n < 0) {
88      n += this.data.length;
89    }
90    return n;
```

```
91    }
92
93    public boolean force(Object object)
94      throws MemoryScopeException
95    {
96      if (isFull()) {
97        synchronized (this.data)
98        {
99          if (isFull())
100         {
101           int last = dec(this.next_write_at);
102           this.data[last] = object;
103         }
104         return true;
105       }
106     }
107     write(object);
108     return false;
109   }
110
111   public boolean write(Object object)
112   {
113     boolean needToNotify = isEmpty();
```

```
114     boolean Result;
115     boolean Result;
116     if (isFull())
117     {
118       Result = false;
119     }
120     else
121     {
122       this.data[this.next_write_at] = object;
123       this.next_write_at = inc(this.next_write_at);
124       Result = true;
125     }
126     if (needToNotify) {
127       synchronized (this.data)
128       {
129         this.data.notify();
130       }
131     }
132     return Result;
133   }
134 }
```

Listing D.3: JamaicaVM WaitFreeDequeue

```
1  package javax.realtime;
2
3  import com.aicas.jamaica.lang.Wait;
4
5  /**
6   * @deprecated
7   */
8  public class WaitFreeDequeue
9  {
10   final Object[] data;
11   volatile int next_write_at = 0;
12   volatile int next_read_at = 0;
13
14   int inc(int i)
15   {
16
17     if (i == this.data.length) {
18       i = 0;
19     }
20     return i;
21   }
22
```

```
23    int dec(int i)
24    {
25
26      if (i < 0) {
27        i += this.data.length;
28      }
29      return i;
30    }
31
32    /**
33     * @deprecated
34     */
35    public WaitFreeDequeue(Runnable writer, Runnable reader, int maximum, MemoryArea memory)
36      throws IllegalArgumentException, MemoryScopeException
37    {
38      if (maximum < 0) {
39        throw new IllegalArgumentException("maximum argument must be larger than 0: " +
              maximum);
40      }
41      memory = WaitFreeReadQueue.getBestMemArea(writer, reader, memory);
42      this.data = ((Object[])memory.newArray(Object.class, maximum + 1));
43      this.next_read_at = 0;
44      this.next_write_at = 0;
```

```
45    }
46
47    /**
48     * @deprecated
49     */
50    boolean isEmpty()
51    {
52      return this.next_read_at == this.next_write_at;
53    }
54
55    /**
56     * @deprecated
57     */
58    boolean isFull()
59    {
60      return inc(this.next_write_at) == this.next_read_at;
61    }
62
63    /**
64     * @deprecated
65     */
66    public Object nonBlockingRead()
67    {
```

```
68      boolean needToNotify = isFull();
69      Object Result = null;
70      if (!isEmpty())
71      {
72        Result = this.data[this.next_read_at];
73        this.next_read_at = inc(this.next_read_at);
74      }
75      if (needToNotify) {
76        synchronized (this.data)
77        {
78          this.data.notify();
79        }
80      }
81      return Result;
82    }
83
84    /**
85     * @deprecated
86     */
87    public void blockingWrite(Object object)
88    {
89      synchronized (this.data)
90      {
```

```
91        while (isFull()) {
92          Wait.wait(this.data);
93        }
94        boolean needToNotify = isEmpty();
95        this.data[this.next_write_at] = object;
96        this.next_write_at = inc(this.next_write_at);
97        if (needToNotify) {
98          this.data.notify();
99        }
100     }
101   }
102
103   /**
104    * @deprecated
105    */
106   public boolean nonBlockingWrite(Object object)
107   {
108     boolean needToNotify = isEmpty();
109     boolean Result;
110     boolean Result;
111     if (isFull())
112     {
113       Result = false;
```

```
114     }
115     else
116     {
117       this.data[this.next_write_at] = object;
118       this.next_write_at = inc(this.next_write_at);
119       Result = true;
120     }
121     if (needToNotify) {
122       synchronized (this.data)
123       {
124         this.data.notify();
125       }
126     }
127     return Result;
128   }
129
130   /**
131   * @deprecated
132   */
133   public Object blockingRead()
134   {
135     Object Result = null;
136     synchronized (this.data)
```

```
137       {
138         while (isEmpty()) {
139           Wait.wait(this.data);
140         }
141         Result = this.data[this.next_read_at];
142         this.next_read_at = inc(this.next_read_at);
143       }
144       return Result;
145     }
146
147     /**
148      * @deprecated
149      */
150     public boolean force(Object object)
151       throws MemoryScopeException
152     {
153       if (isFull()) {
154         synchronized (this.data)
155         {
156           if (isFull())
157           {
158             int last = dec(this.next_write_at);
159             this.data[last] = object;
```

```
160             }
161             return true;
162           }
163         }
164         nonBlockingWrite(object);
165         return false;
166       }
167     }
```


D.2 WaitFreeQueue algorithm

Listing D.4: image distribution thread

```
1 import java.awt.image.BufferedImage;
2 import java.awt.image.DataBufferByte;
3 import java.awt.image.WritableRaster;
4 import java.io.File;
5 import java.io.IOException;
6 import javax.imageio.*;
7 import javax.realtime.*;
8 import java.sql.Timestamp;
9 import java.util.Date;
```

