Integrating Java 8 Streams with The Real-Time Specification for Java

HaiTao Mei
University of York, UK
hm857@york.ac.uk

Ian Gray
University of York, UK
Ian.Gray@york.ac.uk

Andy Wellings
University of York, UK
Andy.Wellings@york.ac.uk

ABSTRACT

The paper investigates the use of the Java 8 stream processing facilities from within the context of the Real-Time Specification for Java. Java 8 stream processing uses the Java 7 Fork/Join framework. We demonstrate that it is not possible, with the current framework, to supply a pool of real-time worker threads with which to perform stream evaluation. We show what changes would need to be made to the framework for it to be used in a real-time context. Our evaluation shows that without such changes, use of the current Java 8 stream processing faculties by real-time threads can result in significant priority inversions. We also consider what hooks the RTSJ would need to provide that would allow real-time Fork/Join pools to be generated without changes to the source code.

1. INTRODUCTION

Java 8 [4] has introduced streams and lambda expressions to support the efficient processing of in-memory data sources (e.g., a Java Collection) in parallel, with functional-style code. One of the primary goals is "to accelerate operations upon large amounts of data by dividing the task between multiple threads (processors)" [5]. The parallel implementation builds upon the `java.util.concurrency` Fork/Join framework introduced in Java 7. The Java 8 stream processing infrastructure is based on three assumptions: its data source has been populated into memory before processing, the size of data source will not change, and the goal is to *process the data as fast as possible using all of the available processors.*

Real-time and embedded platforms have evolved towards multi-core. Parallel programming of these platforms is required if applications are to exploit the extra available performance. Although the next version of the Real-Time Specification for Java (RTSJ) [6] will provide more support for multiprocessors [23], there is currently little work that attempts to integrate either the concurrency utilities or the stream processing infrastructure with the RTSJ. The goal of this paper is to integrate Java 8 stream processing with the Real-Time Specification for Java (RTSJ), in order to provide the efficient parallel processing of bulk data with concise code in a real-time environment. The real-time stream processing framework allows programmers to describe stream computation under real-time constraints.

Real-time multiprocessor platforms can be scheduled by a range of techniques. For example, threads can be globally scheduled across all processors using an earliest-deadline-first policy. Alternatively, the processors can be partitioned into non-overlapping groups, and each thread allocated and scheduled within that group. In the extreme case, the group size can be set to a single processing core. In this situation, the system is said to be fully partitioned, and each thread can only be scheduled on one processor. For our initial work, we only consider these fully partitioned systems. Also, as the RTSJ supports priority-based scheduling, we assume this scheduling policy. Finally, for simplicity, we assume that the stream will be evaluated in the context of RTSJ real-time threads, rather than asynchronous event handlers.

In general, stream data sources can be classified into two types [12, 22]: batched and streaming. A batched data source is where the data is already present in memory, and its content and size will not change during processing. A streaming data source represents data that arrives dynamically, its content and size will change with time, although there is no any modification of the data by the stream itself. This paper focuses on processing batched data sources in real-time, as they more closely relate to Java 8 stream sources.

Thus the real-time stream processing framework presented in this paper:

- assumes a fully partitioned preemptive priority-based scheduled execution platform, and
- processes batched data sources submitted from RTSJ real-time threads.

The goal is to

- process the data as fast as possible whilst having a predictable impact on other real-time activities on the same execution platform, and
- allows the real-time properties of the stream to be calculated.

For a real-time stream processing batched data, the important real-time property is its *worst-case response time.* Where, the response time is defined to be *the time from*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
JTRES '2015 Paris, France
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

when the stream's terminal operation is invoked to its return. This time can then be included in the appropriate schedulability analysis for the entire execution platform.

The paper is structured as follows. First, the Java 8 stream processing framework is introduced in section 2. Section 3 then discusses the issues of extending this framework for a real-time environment, and describes the real-time stream processing framework. Section 4 gives a prototyped implementation of the real-time stream processing framework. The evaluation and discussion are given in section 5. Section 6 summarises related work. The conclusions and future work are given in section 7.

2. JAVA 8 STREAMS

Streams and Lambda expressions are the most notable features that have been added in Java SE 8. The Stream API and lambda expressions are designed to facilitate simple and efficient processing of data sources (such as from Java collections) in a way which can be easily pipelined and parallelised.

A lambda expression is an anonymous method, which consists of arguments and corresponding processing statements for these arguments. For example, `(a,b)->a+b` defines a Lambda expression that sums two arguments. Lambda expressions make code more concise, and extend Java with functional programming languages concepts. Internally, a lambda expression will be compiled into a *functional interface* by the Java compiler. Functional interfaces were introduced by Java 8, and are interfaces which contain only one method. In addition, lambda expressions use target typing [21], i.e. the type of arguments will be automatically determined by the compiler during compilation, rather than required to be specified by programmers. This feature enables passing methods as arguments, rather than constructing an object of a specified class. With suitable frameworks, a programmer can easily construct graphs and pipelines of functional operations.

The chain of a sequence of operations and the data source forms a pipeline. Streams make use of lambda expressions to enable passing different methods into each operation in the pipeline if required. A pipeline consists of a source, zero or more intermediate operations, and a terminal operation [9] [10]. An intermediate operation always returns a new stream, rather than perform methods on the data source. One example of intermediate operations is `map`, which maps each data elements in the stream into a new element in the new stream. A terminal operation forces the evaluation of the pipeline, consumes the stream, and returns a result. Thus, streams are lazily evaluated. An example of terminal operations is `reduce`, which performs a reduction on the data elements using an accumulation function. A simple word count example can be described by the following code using the Stream API and Lambda Expressions:

```
Collection<String> datatoProcess = WordsToCount;
Pattern pat=Pattern.compile("\\s+");
Map<Object, Long> result =
  datatoProcess.parallelStream()
  .flatMap(line->Stream.of(pat.split(line)))
  .collect(Collectors.groupingBy(
            w -> w,
            TreeMap::new,
            Collectors.counting()));
```

