Real-Time Stream Processing in Embedded Systems

Haitao Mei

Doctor of Philosophy

University of York
Computer Science
September 2017

Abstract

Modern real-time embedded systems often involve computational-intensive data processing algorithms to meet their application requirements. As a result, there has been an increase in the use of multiprocessor platforms. The stream processing programming model aims to facilitate the construction of concurrent data processing programs to exploit the parallelism available on these architectures. However, most current stream processing frameworks or languages are not designed for use in real-time systems, let alone systems that might also have hard real-time control algorithms. This thesis contends that a generic architecture of a real-time stream processing infrastructure can be created to support predictable processing of both batched and live streaming data sources, and integrated with hard real-time control algorithms.

The thesis first reviews relevant stream processing techniques, and identifies the open issues. Then a real-time stream processing task model, and an architecture for supporting that model is proposed. An approach to the integration of stream processing tasks into a real-time environment that also has hard real-time components is presented. Data is processed in parallel using execution-time servers allocated to each core. An algorithm is presented for selecting the parameters of the servers that maximises their capacities (within an overall deadline) and ensures that hard real-time components remain schedulable. Response-time analysis is derived to guarantee that the real-time requirements (deadlines for batched data processing, and latency for each data item for live data) for the stream processing activity are met. A framework, called SPRY, is implemented to support the proposed real-time stream processing architecture. The framework supports fully-partitioned applications that are scheduled using fixed priority-based scheduling techniques. A case study based on a modified Generic Avionics Platform is given to demonstrate the overall approach. Finally, the evaluation shows that the presented approach provides a better schedulability than alternative approaches.

Contents

List of Figures

List of Tables

Acknowledgement

Firstly, I would like to present my gratitude to my supervisors: Prof. Andy Wellings and Dr. Ian Gray, for their guidances and help during my whole study. I received a lot of help from them not only for the research, but also for my daily life so that the whole research procedure has been very rewarding.

I would also like to thank my family. Without the support from my parents, I would not be able to finish my Ph.D study. Their help allowed me to focus on the research, rather than also considering the living costs, etc.

I also acknowledge the Computer Science Department's scholarship, without which I would not have been able to fund the Ph.D program.

I would also like to thank the members in the Real-time Systems group, they are so patient to explain the problems I had, and encouraged me to carry on my research.

Last but not least, I want to thank my friends, for their help for both study, and especially life. They helped me to pass each obstacle in daily life, and ease the pain.

Declaration

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

Parts of this thesis have previously been submitted or published in the following papers:

- H. Mei, I. Gray, and A. Wellings. *Integrating Stream Processing into Embedded Real-Time Systems.* Submitted to the Journal of Real-time Systems. Springer, 2017.
- H. Mei, I. Gray, and A. Wellings. *Real-time Stream Processing in Java.* In Ada-Europe International Conference on Reliable Software Technologies (AdaEurope 2016), page 44-57. Springer, 2016.
- H. Mei, I. Gray, and A. Wellings. *A Java-based Real-Time Reactive Stream Framework.* In 19th International Symposium on Real-Time Distributed Computing (ISORC 2016), page 204-211. IEEE, 2016.
- H. Mei, I. Gray, and A. Wellings. *Selecting Execution-Time Server Parameters for Real-Time Stream Processing Systems.* In the 9th York Doctoral Symposium on Computer Science and Electronics (YDS 2016), page 16-25. University of York, 2016.
- H. 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 10-20. ACM, 2015.

Chapter 1

Introduction

Embedded systems are widely used in the world, for an estimation, 99% of the microprocessors are used for embedded systems [32]. A key feature of embedded systems that are used in critical domains, such as flight control, is that their time constraints must be guaranteed. These systems are also usually real-time systems. By definition, given by Burns and Wellings, a real-time system represents "any information processing activity or system which has to respond to externally generated input stimuli within a finite and specified period" [38]. For example, a flight control system of an aircraft must respond to an input stimuli within a deadline, because any deadline miss could results in a serious failure which may cause death or aircraft crash. In real-time systems, tasks are often classified as being hard or soft. Hard real-time tasks provide services within deadlines that must be met; whereas soft real-time tasks' deadlines although important can occasionally be missed without affecting the correct functioning of the system [38].

Due to increased computational demands, modern real-time systems now execute on multiprocessor platforms. Parallel programming of these platforms is required if applications are to exploit the extra available performance. The stream processing programming model [84] that consists of a collection of modules that compute in parallel and communicate via channels. 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). Figure 1.1 is a simple illustration of stream processing.

Stream processing enables users to facilitate the construction of concurrent programs to exploit the parallelism available on multiprocessor architectures.

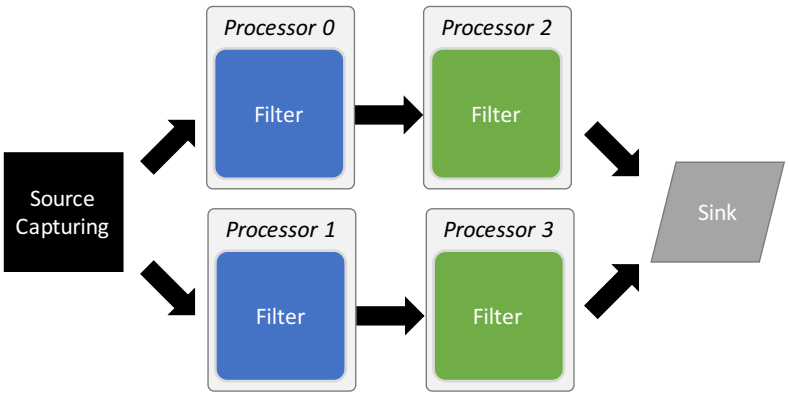


Figure 1.1: A stream processing example

Nowadays, stream processing has been widely adopted in different application domains, such as multimedia systems, signal processing systems, reactive systems, and Big Data systems [4, 5, 37, 44, 56, 64, 84, 87].

In addition to these stream processing frameworks, many programming languages also provide support for programming stream processing applications. StreamIt [24] focused on developing a new language that was specifically designed for processing data streams on platforms ranging from embedded systems to large scale and high performance systems. In addition, the most recent version of Java (Java 8) has introduced Streams and lambda expressions to support the stream processing paradigm, with functional-style code.

1.1 Motivation

Stream processing is suitable for several time-critical domains, such as real-time signal processing [62]. For example, an unmanned aerial vehicle (UAV) uses radar to identify potential obstacles and chose an avoidance path [77]. The continuous radar signals are processed by a on-board multiprocessor computer, and the processing is associated with real-time constraints to avoid potential hazards to the safety of individuals and communities. However, most stream processing architectures are not targeted towards real-time systems.

Often, many stream processing frameworks provide real-time performance by using high performance computation platforms, therefore increasing the speed of the stream processing so that the overall time which is required to handle the requests is reduced. Unfortunately, ***real-time guarantees*** are unlikely to be provided for every request using this approach even though

significant power and computation resources are employed. The reason is that these stream processing architectures pin their hopes on being sufficiently fast [85], rather than targeting towards real-time systems, to deliver "real-time" performance which is actually an illusion of real-time.

1.1.1 Real-time Stream Processing

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, stream processing either divides incoming data into partitions and fully processes each partition before the next one arrives [22] (for example, object tracking, or radar beamforming), or directly operates on each incoming data items (for example, wheel speed sensor signals in a car's ABS system). In the most general case, stream processing components may share the same computing platform, and interact with, other real-time components some of which might have hard real-time requirements.

In general, the data sources of stream processing systems can be classified into two types [71]: batched and live 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 live streaming data source represents data that arrives dynamically, its content and size will change with time.

The thesis intends to create an architecture using real-time principles, and implement a framework with real-time technologies, so that real-time guarantees can be provided to the stream processing.

1.1.2 Motivating Case Study

Consider an aircraft equipped with a spotlight synthetic aperture radar (SAR), and has a mission to generate images of a series of target areas using SAR, whilst its defence systems aim to guarantee its safety during the flight, as shown in Figure 1.2. According to the mission requirement, there is a real-time deadline for the imagery generation of each target area once the aircraft flies over it. At the same time, all the hard real-time tasks in the defence system must still meet their deadlines. All these tasks are executed by a multiprocessor mission control computer.

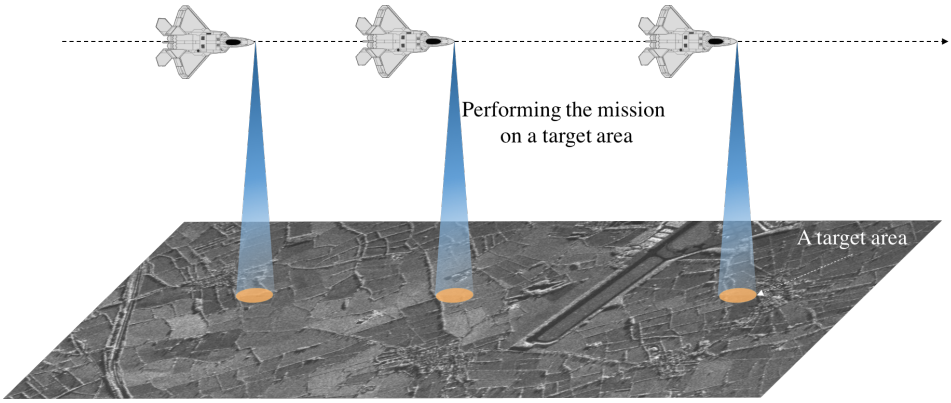


Figure 1.2: The mission of the generating images of target areas using SAR.

This is a scenario of the real-time stream processing, which inputs from a live streaming data source. However, it is difficult to employ existing stream processing techniques to generate images for each target area within the deadline, whilst guaranteeing that all the hard real-time tasks in the defence system can still meet their deadlines. The reason is that these stream processing techniques are designed for time-sharing systems and so do not provide predictability of execution. For example, there is no interface to configure the deadline for a job. More reasons are discussed in Chapter 2.

This case study will be addressed using our proposed real-time stream processing architecture in Section 5.6, along with its configuration. The derived response-time analysis guarantees that the time requirements of the mission are met.

1.2 Thesis Aims

The overall objective of the thesis is to develop a real-time stream processing architecture, and a prototype implementation, along with corresponding schedulability analysis techniques. The challenges and contributions of the thesis are described in this section.