```
10 public class imageDistThread extends RealtimeThread {
11         // local variables
12         WaitFreeQueue localPool; // reference to the imageDataPool
13         byte[] imgArray; // reference to the image
14         int[] segmentNumber = new int[1];
15         int numberOfProcessingThreads;
16         // the segment size, changes according to the number of threads
17         int segmentSize=4480438;
18
19         public imageDistThread(int numberOfActiveThreads, WaitFreeQueue lPool,
               PriorityParameters pri, PeriodicParameters per, String name) {
20                 super(pri, per);
21                 this.setName(name);
22                 this.localPool = lPool;
23                 this.numberOfProcessingThreads = numberOfActiveThreads;
24         }
25
26         // prepare the image for processing, change the image to a binary version
27         public void prepImage(){
28                 BufferedImage bImage = null;
29                 try {
30                         bImage = ImageIO.read(new File("large.jpg"));
31                 } catch (IOException e) {
```

```
32                      // TODO Auto-generated catch block
33                      e.printStackTrace();
34              }
35              WritableRaster raster = bImage.getRaster();
36              DataBufferByte data   = (DataBufferByte) raster.getDataBuffer();
37              imgArray = data.getData();
38      }
39
40      // runnable object to enqueue the segment details in the imageDataPool
41      Runnable r = new Runnable() {
42              int[] sNumber = segmentNumber;
43              int sSize = segmentSize;
44              byte[] pixelArray = imgArray;
45              public void run(){
46                      LTMemory myMem = (LTMemory) RealtimeThread.getCurrentMemoryArea();
47                      PinData pthread = new PinData();
48                      pthread.start();
49                      // store the segment details
50                      pthread.setSegmentDetails(imgArray, sNumber,  sSize);
51                      myMem.setPortal(pthread);
52              }
53      };
54
```

```
55          public void run(){
56                  prepImage();
57                  boolean result;
58                  segmentNumber[0] = 0;
59                  int counter = 0;
60                  // outer loop; repeat the same process 50K times.
61                  do {
62                          // inner loop; enqueue the segments details in the imageDataPool
63                          do {
64                                  boolean throwAway = waitForNextPeriod();
65                                  if (localPool.enqueue(r)) {
66                                          segmentNumber[0]++;
67                                  }
68                          } while(segmentNumber[0]<numberOfProcessingThreads);
69                          segmentNumber[0] = 0;
70                          counter++;
71                  } while(counter<50000);
72          }
73  }
```

Listing D.5: Pin Data thread

```
1  import java.util.*;
2  import javax.realtime.*;
3  public class PinData extends RealtimeThread {
4          int[] pixelcount = new int[256];
5          int startOfSegment;
6          int endOfSegment;
7          byte[] imageArrayRef;
8          // The constructor
9          public PinData(){
10                 int maxPrio = PriorityScheduler.instance().getMaxPriority();
11                 RealtimeThread t = RealtimeThread.currentRealtimeThread();
12                 ((PriorityParameters) t.getSchedulingParameters()).setPriority(maxPrio);
13         }
14
15         // Copy the segment of the image to the scoped memory area
16         public void setSegmentDetails(byte[] imgArr, int[] segmentNumber, int segmentSize){
17                 this.imageArrayRef = imgArr;
18                 this.startOfSegment = segmentNumber[0] * segmentSize;
19                 this.endOfSegment = startOfSegment + segmentSize;
20         }
21
22         // return the count
```

```
23          public int getpixelCount(int index){
24                  return pixelcount[index];
25          }
26
27          // perform the count process
28          public void pCount(){
29                  int index;
30                  int value;
31                  for (index=startOfSegment;index<endOfSegment; index++){
32                          value = (int) imageArrayRef[index] & 0xff;
33                          pixelcount[value]++;
34                  }
35          }
36
37
38          // start the thread
39          public void run(){
40                  try {
41                          sleep(9999999);
42                  } catch (InterruptedException e) {}
43          }
44  }
```

Listing D.6: Pixel counting thread

```
1  import java.util.*;
2  import javax.realtime.*;
3  public class pixelCountingThread extends RealtimeThread {
4          imageContentPool localDataPool;
5          imageContentPool localCountPool;
6          int[] partialCount = new int[256];
7          int imgIndex;
8          String tName;
9
10         public pixelCountingThread(imageContentPool dPool, imageContentPool cPool,
               PriorityParameters pri, PeriodicParameters per, String name) {
11                 super(pri, per);
12                 this.setName(name);
13                 tName = name;
14                 this.localDataPool = dPool;
15                 this.localCountPool = cPool;
16         }
17                 // executable to enqueue the partial histogram in the imageCountPool
18                 Runnable ren = new Runnable() {
19                 int[] temp = partialCount;
20                 public void run(){
21                         LTMemory myMem = (LTMemory) RealtimeThread.getCurrentMemoryArea();
```

```
22                      PinCount pthread = new PinCount();
23                      pthread.start();
24                      pthread.setPixelCount(temp);
25                      myMem.setPortal(pthread);
26              }
27      };
28      // executable required to read the segment details from the imageDataPool
29      Runnable rde = new Runnable(){
30              int[] temp = partialCount;
31              int i;
32              public void run(){
33                      LTMemory myMem = (LTMemory) RealtimeThread.getCurrentMemoryArea();
34                      PinData pthread = (PinData) myMem.getPortal();
35                      if (pthread == null)
36                              return;
37                      }
38                      else {
39                              pthread.pCount();
40                              for(i=0;i<256;i++){
41                                      temp[i] = pthread.getpixelCount(i);
42                              }
43                              pthread.interrupt();
44                      }
```

```
45                  }
46          };
47
48          public void run() {
49                  int counter=0;
50                  do {
51                          boolean throwAway = waitForNextPeriod();
52                          // try to read a segment details from the imageDataPool
53                          if(localDataPool.dequeue(rde)) {
54                                  // if succeed, enqueue the produced histogram in the
                                       imageCountPool
55                                  localCountPool.enqueue(ren);
56                                  counter++;
57                          }
58                          // the counter limit varies according to the number of threads
59                  } while (counter<25000);
60          }
61  }
```

Listing D.7: Image Histogram application