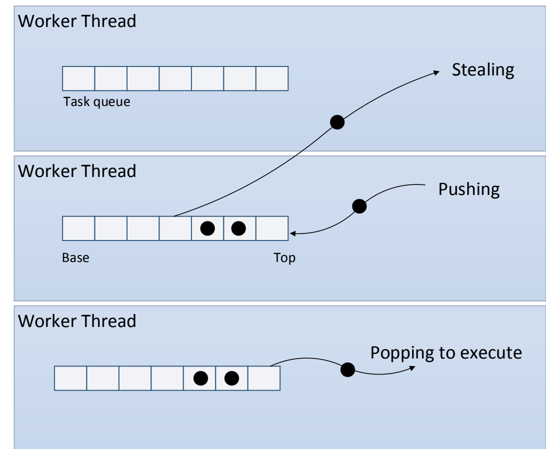

Figure 1: Tasks stealing, pushing and popping within worker threads

2.1 Stream Evaluation Model

One of the main advantages of streams is that they can be either sequentially evaluated, or evaluated in parallel. Sequential evaluation is carried out by performing all the operations in the pipeline on each data element sequentially by the thread which invoked the terminal operation of the stream. When a stream is evaluated in parallel, it uses a special kind of iterator called a *Spliterator* to partition the processing, and all the created parts will be evaluated in parallel with the help of a ForkJoin thread pool. Efficiency is achieved by the work stealing algorithm that is used by the ForkJoin pool.

2.1.1 The ForkJoin Thread Pool

Introduced in Java SE 7, the ForkJoin thread pool is a parallel framework in which tasks are computed by splitting themselves into small subtasks that will be computed in parallel, waiting for them to be completed, and then composing the results [14]. More specifically, the small subtasks are computed by the ForkJoin thread pool with a work stealing algorithm to balance the load of its workers.

A ForkJoin thread pool maintains a task queue, and creates worker threads with a thread factory. In addition, the thread factory can be configured. The number of worker threads usually corresponds to the number of available processors in the platform. In overview, worker threads take tasks from the queue associated with the ForkJoin pool, and execute the task. The task may split into small subtasks, and these smaller tasks are pushed into the worker's own task queue. The worker thread pops tasks out from its queue and executes them, when its current task is completed. A worker thread tries to take a task from other worker threads' queues when its queue is empty, using a work stealing algorithm.

2.1.2 The Java 8 Work Stealing Algorithm Details

A work stealing algorithm is the heart of the ForkJoin thread pool. The details of the execution of a worker thread using the work stealing algorithm are summarised by the following, according to the publication of Lea [14] and the source code of java.util.concurrent package:

1. Each worker thread maintains its own task queue. The queue is a double-ended queue, which enables access to the data from both the top and bottom.
2. Within one worker thread, subtasks that are generated by splitting its tasks will be pushed onto the top of the worker thread's own queue.
3. Each worker thread executes its currently task first, then executes tasks in its queue in LIFO order, i.e. by popping tasks from the top of the queue.
4. When a worker thread has no tasks to execute, it tries to take a task from another randomly chosen worker thread's queue in FIFO order.
5. When a worker thread waits for a task to finish, it will process other tasks with the help of the ForkJoin pool until it is notified of completion (via `ForkJoinTask.isDone()`). Tasks otherwise run to completion without blocking.
6. When a worker thread is idle, and fails to steal tasks from other worker threads, it backs off, e.g. yields.

The internals of worker threads employing the work stealing algorithm are illustrated by Figure 1.

2.1.3 *Parallel Evaluation of a Stream with the ForkJoin Pool*

A stream starts to be evaluated once its terminal operation is called. Once a terminal operation is invoked, the corresponding terminal operation task, which inherits from the ForkJoin task, is executed. Thus, the evaluation of a stream is represented by the execution of a ForkJoin task. With parallel evaluation, the stream is evaluated by the current thread alongside the worker threads in the default ForkJoin pool. Note that, the current thread can be a worker thread in a ForkJoin pool, when the evaluation of a stream is submitted to that pool. The evaluation of a stream is split to small subtasks, and these subtasks are then evaluated using the work stealing algorithm. By default, a stream splits into four pieces for each worker thread in the ForkJoin Pool, so a thread being executed by a pool with 4 threads will split at most 16 times.

For example, one stream is submitted to a pool with 2 worker threads. The parallel evaluation of this stream is illustrated by Figure 2. One worker thread takes the evaluation task from the pool first, then executes (see time 1). The task splits into 2 subtasks, and one of them is pushed into the task queue at time 2. Work stealing is assumed to occur at time 3, in reality, it can be later or earlier. When all the tasks shown at time 9 have been executed, the stream has been successfully evaluated. Note that, in this example, we assume this stream can be split as often as it requires, and all the worker threads within that pool have been successfully created before evaluation.

3. INTEGRATING JAVA 8 STREAMS WITH THE RTSJ

The Java 8 Stream API enables pipelined or parallelised processing of data sources with concise code. However, the Java 8 Stream API has not been designed to address real-time concerns. Hence, execution in an RTSJ environment has not been envisaged.

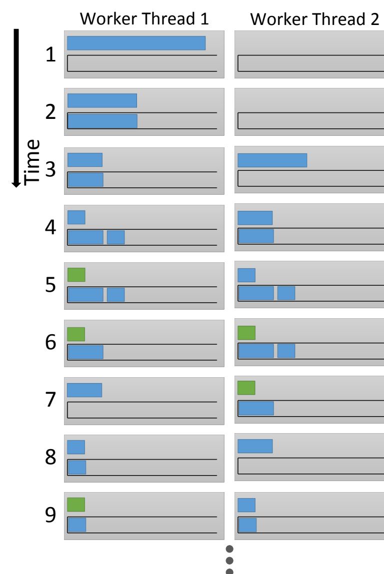

Figure 2: The parallel evaluation of a Stream by a pool with 2 threads. The grey box represents thread; the open rectangle represents the task queue; the blue block represents the task or the subtask that wait to be executed; the green block represents the task that is being executed.

