Real-Time Stream Processing in Java

HaiTao Mei, Ian Gray and Andy Wellings

University of York, UK

Abstract. This paper presents a streaming data framework for the Real-Time Specification for Java, with the goal of levering as much as possible the Java 8 Stream processing framework whilst delivering bounded latency. Our approach is to buffer the incoming streaming data into micro batches which are then converted to collections for processing by the Java 8 infrastructure which is configured with a real-time ForkJoin thread pool. Deferrable servers are used to limit the impact of stream processing activity on hard real-time activities.

1 Introduction

A stream processing system consists of a collection of modules that compute in parallel and communicate via channels [16]. Modules can be either *source capturing* (that pass data from a source into the system), *filters* (that perform atomic operations on the data) and *sinks* (that either consume the data or pass it out of the system). Real-time stream processing systems are stream processing systems that have time constraints associated with the processing of data as it flows through the system from its source to its sink. Typically, the sources of streaming data may originate from an embedded system (for example, the Large Hadron Collider can output a raw data stream of approximately 1PB/s [17]) or from a variety of internet locations (e.g., Twitter's global stream of Tweet data). In the context of this work, we assume stream processing is computationally intensive and is a soft real-time activity. Hence, we are interested in the latency of processing each element in the stream and bounding the impact that stream processing has on other hard real-time activities that might be sharing the same computing platform.

The most recent version of Java (Java 8) has introduced Streams and lambda expressions to support the efficient processing of in-memory stream 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.concurrent` ForkJoin 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*. Hence it is targeted at batched streams.

Previously we have evaluated the efficacy of Java 8 Streams as a framework for processing real-time batched streams and have found it inadequate [14] even

when used in conjunction with the Real-Time Specification for Java (RTSJ). The essence of the problem is that using the ForkJoin framework introduces priority inversions, as it is not possible to construct the worker threads as real-time threads. (By definition, all standard Java threads have a lower priority than all real-time Java threads.) We have suggested changes to JSR 282[1] to circumvent this problem, which have now been adopted. We have also presented a real-time stream processing framework for batched data, which includes a real-time ForkJoin pool [14]. In this paper, we consider how real-time *streaming* data sources can be handled, and propose an extended framework. We assume the presence of a multicore platform hosting version 2.0 of the RTSJ. Our goal is to, where possible, use the proposed Java 8 Streams and evaluate their adequacy for a real-time environment.

The paper is structured as follows. Section 2 introduces Java 8 Streams. Related work is considered in Section 3. In Section 4, our overall approach is discussed. This is followed in Section 5 by a description of our implementation. Section 6 then evaluates our approach by comparing its performance against the regular Java concurrency framework. Finally we present our conclusions.

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, which cannot have a default implementation.

A sequence of operations with a 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. 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

[1] The JCP Expert Group are due to release a new version of the RTSJ (Version 2.0) in early 2016. This version will be compatible with Java 8.

simple word count example can be described by the following code using the Stream API and Lambda Expressions:

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

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.

3 Related Work

The StreamIt [7] language is specifically designed for processing data streams on platforms ranging from embedded systems to large scale and high performance system. StreamIt defines several data flow abstractions for stream processing, such as `filter` (similar to the `filter()` method in Java 8 Streams), and a Java-like high-level API to access these abstractions. StreamIt uses the synchronous data-flow model and allows thus very aggressive compiler optimisations. Borealis [8] focuses on distributed stream processing, and defines a set of stream operations, e.g., `map`, `join` etc., written in the Java API. Neither StreamIt nor Borealis provide real-time support.

Storm [3], Heron [11], and Samza [2] are distributed stream processing frameworks. Computation graphs (typically directed acyclic graphs) can be constructed to represents the stream processing logic, where edges represent data flow and vertexes represent computation. A data push model is employed for stream dispatching. Spark Streaming [6] is a distributed stream processing library that is built on top of Spark [1]. Spark Streaming periodically groups the received data in streams into a micro batch, and process it with the Spark engine. However, none of the above are integrated into a real-time environment.