1.2.1 Challenges in Real-Time Embedded Stream Processing

Handling stream processing in a system that also host many hard real-time activities to meet the given time constraints, whilst the hard real-time components remain schedulable is challenging. The reasons include:

- Typically, the stream processing paradigm uses multiple processors in the system, and its input data source can be either static, i.e., a batched data sources or data which arrives dynamically, i.e., a live streaming data source.

  - When processing a batched data source, how should the real-time stream processing of a batched data source be modelled into a real-time activity, so that the real-time characteristics, such as its deadline, can be captured, and multiple processors can be utilised?
  - For a live streaming data source, as each data item arrives dynamically, typically there is a deadline for completing each individual data item's processing. Therefore, how to create a parallel real-time model to queue (if necessary) and process these data items within a stream processing paradigm also needs to be addressed.

- Stream processing is often computationally intensive, it can be either hard real-time or soft real-time. For the later case, it might be difficult to predict the volume and the cost of processing the data. If the stream processing activity is executed at a very high priority, it is likely to meet its deadline, but it may also cause hard real-time tasks in the same system to miss their deadlines. If the stream processing activity is assigned with a too low priority, such as the background priority, it may miss its target deadline because of suffering interference from higher priority hard real-time activities during its execution.

  Therefore, how to execute a stream processing activity so that its deadline can be met, whilst the hard real-time tasks remain schedulable raises another challenge.

- Another requirement for real-time stream processing is to derive appropriate schedulability analysis. A real-time activity is schedulable if its response time is less or equals to its deadline. The response time is the time interval from when the input arrives to when the output is generated in a system. Once the stream processing activity is executed by multiple processors, the worst-case response time of the stream processing is required to be calculated in order to test its schedulability.

  More specifically,

– For a batched data source, the worst-case response time of the real-time activity which processes the whole batched data is required to be analysed.

– For a live streaming data source, as each data item is associated with a deadline, typically called latency, which represents the time from when the data arrived in the system to when the data processing has finished. Therefore, for every data item, the worst-case queueing time and response time of its processing is required to be analysed.

1.3 Thesis Hypothesis

This thesis addresses the hypothesis that:

> Programming languages or existing frameworks' support for stream processing is insufficient for addressing real-time requirements. However, a generic architecture of a real-time stream processing infrastructure can be created to support predictable and analysable processing of both batched and live streaming data sources, and can be used in high-integrity real-time embedded systems. Moreover, the architecture can be implemented as a framework using Java, with the Java Fork/Join framework and the Real-Time Specification for Java.

1.4 Success Criteria and Contributions

To assist with evaluating the work created as part of this thesis, the following success criteria were developed:

SC1 The definition of a generic architecture of a real-time stream processing infrastructure, which supports both batched data and live streaming data sources processing with real-time constraints, and is programming language independent.

SC2 A process for engineering real-time systems that have both hard real-time and hard or soft stream processing components, which focuses on how this architecture is to be mapped to the physical platform and how

the stream processing activity for both batched data and live streaming data sources is configured.

SC3 Response time analysis to determine the schedulability of stream processing for a batched data source, and latency for a live streaming data source.

SC4 A framework for integrating real-time stream processing activities with hard real-time components, and its implementation using the Real-Time Specification for Java (RTSJ).

SC5 An evaluation that demonstrates that the proposed model is as effective as a more typical real-time systems model that does not use the stream processing paradigm.

In addition to the above success criteria, a number of additional contributions were also made during the development of this work. These were:

- The first use of execution-time servers for performing stream processing in the context of hard real-time control system.
- An algorithm for selecting the number of servers and their parameters, which maximises the processor time that can be allocated to real-time stream processing within the deadline, yet guarantees the deadlines of the hard real-time components.
- A *bound* task is free of 'double-hit' (see Section 5.1.1) introduced by higher priority deferrable servers, therefore maximising the capacity that can be reclaimed by deferrable servers. This observation has been proved, and is a supplement to the original RTA (as described in Section 5.1).
- A comparison of the relative efficiency of the Java and StreamIt stream processing models.
- An evaluation of the suitability of the Java stream processing framework for use within a real-time environment.

1.5 Structure of the Thesis

The thesis is structured as follows:

- **Chapter 2** provides some necessary background material before reviewing stream processing frameworks, and programming languages.
- **Chapter 3** described the architecture of a proposed real-time stream processing infrastructure, which enables processing both batched and live streaming data sources in real-time. This architecture is assumed by the proposed approach in Section 4, and analysis in Chapter 5. This chapter provides the material needed to meet `SC1`.
- **Chapter 4** presents the overall approach to configure and schedule real-time stream processing tasks for both batched and live streaming data sources to meet the real-time constraints. In addition, the assumptions we make on the underlying real-time platform are also described in this chapter. This chapter provides the material needed to meet `SC2`.
- **Chapter 5** draw upon the research in schedulability analysis for the proposed real-time stream processing task model for both batched and live streaming data sources. An example of the scheduling, configuring and schedulability analysis for a real-time batched data processing task, and a case study of a real-time live streaming data processing application are also given in this chapter. This chapter provides the material needed to meet `SC3`.
- **Chapter 6** investigates two different stream processing models, and describes the implementation (SPRY) of the architecture of the proposed real-time stream processing infrastructure using RTSJ, along with the implementation of the case study using SPRY. This chapter provides the material needed to meet `SC4`.
- **Chapter 7** evaluates the presented real-time stream processing approach to the traditional embedded approach, when processing batched data and live streaming data sources. This chapter provides the material needed to meet `SC5`.
- **Chapter 8** draws the conclusions and summarises future work.

Chapter 2

Literature Review

This chapter first introduces some necessary background material on computer architectures and real-time in order to set the landscape within which this research has been conducted. A brief history of stream processing is then presented. Several typical stream processing techniques, including frameworks, programming languages, are then reviewed in detail.

2.1 Parallel Computer Architectures

In recent years, processor manufacturers have turned to parallelism to speed up computation, rather than increasing the clock speed [54], and thus computers have evolved towards multiprocessor architectures. According to their memory access model, parallel computers can be classified into three typical architectures:

- **Uniform Memory Access (UMA)**
  All the processors use the same memory, and they have equal access and access times to memory.

- **Non-Uniform Memory Access (NUMA)**
  Processors are divided into groups, processors in each group access memory using UMA. Not all processors have equal access time to all memories. Cache Coherent NUMA (CCNUMA) is one type of NUMA architecture that provides cache coherency. Most modern processors running on servers use this architecture.

- **Distributed Memory**
  Processors have their own local memory in distributed memory systems,

processors can not access data in other processor's local memory. Processors communicate over networks.

This thesis is concerned primarily with UMA architectures. However, our underlying approach is also appropriate for NUMA architectures.

2.2 Real-Time Systems Model

The literature in real-time systems is broad. Here we present a top level view in order to place our work in context. More details will be given on particular techniques when they are used later in the thesis.

We introduce: the scheduling approaches used in the real-time literature, the problem of task allocation in a multiprocessor systems, and the role of execution-time servers.

2.2.1 Scheduling

The thesis focusses on the task-based scheduling of real-time systems, which have been summarised by Burns and Wellings [38]:

- Fixed-Priority Scheduling (FPS)
  In a fixed priority scheduled system, each task has a fixed priority, which does not change with time. The tasks' running order is determined by their priorities.

- Earliest Deadline First (EDF)
  The running order of the runnable tasks is determined according to their absolute deadline, the task that has the nearest deadline executes prior to the rest of the tasks.

- Least Laxity (LL)
  The runnable tasks are executed according to their slack, which is the deadline minus the required computation time. The next task to execute is the task with the shortest slack.

- Value-Based Scheduling (VBS)
  This algorithm considers system where overloaded are possible. In this type of systems, each task is allocated with a value, and an online value-based scheduling approach is employed to determine which one is the next task to run. The one with the highest value is the one chosen.

In this thesis, we are concerned with priority-based scheduling as this is the most widely used approach [39] and the one supported by all real-time operating systems [38].

2.2.1.1 Preemption

In a *preemptive* system, when a higher priority task is released and a lower priority task is executing, the execution is immediately switched to the higher priority task. In contrast, in a *non-preemptive* system, the higher priority task has to wait for the lower priority task until it finishes its execution.

The preemptive scheme is adopted in this thesis as it makes higher priority tasks more responsive.

2.2.2 Task Allocation

Given a set of application tasks, a multiprocessor execution platform and preemptive priority-based scheduling, there are essentially three approaches to scheduling the tasks on the platform [38].

- Global Scheduling
  A globally scheduled system is a system where all the tasks can execute on any available processor. A task that is executing on one processor can switch to another processor, i.e., a task can start its execution on one processor and then migrate to another process to continue/finish its execution.

- Fully-Partitioned Scheduling
  A task in a fully partitioned system is not allowed to migrate to another processor once it has been allocated to one processor.

- Semi-Partitioned Scheduling
  Semi-partitioned scheduling is between global scheduling and fully partitioned scheduling. In a semi-partitioned system, it limits which tasks may migrate, and where they may migrate to.

Normally, a single processor only resides in a single partition. This thesis addresses only Fully-Partitioned Systems as the schedulability analysis for such systems is a major domain in the real-time literature [38]. For example, in Chapter 5, we build our analysis on [47], which is based on fully-partitioned systems rather than globally scheduled systems.

2.2.3 Execution-Time Servers

In the real-time community, execution-time (or aperiodic) servers [38] are used to give tasks that might demand unbounded CPU time a good response time, while limiting their impact on other tasks so that, e.g., a hard real-time task will not miss its deadline. An execution-time server has a capacity, and a replenishment policy. When a client task execute under a execution-time server, it consumes the capacity. When the capacity is empty, the client task is not allowed to run and has to wait for the next replenishment.

For a periodic server [38], it has a capacity, which is periodically replenished. The capacity is consumed even if there is no client task. For example, a periodic server has a period of 10, capacity of 5, released at time 0. It has only 3 time units capacity left at time 12, because 2 time units capacity has idled away.

The POSIX standard supports Sporadic Servers [65,83]. A sporadic server has a replenishment period, a budget (or capacity), and two priorities: high priority and low priority. When handling aperiodic events, the server executes at the high priority when it has budget, otherwise runs at the low priority. When the server runs at the high priority, the amount of execution time that has been consumed is subtracted from its budget. The budget remains indefinitely if not consumed. If consumed, e.g., at time t, the budget will be replenished at $t+$ its replenishment period.