One approach to providing real-time stream processing is to introduce the notion of a real-time stream, which would be created explicitly by the application in a similar manner to the way parallel streams are created. So, for example, a real-time stream could be generated from a Java collection by calling a `Collection.realtimeStream(Priority p)`. There are two reason we decided not to follow this approach. Firstly, as shown in section 2, the main execution engine of a Java 8 stream is a ForkJoin thread pool. Streams can be evaluated with real-time constraints when evaluated by a real-time ForkJoin thread pool. The real-time constraint is on the processing of the stream, not the stream itself. Secondly, streams are usually created from static factory methods on the stream classes, such as `Stream.of(Object[])`, and a Collection via the `stream()` method. Extending the current stream API requires modifying the Java Collection class. We prefer to keep our changes as minimal as possible.

Hence, our overall approach is to focus on modifying the behavior of the ForkJoin pool so that the worker threads are real-time threads rather than `java.lang` threads. Section 3.1 first considers the difficulties of doing this with the current framework. In doing so, it provides the overall rationale for why we have to make some modification to the Java 8 libraries. This is followed by a discussion of our proposed real-time stream processing framework.

3.1 Difficulties In Creating a Real-Time Thread Pool

The ForkJoin thread pool has been designed so that the programmer has some control over its configuration; in par-

ticular the number of worker threads. It even allows the application to provide its own factory for creating these worker threads. The intention is that the factory should return a thread whose class extends the predefined `ForkJoinWorkerThread` class. This class has two methods that can be overridden: `onStart()` and `onTermination()`, which are called immediately a new worker thread is created and before a worker thread terminates respectively. Hence, the application can provide some limited context within which the threads execute.

Unfortunately, the framework is not flexible enough to allow the introduction of real-time threads because creating a customized ForkJoin pool requires a thread factory that *must* produce threads that inherit from `ForkJoinWorkerThread`. This class is a subclass of `java.lang.Thread`. In the RTSJ all real-time threads must extend `javax.realtime.RealtimeThread`, which itself extends `java.lang.Thread`. Java does not support multiple inheritance, so the requirements are conflicting.

Given that the main `run()` method of the `ForkJoinWorkerThread` is not final, we thought we might be able to adopt a delegation approach. With this approach, each fork-join worker thread creates a local real-time thread and delegates all processing to that real-time thread. The following illustrates the approach:

```
public class RealtimeForkJoinWorkerThread extends
     ForkJoinWorkerThread {
  private RealtimeDelegate rtwt = new
       RealtimeDelegate(this);
    //Constructor and other methods ...
  @Override
  public synchronized void start() {
    rtwt.setDaemon(true);
    rtwt.start();
  }
}
```

where

```
import javax.realtime.RealtimeThread;
class RealtimeDelegate extends RealtimeThread{
  private RealtimeForkJoinWorkerThread parent;
  public RealtimeDelegate(RealtimeForkJoinWorkerThread
       parent){
    this.parent=parent;
  }
  @Override
  public void run(){
    // The run() method of RealtimeDelegate
    // is inherited from ForkJoinWorkerThread
    parent.run();
  }
}
```

Although, this has the appearance of creating a real-time thread pool, it does not have the desired effect when used in conjunction with the main fork and join processing class. This is because the `fork()` method checks to see if the calling thread is an instance of `ForkJoinWorkerThread`. If it is, it submits the new task to the current pool; if it is not, it submits the new task to the default common (and, therefore, non real-time) pool. Of course, with the delegate approach, the calling thread is not an instance of this class. Furthermore, the common pool is final and cannot be modified.

Hence, reluctantly, we conclude that integrating the RTSJ with the fork-join thread pool requires the source code to be modified.

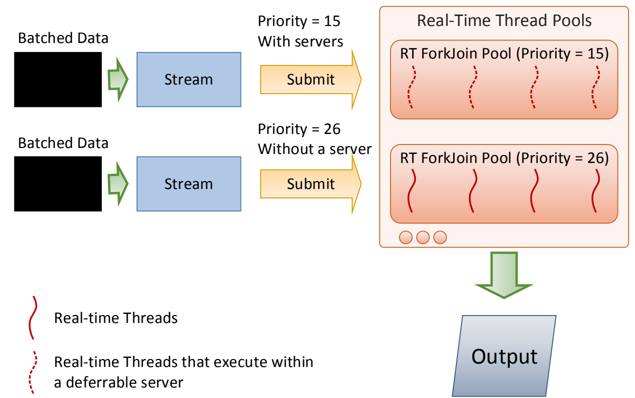

Figure 3: Overview of the Real-time Stream Framework.

3.2 The Real-Time Stream Processing Framework

The real time streaming processing model builds on the standard RTSJ system model. In this model, a system consists of a set of sporadic, periodic, and aperiodic real-time threads. An RTSJ-aware stream, like a Java 8 stream, encapsulates data which can be processed in parallel using functional-style programming constructs and usually originates from a Java Collection, or I/O class. Operation on streams are performed by standard RTSJ real-time threads. These threads have deadlines, priorities, worst-case execution time constraints. Aperiodic threads can also have execution time servers (see section 3.2.2) that regulate how much CPU time the thread is given. Hence, the real-time stream framework executes within this context.

Where Java 8 Streams are evaluated by a ForkJoinPool, RTSJ-aware streams are evaluated by a RealTimeForkJoinPool that is defined as a fixed size pool of real time aperiodic threads. The RealTimeForkJoinPool is created with a priority (which sets the priority of all of its constituent threads) and optionally may be created with an execution-time server. The real-time stream processing framework is illustrated in Figure 3.

