Real-Time Stream Processing in Java

HaiTao Mei, Ian Gray, Andy Wellings

Real-Time Systems Group

Department of Computer Science

University of York, UK

Summary

- Introduction
- Project Goals
- A Real-time Data Streaming Framework
- Evaluations
- Conclusions

Introduction

Java 8 has introduced *Streams* and *lambda* expressions to support the efficient processing of in-memory ***static*** data sources (e.g. a Java Collection)

```
//myList = ["a1", "a2", "b1", "c2", "c1"]

myList.parallelStream()
      .filter(s -> s.startsWith("c"))
      .map(String::toUpperCase)
      .forEach(System.out::println);
```

Introduction

- Pipeline – A sequence of operations and the data source
- A pipeline consists zero or more intermediate operations, and a terminal operation
    - Intermediate operations return a new stream
    - Terminal operations force the evaluation of a stream, and return a result

Introduction

- Sequential Case

  Performing all the operations in the pipeline on each data element sequentially by the thread which invoked its terminal operation

- Parallel Case

  Parallel stream will partition the processing, and all the created parts will be evaluated in parallel with the help of a ForkJoin thread pool

Project Goals

- Develop a streaming data framework for real-time (RTSJ) Java applications
- Using the facilities of Java 8 Streams

The Nature of Real-Time Streaming

- Data item processing is sensitive to the *latency*
- Data items arrive *sporadically*
- *Micro batching* (i.e., collect the individual data items into micro batches) improves efficient of processing, for example the Spark streaming framework
  - In addition, makes it possible to use Java 8 (Parallel) Stream operations, e.g. map, filter, reduce etc., when processing data flows in real-time

RTS York

Using Micro Batching

- Latency → *Timeout*

  i.e., data must be processed within a finite time after arriving

  For example, data items rate is 1/second, processing time is 50ms, max latency is 200 ms → *Timeout* should be at most 150ms

- Sporadically → *Batch size*

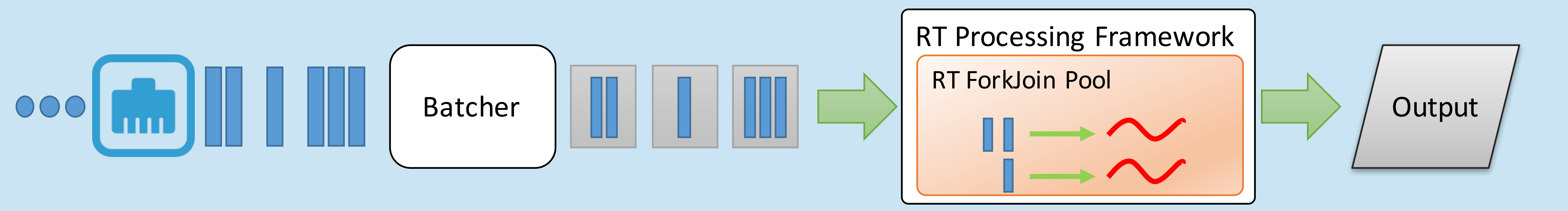
  i.e., data must be processed once batch reaches a size

  For example, same data flow, but *occasionally* has bursts (3 items at the same time), →