Inspired by StreamIt and the RTSJ, StreamFlex [15] is a stream processing framework which provides bounded latency. StreamFlex provides a set of classes, such as `filters`, which are used to construct computation graphs for stream processing. The processing latency is bounded by changing the virtual machine to support real-time periodic execution of threads, computational activities isolation, and a memory model that avoids the use of garbage collectors. However, as a result StreamFlex is a very different programming model to more standard languages and is not compatible with Java 8 Streams. Also it does not

support priority assignment to limit the impact of soft real-time streaming work on hard real-time activities.

AdaStreams [10] is a stream processing library with a run-time system that targets at multiprocessor platforms. `Filter` that is similar to the one in StreamIt, `Splitter` and `Joiner` are created as stream processing abstracts, and a processing graph can be constructed by connecting them together. However, it does not support real-time constraints.

Mattheis [13] proposed a framework that uses work stealing algorithms in parallel stream processing in soft real-time systems. This work investigated the variance of latency when using work stealing algorithms with different strategies. It determined that latency is reduced by using FIFO ordering when stealing from the global queue, and when using LIFO ordering for stealing from the local queue. This is the approach adopted by Java 8, which we use unchanged.

Extending Storm to provide real-time support is proposed in [9], which defines a real-time processing stack including a real-time OS, a real-time JavaVM, and real-time versions of Storm's classes. Two core concepts in Storm: the `Spout` (source of streams) and the `Bolt` (computation logic in the data flow graph) are extended to be sporadic activities which can be configured with minimum interval times, computation times, and priorities. In addition, a fixed-priority scheduler is provided. A drawback of Storm is that it uses an eager computation model which does not provide all of the optimisation opportunities of the lazy model of Java 8.

4 A RTSJ-based Real-time Stream Processing Framework

The overall goal of the work is to leverage as much as possible the Java 8 Stream processing framework within an RTSJ environment. The fundamental problem that must be addressed is how to map a *streaming* data source into *batched data* so that it can be processed with our current real-time stream processing framework. The proposed approach is to group the streaming data into *micro batches*, each of which can be treated as a static data source. Then a stream can be created to process each micro batch. The overview of this approach is shown in Figure 1.

The size of each micro batch is determined by two factors: the *input data volume* – incoming data is buffered up to an application-defined maximum amount and once the buffer is full the batch is processed; and *time* – individual data elements of the input data stream have an application-defined maximum latency for their processing, so a micro batch must be released early if the processing time of the batch is such that a data item may miss its deadline. Figure 2 illustrates the approach. The handler turns the buffer into a collection, which can then be processed using the stream processing framework. Note that, when the batch is processed in the case of a full buffer, the next timeout will be reset to be $time_{now} + timeout$. The micro batch will be processed using the real-time stream processing framework, and the underlying work is performed by a real-

Fig. 1: The overview of real-time processing streaming data.

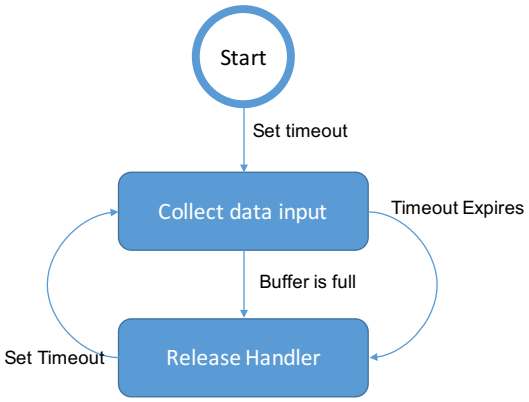


Fig. 2: The Real-Time Micro Batching approach.

time ForkJoin thread pool at a desired priority. The approach is described using the Real-Time Specification for Java (RTSJ). Three classes are defined:

`Receiver`: Maintains a dedicated real-time thread which is used to receive data from a source, e.g., a TCP/IP socket. It also maintains a buffer that stores the received data, and when enough data has arrived it notifies the `Handler`. Users can define their own receivers; for example, to receive data from different data flow sources.

`Timer`: Manages when the next timeout occurs. When fired, the next fire time is automatically reset.

`Handler`: Contains the user-defined processing logic for each micro batch using Java 8 Streams. Once notified, it retrieves data from the receiver as a Collection and performs the processing logic.

4.1 The Real-Time Micro Batching Stream API

The approach described above is implemented in a new framework called `BatchedStreams`. `BatchedStreams` adopts the described micro batching approach to provide real-time behaviour.