A Deferrable Server [65, 83] allows 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. When registered with a deferrable server, an aperiodic thread executes at the server's priority level until either the capacity is exhausted or it finishes its execution. In the former case, the aperiodic thread is suspended or transferred to a background priority. The capacity of a deferrable server is replenished every period. Different from the periodic server, the capacity of a deferrable server is retained as long as possible, rather than idled away.

The response time of a task executing under an execution-time server can be analysed using the techniques provided by Davis and Burns [47]. The impact from a higher priority deferrable server to a lower priority task can be analysed by [34].

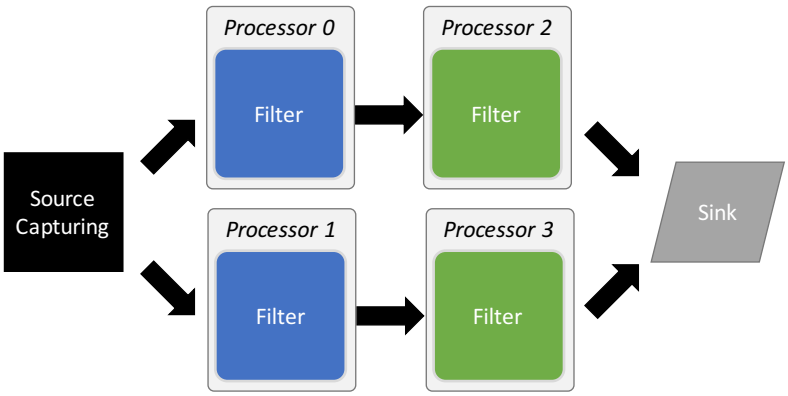


Figure 2.1: A stream processing example

In this thesis, we will use Deferrable Servers as they have superior schedulability compared to Periodic Servers. Furthermore, they are easier to implement than sporadic servers. However, our framework is independent of the server technology used.

2.3 Stream Processing and Related Techniques

Stream processing has been around for decades, and is widely used in data flow systems, signal processing systems, reactive systems, etc. [1, 4, 12, 84, 87].

2.3.1 Stream Processing

A stream processing system uses a collection of modules to compute the input in parallel, and communicates via channels [84]. 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). For example, a stream processing system that has 4 filters computing in parallel can be illustrated in Figure 2.1 (a replication of Figure 1.1 for convenience of presentation). The filters in processor 0 and 1 have the same functionality, and their outputs flow into filters in processor 2 and 3 separately.

2.3.2 Stream Processing Data Source Classification

As has been introduced in Section 1.1.1 the data sources of stream processing systems can be classified into two types [71]: *batched* and *live streaming*.

Note however, the live streaming data may has other names, for example, in Big Data community, the live streaming data is also called real-time data. However, the real-time in this thesis represents "any information processing activity or system which has to respond to externally generated input stimuli within a finite and specified period" [38]. For the clarification, the term *live streaming* used to to represent this type of data sources.

2.3.3 A Brief History of Stream Processing Techniques

The earliest recorded work of stream processing is data flow programming in 1960s [84], even though it was not termed as *data flow* at that time. Then, several research projects targeting stream processing were performed. For example, in 1970s, Kahn Process Networks were proposed as an asynchronous programming model for data flow, i.e., filter processing without synchronisation with respect to other filters. Synchronous data flow [64] was proposed in 1980s for stream processing, where synchronisation was supported when collaboration between filters is required. In 1990s, LUSTRE [57] was proposed as a programming language to support synchronous data flow. More related stream processing work in the past decades is reviewed in [84].

In 2002, StreamIt [87] was created as a new language for stream processing on platforms ranging from embedded systems to large scale and high performance system. In 2003, Brook [36] was proposed as a stream processing specification, and its main follow-up work is Brook for GPUs [37], which was developed for the stream processing on GPUs. In the same year, STREAM [28], Aurora [26], and Medusa [43] were created to support stream processing mainly in data management systems.

In order to address the requirement of large data sets processing in a distributed computer cluster, MapReduce [50] was announced by Google in 2004. MapReduce partitions the input, distributes the partitions over the computer clusters, performs operations, and fold the results. In addition, Borealis [5] integrated Aurora [26] and Medusa [43] to provide a distributed stream processing system for data management.

In 2007, Microsoft announced Dryad [60], which supports distributed large data sets processing with directed acyclic graphs (DAG) so that more complicated processing logic can be represented. Inspired by StreamIt, StreamFlex [82] was also proposed in 2007, which intends to deliver low-latency stream processing. StreamFlex uses abstractions supported by the Real-Time

Specification for Java (RTSJ) [91], e.g., a memory area that avoids any interference from garbage collectors, to minimise latency.

In 2008, DryadLINQ [93] was proposed to provide a high level language abstraction, which enables the succinct description of a distributed stream processing job. In 2010, FlumeJava [41] was created to provide an easy, efficient data parallel pipeline, which was used by Google internally.

In 2010, S4 [76], and Storm [4] were proposed for distributed live streaming data processing. Spark [1] was created to support in-memory stream processing of large data sets. In addition, MapReduce online [45], Twister [52], HaLoop [35] also tried to refine MapReduce so that it can be used iteratively, in order to provide interactive data processing. In addition, considering the requirement of large-scale graph processing, such as, social networks that has billions of vertices, trillions of edges, Pregel [69] was proposed by Google.

Spark Streaming [19] was developed as a library on Spark in 2013, in order to support live streaming data sources. In the same year, MillWheel [27] was also created at Google, to support live streaming data processing as MapReduce is not fit for live streaming data. Additionally, Flink [40], Heron [61], and Samza [2] were developed in 2014 to 2015 as distributed stream processing frameworks, in order to deliver more scalability and efficiency compared to Storm. In addition, inspired by FlumeJava [41] and PLINQ [15] that provides a parallel implementation of data set operations, Java SE 8 [12] was released in 2014, with Java 8 Streams and lambda expressions, which enables efficient parallel stream processing with functional-style code.

These stream processing techniques introduced above are designed for best effort, time-sharing systems. For real-time systems, some distributed real-time frameworks have emerged in recent years, for example, [33] is a real-time version of Storm [4], and the JUNIPER [31] project. These frameworks provide supports to process large data sets in a distributed computer cluster, with a predictable processing time. However, it is difficult to implement a hard real-time system in a distributed system, because of the unpredictability of the network, they are targeted soft real-time.

2.3.4 Stream Processing Classification

In the most recent stream processing frameworks [1,4,12], stream processing is typically represented by a pipeline with zero or more synchronous stages. Each stage contains one or more filters, which are allocated to different processors

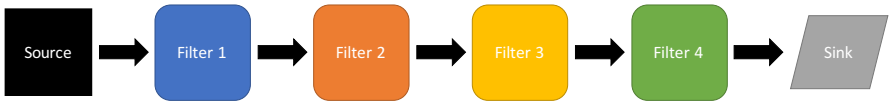
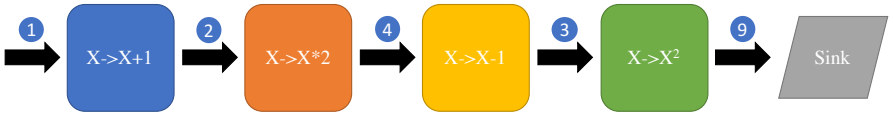


Figure 2.2: A pipeline that has 4 filters



(a) Eager evaluation of a pipeline.

(b) A pipeline that is lazily evaluated.

Figure 2.3: The evaluation of a pipeline.

or computer nodes to execute, in order to exploit the possible parallelism. The whole processing procedure forms a DAG.

According to the executing behaviour and allocation of the filters, stream processing can be classified into different types. Considering a typical pipeline, which contains 4 filters can be illustrated by Figure 2.2, the stream processing can be classified as:

- **Lazy or Eager**

  According to the behaviours of the pipeline's executing, their evaluation can be classified as *eagerly*, and *lazily*.

  - *Eager Evaluation*

    The input is processed eagerly in this model, i.e., any filter in this model triggers the processing immediately. When an input arrives at a pipeline that is eagerly evaluated, the input is immediately processed by the first filter, and generates an intermediate result, which will be an input of the down stream (or next stage) filter. The intermediate results are typically transferred via channels, shared memory buffers, or networks in a distributed computer cluster.

    This model can be illustrated by Figure 2.3a, where a pipeline that contains four filters is used to process numbers. For example, when the number `1` enters the eager evaluated pipeline, it is immediately processed by the filter that increases the input by 1, and generates

`2` as the intermediate result. The intermediate result `2` is then be sent to the next filter that multiplies the input by 2, and we get the number `4`. So on and so forth. In this example, three intermediate results are generated, stored and transferred by channels.

– *Lazy Evaluation*

In a lazily evaluated model, the processing or the input is delayed as long as possible, and only processed when necessary, such as, when the final results are requested or the intermediate results are required to be transferred to another machine in a distributed environment.

Considering the same pipeline again, Figure 2.3b illustrates how it is lazily evaluated. When the input `1` arrives, it is not processed until the sink requests the final result. The filters are combined together to be a super filter, so that the input is processed by this super filter, and no intermediate result is generated or transfered.

Lazy operations not only can avoid unnecessary evaluation, but also can provide potential optimisation opportunities. For example, the following Java 8 pseudo code gives the MD5 hash code of the first number in a given array. Lazy operations enables a return generated upon the first input, instead of calculating all numbers' hash code then finding the first one.

```
Arrays.stream(new int[] { 1, 2, 3,..., 1000000 })
    .map(n -> MD5(n))
    .findFirst()
    .ifPresent(System.out::println);
```

However, a lazy pipeline is identical to a eager pipeline when the intermediate results of each filter are required to be transferred to another machine. Typically, when evaluating a lazy pipeline, the application travels through the pipeline until a filter that triggers the processing is met. Then the application travels back to the first filter and perform operations in down-stream filters one by one. Compared to a eager pipeline, this introduces overheads when there is no optimisation opportunities.