3.2.1 *The Real-Time ForkJoin Pool*

A RealTimeForkJoinPool is designed to be a Java ForkJoin thread pool, in which each worker thread is a real-time thread, and the priority of each worker thread is configured when the pool is created. In addition, each worker thread of a pool can be executed under the control of an execution-time server. Typically each pool will inherit the priority of its calling real-time thread. Where that real-time thread is aperiodic, it will inherit the caller's execution-time server (if set). However, for greater flexibility, we also provide mechanisms for these to be set explicitly.

In standard Java, the default ForkJoin pool creates the same number of worker threads as there are processors on the execution platform minus one. For the RealTimeForkJoinPool, the default is to create a set of pools, one per priority level. Each pool contains one worker thread per core. This

is preferred to a single pool with one thread per priority level per processor as the work stealing algorithm would need be modified to ensure that tasks from high priority streams are given preference over tasks from low priority schemes. Thus, multiple pools ensures that high priority stream processing will not be delayed by low priority stream processing. There are no execution-time servers associated with the default pools. This is because they are shared.

A real-time ForkJoin pool instance can also be created by invoking the constructor with the priority and server arguments. The `RealTimeForkJoinPool` class is show below

```
public class RealtimeForkJoinPool extends ForkJoinPool {
  // constructor
  public RealtimeForkJoinPool(PriorityParameters
       priority, ProcessingGroup server){ ...}
  // return a real-time ForkJoin pool with requested
  // priority and assign a server to the worker threads

  public static ForkJoinPool
       getRTForkJoinPool(PriorityParameters
       priority){/*...*/}
    // return the default real-time ForkJoin pool
    // with requested priority
}
```

With this approach, other than imposing priorities (and execution-time servers) on the worker threads and ensuring tasks are submitted to the correct pool, there are no other changes required to the Java 8 stream processing framework.

3.2.2 *Execution-Time Servers*

Typically stream processing is computationally intensive. In an RTSJ context, it is most likely to occur within a soft real-time task. With all such soft real-time activities, there is tension between getting good response time without jeopardizing any hard real-time activities. Running stream processing at the lowest priority in the system will not give good response times, but running it at too high a priority might cause critical activities to miss their deadlines. Hence, an appropriate priority level must be found, and any spare CPU capacity that becomes available must be made available as soon as practical.

The real-time community has addressed this problem in the context of aperiodic (or execution-time) servers. Essentially, as aperiodic activities have no worst-case arrival rates, they cannot be guaranteed to meet their deadlines on any platform. Hence, they are viewed as soft real-time. The goal has been to service all such activities as fast as possible without undermining the guarantees given to the periodic and sporadic hard real-time activities. To this end, several types of servers have been defined by the real-time community. The POSIX standard supports the Sporadic Server[13, 20, 15]. A sporadic server assigns a limited amount of CPU capacity to handle aperiodic events. It has a replenishment period, a budget, and two priorities. The server runs at a high priority when it has some budget left and a low one when its budget is exhausted. When a server runs at the high priority, the amount of execution time it consumes is subtracted from its budget. The amount of budget consumed is replenished at the time the server was activated plus the replenishment period. When its budget reaches zero, the server's priority is set to the low value.

With the Deferrable Server [20, 15], an analysis is undertaken that enables a new logical thread to be introduced at a particular priority level. This thread, the server, has a period and a capacity. These values can be chosen so that all the periodic schedulable objects in the system remain schedulable even if the server executes periodically and consumes its capacity. At run-time, whenever an aperiodic thread is released, and there is capacity available, it starts executing at the server's priority level until either it finishes or the capacity is exhausted. In the latter case, the aperiodic thread is suspended (or transferred to a background priority). With the deferrable server model, the capacity is replenished every period.

When RTSJ's processing group parameters are assigned to one or more aperiodic real-time threads[1], a server is effectively created. The server's start time, cost (capacity), and period is defined by the particular instance of the parameters. These collectively define the points in time when the server's capacity is replenished. Any aperiodic schedulable object that belongs to a processing group is executed at its own defined priority. However, it only executes if the server still has capacity. As it executes, each unit of CPU time consumed is subtracted from the server's capacity. When the capacity is exhausted, the aperiodic real-time threads are not allowed to execute until the start of the next replenishment period. If the application only assigns aperiodic real-time threads of the same priority level to a single `ProcessingGroupParameters` object, then the functionality of a deferrable server can be obtained.

The RTSJ is, however, a little more general. It allows the "servers" to be given deadlines, and cost overrun and deadline-miss handlers to be specified. If used within the context of an aperiodic server, a cost overrun potentially allows the application to, for example, modify the priorities of associated aperiodic schedulable objects.

Servers can be employed to bound the impact of stream processing. The real-time stream processing framework uses the approach suggested in [24] to allow a range of servers to be associated with real-time thread pools

```
public abstract class ProcessingGroup {
  ProcessingGroup(ProcessingGroupParameters PGP};

}

public abstract class AperiodicServer
       extends ProcessingGroup {
  AperiodicServer(ProcessingGroupParameters PGP,
         PriorityParameters SP);
  ...
}
```

Subclasses of `AperiodicServer` support the common techniques for serving aperiodic activities; for example:

```
public class PollingServer extends AperiodicServer {
  public PollingServer(ProcessingGroupParameters PGP,
                       PriorityParameters SP);
}

public class DeferrableServer extends AperiodicServer {
  public DeferrableServer(
            ProcessingGroupParameters PGP,
            PriorityParameters SP,
            PriorityParameters background);
}
```

[1]In general the RTSJ supports the notion of schedulable objects; a real-time thread is on type of schedulable object.

Note the Deferrable server runs at background priority when its budget is exhausted.