This overall approach is quite straightforward and allows the data flow behaviour to be captured well. However, the micro batching approach is difficult to implement in a way which allows user code to be as concise as when using standard Java 8 Streams. This is because a Java 8 Stream pipeline (e.g., `.map().filter().forEach()`) cannot be created outside of the context of a Stream, and a Stream can only have a single source of input data.

To address this problem, defined as part of `BatchedStreams` are `ReusableStreams`. `ReusableStreams` implement the standard Java Streams API, but also allow their processing pipeline to be reused over different input Collections (i.e., to apply to multiple batches) once its terminal operation has been invoked. `ReusableStreams` also allow more concise code through the use of Java 8 lambda expressions to specify the processing logic, as detailed in the follow sections.

`ReusableReferencePipeline` implements the `ReusableStream` interface, and represents a reusable stream of Java objects. In addition, we have implemented the equivalent classes for Java's primitive types.

The Structure of BatchedStream `BatchedStreams` maintain instances of `Receiver`, `Timer` and `Handler`. The instance of `ReusableStream` that is used to represent processing logic is also maintained by the `BatchedStream`. The `BatchedStream` starts the timer and the receiver, and sets their handler. Once a micro batch is released, the handler processes it using the `ReusableStream`, and optionally, using the `BatchedStreamCallback` to further process (e.g., to accumulate) the result. The `BatchedStreamCallback` is a functional interface, the method of which is invoked by the `ReusableStream` once its terminal operation returns, and acquires the returned result. The reusable pipeline must be initialised before processing any micro batch. A reusable pipeline can either be initialised then passed to the constructor of the `BatchedStream`, or be initialised by a functional interface named `ReferencePipelineInitialiser`, which is required by the constructor. Functional interfaces enable the `BatchedStream` to take the advantage of Java's lambda expressions to make code more concise. An example, which calculates how many words have been received from a TCP/IP socket, is described as follows:

```
long count = 0;
BatchedStream<String> textStreaming = new BatchedStream<>(
    new StringSocketRealtimeReceiver(...),
    p -> p.flatMap(line -> Stream.of(line.split("\\W+"))).count());
textStreaming.setCallback( r -> count += (long) r );
textStreaming.start();
```

The pipeline here counts how many words are within a micro batch, and is the same as it would be with normal Java 8 Streams. The pipeline is initialised using a lambda, and the callback that accumulates all the local results is set.

The `BatchedStream` cannot extend the `ReusableReferencePipeline` class because several terminal operations that are defined in the `Stream` interface are required to return a result. Applying terminal operations, such as `reduce`, on a `BatchedStream` represents a reduction of all the data elements from the data source. However, the `BatchedStream` generates one local result for every micro batch release.

Stream Processing in Real-Time To evaluate a Stream under real-time constraints requires the use of a real-time ForkJoin thread pool. The standard

Java thread pool is insufficient because all standard Java 8 Streams execute using the same system-wide ForkJoin pool. This pool consists of standard Java threads which do not support real-time properties [14]. Furthermore, standard Java streams are defined to have a lower priority than all real-time (RTSJ) threads.

The real-time constraints are met by `BatchedStream`s that submit each micro batch and its corresponding reusable pipeline to a real-time ForkJoin thread pool [14]. This is a pool in which each worker thread is an aperiodic real-time thread and the priority of each worker thread is assigned when the pool is created.

Bounding the Impact of BatchedStream Typically stream data processing is computationally-intensive, and the unpredictability of data flows makes the corresponding CPU demand unpredictable. In an RTSJ runtime environment, we assume that stream processing occurs within a soft real-time task. With all such soft real-time activities, there is tension between achieving a short response time without jeopardising any hard real-time activities. Running stream data 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 impact of stream data processing can be bounded by associating servers that are described in [14] with real-time thread pools. Performing the previous example with real-time constraints requires the server and the priority to be configured. A real-time ForkJoin thread pool with the desired priority associated is created to process each micro batch using the given pipeline.

```
long count = 0;
BatchedStream<String> textStreaming = new BatchedStream<>(
    new StringSocketRealtimeReceiver(...), new PriorityParameters(26),
    new  DeferrableServer(...),
    p -> p.flatMap(line -> Stream.of(line.split("\\W+"))).count());
textStreaming.setCallback( r -> count += (long) r );
textStreaming.start();
```