  timeout=50 is inefficient ✘

  Batch size should be 3 ✓

The Real-Time Micro Batching



- Batcher

    groups a streaming data source into batched data

- Processing Framework

    Java 8 streams and our real-time ForkJoin thread pool

Determine The Micro Batch Size

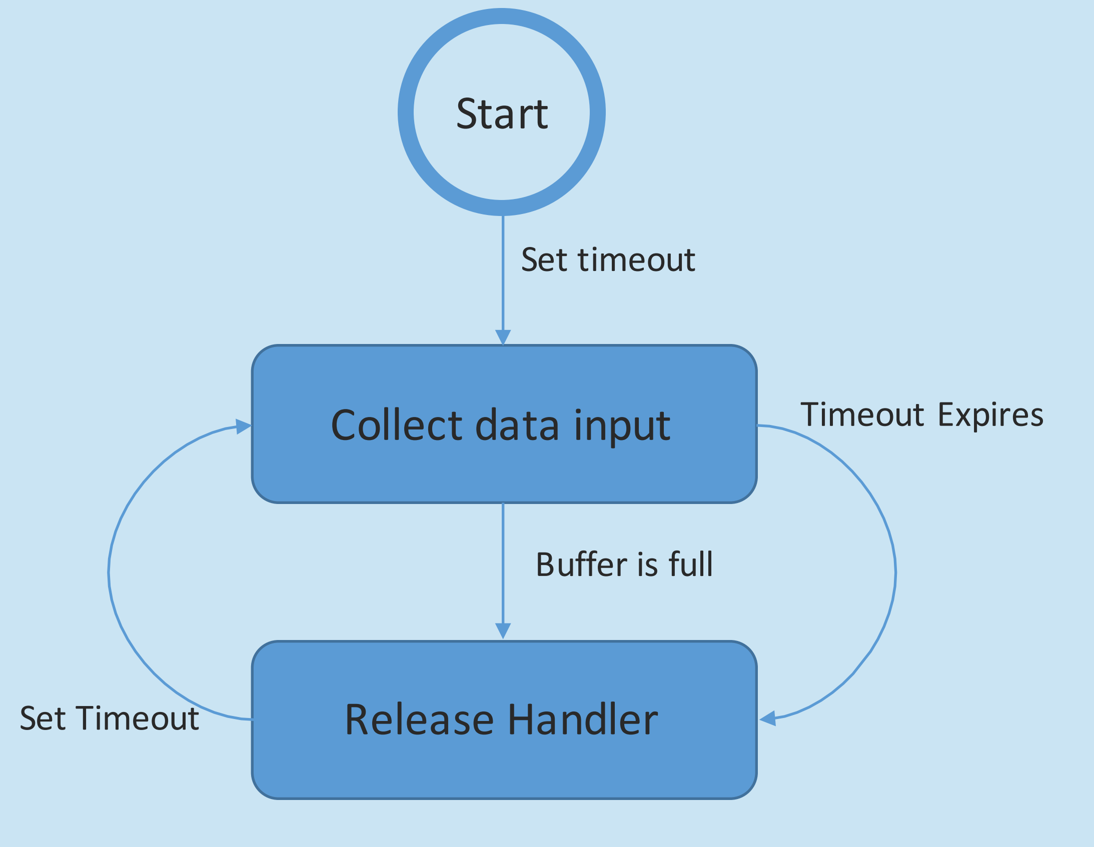
The processing of each micro batch is triggered by two factors:

- Either, 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

- Or, timeout

  A micro batch must be released early if the processing time of the batch is such that a data item may miss its deadline



The Real-Time Data Streaming Framework

- Receiver – receives data from a data source
- Timer – maintains the timeout
- Handler – does the actual processing (in parallel)

The Receiver

- Maintains a dedicated real-time thread which is used to receive data from a source, e.g., a TCP/IP socket

- Maintains a buffer that stores the received data
  - When enough data has arrived it notifies the Handler, and
  - Reset the next timeout

The Timer

- Manages when the next timeout occurs
    - When fired, the next fire time is automatically reset

The 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

Bounding the Impact of data Streaming

- Typically data flow processing is computationally intensive
- Usually occurs within a soft real-time task
- Running it at the lowest priority -> <u>bad response times</u>
- Running it at too high a priority -> <u>cause critical activities to miss their deadlines</u>

Bounding the Impact of data Streaming

Servers are typically used to support computationally intensive soft real-time tasks to give them good response times but bound their impact on hard real-time tasks

We use the approach suggested by Wellings and Kim [2] to allow a range of servers to be associated with our data processing tasks

[2] Wellings, Andy, and MinSeong Kim. "Processing group parameters in the real-time specification for Java." *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*. ACM, 2008.