With this model, for example, an instance of the `DeferrableServer` class is attached to the real-time ForkJoin pool via the `ProcessingGroup` within the appropriate constructor:

```
public RealtimeForkJoinPool(PriorityParameters
     priority, ProcessingGroup server)
```

3.2.3 *Using the Real-Time Stream Evaluation Framework*

Section 2 presented the following simple example of code that evaluates a stream

```
Collection<String> datatoProcess = WordsToCount;
Pattern pat=Pattern.compile("\\s+");
Map<Object, Long> result =
  datatoProcess.parallelStream()
  .flatMap(line->Stream.of(pat.split(line)))
  .collect(Collectors.groupingBy(
            w -> w,
            TreeMap::new,
            Collectors.counting()));
```

If this code is to be executed by the default real-time thread pool, it requires no change. The real-time stream processing framework will determine the priority of the caller and use the appropriate pool. When embedded within a periodic or sporadic real-time thread the execution of the thread can be illustrated in Figure 4.

To use a real-time ForkJoin pool that is subject to execution time constraints, it is necessary to create a bespoke pool. Consider, for example the above code which is now to be invoked from within an aperiodic real-time thread.

```
// assuming the following have be set appropriately
ProcessingGroupParameters pgp;
PriorityParameters pri, backgroundPri, priForPool;

DeferrableSerrver server=DeferrableSerrver(pgp, pri,
     backgroundPri);

Map<Object, Long> result= new
     RealtimeForkJoinPool(priForPool, server)
      .submit(() -> {
         Pattern pat = Pattern.compile("\\s+");
         return datatoProcess
            .parallelStream()
            .flatMap(line -> Stream.of(pat.split(line)))
            .collect(Collectors.groupingBy(w -> w,
                 TreeMap::new,Collectors.counting()));
      }).get();
```

The real-time pool is created explicitly, and a stream is submitted. Note, this is exactly the same structure that would be needed if a stream is to be evaluated within an application created ForkJoin pool.

The resulting execution of the aperiodic thread is illustrated in Figure 5.

4. IMPLEMENTATION

The real-time stream processing framework has been implemented using aicas's Jamaica implementation of the RTSJ [3], which includes support for multiprocessor applications including affinity sets. There are two components: modifications to the source code of the fork-join pool and implementation of the `ProcessingGroup` class hierarchy.

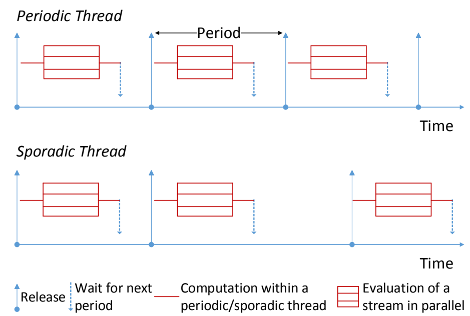

Figure 4: Evaluating a Stream within a Periodic/Sporadic Thread.

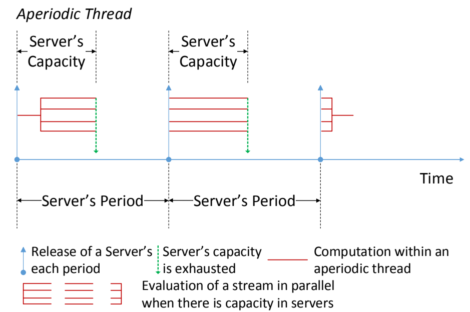

Figure 5: Evaluating a Stream Using The Deferrable Server, In an Aperiodic Thread.

4.1 Modifications to the ForkJoin Framework

Our modifications to the ForkJoin framework are of the following main source files

- `ForkJoinPool.java`
- `ForkJoinTask.java`
- `CountedCompleter.java`
- `ForkJoinWorkerThread.java`

The changes made to the first three files are solely concerned with handling the common pool. For example, in the `ForkJoinPool.java` file there is a static method

```
/**
 * Returns common pool queue for a thread that has
 * submitted at least one task.
 */
static WorkQueue commonSubmitterQueue() {
    ForkJoinPool p; WorkQueue[] ws; int m, z;
    return ((z = ThreadLocalRandom.getProbe()) != 0 &&
            (p = common) != null &&
            (ws = p.workQueues) != null &&
            (m = ws.length - 1) >= 0) ?
        ws[m & z & SQMASK] : null;
}
```

This must be modified to select the appropriate priority queue.

```java
/**
 * Returns common pool queue for a thread that
 * has submitted at least one task.
 */
static WorkQueue commonSubmitterQueue() {
    ForkJoinPool p; WorkQueue[] ws; int m, z;
    ForkJoinPool common;
    int prio=Thread.currentThread().getPriority();
    if(Thread.currentThread() instanceof
         RealtimeThread){
      prio= ((PriorityParameters)RealtimeThread
      .currentRealtimeThread()
      .getSchedulingParameters()).getPriority();
    }
    common= getCommonPool(prio);
    return ((z = ThreadLocalRandom.getProbe()) != 0 &&
            (p = common) != null &&
            (ws = p.workQueues) != null &&
            (m = ws.length - 1) >= 0) ?
        ws[m & z & SQMASK] : null;
}
```

The changes made to `ForkJoinWorkerThread.java` is simply to make it extend `RealtimeThread`

```java
public class ForkJoinWorkerThread extends
    RealtimeThread {//...}
```

We then use a bootstrap class path option, i.e. `-Xbootclasspath` (when invoking the real-time JVM), which allows programmers to use a different set of core classes, to ensure that our modifications are loaded in preference to the standard files.

4.2 The ProcessingGroup Class Hierarchy

In our current implementation we only support deferrable servers, as they most closely match the support provided by the current version of the RTSJ. It maintains a periodic timer, a list of zero or more real-time threads that are associated with the server, and a cost overrun handler. The timer starts at the same time as the server, every time this timer fires, an asynchronous event handler sets the priority of all the registered threads back to their base priorities. The cost overrun handler is released automatically when the budget of the deferrable server has been consumed. Its purpose is to lower all the registered real-time threads' priorities. A thread is dropped from a server's monitoring list, when it terminates.