```
1  import javax.realtime.*;
2  public class imageHistoSim extends RealtimeThread {
3
4          public imageHistoSim() {
5                  super();
6          }
7
8          public void run(){
9                  ImmortalMemory.instance().executeInArea(new Runnable() {
10                         public void run(){
11                                 int numberOfReaders = 2;
12                                 int numberOfWriters = 1;
13                                 int prip = PriorityScheduler.instance().getMinPriority();
14                                 RelativeTime start = new RelativeTime(10 /* ms */, 0 /* ns
                                       */);
15
16                                 // parameters for imageDistribution and imageCountCollection
                                       threads
17                                 PriorityParameters pri = new PriorityParameters(prip + 4);
18                                 RelativeTime perod1 = new RelativeTime(50 /* ms */, 0 /* ns
                                       */);
```

```
19          RelativeTime deadline = new RelativeTime(50 /* ms */, 0 /*
                ns */);
20          PeriodicParameters period =     new
                PeriodicParameters(start, perod1, deadline, null, null,
                null);
21
22          // parameters for pixelCountingThreads
23          PriorityParameters pri1 = new PriorityParameters(prip + 4);
24          RelativeTime per1 = new RelativeTime(50 /* ms */, 0 /* ns
                */);
25          RelativeTime dline = new RelativeTime(50 /* ms */, 0 /* ns
                */);
26          PeriodicParameters period1 =    new
                PeriodicParameters(start, per1, dline, null, null, null);
27
28          // pool for the image segments
29          WaitFreeQueue imgDataPool = new
                WaitFreeQueue(numberOfReaders, numberOfWriters, 900000);
30
31
32          // pool for the pixel count
33          WaitFreeQueue imgCountPool = new
                WaitFreeQueue(numberOfWriters, numberOfReaders, 590000);
```

```
34
35                  // thread to segment the image
36                  imageDistThread imgDistThread = new
                        imageDistThread(numberOfReaders, imgDataPool, pri,
                        period, "ImageDistributionThread");
37
38
39                  // threads to perform the pixel counting
40                  pixelCountingThread processingThread1 = new
                        pixelCountingThread(imgDataPool, imgCountPool, pri1,
                        period1, "pixelCountingThread1");
41                  pixelCountingThread processingThread2 = new
                        pixelCountingThread(imgDataPool, imgCountPool, pri1,
                        period1, "pixelCountingThread2");
42
43                  // thread to collect the counts
44                  countCollectionThread countCollectionThread = new
                        countCollectionThread(imgCountPool, pri1, period1,
                        "countCollectionThread");
45
46                  //starting the threads
47                  imgDistThread.start();
48                  processingThread1.start();
```

```
49                processingThread2.start();
50                countCollectionThread.start();
51            }
52        });
53    }
54    public static void main(String args[]){
55        imageHistoSim ihg = new imageHistoSim();
56        ihg.start();
57    }
58 }
```

Listing D.8: Pin thread example

```
1  import javax.realtime.*;
2  public class PinThread extends RealtimeThread {
3          // local variables, if reuquired
4          // The constructor, if reuquired
5          public PinThread(){
6          }
7          // Produce data method
8          public void produceData(){
9                  // The code required to store the data in the scoped memory backing store.
10         }
11         // Consume the data
12         public void consumeData(){
13                 // The code required to consume the data stored in the scoped memory backing
                      store.
14         }
15
16         public void run(){
17                 // The thread performs no actions, the reader thread terminates this thread
                      when data consumptions is finished.
18         }
19 }
```

Listing D.9: Writer thread example

```
1  import javax.realtime.*;
2  public class WriterThread extends RealtimeThread {
3          // local variables, the variables required to perform the data production.
4          WaitFreeQueue localPool;
5
6          // the constructor
7          public WriterThread(WaitFreeQeueue wfq, PriorityParameters pri, PeriodicParameters
               per, String name) {
8                  super(pri, per);
9                  this.setName(name);
10                 this.localPool = wfq;
11         }
12
13     // the runnable object, should be designed to\\
14         // perform the data production.
15         Runnable r = new Runnable() {
16                 // local varaibles that are required during the data production process.
17
18                 public void run(){
19                         // identify the scoped memory area
20                         LTMemory myMem = (LTMemory) RealtimeThread.getCurrentMemoryArea();
21                         // PinThread supposed to simulate the pinnable scoped memory area.
```

```
22                          PinThread pthread = new PinThread();
23                          pthread.start();
24
25                          // store the data in the PinThread, this is application
26                          // dependent, the programmer should identify the type of the data.
27                          pthread.produceData();
28                          // link the PinThread to the
29                          myMem.setPortal(pthread);
30                  }
31          };
32
33          public void run(){
34                  // accessing the enqueue method
35                  localPool.enqueue(r);
36                  // after finshing the data production, wait for the next period
37                  // the writer thread assumed to access the enqueue method once every period,
38                  //  in order to estimate the data production rate.
39                  boolean throwAway = waitForNextPeriod();
40
41  }
```

Listing D.10: Reader thread example

```
1  import javax.realtime.*;
2  public class ReaderThread extends RealtimeThread {
3          // local variables
4          WaitFreeQueue localPool;
5
6          // The constructor
7          public ReaderThread(WaitFreeQueue lPool, PriorityParameters pri, PeriodicParameters
               per, String name) {
8                  super(pri, per);
9                  this.setName(name);
10                 this.localPool = lPool;
11         }
12
13         Runnable r = new Runnable() {
14                 // local varaibles that are required during the data production process.
15
16                 public void run(){
17                         // identify the scoped memory area
18                         LTMemory  myMem = (LTMemory) RealtimeThread.getCurrentMemoryArea();
19
20                         // access the portal of the scoped memory backing store
21                         PinThread pthread =  (PinThread) myMem.getPortal();
```