- **Control Parallel or Data Parallel**

Considering a stream processing system with the pipeline shown in Fig-

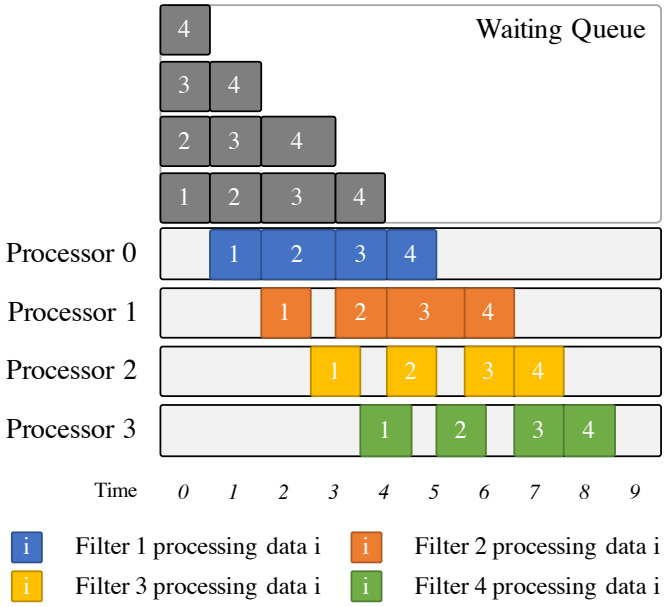


Figure 2.4: A control-parallel pipeline.

ure 2.2 again, and a 4 processors SMP CPU. The inputs are 4 data items, which arrive at time 0, and are stored in a waiting queue.

According to the processor allocation, a pipeline can be either mapped across different processors, i.e., control parallel, or duplicated on each processor, i.e., data parallel.

– *The Control-Parallel Pipeline*

The control-parallel pipeline behaves similar to an instruction pipeline within a modern CPU. In this scheme, one or more filters are mapped to a processor, but a same filter does not reside on more than one processor. In this example, each filter is mapped to different processors as shown in Figure 2.4. The processor 0 takes an input from the waiting queue, processes it, passes the intermediate result to the down-stream filter that is running on processor 1, then takes another input from the waiting queue. Note that, the result merging is not shown.

Multiple processors can be utilised to exploit the parallelism, however, the control-parallel pipeline has the following disadvantages:

∗ When the pipeline contains too few stages compared to the number of available processors, some of the processors cannot

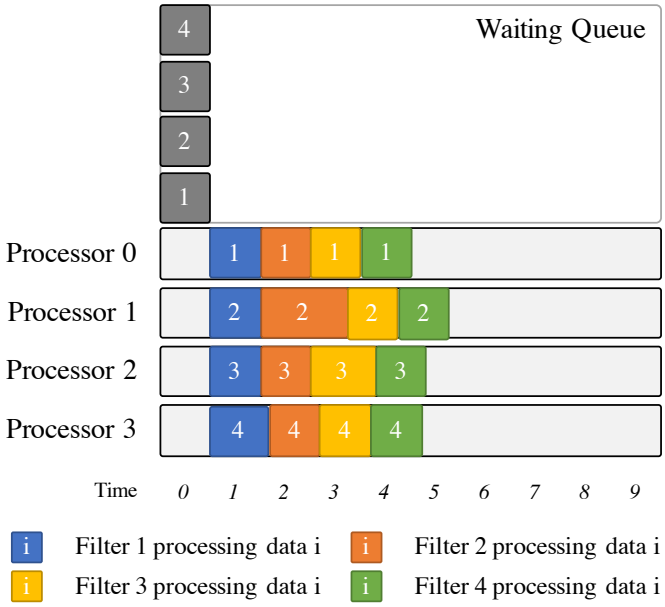


Figure 2.5: A data-parallel pipeline.

be utilised, therefore, making the system inefficient.

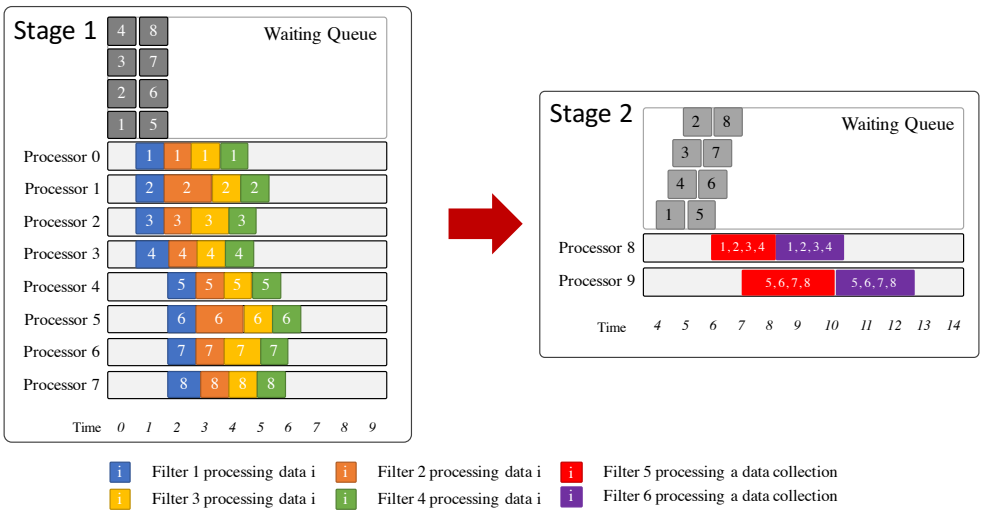
* If the pipeline is unbalanced (i.e., computation time, of each filter is not identical), or up-stream filters' processing is delayed, for example, due to receiving interference from other activities, the system efficiency is reduced. The reason is that, when any up-stream filter requires more time to finishes its processing, the down-stream filter has to wait for it idly.
* Moreover, inter-processor communication introduces extra overheads.

– *The Data-Parallel Pipeline*

The data-parallel pipeline duplicates the entire pipeline to different processors, as shown in Figure 2.5. The data is allocated to different processors. In this example, each processor works independently, and there is no waiting gap. The data-parallel pipeline is suitable for lazy evaluation, as all the filters are allocated into the same processor. Again, the result merging is not shown in the figure.

However, the drawback of a data-parallel pipeline is making the the pipeline span different computation resources impossible. For example, one of the filter within the pipeline requires to access a GPU or FPGA.

(a) The pipeline used in the hybrid pipeline example.



(b) The processing of the example hybrid pipeline.

Figure 2.6: The hybrid pipeline.

– *The Hybrid Pipeline*

With a hybrid pipeline, the pipeline can span different nodes in a distributed system, while within each node, the pipeline is duplicated according to its data source partitions.

For example, a hybrid pipeline can be illustrated by Figure 2.6. Where the logic of the pipeline is shown in Figure 2.6a, the processing of this pipeline is illustrated by Figure 2.6b. This example inputs data collections, which are firstly processed by the first 4 filters using a data-parallel model. For example, the first input collection $1, 2, 3, 4$ are partitioned, and processed by processor 0, 1, 2, and 3. This can be illustrated as stage 1 in Figure 2.6b. The intermediate results are merged, and sent to down-stream filters. For example, the merged intermediate results are then processed by filter 5 and 6 using a data-parallel model, with processor 8 and 9. This can be illustrated as stage 2 in Figure 2.6b. The sub-pipelines in Stage 1 and 2 are evaluated using a control-parallel model.

The following subsections consider in more depth several of the stream

processing techniques that were discussed previously. A summary of those techniques and their characteristics is given in Table 2.1. The remainder of this subsection justifies our choice of the set of representative techniques to consider.

Table 2.1: Stream Processing Techniques Classification

Technique	Type	Behaviour	Pipeline Type	RT
StreamIt	Language	Eager	Control parallel	No
Spark	Framework	Lazy	Hybrid	No
Java 8 Streams	Framework	Lazy	Data parallel	No
Storm	Framework	Eager	Hybrid	No
JUNIPER	Infrastructure	Hybrid	Hybrid	Soft
RT-Storm	Framework	Eager	Hybrid	Soft

StreamIt is chosen as it targets embedded systems and provides flexible support for the development of stream processing applications [87]. It uses an eager pipeline as any filter triggers the processing, and a control-parallel model. It is also a widely referenced stream processing language.

In order to address the requirement of large data sets processing challenges introduced by the rapid growth in data production, MapReduce [50], Hadoop [3], and Dryad [60] were created. Recently, Spark [1] has successfully succeeded these frameworks. Spark uses lazy evaluations (as only certain types of filters trigger the processing), and a hybrid pipeline. We, therefore, review Spark as an example of a batched stream framework for large scale data processing applications.

Java is a popular programming languages, and used widely in modern stream processing domain, for example, Hadoop, Spark, Flink [40], Storm, etc., are based on Java platforms. In the most recent version (Java 8) a stream processing library has been included to support efficient batched data stream processing in parallel. Java 8 Streams are lazily evaluated, with a data-parallel pipeline. Java is reviewed as it is used in this thesis to implement our proposed real-time stream processing architecture. It is also an example of a framework that supports batched stream processing.

Storm [4] was created to target live streaming data processing, and has been widely adopted in commercial areas [90]. Storm is considered because the hybrid pipeline model, and uses the eager evaluation model (because any filters

in Storm triggers data processing). Storm is, therefore, reviewed as an example of a commercially successful live streaming data processing framework.

In order to address the real-time requirement of stream processing, JUNIPER [31] and a real-time version of Storm [33] were created. JUNIPER mainly targets batched data processing in real-time, while real-time Storm focuses predictable live streaming data processing. We consider these two as they are two state-of-the-art frameworks that focus on real-time streaming issues (albeit in a distributed environment).

2.3.5 StreamIt

StreamIt [87] is mainly based on Java, but provides its own compiler (it compiles StreamIt source code to Java code, then translates the Java code to C++ code using a third party library, and finally generates a binary executable file using G++) and tool set. StreamIt defines several concepts. The basic concept is the *filter*, which is a computation unit of StreamIt, and contains user defined data processing code. StreamIt also defines global variables, that can be accessed by any of the filters.

A simple stream processing program can be defined using the following StreamIt code:

```
void->void pipeline Example() {
   add IntegerSource();
   add IntegerPrinter();
}
void->int filter IntegerSource {
   int i;
   init { i = 0; }
   work push 1 { push(i++); }
}
int->void filter IntegerPrinter {
   work pop 1 { print(pop()+" "); }
}
```

This program defines a pipeline, which contains two filters:

- the `IntegerSource` filter, which takes nothing as the input, and generates an incremental integer each time (via the "push" statement). The push statement writes the results into the communication channel.
- the `IntegerPrinter` filter, which inputs one integer at a time (via the

"pop" statement), and prints it out. The pop statement reads numbers of intemediate results from the communication channel.