4.3 Real-Time Stream Evaluation

The programmer evaluates streams at different priority levels by submitting them to real-time ForkJoin pools at the desired priority.

In a partitioned system, each thread is constrained to execute on a single processing core, task migration is not allowed. The real-time ForkJoin thread pool creates one worker thread for each available processor on the system. Task migration is prevented by using CPU affinity. The implementation uses `javax.realtime.AffinitySet` to pin each worker thread in a pool to different processors.

The last requirement is to register each worker thread with an execution-time server, usually required by evaluating a stream in an aperiodic thread (see section 3.2.3). When a real-time ForkJoin thread pool is passed an instance of RTSJ ProcessingGroup during construction, each worker thread is registered with that group.

5. EVALUATION AND DISCUSSION

The main goal of our evaluation is to determine the impact of stream processing when performed using the our streaming processing framework compared to the standard Java 8 framework. We consider both the case of a periodic real-time thread and an aperiodic real-time thread evaluating streams and the impact this has on other real-time activity on the same platform.

The experiments were performed on a 3.4 GHz Intel Core i7-2600K processor with 4 physical cores, running Debian 7 Linux with a 3.2.0-4-rt-amd64 real-time kernel. Two physical cores were selected to be used by experiments using the Linux "taskset" shell command, and hyperthreading was turned off. The RTSJ VM uses aicas Jamaica version 6.4. Each experiments is performed 30 times and the error bands on the graph show the variation of the results obtained.

5.1 Real-Time Streams within Periodic Threads

This experiment considers three periodic real-time threads running on a single processor (Processor 1). The threads have the real-time characteristics shown in Table 1, all times are in milliseconds. The thread with the medium priority $T2$ evaluates a stream within its period. The execution time of this thread is measured using the standard sequential Java stream framework.

Table 1: Periodic Real-time Threads Characteristics

Name	Priority	WCET	Release	Period	Deadline
T1	High	821	0	4000	1000
T2	Mid	2704	0	4000	2500
T3	Low	1718	0	4000	4000

We first run the tasks set with sequential stream evaluation. The results are illustrated in Figure 6. As can be seen, the medium priority thread is unable to make its deadline. We now modify $T2$ so that it uses a parallel stream and configure the system so that the ForkJoin pool can use the same processor (Processor 1) and one extra processor (Processor 2). The results are illustrated in Figure 7. This demonstrates that now, although two processors are involved in evaluating the stream, $T2$ still does not make its deadline; in fact its response time increases. This is because the evaluation suffers from priority inversion on Processor 1, where $T3$ will execute ahead of the worker threads in the ForkJoin pool. Finally, in Figure 7 we show the results when a real-time stream is used. Priority inversion is avoided and the full benefits of parallel stream evaluation can be obtained, allowing $T2$ to make its deadline.

5.2 Real-Time Streams within Aperiodic Threads

This experiment investigates the interference that is introduced on a hard real-time activity by evaluating a real-time stream submitted by an aperiodic real-time thread. To simulate the situation where we want to get a good response time for the stream and yet still meet the hard real-time deadline, we assume that the hard real-time task is given a lower priority than the aperiodic soft task. A deferrable server is then associated with the aperiodic to bound the interference.

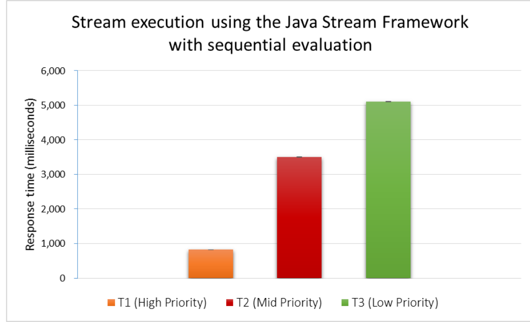

Figure 6: Threads Response Time in Millisecond. T2 Uses the Java Stream Framework with Sequential Evaluation.

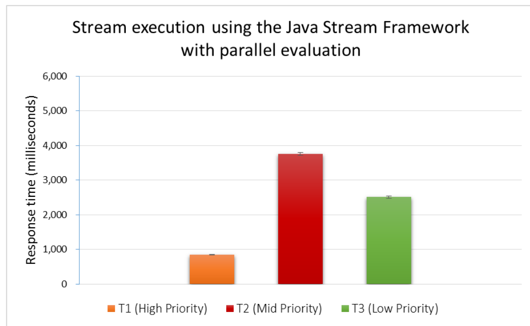

Figure 7: Threads Response Time in Millisecond. T2 Uses the Java Stream Framework.

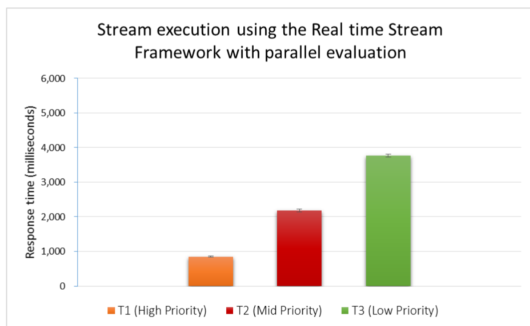

Figure 8: Threads Response Time in Millisecond. T2 Uses the Real-time Stream Framework.

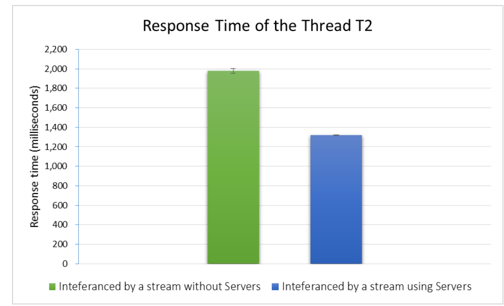

Figure 9: The Response Time of T2 in Millisecond.