```
22
23                          // Read the stored data, the code should be developed here.
24                          pthread.consumeData();
25
26                          // Interrupt the PinThread, when the reader thread leaves
27                          // the scoped memory area RTSJ reclaims the allocated memory.
28                          pthread.interrupt();
29                  }
30          };
31
32          public void run(){
33                  // Access the dequeue method
34                  localPool.dequeue(r)
35                  // The reader thread supposed to access the dequeue method once
36                  // every period, in order to estimate the data consumption rate.
37                  boolean throwAway = waitForNextPeriod();
38          }
39  }
```

Listing D.11: Enqueue Method

```
1         public boolean enqueue(Runnable r) {
2                 // Enqueue method variables
3                 int reservedNode = -1;
4                 int enqueueLocation;
5                 int currentLocation;
6                 boolean dataStrucUpdated=false;
7                 int inV;
8
9                 int maxPrio = PriorityScheduler.instance().getMaxPriority();
10                RealtimeThread rtThread = RealtimeThread.currentRealtimeThread();
11                int defPrio = ((PriorityParameters)
                     rtThread.getSchedulingParameters()).getPriority();
12
13                ((PriorityParameters)
                     rtThread.getSchedulingParameters()).setPriority(maxPrio);
14                do {
15                        reservedNode++;
16                        if (reservedNode == queueSize-1)
17                                return false; // QUEUE IS FULL
18                } while(!scopedMemoryPool[reservedNode].reserved.compareAndSet(false, true));
19                ((PriorityParameters)
                     rtThread.getSchedulingParameters()).setPriority(defPrio);
```

```
20
21                scopedMemoryPool[reservedNode].SMemory.enter(r);
22
23                ((PriorityParameters)
                     rtThread.getSchedulingParameters()).setPriority(maxPrio);
24                do {
25                        currentLocation = readListHead.get();
26                        enqueueLocation = (currentLocation + 1) % queueSize;
27                        inV = readListInvalid[enqueueLocation]; // ABA avoidance mechanism
28                if (readListHead.compareAndSet(currentLocation, enqueueLocation)){
29                readList[enqueueLocation].set(reservedNode);
30                                dataStrucUpdated = true;
31                        }
32                } while (!dataStrucUpdated);
33                ((PriorityParameters)
                     rtThread.getSchedulingParameters()).setPriority(defPrio);
34
35                assert throwAway > 0 : "atomic integer failed";
36                return true;
37        }
```

Listing D.12: Simple multi-threaded application

```
1  import javax.realtime.*;
2  public class ApplicationSample extends RealtimeThread {
3
4          public ApplicationSample() {
5                  super();
6          }
7
8          public void run(){
9                  // initiate the queue in the Immortal memory
10                 ImmortalMemory.instance().executeInArea(new Runnable() {
11                         public void run(){
12                                 // define the number of threads concurrently accessing the
                                       queue.
13                                 int numberOfReaders = 2;
14                                 int numberOfWriters = 2;
15
16                                 // define the realtime properties.
17                                 int prip = PriorityScheduler.instance().getMinPriority();
18                                 RelativeTime start = new RelativeTime(10 /*ms*/, 0 /*ns*/);
19                                 // Set the priority of the realtime thread, here it is
20                                 // set to minimum priority + 4
21                                 PriorityParameters pri = new PriorityParameters(prip + 4);
```

```
22                  RelativeTime deadline = new RelativeTime(50/*ms*/,0/*ns*/);
23                  PeriodicParameters period =     new
                        PeriodicParameters(start, perod1, deadline, null, null,
                        null);
24
25                  // The waifree queue
26                  WaitFreeQueue wfq = new WaitFreeQueue(numberOfReaders,
                        numberOfWriters, 900000);
27
28
29                  // Writer threads
30                  WriterThread Tw1 = new WriterThread(wfq, pri, period, "Tw1");
31                  WriterThread Tw2 = new WriterThread(wfq, pri, period, "Tw2");
32
33                  // Reader threads
34                  ReaderThread Tr1 = new ReaderThread(wfq, pri, period, "Tr1");
35                  WriterThread Tr2 = new WriterThread(wfq, pri, period, "Tr2");
36              }
37
38              // start the threads
39              Tw1.start();
40              Tw2.start();
41              Tr1.start();
```