The program is controlled by a loop, with a user configured total iteration times. Within each iteration, an input is read into the system, and then sent to the down-stream filters for processing. For example, after compilation, the user can run this program with the following command:

`./Example -i 5`

The program iterates 5 times, and generates the output: `0 1 2 3 4`.

2.3.5.1 Connecting the Filters

The notion of *stream* in StreamIt is defined as a component, which has one or more connected filters and with data flows into and out. Three structures of stream processing logic are defined by StreamIt, by connecting filter in different ways: the *Pipeline*, the *SplitJoin*, and the *FeedbackLoop*.

Pipeline

The Pipeline is used to construct a sequential stream, which has a series of filters connected linearly using the *add* command (see line 2, and 3 in the above code). An example of the pipeline has been introduced above.

SplitJoin

The SplitJoin splits the input data stream to different branches, which can process data items in parallel, and merge the intermediate results into a common joiner. For example, the following code distributes the input to two filters in a round-robin fashion.

```
void->void pipeline SJExample() {
   add IntegerSource();
   add SJ();
   add Printer();
}
void->int filter IntegerSource { int i;
   init { i = 0; }
   work push 1 { push(i++); }
}
int->int splitjoin SJ () {
   split roundrobin;
```

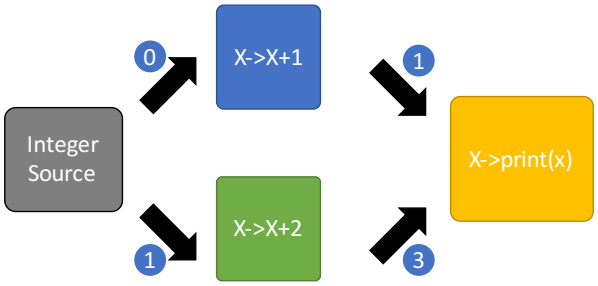


Figure 2.7: StreamIt SplitJoin example, the cycle with numbers represent the input data or an intermediate result.

```
   add Adder(1);
   add Adder(2);
   join roundrobin;
}
int->int filter Adder(int increment) {
   work pop 1 push 1 { push(pop()+increment); }
}
int->void filter Printer {
   work pop 1 { print(pop()+" "); }
}
```

The program can be illustrated by Figure 2.7. Run the program with 2 iterations, the source generates the input (i.e., `0`), and the second input (i.e., `1`). The first input is sent to the filter: $x->x+1$, while the second input is sent to the filter: $x->x+2$. Finally, `1 3` is printed.

StreamIt supports three types of *SplitJoin*:

1. Duplicate
   Each input is duplicated, and sent to every added filter.

2. RoundRobin
   The inputs are distributed to added filters in a round-robin fashion. The first data is sent to the firstly added filter, the next data is sent to the secondly added filter, and so on.

3. Null
   It considers the parallel paradigm where there is no input is required by the added filters.

FeedbackLoop

The FeedbackLoop is used to create cycles in the stream processing graph. For example, a stream that calculates the Fibonacci numbers can be described by the following example, which is taken from StreamIt benchmarks [23]. The feedbackloop takes two numbers each turn, generates the output by adding them together. The output of the feedbackloop is copied to 2 pieces: one goes to the printer, one is go back to the feedbackloop as an input in the next iteration.

```
void->void pipeline Fib {
   add feedbackloop {
      join roundrobin(0, 1);
      body PeekAdd();
      split duplicate;
      enqueue 0;
      enqueue 1;
   };
   add IntPrinter();
}

int->int filter PeekAdd {
      work push 1 pop 1 peek 2 {
      push(peek(0) + peek(1)); pop();
   }
}

int->void filter IntPrinter {
   work pop 1 {
         println(pop());
   }
}
```

2.3.5.2 Parallel/Distributed Execution

StreamIt is supported on Linux. The StreamIt compiler compiles the code into Java source code, and then generates C code. Finally, the C code is compiled into binaries using GNU G++.

Processor Allocation

When compiling a StreamIt program, the number of processors that are allocated to the program is required to be given, otherwise, by default, the code is compiled to be a sequential program. In addition, there is also a configuration file, which specifies the host, i.e., which node in a computer cluster, where each processor is located in.

Each filter in the code is compiled to be a C function, and the data flow between two filters is implemented using shared memory buffers in a multiprocessor CPU, or TCP/IP sockets in a distributed system.

When there are more filters than processors, the StreamIt compiler combines several filters to be a super filter. The compiler generates several super filters as many as the available processors. These super filters are allocated to POSIX threads for execution.

However, when there are more processors than filters, some of the unallocated processors are idle as StreamIt does not duplicates filters.

Performance Optimisation

The StreamIt compiler estimates the computation load of each filter using simulation, then allocates different numbers of filters into different super filters so that the super filters have an identical amount of computation load.

In addition, StreamIt also employs function inlining, array scalarization, and loop unrolling to optimise the performance [55].

2.3.5.3 Discussion

StreamIt is a stream processing programming language, which enables data flow processing program can be written using concise code.

However, the main drawback of StreamIt is that it is a special purpose language, so that it is hard to integrate with general purpose languages.

In addition, the pipeline in StreamIt is not replicated by default. Therefore, when there are more processors, it relies on users to construct a parallel structure, which can utilise all the processors. For example, it assumes users will use the SplitJoin to duplicate the filters. Otherwise, some of the processors are idle.

As StreamIt is not designed for real-time systems, it is impossible to be directly integrated with real-time systems. This is because the threads may

demand unlimited CPU time, therefore causing the hard real-time tasks in the same system to miss their deadlines.

2.3.6 Spark

Spark [1] was created at UC-Berkeley and implemented in the Scala programming language, which runs on the JVM and targets batched data processing. Spark provides a functional programming interface, which enables programming with concise code. Spark Streaming [19] is an extension of Spark, and allows the live streaming data to be processed using the existing Spark runtime.

The data structure used by Spark is the Resilient Distributed Dataset (RDD), which represents a read-only collection of data located in a set of machines. Data that is corresponding to the RDDs can be parallel processed by invoking multiple parallel operations, for example, *map*, *reduce* etc. An example is described by the following Scala code:

```
1 val InputFilesRDD = spark.textFile("hdfs://...")
2 val ResultRDD = InputFilesRDD.flatMap(line => line.split(" "))
3                              .map(word => (word, 1))
4                              .reduceByKey(_ + _)
5 //save the result...
```

In this example, a RDD named *InputFilesRDD* is created from the text files that are stored in HDFS (see line 1). The *InputFilesRDD* references to all the texts in these text files. Spark splits each line into words using the *flatMap* operation (see line 2), and these words are represented by a new RDD. Each word in this new RDD is mapped into pairs *(word,1)* by invoking the *map* operation (see line 3), and the generated pairs are represented by another newly created RDD. Spark performs the *reduceByKey* operation on this RDD, and generates the final result.

The Spark application runs as a set of Java JVM processes on a cluster, with a master-slave architecture.

2.3.6.1 The Resilient Distributed Dataset (RDD)

An RDD is a read-only, distributed, partitioned collection of records, and it is the core concept of Spark [95]. An RDD is an object that references the data source, and provides several parallel operations for data processing. Zaharia

Figure 2.8: Spark Streaming Overview [20]

claims that RDDs are so general that RDDs can emulate any distributed system [94].

RDDs can be created by invoking deterministic operations on either data in stable storage or other RDDs. The RDDs are lazy evaluated, which means that the data is evaluated only when *action operations* (they are similar to terminal operations defined in Java 8) are invoked, rather than all of the operations in pipeline are performed on data immediately. In addition, programmers can call a *persist* method to indicate which RDDs are going to be reused in future.

Spark also connects and performs multiple operations in a pipeline on RDDs to optimise the performance, in the same machine. For example, RDDs can be evaluated by applying a *map* followed by a *filter* operation on the same node. Thus, transferring the intermediate results among nodes is avoided.

2.3.6.2 Spark Streaming

Spark Streaming is an extension to Spark, which is designed for live streaming data processing. The core concept of Spark Streaming is Discretized Streams (D-Streams), which were created in order to enable the Spark to provide live streaming data items with an interactive response time. By using D-Streams, a DAG can be created to represent the processing logic.

The key idea of Spark Streaming is that D-Streams treats a live streaming computation as a series of deterministic batch computations on small time intervals [96]. For example, in order to process live streaming data, we can group data received every second into a batch, and processes each batch using Spark. This can be shown in Figure 2.8, Spark streaming groups live streaming data into batches periodically, and processed using the existing Spark runtime.

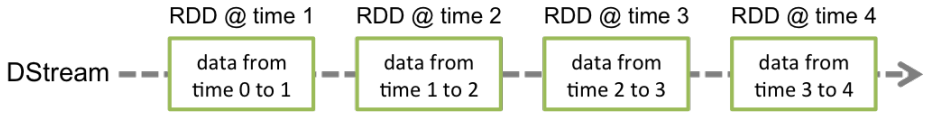


Figure 2.9: Structure of D-Streams [20]

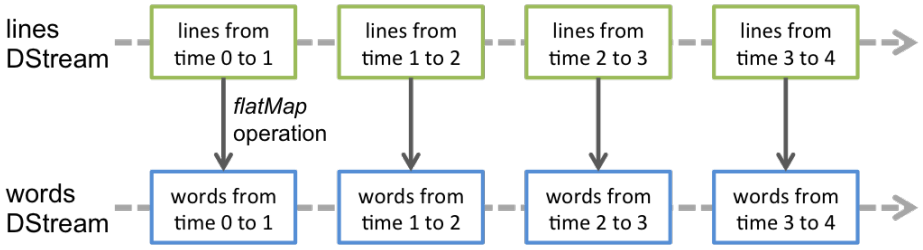


Figure 2.10: Applying the flatMap operation on a D-Streams [20]

The D-Streams Computation Model

A D-Stream is a sequence of immutable, partitioned datasets (RDDs) that can be parallel processed by numbers of operations [94]. These operations can yield new D-Streams or generate outputs, and any operation that is applied on D-Stream will be translated to operations on the underlying RDDs. In a D-Stream, each RDD contains data from a certain interval. Figure 2.9 illustrates the structure of a D-Stream, in this example, the D-Stream consists of RDDs with the period of 1 second. For example, in the WordCount example, the first stage is converting a stream of lines to words. The *lines* D-Stream is transformed by a *flatMap* operation, as described by Figure 2.10. Each RDD in the *lines* D-Stream is evaluated by the *flatMap* operation, and the RDDs representing the words are generated. Finally, the newly generated RDDs form the *words* D-Stream. In this example, the Spark Streaming framework generates a batch processing task (i.e., underlying RDD transformations) every second, and these tasks will be processed by the Spark Engine.