5 Implementation

The real-time stream processing framework is implemented in the RTSJ. The RTSJ execution environment used in this work was JamaicaVM [4]. JamaicaVM provides support for multiprocessor applications including affinity sets. Timers are implemented using the RTSJ's `PeriodicTimer` class. Handlers are implemented using the RTSJ `AsyncEventHandler`, which submits a micro batch to be processed when either the event buffer is full or the next timeout occurs. The processing infrastructure uses our real-time ForkJoin thread pool, which is

described in [14]. Repeatedly applying the same pipeline on each micro batch is achieved by using our `ReusableStream` framework, described below.

5.1 The `ReusableStream` Pipeline

Recall that the purpose of `ReusableStream`s is to create a pipeline of operations which may be repeatedly applied to different data collections. In addition, the `ReusableStream` must remain compatible with the existing Java Stream API.

`ReusableStream`s were defined as an interface that extends the Java `Stream` interface. They define a method named `processData` which takes a reference to a data source (Java Collection) to be processed, and optionally a callback which is called to present the result.

In a `ReusableStream`, operation pipelining uses a linked list. Each node maintains one intermediate operation and its arguments, and each intermediate operation returns a new node that will be appended to the tail of the linked list. When the terminal operation is invoked, the execution thread travels through the pipeline, and performs each operation on each data element. In order to make a pipeline reusable, the terminal operation is added to the linked list as well, rather than forcing stream evaluation. This is the only difference between the use of standard Java streams and `ReusableStream`s.

5.2 Real-Time Stream Processing

The stream is processed using `BatchedStream`s at different priority levels by submitting the `ReusableStream` that is used to process each micro batch to a real-time ForkJoin pool at the desired priority. In a globally scheduled system, each worker thread within the real-time ForkJoin thread pool can execute on, or migrate to any available processor. No CPU affinity is applied. In a fully-partitioned system, each worker thread within the real-time ForkJoin thread pool is constrained to execute on one processor, and task migration is forbidden using CPU affinity. The implementation uses `javax.realtime.AffinitySet` to pin each worker thread within a real-time ForkJoin pool to different processors. A semi-partitioned system is a mix of these two schemes. The semi-partitioned system extends the fully partitioned system, so that a certain number of tasks can migrate to a set of allowed processors. In a semi-partitioned system, different worker threads are allocated with different affinity sets, which determine the set of processors the task can migrate to.

6 Evaluation

The main goal of the evaluation is to determine the latency of stream processing and its impact on other real-time activities. First, we compare the latency of processing each data element in a stream using `BatchedStream` and the real-time stream processing framework (described in Section 4.1) with the standard Java 8 Stream processing framework. The experiments were performed on a 3.7

GHz Intel Core i7 processor (with 4 physical cores) platform, running Debian 7 Linux with a 3.2.0-4-rt-amd64 real-time kernel. Three physical cores were selected to be used by experiments using the Linux "taskset" shell command, and hyperthreading was turned off. The RTSJ VM uses the aicas JamaicaVM version 6.5.

6.1 Latency of Stream Processing

This experiment considers stream processing activities using a `BatchedStream` running on one processor (Processor 2). The same processor also hosts three periodic real-time threads at the same time. The experiment demonstrates that bounded latency can be provided when using `BatchedStreams` to process streaming data.

The underlying real-time stream processing framework that is used by `BatchedStreams` and the real-time thread employed by the receiver have medium priority. The experimental data flow is simulated using a real-time thread running on Processor 1 which sends one pre-generated string text per random interval at a low rate (minimum inter-arrival time (MIT) = 200 milliseconds, maximum inter-arrival time (MAT) = 400 milliseconds). The execution time for processing each string is set to 34 milliseconds, and the deadline is 60 milliseconds, thereby illustrating the computationally intensive nature of the processing required. We set the period of micro batching in the `BatchedStream` to be 10 milliseconds. These values have been chosen to highlight the impact of a varying data arrival rate on our framework. In addition, the buffer size of the receiver is set to be 1024 elements, which ensures that storing all the elements within the data flow in this experiment will not trigger the early release of the micro batch. The other real-time threads have the real-time characteristics shown in Table 1, all times are in milliseconds.