```
42                 Tr2.start();
43
44             });
45         }
46         public static void main(String args[]){
47             ApplicationSample sample1 = new ApplicationSample();
48             sample1.start();
49         }
50     }
```

References

[1] T. Acharya and A.K. Ray. *Image Processing: Principles and Applications*. Wiley, 2005.

[2] Yehuda Afek, Eli Gafni, and Adam Morrison. Common2 extended to stacks and unbounded concurrency. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC '06, pages 218–227. ACM, 2006.

[3] Kunal Agrawal, Jeremy T. Fineman, and Jim Sukha. Nested parallelism in transactional memory. In *In PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 163–174, 2008.

[4] Juan Alemany and Edward W. Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the eleventh annual ACM symposium on Principles of distributed computing*, PODC '92, pages 125–134. ACM, 1992.

[5] A. Alexandrescu. Lock-free data structures, October 2004.

[6] JamesH. Anderson, Srikanth Ramamurthy, Mark Moir, and Kevin Jeffay. Lock-free transactions for real-time systems. In Azer Bestavros, Kwei-Jay Lin, and SangHyuk Son, editors, *Real-Time Database Systems*, volume 396 of *The Springer International Series in Engineering and Computer Science*, pages 215–234. Springer US, 1997.

[7] USENIX Association, IEEE Computer Society. Technical Committee on Operating Systems, and ACM Special Interest Group in Operating Systems. *ACM Symposium on Operating Systems Design and Implementation*. Number v. 2 in Conference proceedings (USENIX Association). USENIX Association, 1996.

[8] Hagit Attiya, Nancy Lynch, and Nir Shavit. Are wait-free algorithms fast? *J. ACM*, 41(4):725–763, 1994.

[9] Woongki Baek, Chi Cao Minh, Martin Trautmann, Christos Kozyrakis, and Kunle Olukotun. The opentm transactional application programming interface. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 376–387, Washington, DC, USA, 2007. IEEE Computer Society.

[10] C. Baier and J.P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[11] Lee Baugh and Craig Zilles. Analysis of I/O and syscalls in critical sections and their implications for transactional memory. In *TRANSACT '07: 2nd Workshop on Transactional Computing*, August 2007.

[12] Timothy Beneke and Tori Wieldt. Javaone 2013 review: Java takes on the internet of things, 2013.

[13] Ethan Blanton and Lukasz Ziarek. Non-blocking inter-partition communication with wait-free pair transactions. In *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '13, pages 58–67, New York, NY, USA, 2013. ACM.

[14] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Deconstructing transactions: The subtleties of atomicity. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*. International Symposium on Computer Architecture (ISCA), June 2005.

[15] Hans-J. Boehm. An almost non-blocking stack. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 40–49. ACM, 2004.

[16] Greg Bollella, Ben Brosgol, Steve Furr, David Hardin, Peter Dibble, James Gosling, Mark Turnbull, and Rudy Belliardi. *The Real-Time Specification for Java*. Addison-Wesley, Boston, MA, USA, 2000.

[17] JonathanP. Bowen and MichaelG. Hinchey. Seven more myths of formal methods: Dispelling industrial prejudices. In Maurice Naftalin, Tim Denvir, and Miquel Bertran, editors, *FME '94: Industrial Benefit of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*, pages 105–117. Springer Berlin Heidelberg, 1994.

[18] K. Breitman and R.N. Horspool. *Patterns, Programming and Everything*. SpringerLink : Bücher. Springer London, 2012.

[19] A. Burns and A. Wellings. *Concurrent and Real-Time Programming in Ada*. Cambridge University Press, 2007.

[20] A. Burns and A.J. Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. International Computer Science Series. Addison-Wesley, 2009.

[21] Francois Carouge and Michael Spear. A scalable lock-free universal construction with best effort transactional hardware. In NancyA. Lynch and AlexanderA. Shvartsman, editors, *Distributed Computing*, volume 6343 of *Lecture Notes in Computer Science*, pages 50–63. Springer Berlin Heidelberg, 2010.

[22] Daniel Cederman. *Concurrent Algorithms and Data Structures for Many-Core Processors*. PhD thesis, The school where the thesis was written, Chalmers University of Technology, 2011.

[23] Jing Chen and Alan Burns. Asynchronous data sharing in multiprocessor real-time systems using process consensus. In *Proc. 10th Euromicro Workshop on Real-Time Systems*, pages 2–9, 1998.

[24] Chergert. Came for the beer, stayed for the freedom: Programming is not a spectator sport, 4 2007.

[25] Phong Chuong, Faith Ellen, and Vijaya Ramachandran. A universal construction for wait-free transaction friendly data structures. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 335–344. ACM, 2010.

[26] Cliff Click. Just what-the-heck is a "wait-free" algorithm, 2008. accessed: 25-March-2010.

[27] R. Colvin and L. Groves. A scalable lock-free stack algorithm and its verification. In *Software Engineering and Formal Methods, 2007. SEFM 2007. Fifth IEEE International Conference on*, pages 339–348, 2007.

[28] W. Damm and E.R. Olderog. *Formal Techniques in Real-Time and Fault-Tolerant Systems: 7th International Symposium, FTRTFT 2002, Co-sponsored by IFIP WG 2.2, Oldenburg, Germany, September 9-12, 2002. Proceedings*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003.

[29] I.F. Darwin. *Checking Java Programs*. O'Reilly short cut. O'Reilly Media, 2007.

[30] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. Lock-free dynamically resizable arrays. In MariamMomenzadehAlexanderA. Shvartsman, editor, *Principles of Distributed Systems*, volume 4305 of *Lecture Notes in Computer Science*, pages 142–156. Springer Berlin Heidelberg, 2006.

[31] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. Understanding and effectively preventing the aba problem in descriptor-based lock-free designs. In *Proceedings of the 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, ISORC '10, pages 185–192, Washington, DC, USA, 2010. IEEE Computer Society.