2.3.6.3 Scheduling Spark Applications

By default, Spark schedules applications in FIFO order, each application are allocated with a fixed amount of resources (the number of processors and the size of memory). Dynamic resource allocation is introduced in Spark version 1.2. This approach allows an application to give resources back to the scheduler when it does not use them, and request resources again later when needed. For example, when some tasks within an application are waiting for I/O, the processors that are allocated to these tasks can be given back to the scheduler temporarily.

2.3.6.4 Scheduling Within The Spark Application

Spark hides the details of resource allocation for the pipeline, for example, how many workers are involved by each operation. Spark uses the following concepts and schemes to execute a pipeline in parallel, or in a distributed system.

A Spark application may generate several RDDs as the results. Additionally, one or more operations in a pipeline and source RDDs are used to generate each target RDD.

In Spark, responding to a Spark action, e.g., generating a target RDD, is defined as a *job*. Each job will be compiled into multiple *tasks*. The task in Spark is executable code, typically part of the code within an operation, e.g., the processing logic in a *map* operation. The tasks are executed by the executors, which are JVM processes running the worker node. This section describes how a job is compiled to tasks, and how Spark executes these tasks.

When an RDD is required to be generated, i.e., a job is generated, the scheduler examines all the operations and required input RDDs, then builds multiple *stages* to execute this job. Stages are generated using the following principles:

- Each stage should contain as many pipelined operations with narrow dependencies as possible. The narrow dependencies are the relationship in where multiple operations on a RDD can be composed together into a single operation. For example, in WordCount, there are two operations: mapping lines to words, and mapping words to (word,1) pairs. These two operations can be put into a single operation, because the data/partitions is transformed in a one-to-one relation.

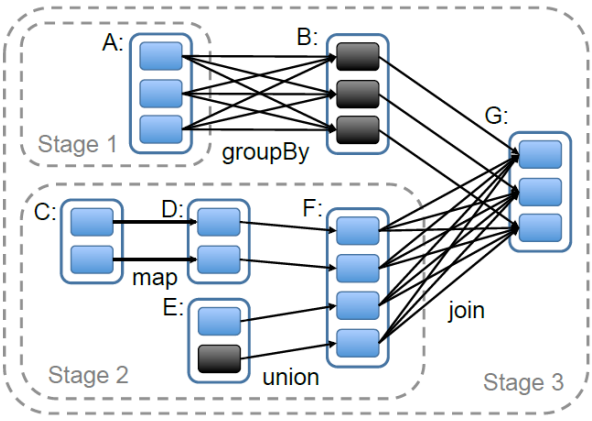


Figure 2.11: The stages in a job. Boxes with solid outlines are RDDs, Shaded rectangles are partitions, black rectangles means partitions are already in memory. [94]

- The boundaries of the stages are either the shuffle operations, or any already computed partitions, which determines that their parent RDD is not required to be computed.

Figure 2.11 illustrates an example of how Spark determine the stages of a job. In this example, in order to generate RDD *G*, Spark builds three stages according to the above principles:

1. Stage 1
   The goal of it is to generate RDD *B*. Note that, in this example, the results have been generated.

2. Stage 2
   The goal of this stage is to generate RDD *F*. The *map* and *union* operations in this stage represent a one-to-one relation, therefore, Spark merges them in the same stage for the optimisation.

3. Stage 3
   In this stage, the *join* operation is required to be performed on RDD *B* and RDD *F*, and then generate the finally required result, i.e., RDD *G*.

As the output RDD of stage 1 is already in memory, therefore Spark runs stage 2, then stage 3. Once the stages are determined, the scheduler generates tasks, which are to compute the missing partitions for each stage. Finally, the target RDD is computed.

The number of tasks spawned by each stage equals to the number of partitions from the target RDD within this stage. In this example, 4 tasks are generated in stage 2, 3 tasks are generated in stage 3, and 1 shuffle task are created between stage 2 and 3.

The tasks will be executed by executors, which are JVM processes. The total number of number of executors, and processors that each executor can use is configured when the application is deployed.

Scheduling The Generated Jobs

The default scheduler is a FIFO scheduler. When scheduling jobs, the first arriving job has the highest priority, and all its tasks inherit its priority. Considering an application may create multiple jobs in parallel, Spark also provides fair scheduling between jobs (called the fair scheduler), in which Spark assigns tasks between jobs in a Round-Robin fashion, so that a short job can receive resources while a long job is running therefore get a good response time. In addition, the fair scheduler supports grouping jobs into pools with different scheduling options (e.g., weight). This can be used to create a high priority pool for more important jobs.

2.3.6.5 Discussion

In Spark, the data transformations in RDDs is quite similar to the one in Java 8 streams [12], and both of them are lazily evaluated. Additionally, the Spark engine employs executors that are distributed over a cluster to execute the generated tasks from evaluating a pipeline, while Java 8 uses ForkJoin Thread Pool to achieve this purpose.

However, as a time-sharing framework, it is not easy to add real-time constraints on Spark:

- Spark runs on standard JVM, the overall runtime of the Spark engine lacks real-time features.
- Preemption is not supported by the Spark scheduler. If a new higher priority job is submitted, and there if no idle resources, it is not possible for the scheduler to take certain amount of computation resource back, and runs this higher priority job.
- The execution of threads in Spark can not be bounded, therefore, it is

difficult to integrate Spark into a real-time system, which also has hard real-time component.

2.3.7 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.

Lambda expressions provide a clear and concise way to represent one method interface using an expression [10], 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*. Functional interfaces were introduced by Java 8, and are interfaces that contain exactly one abstract method which can not have a default implementation. They may define other methods as long as those methods do have default implementations. For example, `java.util.function.Consumer<T>` is a functional interface. It has only one abstract method (see line 4), and its source is described as follows.

```
1 @FunctionalInterface
2 public interface Consumer<T> {
3     /** Performs this operation on the given argument. */
4     void accept(T t);
5
6     default Consumer<T> andThen(Consumer<? super T> after) {
7         Objects.requireNonNull(after);
8         return (T t) -> { accept(t); after.accept(t); };
9     }
10 }
```

In addition, lambda expressions use target typing [86], 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.

A Java 8 stream is a sequence of operations and a data source. The

`Stream` itself is an interface, which defines all the operations supported by the Java 8 Stream framework. The actual implementation of streams are pipelines, for example, the implementation for a stream of Java objects is `java.util.stream.ReferencePipeline`. In addition, Java 8 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 performing methods on the data source. One example of intermediate operations is *map*, which maps each data element 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:

```
1 Collection<String> datatoProcess = WordsToCount;
2 Map<Object, Long> result = datatoProcess
3   .parallelStream()
4   .flatMap(line->Stream.of(Pattern.compile("\\s+").split(line)))
5   .collect(Collectors.groupingBy(
6     w -> w, TreeMap::new, Collectors.counting())
7   );
```

It first create a stream, which inputs from a collection of strings (see line 3). Each string is split to words (see line 4), these words are counted, and accumulated into a Java `Map` (see line 5). The procedure is performed in parallel, the details of how the processing is parallelised are given in the following sections.

2.3.7.1 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. To be able to be evaluated in parallel, it requires the data source to be splitable. Efficiency is achieved

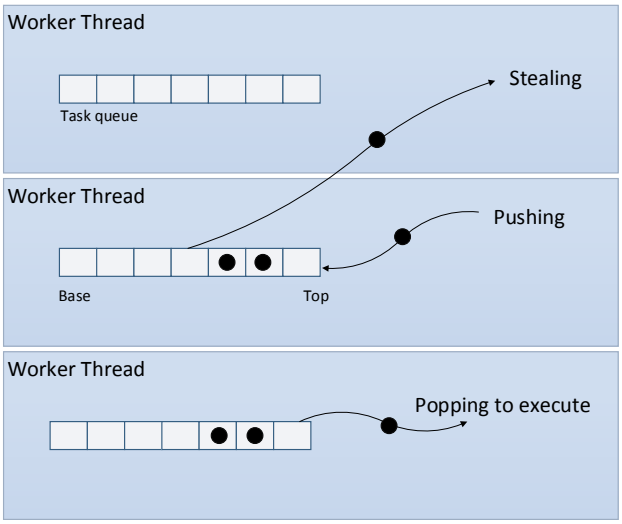


Figure 2.12: Tasks stealing, pushing and popping within worker threads

by the work stealing algorithm that is used by the ForkJoin pool.

2.3.7.2 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 [63]. 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 on 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.

2.3.7.3 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 [63] 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 current 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 2.12.

2.3.7.4 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

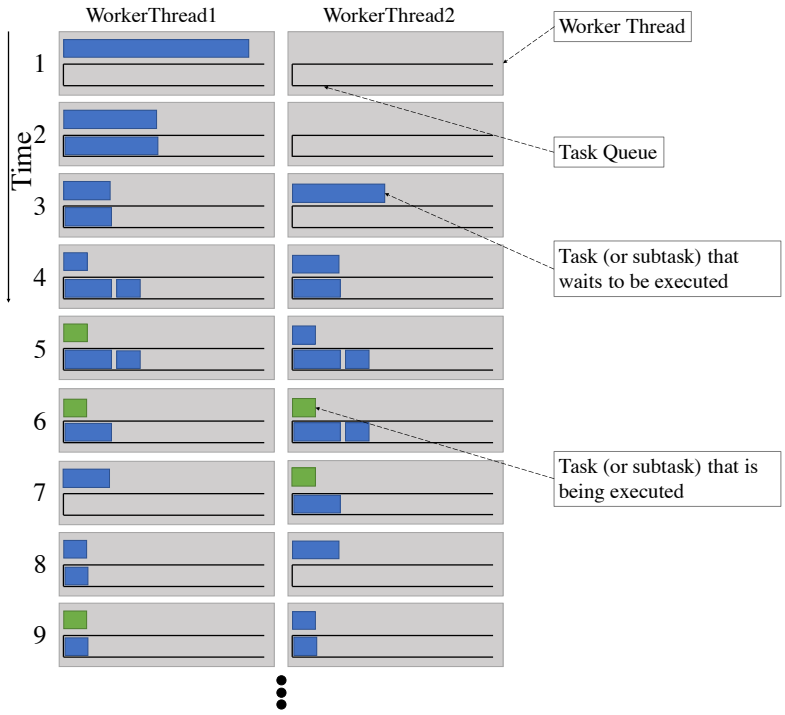


Figure 2.13: The parallel evaluation of a Stream by a pool with 2 threads.

to that pool directly. 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 2.13. 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.