Table 1: Periodic Real-time Threads Characteristics

Name	Priority	WCET	First Release	Period	Deadline	Processor ID
T1	Low	28	0	100	100	2
T2	Low	28	130	200	200	2
T3	Low	28	50	400	400	2

We start the stream processing at time 0, and real-time threads according to their release characteristics. The thread's first release times are offset to ensure a more balanced background load. The data flow starts 400 milliseconds after the stream processing and generates 100 strings. The latency of each data element is measured, and illustrated in Figure 3. As we can see, the latency of each data element in the data flow varies significantly when using the Java Stream framework as the processing infrastructure. As a consequence, some of data elements

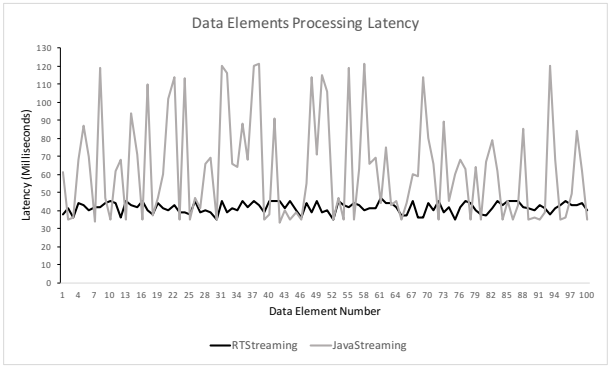


Fig. 3: The Latency of Data Elements In Data Streams

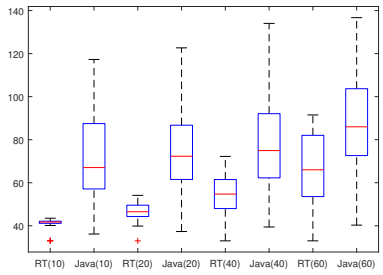
miss their deadlines. This is because the processing suffers from priority inversion on Processor 2, where all the periodic real-time threads will pre-empt the worker threads in the standard Java ForkJoin pool. The variance of the latency is notably reduced when using the real-time stream processing framework, as shown by the black line in Figure 3. Priority inversion is avoided, and all data elements meet their deadlines. Note that, the variance of the latency when using the real-time stream processing framework is due to the variation of the time waiting in the buffer, as data can arrive at any time within the micro batching interval.

We repeat each experiment 30 times and with different intervals of micro batching (10, 20, 40, and 60 milliseconds). The distribution of latency of using the different frameworks are illustrated in Figure 4a. This shows that the latency is well bounded when employing the `BatchedStream` with the help of the real-time stream processing infrastructure. As expected, the larger the micro batching interval, the larger the variance in the latency. With standard Java, the pattern is the same, only with much larger variance in the response times.

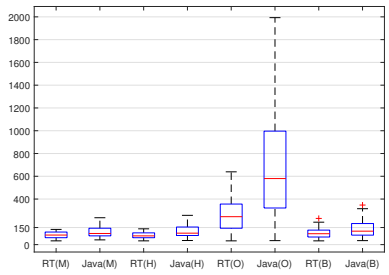
6.2 Different Data Rates

The experiments performed in Section 6.1 had an inter-arrival time between strings in the range of 200-400 milliseconds. This represents a low load on the system. The experiments reported in this section consider the impact of varying this arrival rate to represent medium (M), high (H) and overload (O) workloads. The experiments and the streams investigated and their underlying processing frameworks are described in Table 2. The configuration of processors, and interference from real-time threads are the same as the previous experiment. The buffer size is 1024, and the interval of micro batching is 100 milliseconds. Each stream under this experiment contains 100 data elements, and the sequence of arrival times of each data element was generated before the experiment.

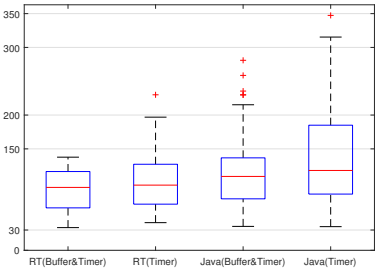
Again, each experiment was repeated 30 times, and the results are shown in Figure 4b. The latency of streams at the medium and high rate is well bounded



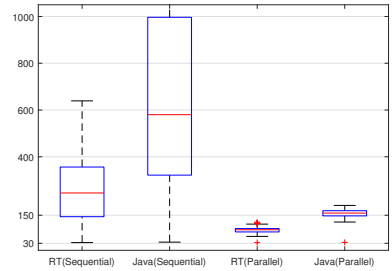
(a) The Latency of Data Elements in Streams with Different Intervals.