RT Data Streaming Framework Example

```
long count = 0;

BatchedStream<String> textStreaming = new BatchedStream<>(
  new StringSocketRealtimeReceiver(...),
  new RelativeTime(5000,0),      /* timeout = 5s */
  new PriorityParameters(26),    /* priority */
  new DeferrableServer(...),     /* execution-time server */
  p -> p.flatMap(line -> Stream.of(line.split("\\W+"))).count());

textStreaming.setCallback( r -> count += (long) r );
textStreaming.start();
```

Evaluation

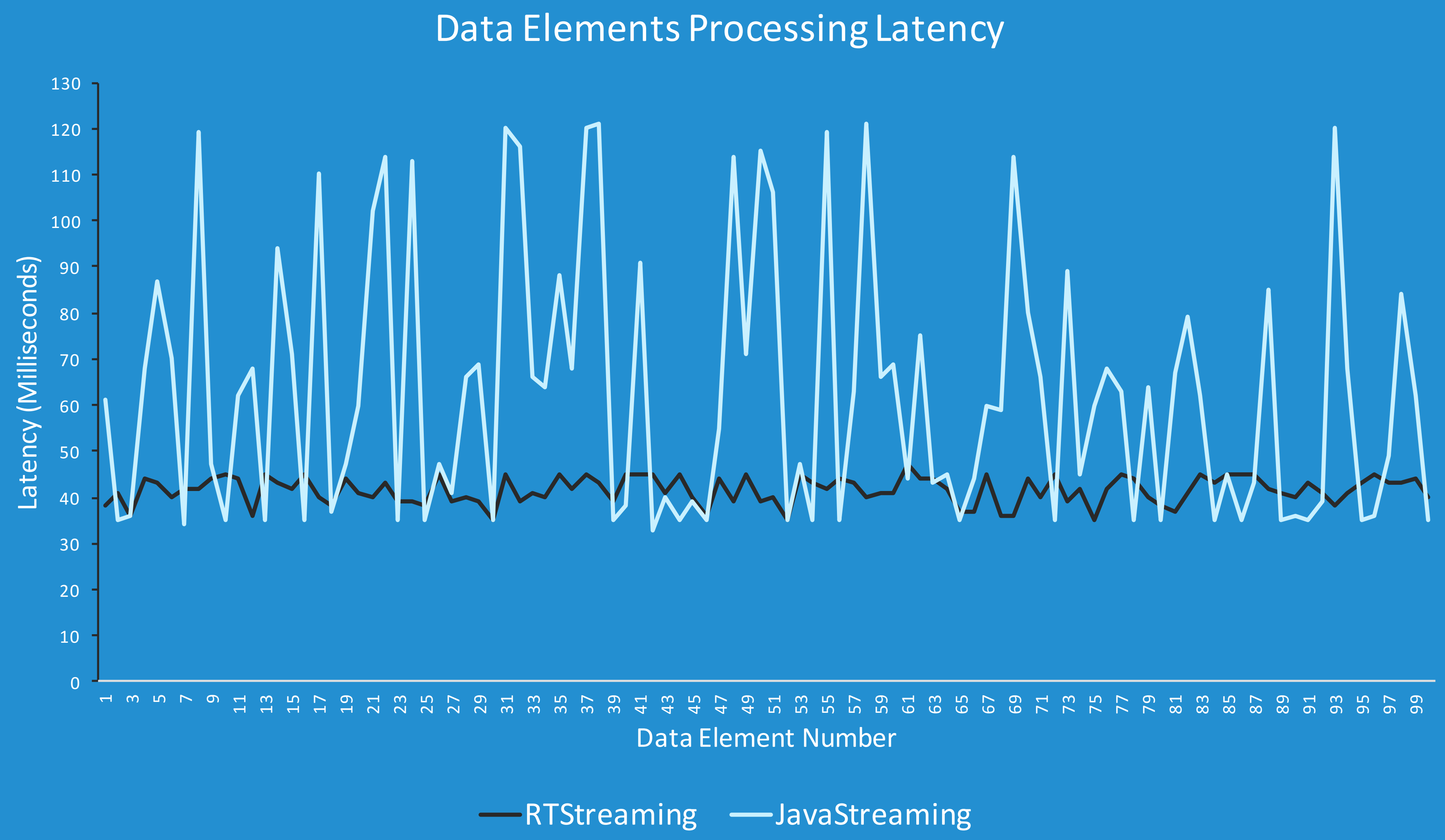
1. Latency of Stream Processing
2. Different Data Flow Rates
3. Burst Handling
4. Parallel Processing