2.3.7.5 Discussion

The Java 8 Stream API enables pipelined and parallelised processing of data sources in a Fork/Join manner, with concise code. However, by connecting multiple streams in different ways, more complicated parallel processing algorithms can be obtained.

The Java 8 Stream API has not been designed to address real-time concerns. Firstly, even with a real-time Java virtual machine, there is no way to place real-time constraints on the program.

Secondly, the executing of worker threads in a pool, which evaluates the stream processing, is not bounded. Therefore, it may demand unlimited CPU time, therefore causing the hard real-time tasks in the same system to miss their deadlines.

Moreover, Java 8 streams assume the data has already been stored in memory, therefore, the live streaming data is not supported by Java 8 streams.

2.3.8 Storm

Apache Storm [4] is a stream processing framework developed at Twitter using the Clojure programming language, and provides multiple programming language APIs, including Java, Python etc. Storm has seen wide commercial adoption from companies such as Yahoo!, The Weather Channel, Alibaba, Baidu, Groupon and Rocket Fuel [90].

Storm defines five basic concepts: *streams*, *tuples*, *spouts*, *bolts* and *topologies*. A tuple is a data structure that stores values. A stream is an unbounded sequence of tuples. Unlike Java 8 Streams which carry references to heap data, Storm streams pass the data itself. In addition, the stream in Storm is eagerly evaluated, data elements at a stream are transformed immediately in each stage of a pipeline. A spout is a source of stream which emits tuples. A bolt processes one or more input streams, produces new tuples and passes them to one or more new output streams. By connecting spouts and bolts together, data elements can flow through the stream. The graph of this connection is named the topology, where the edges represent the data flow and vertices are computation components (spouts or bolts).

Figure 2.14 illustrates a simple topology that counts the words occurring in a stream of sentences, and also illustrates how the data in the stream is moved and processed. There are 1 spout (emits sentences to a stream) and

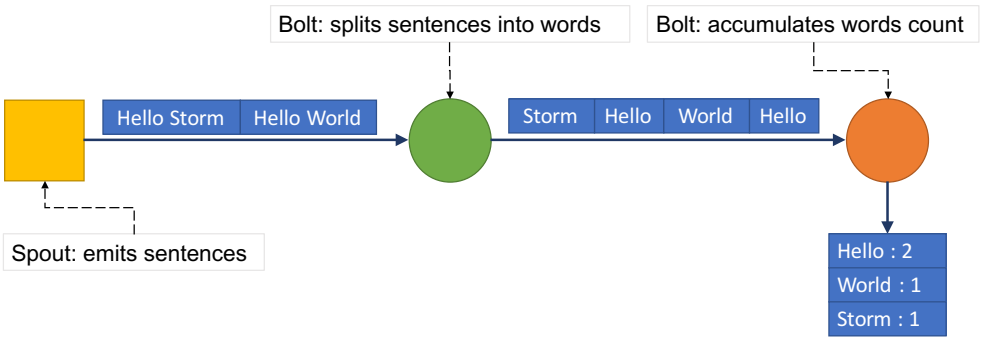


Figure 2.14: WordCount topology and data flow in it.

2 bolts (one bolt parses received sentences, another bolt counts the words) in this topology. In this example, two sentences (<Hello World> and <Hello Storm>) are emitted to the stream by the spout. Once the first bolt receives a sentence, it parses the received sentence immediately and emits <Hello> and <World> to the downstream. Similarly, when the second sentence is received, <Hello>, <Storm> are parsed out and emitted to the downstream. The last bolt accumulates the number of each words occurring, and generates the final result.

2.3.8.1 Storm Runtime Overview

Storm runs on a distributed compute cluster, using a master-slave architecture. The actual work is done by *worker processes* that are running on the worker node. Each worker process is an OS process that is running a separate JVM, and it spawns threads called *executors* to perform the processing. Note that, each worker process only executes parts of a single topology, multiple worker processes on the same node may execute different part of the same topology.

The actual computation of the data processing of a spout or a bolt is encapsulated into a *task*. The parallelism is achieved by running multiple executors, each of which may execute one or more tasks.

Data items from spouts/bolts (producer) are shuffled to tasks within bolts (consumer) in a storm topology, for the load balancing. Several built-in shuffling algorithms are provided, such as, data items are evenly distributed to the down-stream consumers. Additionally, users can implement a customised shuffler by implementing the *CustomStreamGrouping* interface.

2.3.8.2 Scheduling a Storm Topology

This section introduces how a Storm topology is scheduled by introducing an example, which considers a topology consisting of three components: 2 spouts called yellow-spouts, 4 bolts called blue-bolts and 4 bolts called green-bolts. The components are linked so that yellow-spouts send their outputs to blue-bolts, which in turn send their outputs to green-bolts. This topology is defined by the following code based on a Storm version 2.0 example [25]:

```
1 Config conf = new Config();
2 conf.setNumWorkers(2); /* set 2 worker process */
3
4 TopologyBuilder builder = new TopologyBuilder();
5
6 /* 2 spouts */
7 builder.setSpout("yellow-spout", new YellowSpout(), 2);
8
9 /* 4 blue-bolts, and each has 2 tasks */
10 builder.setBolt("blue-bolt", new BlueBolt(), 4)
11         .setNumTasks(2).shuffleGrouping("yellow-spout");
12
13 /* 4 green-bolts */
14 builder.setBolt("green-bolt", new GreenBolt(), 4)
15         .shuffleGrouping("blue-bolt");
16
17 StormSubmitter.submitTopology("example-topology",
18                                conf, builder.createTopology());
```

In the code, the topology is configured to use 2 worker processes (see line 2). The yellow-spouts are defined to use 2 executors and each yellow-spout is encapsulated into 1 task by default (see line 7). Note that, the actual work is performed through a task, each task contains the code corresponding to the user-defined function a spout/bolt. Similarly, the blue-bolts use 4 executors and each blue-bolt is encapsulated into 2 task (see line 10-11). The green-bolts use 4 executors and each blue-bolt is encapsulated into 1 task (see line 14-15).

The Storm scheduler allocates all $(2 + 4 + 4 = 10)$ the executors into 2 worker processes evenly, and this procedure can be illustrated by Figure 2.15. The left part shows the architecture of the topology. The right part illustrates how tasks are allocated into executors, and how executors are packed into worker processes.

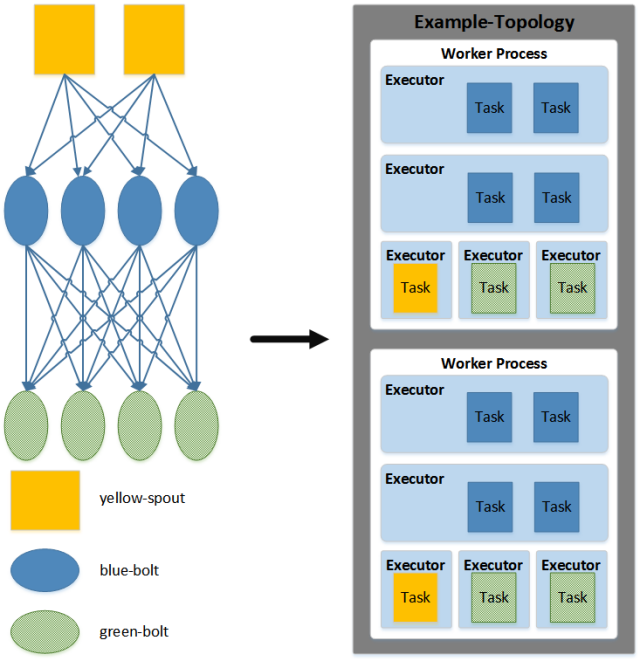


Figure 2.15: Mapping a storm topology to worker processes, executors and tasks

More specifically, from the point view of the OS, the worker processes are separate JVM processes, executors are threads running within each worker process, and a task is a code fragment defined by users that will be executed by threads(i.e., executors). In this example, Storm spawns 2 JVM processes, and each JVM process spawns 5 threads within it. Figure 2.16 illustrates how this example is executed within a worker node in a storm cluster. The two worker processes in this example are mapped into 2 JVM processes, and five executors in each worker processes are mapped into corresponding JVM threads. Each slot allows one process to be created, the total number of slots in a worker node is configured in the deployment.

2.3.8.3 Discussion

Storm is a fast in-memory stream processing framework, however, Storm is not designed for real-time systems. There are many difficulties with using Storm in a real-time system:

- The stack of Storm's runtime lacks real-time features. For example, Storm is developed with the Clojure programming language and runs on

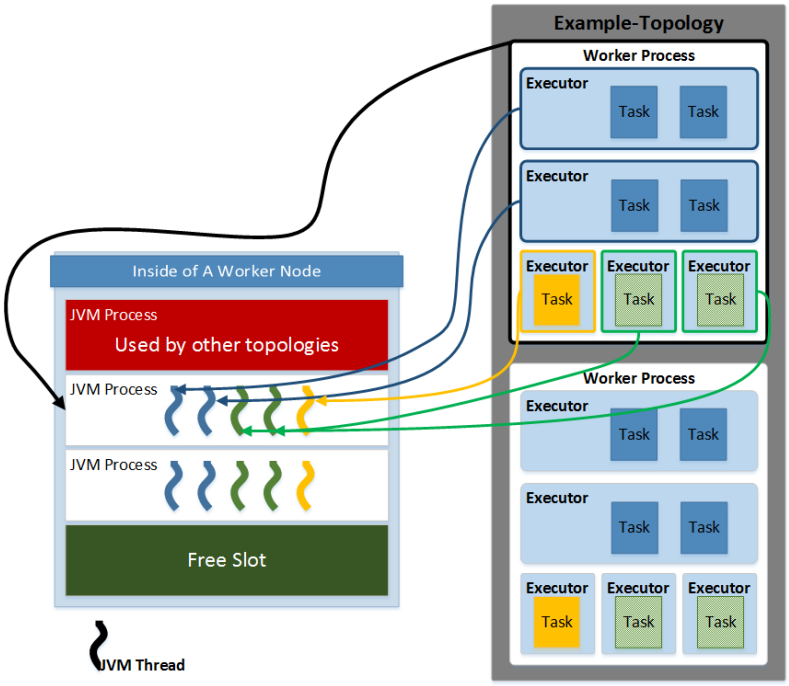


Figure 2.16: Inside of a node in Storm Cluster

a standard JVM.