(b) Latency Distribution of Processing Different Rate Streams.



(c) Using Suitable Buffer Size To Handle Bursts.



(d) Latency Distribution of Parallel Stream Processing.

Fig. 4: Latency Distribution Experiment Results.

by employing `BatchedStreams` with the real-time processing framework. However, there are few deadline misses when the stream is bursty, and there are many deadline misses when the rate is very high (and therefore results in system overload). These issues will be described in the following sections, and proposed solutions will be given. For all the cases, the latency of using standard Java frameworks cannot be guaranteed to meet the deadline because of the priority inversion issue.

6.3 Burst Handling

With a bursty stream, there were deadline misses when releases of each micro batch within the `BatchedStream` was purely triggered by timeouts. The reason is that the waiting time of a data element can result in deadline misses. For example, consider 4 data elements (d_1, d_2, d_3, d_4) that arrive in the system at time t when a burst occurs, while the next timeout is $t + 90$, thus, the latency of the last data element $Latency_{d_4} = 90 + ResponseTime_{d_4}$.

The minimum latency of d_4 in this case is determined when the response time of each data element equals their execution time (28 ms) and when there is no preemption or blocking.

Table 2: Streams And Their Processing Frameworks, MIT and MAT Represents The Minimum and Maximum Interval. Times Are In Milliseconds.

Name	Processing Framework	MIT	MAT	Burst Size	WCET	Deadline
RT(M)	Real-Time	100	200	0	28	150
Java(M)	Java	100	200	0	28	150
RT(H)	Real-Time	50	100	0	28	150
Java(H)	Java	50	100	0	28	150
RT(O)	Real-Time	20	40	0	28	150
Java(O)	Java	20	40	0	28	150
RT(B)	Real-Time	200	400	4	28	150
Java(B)	Java	200	400	4	28	150

$$Min(Latency_{d_4}) = 90 + WCET_{d_1} + WCET_{d_2} + WCET_{d_3} + WCET_{d_4}$$

Thus, the best-case latency of d_4 in this case is 202 milliseconds, therefore missing its deadline.

One possible solution to this problem is to reduce the interval of micro batching, i.e., the timeout, so that the latency is within the deadline even when bursts occur. In this experiment, the maximum interval is $deadline - (WCET_{d_1} + WCET_{d_2} + WCET_{d_3} + WCET_{d_4})$, i.e., $150 - (28 + 28 + 28 + 28) = 38$ milliseconds. However, the stream rate in this experiment is generally slow, bursts only occur infrequently. Hence, using this interval to handle this stream is not efficient, because this introduces many releases of the handler where there is no data in the buffer.

An alternative approach, and the one we adopt, is to vary the buffer size to enable data to be processed immediately when bursts occur. The waiting time will be reduced, and therefore, the data within bursts can meet their deadlines. In this experiment, the buffer size of the `BatchedStream` is configured to be 4 elements, i.e., the burst size. Redoing the experiments for the bursty stream, the results are illustrated in Figure 4c. The latency is reduced so that all the data elements now meet their deadlines, which is shown in the first plot in Figure 4c. The second and the forth plots are taken from from the last experiment for easy comparison. The third plot represents the latency distribution of employing standard Java framework.

The interval of micro batching, i.e., the timeout, the maximum count of data arrived during this interval, and the execution time of each data determine the maximum latency of a stream. For example, assuming the maximum data arrival during the interval is N, the maximum latency can be represented by the following formula when there is no preemption from higher priority activity.
$Max(Latency) = Interval + \sum_{i=1}^{N} WCET_{d_i}$
When the maximum latency equals the deadline, N can be calculated. Thus, in the bursty case where the burst size is unknown, the buffer size should be configured to be at most N in order to provide bounded latency for bursts. Note

that, this is based on the assumption that there are always enough computation resources so that even a very large burst can be processed within the deadline.