[32] Ashley M. DeFlumere and Sadaf R. Alam. Exploring multi-core limitations through comparison of contemporary systems. In *The Fifth Richard Tapia Celebration of Diversity in Computing Conference: Intellect, Initiatives, Insight, and Innovations*, TAPIA '09, pages 75–80. ACM, 2009.

[33] David L. Detlefs, Paul A. Martin, Mark Moir, Guy L. Steele, and Jr. Lock-free reference counting. In *in Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, pages 190–199, 2001.

[34] Srinivas Dharmasanam. Multiprocessing with real-time operating systems, 2003. Accessed: 13-June-2013.

[35] Dr. Peter C Dibble. *Real-Time Java Platform Programming: Second Edition*. BookSurge Publishing, 2nd edition, 2008.

[36] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 155–165. ACM, 2009.

[37] Akos Dudas and Sandor Juhasz. Blocking and non-blocking concurrent hash tables in multi-core systems. *WSEAS Transactions on Computers*, 12:74–84, 2013.

[38] Robert J. Dugan. System/370 extended architecture: A program view of the channel subsystem. In *ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture*, pages 270–276. ACM, 1983.

[39] Haggai Eran, Ohad Lutzky, Zvika Guz, and Idit Keidar. Transactifying apache's cache module. In *SYSTOR '09: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–9. ACM, 2009.

[40] J. Esparza and R. Majumdar. Tools and algorithms for the construction and analysis of systems. In *Proceedings of the 16th International Conference, TACAS 2010, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-29, 2010*, Advanced research in computing and software science. Springer, 2010.

[41] C. Evequoz. Non-blocking concurrent fifo queues with single word synchronization primitives. In *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, pages 397–405, 2008.

[42] S. Fahmy and B. Ravindran. On stm concurrency control for multicore embedded real-time software. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 1–8, 2011.

[43] Panagiota Fatourou and Nikolaos D. Kallimanis. A highly-efficient wait-free universal construction. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 325–334. ACM, 2011.

[44] Nathan Wayne Fisher. *The Multiprocessor Real-Time Scheduling of Gnereal Task Systems*. Chapel Hill, 2007.

[45] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), May 2007.

[46] F. Gebali. *Algorithms and Parallel Computing*. Wiley Series on Parallel and Distributed Computing. Wiley, 2011.

[47] Aicas GmbH. Jamaicavm, 2013. accessed: 1-May-2011.

[48] B. Goetz, T. Peierls, J. Bloch, J. Bowbeeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison Wesley, 2006.

[49] Chun Gong and Jeannette M. Wing. A library of concurrent objects and their proofs of correctness. Technical report, Carnegie Mellon University, Research Showcase, 1990.

[50] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[51] Alexey Gotsman, Byron Cook, Matthew Parkinson, and Viktor Vafeiadis. Proving that non-blocking algorithms don't block. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 16–28. ACM, 2009.

[52] Justin E. Gottschlich, Manish Vachharajani, and Jeremy G. Siek. An efficient software transactional memory using commit-time invalidation. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 101–110. ACM, 2010.

[53] D. Gove. *Multicore Application Programming: For Windows, Linux, and Oracle Solaris*. Developer's Library. Addison-Wesley, 2010.

[54] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic Contention Management. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC'05)*, volume 3724 of *Lecture Notes in Computer Science*, pages 303–323, 2005.

[55] R. Guerraoui, M. Kapałka, and N. Lynch. *Principles of Transactional Memory*. Synthesis lectures on distributed computing theory. Morgan & Claypool Publishers, 2010.

[56] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 102. IEEE Computer Society, 2004.

[57] Tim Harris. Exceptions and side-effects in atomic blocks. *Science of Computer Programming*, 58(3):325 – 343, 2005. <ce:title>Special Issue on Concurrency and synchonization in Java programs</ce:title> <xocs:full-name>Special Issue on Concurrency and synchronization in Java programs</xocs:full-name>.

[58] Tim Harris, Adrián Cristal, Osman S. Unsal, Eduard Ayguade, Fabrizio Gagliardi, Burton Smith, and Mateo Valero. Transactional memory: An overview. *IEEE Micro*, 27(3):8–29, May 2007.

[59] Tim Harris and Keir Fraser. Language support for lightweight transactions. *SIGPLAN Not.*, 38(11):388–402, October 2003.

[60] Tim Harris, James Larus, and Ravi Rajwar. Transactional memory, 2nd edition. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, 2010.

[61] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, London, UK, UK, 2001. Springer-Verlag.

[62] K. Havelund, J. Penix, and W. Visser. *SPIN Model Checking and Software Verification: 7th International SPIN Workshop Stanford, CA, USA, August 30 - September 1, 2000 Proceedings*. Forschungen Zur Kirchen- Und Dogmengeschichte. Springer, 2000.

[63] D. Hendler and N. Shavit. Work dealing. In *Proc. of the Fourteenth ACM Symposium on Parallel Algorithms and Architectures*, pages 164–172, 2002.

[64] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 206–215. ACM, 2004.

[65] M. Herlihy. A methodology for implementing highly concurrent data structures. *SIGPLAN Not.*, 25(3):197–206, February 1990.

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

[67] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, PODC '03, pages 92–101. ACM, 2003.

[68] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

[69] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[70] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[71] T. Higuera-Toledano and A.J. Wellings. *Distributed, Embedded and Real-time Java Systems*. Springer, 2012.

[72] Owen S. Hofmann, Donald E. Porter, Christopher J. Rossbach, Hany E. Ramadan, and Emmett Witchel. Solving difficult htm problems without difficult hardware. In *In ACM TRANSACT Workshop*, 2007.