One periodic (`T2`, low priority) real-time thread, and an aperiodic thread (`T1`, mid priority) were executed in this experiment. The thread `T1` evaluates a stream in its priority using the real-time stream framework. Their characteristics are described by Table 2, time unit are in millisecond. The different response time of thread `T2` are measured with $T1$ evaluating a stream using a deferrable servers (Period=3000, Cost=500), and without using a deferrable server.

Table 2: Real-time Threads Set Characteristics

Name	Priority	WCET	Release	Period	Deadline
T1	Mid	1200	0	-	-
T2	Low	820	0	3000	1600

The response time of thread `T2` is illustrated by Figure 9. Thread `T2` meets its deadline when a deferrable server is employed to bound `T1`'s stream evaluation. When there is no server employed, the stream evaluation in `T1` makes potentially unboundedly CPU demands, and results in $T2$ missing its deadline.

5.3 Discussion

The approach adopted by this work has assumed that the RTSJ is fixed and cannot be altered. From a pragmatic perspective, it is easier to obtain the code of the Java utilities and produce new versions rather than modify the `javax.realtime` package and the real-time VM. However, it is worth considering what changes would be need to be made to the RTSJ to allow a real-time thread pool to be created *without* changing the Java libraries.

Recall that the ForkJoin thread pool has been designed so that the programmer can provide its own factory for creating worker threads. The intention is that the factory should return a thread whose class extends the predefined `ForkJoinWorkerThread` class. This class has two methods that can be overridden: `onStart()` and `onTermination()`, which are called immediately a new worker thread is created and before a worker thread terminates respectively. Now suppose that the RTSJ provided a mechanism that allowed a Java thread to be given a real-time priority, which would result in it being scheduled by the RTSJ base priority scheduler. The thread becomes equivalent, from a scheduling perspective, to a real-time thread (rather than, say, an RTSJ no-heap real-time thread). From a semantic view point, it is still a Java thread. It cannot enter into RTSJ's scoped memory areas, be asynchronously interrupted etc.

Consider two possibilities: a thread can become real-time scheduled at any point in its execution by, for example, calling an appropriate static method

```
public static void makeRealtime(Thread t,
    PriorityParameters pri);
```

or, the thread can only be made real-time scheduled *before* it is started; so the above method would throw an `IllegalStateException` if the thread is already started.

In the former case, we can write a worker thread factory using the following approach:

```
static public class RealtimeWorker extends
    ForkJoinWorkerThread {
  public RealtimeWorker(ForkJoinPool pool,
      PriorityParameters pri) {
    super(pool);
    priority = pri;
  }
  @Override
  public void onStart() {
       makeRealtime(this, priority);
  }
  private PriorityParameters priority;

}

static final class RealtimeForkJoinWorkerThreadFactory
    implements ForkJoinWorkerThreadFactory {
  public RealtimeForkJoinWorkerThreadFactory(
       PriorityParameters pri) {
    this.priority = pri;
  }
  public final ForkJoinWorkerThread
      newThread(ForkJoinPool pool) {
    return new RealtimeWorker(pool, priority);
  }
  private PriorityParameters priority;
}
```

The latter case requires us to do the conversion in the constructor.

```
static public class RealtimeWorker extends
    ForkJoinWorkerThread {
  public RealtimeWorker(ForkJoinPool pool,
      PriorityParameters pri) {
    super(pool);
    priority = pri;
    makeRealtime(this, priority);
  }
}
```

In either approach, the default thread pool uses Java threads. Only if the application explicitly provides their own pool can they provide their own factory.

6. RELATED WORK

One of the main motivations for the introduction of Java 8 streams was to facilitate the processing of Big Data. Hence, much of the related work occurs in this context. Here we mention just a small section of this work. Perhaps the best known example of a Big Data framework is Apache Hadoop [1] (written in Java) which enables processing large data sets across clusters using the MapReduce [11] paradigm. Java 8 Streams, are of course not distributed, but the local processing perform by MapReduce could be rewritten using the Java 8 stream framework. Following on from Hadoop, much work has been done to try to optimize performance. For example, Spark [2] and MR3 [18] primarily use memory storage instead of writing all the intermediate results to hard-drive to achieve higher performance.

Languages that support the notion of streams, tend to focus on streaming data rather than batched data. StreamIt [22] is a programming language and a compiler specifically designed for processing infinite sequence of data, i.e. stream, on the platforms ranging from embedded systems to large scale and high performance system. StreamIt provide Java-like high-level data flow abstractions to improve programmer productivity, which includes several classes for stream processing, for example, the filter. The functionality of the filter is similar to the `filter()` or `map()` method in Java 8 Stream API. Borealis [7] also targets stream processing on distributed systems. It provides a Java API, and defines a set of stream operations, e.g. `map`, `join` etc.. However, both StreamIt and Borealis lack of real-time supports.

StreamFlex [19] extends Java and provides a stream programming framework, which allows program optimization and provides latency guarantees. StreamFlex is inspired by StreamIt and the Real-time Specification for Java (RTSJ). A graph for processing data can be constructed with provided classes, like filers etc.. The guarantees are provided by changing the virtual machine to support real-time periodic execution of threads, isolation of computational activities, a memory model that avoids suffering from garbage collectors.

Using work stealing algorithms for parallel stream processing in soft real-time systems is proposed in [17]. This work is based on the observation that using work stealing algorithms results in unpredictable latency. The latency is reduced by revising the work stealing strategies, for example, threads execute tasks in FIFO order, rather than LIFO order. Their work is targeted at streaming data. For batch processing, the LIFO order used by Java 8 is adequate as it timing constraint in on the completion of the streams terminal operation. Also, some important real-time features, for example, priority, were not considered. Anselmi and Gaujal [8] have proposed a framework for analysis for a real-time streaming model, which can predict the mean service time and waiting time of the stream element.