Latency of Stream Processing

- A BachtedStream (Mid Priority) processing a data flow on Processor 2
  - Period of micro batching: 10 milliseconds
  - Buffer size: 1024 data elements
- Data Flow: MIT=200, MAT=400, WCET=34, Deadline=60, generate from Processor 1 (illustrating the computationally intensive nature of the processing required)
- Processor 2 also executes three periodic real-time threads at the same time:

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

Latency of Stream Processing



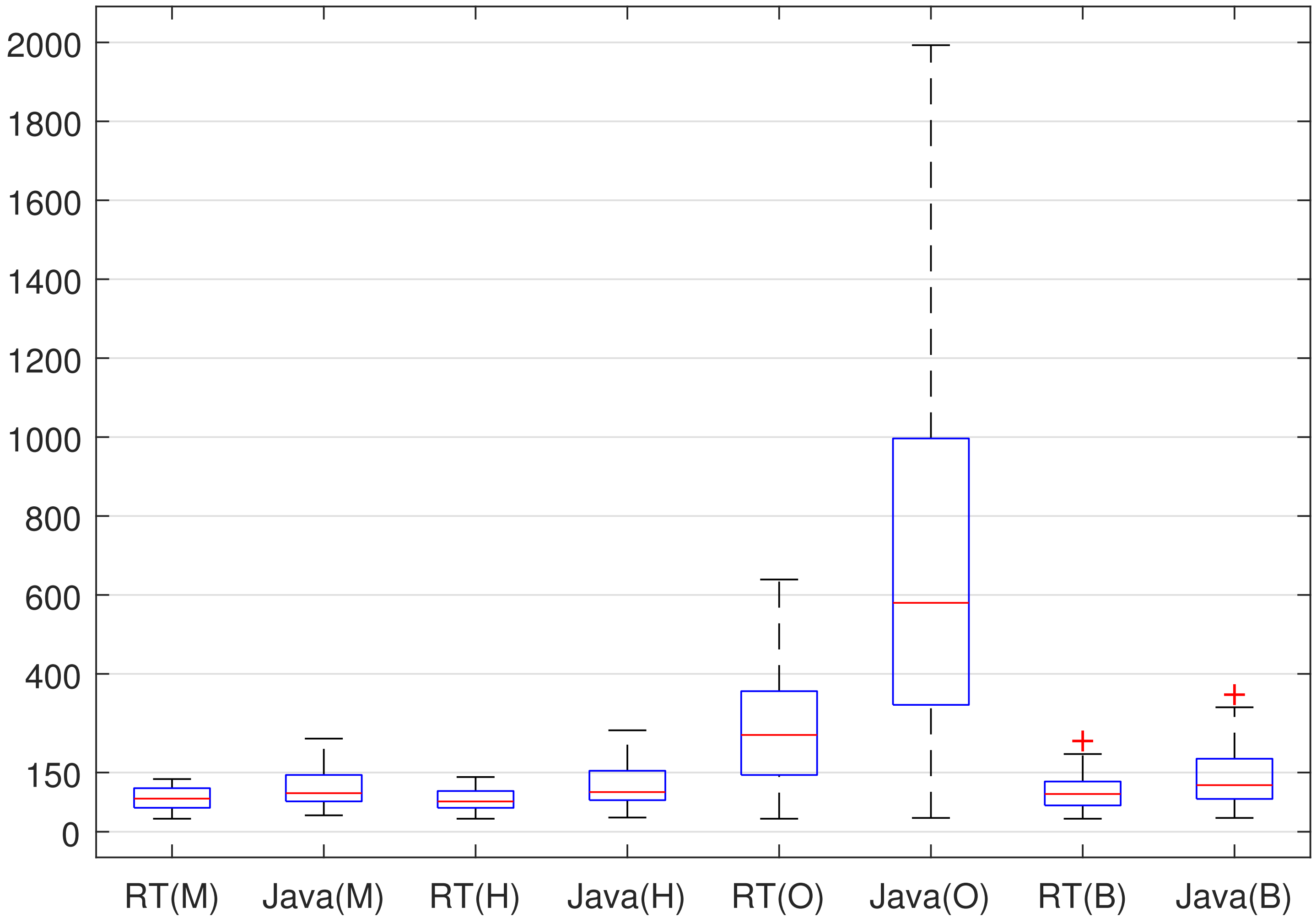
Difference Data Flow Rates

Considers the impact of different arrival rate: medium (**M**), high (**H**) and overload (**O**) workloads

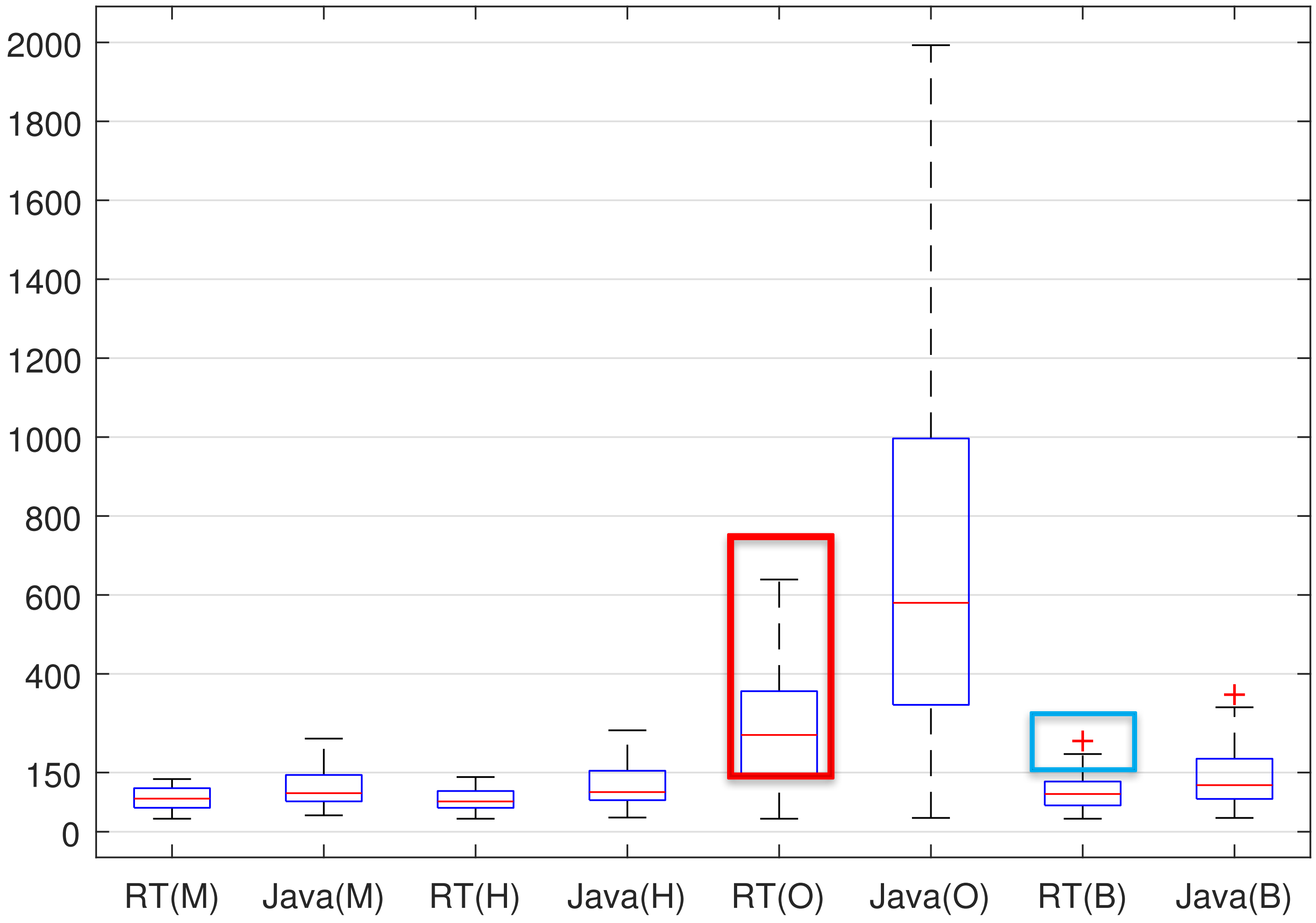
Period of micro batching: 90 milliseconds; Buffer size: 1024 data elements