[73] J. Hunt and A.G. McManus. *Key Java: Advanced Tips and Techniques*. Practitioner Series. Springer London, 2013.

[74] IBM. Websphere real time, 2014. accessed: 27-March-2014.

[75] Intel. 4th generation intel® core™ i7 processor, 2013. accessed: 28-June-2013.

[76] Idaku Ishii, Tomoki Ichida, Qingyi Gu, and Takeshi Takaki. 500-fps face tracking system. *Journal of Real-Time Image Processing*, 8(4):379–388, 2013.

[77] Java. Learn about java technology, 2013. accessed: 25-June-2013.

[78] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4):21:1–21:54, October 2009.

[79] G. Wade Johnson. Programmer musings, 2005. accessed: 20-June-2013.

[80] Lawrence Kesteloot. A survey of mutual exclusion algorithms for multi-processor operating systems, 1995. Accessed: 3-March-2010.

[81] M. Kowalczyk. Asynchronous communication between threads. In *In 6th Symposium on Trends in Functional Programming, TFP 2005: Proceedings, Institute of Cybernetics, Tallinn, 2005*, TFP 2005, page 76 – 87, 2005.

[82] Lamport L. Specifying concurrent program modulus. In *ACM Transactions on Programming Languages and Systems*, volume 5 of *PODC '83*, pages 190–222, 1983.

[83] Edya Ladan-mozes and Nir Shavit. An optimistic approach to lock-free fifo queues. In *In Proceedings of the 18th International Symposium on Distributed Computing, LNCS 3274*, pages 117–131. Springer, 2004.

[84] Anthony LaMarca. A performance evaluation of lock-free synchronization protocols. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, PODC '94, pages 130–140. ACM, 1994.

[85] Andreas Larsson, Anders Gidenstam, Phuong H. Ha, Marina Papatriantafilou, and Philippas Tsigas. Multiword atomic read/write registers on multiprocessor systems. *J. Exp. Algorithmics*, 13:1.7–1.30, 2009.

[86] E. A. Lee. The problem with threads. *Computer*, 39(5):33 – 42, May 2006.

[87] Patrick P. C. Lee, Tian Bu, and Girish Chandranmenon. A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring. In *in 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Atlanta, Georgia, 2010. IEEE.

[88] Leonard Lensink. *Applying Formal Methods in Software Developmen.* PhD thesis, Radboud University Nijmegen, 11 2013.

[89] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, ExpCS '07, New York, NY, USA, 2007. ACM.

[90] Quan Liu, Xingran Cui, and Xiuyin Hu. Conflict resolution within multi-agent system in collaborative design. In *Computer Science and Software Engineering, 2008 International Conference on*, volume 1, pages 520–523, Dec 2008.

[91] J. Lourenço and E. Farchi. *Multicore Software Engineering, Performance, and Tools: International Conference, MUSEPAT 2013, Saint Petersburg, Russia, August 19-20, 2013, Proceedings.* Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013.

[92] Yuval Lubowich and Gadi Taubenfeld. On the performance of distributed lock-based synchronization? *SIGOPS Oper. Syst. Rev.*, 45(2):28–37, July 2011.

[93] R. Mall. *Real-Time Systems: Theory and Practice.* Pearson Education, 2009.

[94] J. Manson, J. Baker, A. Cunei, S. Jagannathan, M. Prochazka, Bin Xin, and J. Vitek. Preemptible atomic regions for real-time java. In *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, pages 10 pp.–71, 2005.

[95] J. Manson, J. Baker, A. Cunei, S. Jagannathan, M. Prochazka, Bin Xin, and J. Vitek. Preemptible atomic regions for real-time java. In *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, pages 10 pp.–71, 2005.

[96] Henry Massalin and Calton Pu. A lock-free multiprocessor os kernel. TechReport CUCS-005-91, Computer Science Department, Columbia University, 1991.

[97] Austen McDonald, JaeWoong Chung, Brian D. Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural semantics for practical transactional memory. *SIGARCH Comput. Archit. News*, 34(2):53–65, 2006.

[98] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '02, pages 73–82. ACM, 2002.

[99] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15:491–504, 2004.

[100] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275. ACM, 1996.

[101] Maged M. Michael and Michael L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51:1–26, May 1998.

[102] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. *SIGARCH Comput. Archit. News*, 35(2):69–80, June 2007.

[103] M. Moir and N. Shavit. *Concurrent data structures*. Chapman and Hall/CRC Press, handbook of data structures and applications, d. metha and s. sahni edition, 2007.

[104] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using elimination to implement scalable and lock-free fifo queues. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 253–262, New York, NY, USA, 2005. ACM.

[105] Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. *SIGPLAN Not.*, 48(8):103–112, February 2013.

[106] Jogesh K. Muppala, Kishor S. Trivedi, and Steven P. Woolet. Real-time systems performance in the presence of failures. *Computer*, 24(5):37–47, May 1991.

[107] NASA. What is jpf?, 2009. accessed: 25-August-2015.

[108] Farhang Nemati. *RESOURCE SHARING IN REAL-TIME SYSTEMS ON MULTIPROCESSORS*. Mälardalen University Press Dissertations, no. 124 edition, 2012.

[109] Marek Olszewski, Jeremy Cutler, and J. Gregory Steffan. Judostm: A dynamic binary-rewriting approach to software transactional memory. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 365–375, Washington, DC, USA, 2007. IEEE Computer Society.

[110] Oracle. Multithreaded programming guide, the producer/consumer problem, 2010. accessed: 13-Sep-2013.

[111] Oracle. The history of java technology, 2011. Accessed: 10-June-2013.

[112] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008.

[113] Filip Pizlo, Marek Prochazka, Suresh Jagannathan, and Jan Vitek. Transactional lock-free objects for real-time java. In *Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs*, pages 54–62, July 2004.

[114] Alice Porfirio, Alessandro Pellegrini, Pierangelo Di Sanzo, and Francesco Quaglia. Transparent support for partial rollback in software transactional memories. In Felix Wolf, Bernd Mohr, and Dieter an Mey, editors,