7. CONCLUSION AND FUTURE WORK

The goal of this work has been to leverage the Java 8 stream processing framework by considering how it can be used in a real-time Java environment, where that environment is defined by the RTSJ. The framework is closely integrated with Java's ForkJoin pool concurrency utilities, which provides some configuration flexibility. Unfortunately, there is a conflict in the assumptions made by its design and the requirements of the RTSJ: specifically, that application-provided worker threads must extend the class `ForkJoinWorkerThread`, which directly extends `java.lang.Thread`. In the RTSJ, real-time threads themselves directly extend `java.lang.Thread`. As a result, to achieve real-time stream processing we have had to make some changes to the Java 8 code. Although these changes are relatively minor, and can easily be implemented with a patch file, any changes are regrettable.

Arguably real-time support, as envisaged by the RTSJ, will never be incorporated into mainstream Java. Hence, the onus is on the RTSJ to provide better hooks to allow `java.lang` threads to be scheduled by a real-time scheduler.

Our initial experiments illustrate that significant priority inversions will occur if real-time threads use the Java 8 stream processing framework. They suggest that major benefits can be obtained from supporting a real-time stream processing framework. The relatively simple change to the RTSJ suggested in this paper would allow a Java 8 stream to be evaluated by a real-time thread pool without any changes to the stream processing framework. Of course, the devil is in the detail of such a proposed change, and the Expert Group in charge of upgrading the RTSJ (JSR 282) are currently evaluating the implication of supporting such a facility.

Our current work is addressing how our real-time streaming framework can be upgraded to support streaming data sources in addition to the batched sources typified by Java collections. This will require us to use different measures for real-time. In particular, latency and throughput will be more important than response time. We imagine that any limitations on the work-stealing algorithm in a real-time Fork/Join pool will become apparent. We will also be considering the impact of globally scheduled systems.

Although our overriding goal is do produce a real-time infrastructure, the schedulability analysis of a fork-join model of computation on multiple processors is still under development . For example, Maia et al. [16] have proposed a response-time analysis approach for a fixed priority global scheduling system. We will continue to monitor this work, with the goal of selecting the appropriate analysis for our approach.

8. ACKNOWLEDGMENTS

This work has been partially funded by the JUNIPER project under European Union's Seventh Framework Programme for research, technological development and demonstration – grant number 318763.

9. REFERENCES

[1] Apache software foundation: Apache hadoop. http://hadoop.apache.org/. Accessed July 3, 2015.

[2] Apache spark - lightning-fast cluster computing. https://spark.apache.org/. Accessed July 3, 2015.

[3] Jamaicavm | aicas.com. https://www.aicas.com/cms/en/JamaicaVM. Accessed July 5, 2015.

[4] JDK 8 Project. https://jdk8.java.net/. Accessed June 17, 2015.

[5] JEP 107: Bulk Data Operations for Collections. http://openjdk.java.net/jeps/107. Accessed June 19, 2015.

[6] RTSJ Main Page. http://www.rtsj.org/. Accessed June 17, 2015.

[7] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The design of the borealis stream processing engine. In *CIDR*, volume 5, pages 277–289, 2005.

[8] J. Anselmi and B. Gaujal. Performance evaluation of work stealing for streaming applications. In *Principles of Distributed Systems*, pages 18–32. Springer, 2009.

[9] Y. Chan, I. Gray, A. Wellings, and N. Audsley. Exploiting multicore architectures in big data applications: The juniper approach. *Proceedings of MULTIPROG*, 2014.

[10] Y. Chan, A. Wellings, I. Gray, and N. Audsley. On the locality of java 8 streams in real-time big data applications. In *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '14, pages 20:20–20:28, New York, NY, USA, 2014. ACM.

[11] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[12] I. Gray, Y. Chan, N. C. Audsley, and A. Wellings. Architecture-awareness for real-time big data systems. In *Proceedings of the 21st European MPI Users' Group Meeting*, page 151. ACM, 2014.

[13] O. Group/IEEE. *The open group base specifications issue 7, ieee std 1003.1, 2013 edition*. IEEE/1003.1 2013 Edition, The Open Group, 2013.

[14] D. Lea. A Java Fork/Join Framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM.

[15] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in a hard real-time environment. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 261–270, 1987.

[16] C. Maia, L. M. Nogueira, L. M. Pinho, and M. Bertogna. Response-time analysis of fork/join tasks in multiprocessor systems. In *25th Euromicro Conference on Real-Time Systems*, 2013.

[17] S. Mattheis, T. Schuele, A. Raabe, T. Henties, and U. Gleim. Work stealing strategies for parallel stream processing in soft real-time systems. In *Architecture of Computing Systems–ARCS 2012*, pages 172–183. Springer, 2012.

[18] A. Shinnar, D. Cunningham, V. Saraswat, and B. Herta. M3r: increased performance for in-memory hadoop jobs. *Proceedings of the VLDB Endowment*, 5(12):1736–1747, 2012.

[19] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. Streamflex: high-throughput stream programming in java. *ACM SIGPLAN Notices*, 42(10):211–228, 2007.

[20] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1:27–69, 1989.

[21] X. Su, G. Swart, B. Goetz, B. Oliver, and P. Sandoz. Changing engines in midstream: A java stream computational model for big data processing. *Proc. VLDB Endow.*, 7(13):1343–1354, Aug. 2014.

[22] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction*, pages 179–196. Springer, 2002.

[23] A. Wellings, P. Dibble, and D. Holmes. Supporting multiprocessors in the real-time specification for java version 1.1. In *Distributed, Embedded and Real-time Java Systems*, pages 1–22. Springer, 2012.

[24] A. J. Wellings and M. S. Kim. Processing group parameters in the real-time specification for java. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '08, pages 3–9, New York, NY, USA, 2008. ACM.