Name	Processing Framework	MIT	MAT	Burst Size	WCET	Deadline
RT(M)	RTSJ RT ForkJoin Pool	100	200	0	28	150
Java(M)	Standard Java ForkJoin Pool	100	200	0	28	150
RT(H)	RTSJ RT ForkJoin Pool	50	100	0	28	150
Java(H)	Standard Java ForkJoin Pool	50	100	0	28	150
RT(O)	RTSJ RT ForkJoin Pool	20	40	0	28	150
Java(O)	Standard Java ForkJoin Pool	20	40	0	28	150
RT(B)	RTSJ RT ForkJoin Pool	200	400	4	28	150
Java(B)	Standard Java ForkJoin Pool	200	400	4	28	150

Difference Data Flow Rates



Difference Data Flow Rates

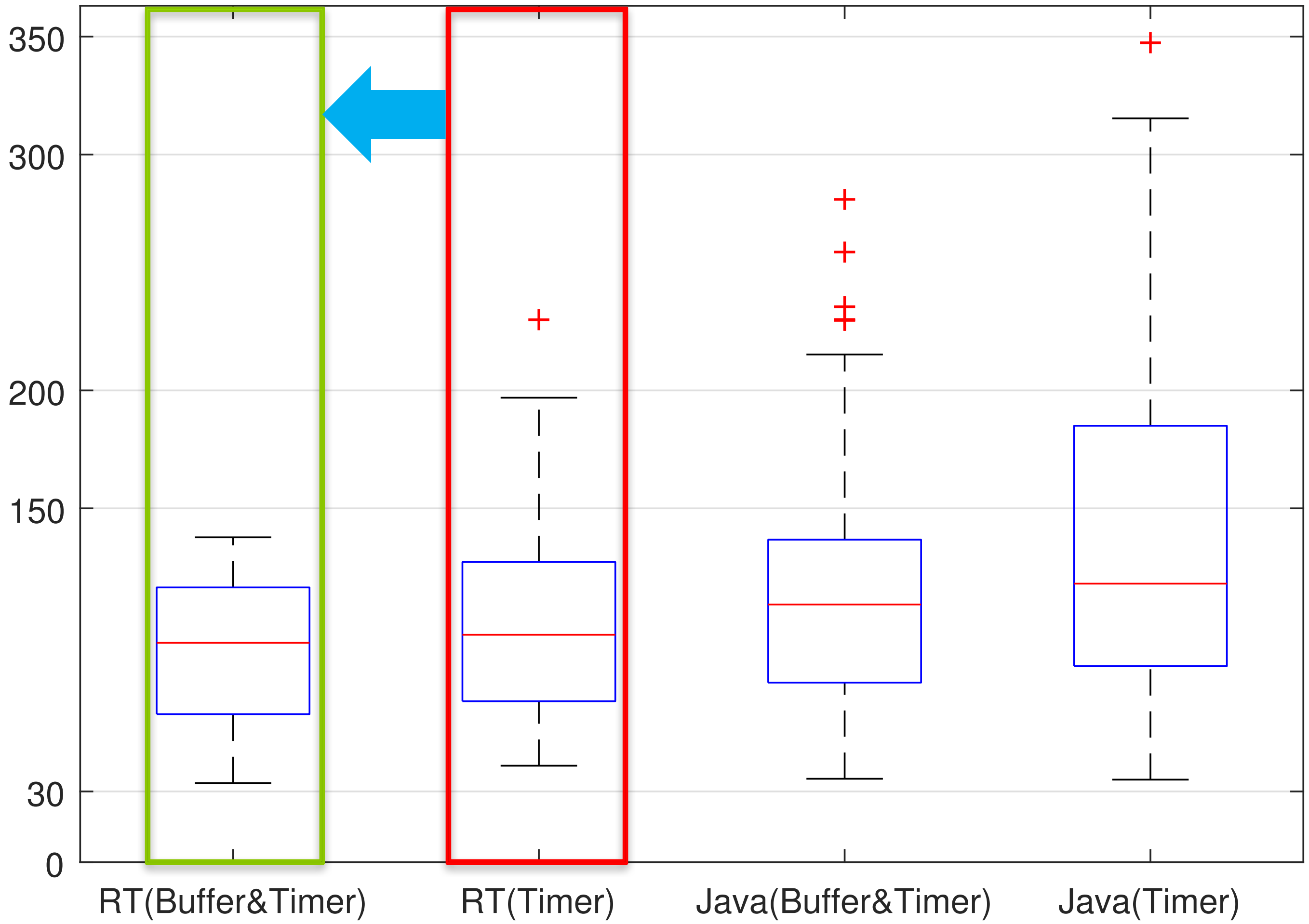


Burst Handling

Vary the buffer size to enable data to be processed immediately when bursts occur

The buffer size of the BatchedStream is configured to be 4 elements, i.e., the burst size, and redo the experiment