Euro-Par 2013 Parallel Processing, volume 8097 of *Lecture Notes in Computer Science*, pages 583–594. Springer Berlin Heidelberg, 2013.

[115] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. The Springer International Series in Engineering and Computer Science. Springer US, 2012.

[116] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, 2010.

[117] R. L. Rockhold. Synchronization in java, September 2012. http://www.cs.utexas.edu/ rockhold/CS439/handouts.

[118] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. Txlinux: using and managing hardware transactional memory in an operating system. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 87–102. ACM, 2007.

[119] RTSJ. Rtsj timeline, 2006. accessed: 22-June-2013.

[120] Bo Sanden. Coping with java threads. *Computing Practices*, 2004.

[121] M. F. Santarelli, V. Positano, and L. Landini. Real-time multimodal medical image processing: A dynamic volume-rendering application. *Trans. Info. Tech. Biomed.*, 1(3):171–178, September 1997.

[122] T. Sarni, A. Queudet, and P. Valduriez. Real-time support for software transactional memory. In *Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA '09. 15th IEEE International Conference on*, volume 24-26, pages 477 –485, 2009.

[123] William N. Scherer, III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, PODC '05, pages 240–248. ACM, 2005.

[124] Martin Schoeberl, Florian Brandner, and Jan Vitek. Rttm: real-time transactional memory. In *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 326–333. ACM, 2010.

[125] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, September 1990.

[126] Niloufar Shafiei. Non-blocking array-based algorithms for stacks and queues. In *Proceedings of the 10th International Conference on Distributed Computing and Networking*, ICDCN '09, pages 55–66. Springer-Verlag, 2009.

[127] Chien-Hua Shann, T.-L. Huang, and Cheng Chen. A practical nonblocking queue algorithm using compare-and-swap. In *Parallel and Distributed Systems, 2000. Proceedings. Seventh International Conference on*, pages 470–475, 2000.

[128] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213. ACM, 1995.

[129] Nir Shavit and Dan Touitou. Elimination trees and the construction of pools and stacks. *Theory of Computing Systems*, 30:645–670, 1997.

[130] F. Siebert. *Hard Real-time Garbage Collection Im Modern Object Oriented Programming Languages*. Aicas realtime, 2002.

[131] H.R. Simpson. Four-slot fully asynchronous communication mechanism. *Computers and Digital Techniques, IEE Proceedings E*, 137(1):17–30, 1990.

[132] Y. Sorel. Real-time embedded image processing applications using the a3 methodology. In *Image Processing, 1996. Proceedings., International Conference on*, volume 1, pages 145–148 vol.2, 1996.

[133] Sorin, Roth, Hill, Wood, Sohi, Smith, Vijaykumar, and Lipasti. Pipelining, 2009. accessed: 19-Sep-2013.

[134] M.F. Spear, M. Silverman, L. Dalessandro, M.M. Michael, and M.L. Scott. Implementing and exploiting inevitability in software transactional memory. In *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, volume 9-12, pages 59 –66, 2008.

[135] Michael F. Spear, Maged Michael, and Michael L. Scott. Inevitability mechanisms for software transactional memory. In *TRANSACT '08: 3rd Workshop on Transactional Computing*, February 2008.

[136] B. Steensgaard, E. Petrank, and G. Kliot. Incremental lock-free stack scanning for garbage collection, September 2 2014. US Patent 8,825,719.

[137] J. M. Stone. A non-blocking compare-and-swap algorithm for a shared circular queue. In *Parallel and Distributed Computing in Engineering Systems*, pages 147–152. Elsevier Science Publishers, 1992.

[138] Sun. Sun java real-time system 2.2, 2014. accessed: 27-March-2014.

[139] Herb Sutter. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[140] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, September 2005.

[141] Dominic Sweetman. *See MIPS Run*. Morgan Kaufmann, edition 2, 2010.

[142] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. Analyzing lock contention in multithreaded applications. *SIGPLAN Not.*, 45(5):269–280, January 2010.

[143] G. Taubenfeld. Efficient transformations of obstruction-free algorithms into nonblocking algorithms. In *In: Pelc, A. (ed.) DISC 2007. LNCS*, volume 4731, pages 450–464. Springer, 2007.

[144] Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. *SIGPLAN Not.*, 47(8):309–310, February 2012.

[145] Kent Treiber. Systems programming: Coping with parallelism. TechReport RJ 5118 (s3r62) 4/23/86, IBM Almaden Research Cente, San Jose, Californi, 1986.

[146] P. Tsigas, Yi Zhang, D. Cederman, and T. Dellsen. Wait-free queue algorithms for the real-time java specification. In *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, pages 373–383, 2006.

[147] Philippas Tsigas and Yi Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *in Proceedings of the 13th ACM Symposium on Parallel Algorithms and Architectures*, pages 134–143. ACM, 2001.

[148] Andras Vajda. *Programming Many-Core Chips*. Springer, Dordrecht, 2011.

[149] J. D. Valois. Lock-free data structures. In *Ph. D. dissertation*. Rensselaer Polytechnic Institute, 1995.

[150] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 214–222. ACM, 1995.

[151] A. Wellings. *Concurrent and Real-Time Programming in Java*. Wiley, 2005.

[152] Andrew Wellings. *Concurrent and Real-Time Programming in Java*. John Wiley & Sons, 2004.

[153] AndyJ. Wellings, Peter Dibble, and David Holmes. Supporting multiprocessors in the real-time specification for java version 1.1. In M. Teresa Higuera-Toledano and Andy J. Wellings, editors, *Distributed, Embedded and Real-time Java Systems*, pages 1–22. Springer US, 2012.

[154] Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Mechanisms for store-wait-free multiprocessors. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 266–277. ACM, 2007.

[155] K. Windisch, K. Kline, and L. Davidson. *Pro SQL Server 2005 Database Design and Optimization*. Apress, 2006.