6.4 Parallel Stream Processing

With the experiments performed in 6.2, a stream whose MIT is 20 and MAT is 40 milliseconds cannot be guaranteed to meet the deadline because the system is overloaded. The computation of each data element may require more time than the data arriving interval (minimum interval is 20 milliseconds, but the execution time is 28 milliseconds). The experiment reported in the section investigates the latency of parallel stream processing, by allocating another processor (Processor 3) to the `BatchedStream`'s underlying processing infrastructure, for both the real-time and standard Java versions. The rest of the configuration remains unchanged. The results are illustrated in Figure 4d, where the first two plots are taken from the original experiment (see Section 6.2). The last two plots represent the latency distributions for the stream that was processed in parallel using the real-time and standard Java infrastructure. Each data element in the stream meets its deadline when using the parallel real-time processing infrastructure. Deadline misses still occur when using standard Java infrastructure because of the priority inversion occurring on Processor 2.

7 Conclusion and Future Work

This work has proposed an efficient general purpose real-time stream processing framework, based on a standard programming language which targets shared memory, multiprocessor platforms. The `BatchedStream` API, that uses the Java 8 Streams framework has been defined. With the help of `ReusableStreams`, `BatchedStreams` enable a real-time stream processing job to be defined with concise code. `BatchedStreams` provide bounded latency, using a real-time micro-batching model in conjunction with an underlying processing infrastructure that utilises a real-time ForkJoin thread pool to avoid priority inversion issues. Configuring the affinity sets of worker threads in a real-time ForkJoin thread pool allows different scheduling schemes, including global, fully partitioned, and semi-partitioned, to be supported.

Whilst `BatchedStreams` provide real-time stream processing facilities, its latency analysis for multiprocessors is subject to future work. Maia et al. [12] have proposed an approach for response time analysis of a `BatchedStream`-like processing model, i.e., ForkJoin pool, on a fixed priority global scheduling system. We will use this approach as a starting point for the analysis of the waiting time of each data element in a stream.

`BatchedStreams` currently only provide pipeline-style stream processing. Our current work is addressing how `BatchedStreams` can provides DAG-style computation logic when processing streams in real-time. This requires the current pipeline evaluation model to be augmented with extra operations (such as `shuffle`, and `collect`).

References

1. Apache Spark - Lightning-Fast Cluster Computing. http://spark.apache.org/. Accessed December 5, 2015.
2. Apache Samza. http://samza.apache.org. Accessed December 5, 2015.
3. Apache Storm. http://storm.apache.org/. Accessed December 5, 2015.
4. JamaicaVM — aicas.com. https://www.aicas.com/cms/en/JamaicaVM. Accessed December 1, 2015.
5. JEP 107: Bulk Data Operations for Collections. http://openjdk.java.net/jeps/107. Accessed December 5, 2015.
6. Spark Streaming — Apache Spark. http://spark.apache.org/streaming/. Accessed December 5, 2015.
7. StreamIt-Research. http://groups.csail.mit.edu/cag/streamit/shtml/research.shtml. Accessed December 5, 2015.
8. 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.
9. P. Basanta-Val, N. Fernández-García, A. Wellings, and N. Audsley. Improving the predictability of distributed stream processors. *Future Gener. Comput. Syst.*, 52(C):22–36, Nov. 2015.
10. G. Hong, K. Hong, B. Burgstaller, and J. Blieberger. Adastreams: A type-based programming extension for stream-parallelism with Ada 2005. In *Reliable Software Technologiey - Ada-Europe 2010, 15th Ada-Europe International Conference on Reliable Software Technologies, Valencia, Spain, June 14-18, 2010. Proceedings*, pages 208–221, 2010.
11. S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM.
12. 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.
13. 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.
14. H. T. Mei, I. Gray, and A. Wellings. Integrating Java 8 Streams with The Real-Time Specification for Java. In *Proceedings of the 13th International Workshop on Java Technologies for Real-time and Embedded Systems*, page 10. ACM, 2015.
15. 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.
16. R. Stephens. A survey of stream processing. *Acta Informatica*, 34:491–541, 1997.
17. X. Vidal and R. Manzano. Taking a closer look at LHC. http://www.lhc-closer.es/1/3/12/.