Burst Handling

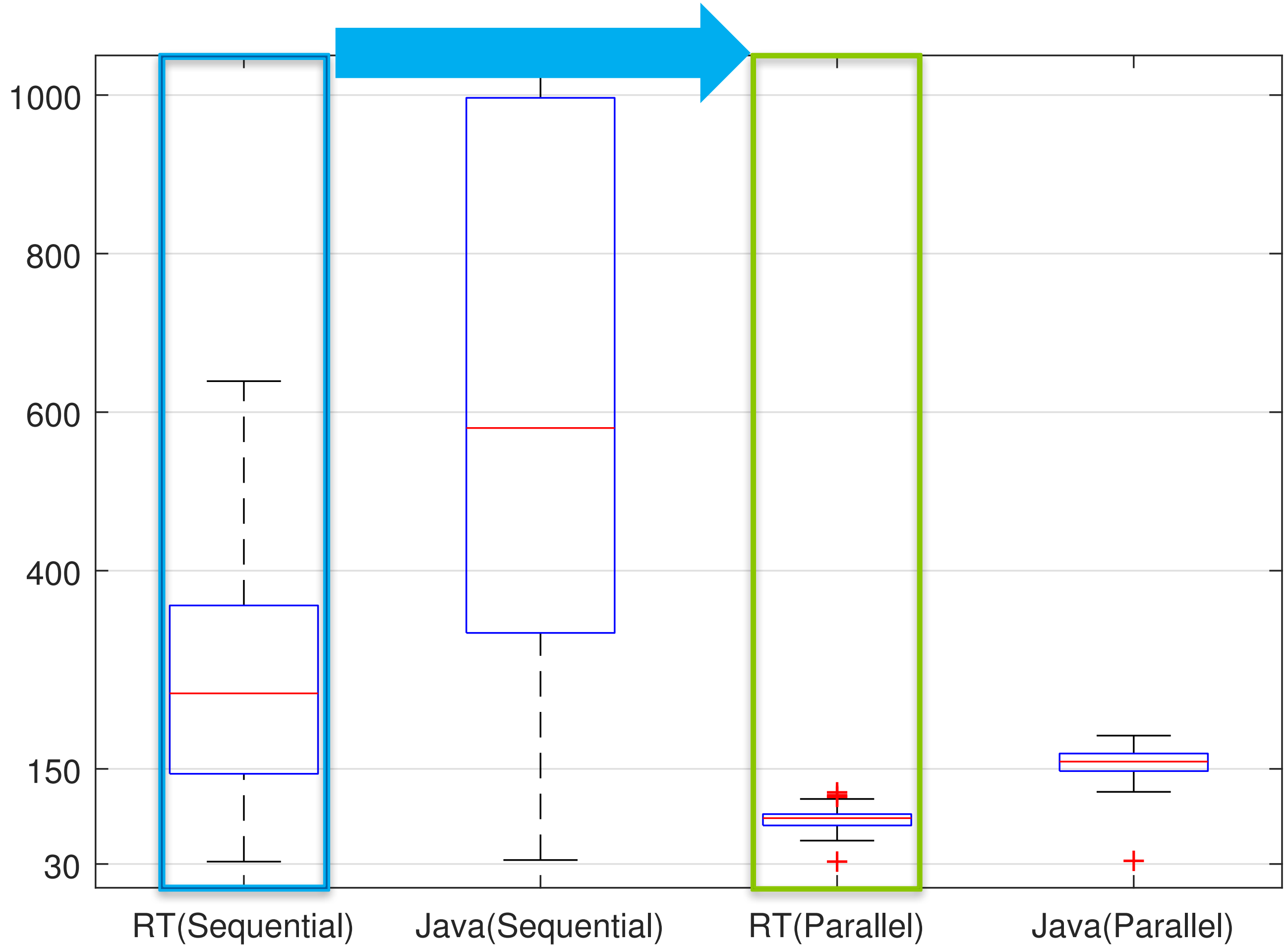


Parallel Processing

A stream whose MIT is 20 and MAT is 40 milliseconds cannot be guaranteed to meet the deadline because the system is overloaded

Allocating another processor (Processor 3) to the BatchedStream's underlying processing infrastructure

Parallel Processing



Conclusion & Future Work

- Proposed a framework for real-time streaming data processing based on micro-batching
- BatchedStreams enable a real-time stream processing job to be defined with concise code
- Evaluation shows that the BatchedStreams framework are predictable

- Response time analysis of a BatchedStream on a fixed priority global/partitioned scheduling system

Thank You

RTS York

Micro-Batching

Mapping data flow into batches, then processing using real-time threads

- Optimised push models of streaming data collect the individual data items into micro batches in order to improve the processing efficiency
- Make it possible for users to use Java 8 (Parallel) Stream operations, e.g. ***map***, ***filter***, ***reduce*** etc., when processing data flows in real-time

Deadline Miss When Bursts Occurs

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, d1, d2, d3, d4 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:

$$\begin{aligned} Latency_{d4} &= 100 + ResponseTime_{d4} \\ &= 100 + 28 + 28 + 28 + 28 \\ &= 212 \end{aligned}$$