- Storm itself does not support real-time notions, such as, priorities, deadlines, etc.

- Storm treats the weight of each workload, i.e., tasks, equally, irrespective of the load of each processor, and creates one or more threads in each processor, then allocate the workload to all the threads evenly.

- The thread in Storm can not be allocated to a specific processor, because its scheduler only aware of the how many available slots can be used in each machine. This means some real-time scheduling schemes can not be supported, for example, the fully-partitioned scheduling.

- The execution of threads in Storm can not be bounded. When running Storm at a high priority, it may cause the hard real-time tasks in the same system to miss their deadlines.

2.3.9 Predictable Stream Processing Frameworks

This section briefly reviews soft real-time stream processing framework, such as the JUNIPER project [31], and a real-time version [33] of Storm. Additionally, an investigation of using work-stealing in soft real-time stream processing system is also reviewed.

2.3.9.1 JUNIPER

JUNIPER [31] is an European Union Seventh Framework project, which provides a Java platform for high-performance and real-time large scale data processing.

JUNIPER provides a real-time operation system based on Linux, a real-time Java virtual machine, and a real-time modelling tool that supports model-driven engineering. JUNIPER also defines its own programming model, which is intended to provide a set of APIs or models so that several existing parallel processing frameworks, e.g., Storm [4] or Spark [1], can be be built upon it, rather than replicating these existing frameworks. For example, a distributed version [42] of Java 8 Streams is developed as a distributed large scale data processing framework.

In addition, JUNIPER employs FPGAs to accelerate Java programs [56] in order to deliver a high performance. A Java to C compiler and a C to hardware description language tool are used, so that FPGA components can be generated directly from Java code.

The JUNIPER programming model is based on Java 8 [12] with the Real-Time Specification for Java (RTSJ) [91], so that the programs can be programed with real-time systems. The JUNIPER programming model defines several real-time components to support real-time programming. For example,

- *Programs*
  The program is written using the RTSJ to capture the real-time constraints.

- *Channels*
  Channels are used to represents analysable data flow between programs. Channels are modelled as either periodic or sporadic, so that periodically or sporadically moving data can be represented.

Disk bandwidth reserving techniques [80] are also developed so that the storage accessing is able to be predictable and analysable.

Discussion

The JUNIPER project uses model-driven engineering to support automatic code generation, rapid deployment, modelling deadline constraints, etc.. In addition, the JUNIPER programming model also enables large scale data processing frameworks to be developed with it.

However, there are several issues, such as, Distributed Streams [42] do not provide any interfaces to configure deadline constraints or priorities. Therefore, a high-level real-time stream processing framework is missing.

2.3.9.2 Real-Time Storm

A real-time version (RT-Stream) of Storm [4] was proposed in [33], to provide predictable stream processing. This work establishes a tool stack including a real-time OS, a real-time JavaVM, and extended Storm classes which support real-time constraints.

The notion of real-time stream is described as "a continuous sequence of data or items whose processing has some real-time requirements like a deadline from the input to the output" in [33]. The idea of this work is to model a real-time stream into a set of real-time activities, and provide related schedulability analysis approaches.

Firstly, Storm's *Spout* (input) and *Bolt* (processing and output) are extended to be periodic activities, or sporadic activities with minimum interval times (MIT). The new classes are called *RTSpout* (input) and *RTBolt*. They allows the period (or MIT if sporadic), worst-case execution time, and the deadline to be given to each *RTSpout* or *RTBolt*. In addition, a fixed-priority scheduler is provided.

Then, the graph of stages of a real-time stream is built by analysing the stream processing graph, which is a DAG of *RTSpouts* and *RTBolts*.

Finally, by performing an end-to-end response time analysis on each *RTSpouts* and *RTBolts* in each stage, the response time of the stream processing can be obtained.

An example from the original paper [33] considers a real-time stream, which has a periodic *RTSpout* with a period of 100 ms, and two *RTBolts* with a period of 200 ms. The worst-case execution time of them are all 10 ms. The stream flows from the *RTSpout* to the two *RTBolts*. The two *RTBolts* runs in parallel. This example makes an assumption that there is no any other

higher priority activities in the system. Therefore, the end-to-end worst-case response time is calculated as: 10ms (for the *RTSpout*) + max(10ms,10ms) for the parallel *RTBolts*.

In addition, an utilisation based schedubility test equation is also given by this work, and it is a sufficient but not necessary analysis. The utilisation based analysis is done by performing analysis on each worker node in the cluster, using the Liu and Layland utilisation bound [67].

Discussion

RT-Storm enables distributed soft real-time stream processing. However, there are some issues:

- The algorithms for determining the period or MIT, and the load for each *RTSpouts* or *RTBolts* in a system which also hosts other real-time activities are not considered.
- The unpredictability of the network, and its impact on the worst-case response time is not addressed.

2.3.9.3 Work Stealing for Parallel Stream Processing in Soft Real-Time Systems

A thread pool is often used to provide the parallel threads needed to perform the stream processing. A major load balancing technique is the work stealing algorithm. In a context of soft real-time systems, the work stealing strategies for parallel stream processing is investigated by [70]. The work stealing is also used by Java 8 ForkJoin framework.

This work considers using multiple threads to processing sequence of inputs in parallel. Each input requires multiple processing stages, which forms a graph or a pipeline. When processing an input, each stage is treated as a *subtask*. Each thread maintains a local queue, which is used to store these generated subtasks. In addition, the system also maintains a shared global queue, which is used to store the input. Similar to Java 8 Fork and Join framework (see Section 2.3.7), when a thread is idle, it tries to steal, i.e., take, work from other threads' local queue, or from the shared global queue.

For example, as shown in Figure 2.17, there are two inputs in the system, which will be processed using a pipeline, with a work-stealing strategy. The pipeline has 3 stages, therefore, 3 subtasks are generated for each input. For

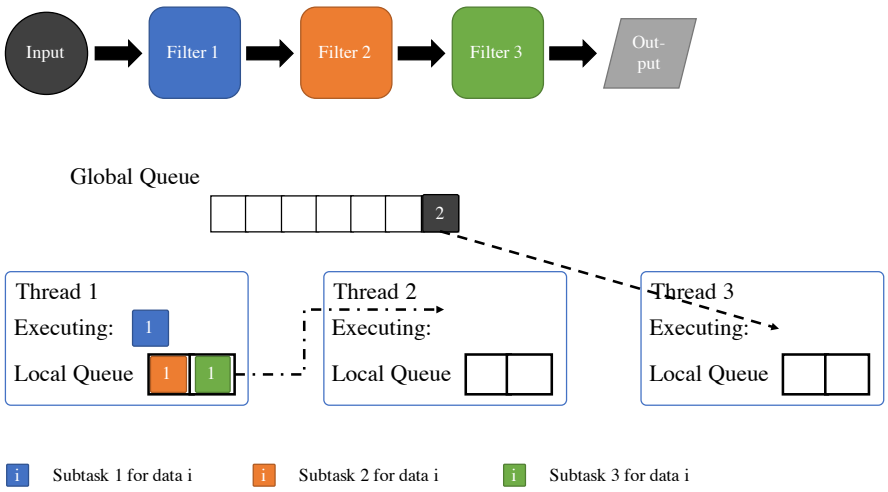


Figure 2.17: Work-stealing in a stream processing system with a pipeline

example, subtask 1 represents performing filter 1 on input 1. In this example, thread 1 has taken the input 1, generated 3 subtasks, and currently executing subtask 1. The subtasks 2 and 3 have been pushed into its local queue. Thread 2 and 3 are idle. Thread 2 is stealing a subtask from thread 1, and thread 3 is taking the next input from the global queue.

This work [70] investigates all the possible policies for inputs or subtasks accessing schemes, such as,

- Local - The input and subtasks goes into the thread's local queue, rather the global queue.
- Global - The input is pushed into the global queue, but subtasks goes into the local queue.
- Stealing First - When a thread is idle, it tries to check other thread's queues first, then the global queue.
- Global First - When a thread is idle, it tries to check the global queue first.

The conclusion is [70]: considering the latency for processing each input, the best combination is the input goes into the global queue, and idle thread tries to check global queue first. The reason is that, with the stealing first strategy, the system suffers from a loss of data locality.

Latency Bound

An approach that is used to calculate the latency bound when using the global queue to store the input is also given in this work [70]. The worst-case execution time w for processing each input is defined as the sum of the execution time of each filter. In addition, the maximum number of inputs that are waiting, including the input that is under the analysis is defined as α.

The worst-case latency bound for the input is w, i.e., the worst-case processing time of itself, + $w(\frac{\alpha-1}{n})$, i.e., the worst-case processing time of inputs before it, where n is the number of processors.

Difference with Java 8 Streams

The difference between the work and Java 8 Streams is that the input in Java 8 Streams is a collection of data, rather than individual data items. In addition, Java 8 Streams is using a stealing first strategy. This is because Java 8 Streams aim to minimise the response time of processing each collection, while this work targets at the throughput.

Discussion

This work evaluates different policies in a stream processing system with a work-stealing algorithm, and gives the conclusion about the best policies. However, this work assumes that the stream processing is using a dedicated system, there is no any other activities. When there is any other activity, such as the operating system, the response time can be bigger to the worst-case execution time.

In addition, the issues that have been discussed in the previous sections of integrating stream processing into a real-time system, which also has hard real-time activities, are still open.

2.4 Summary

This chapter has briefly introduced parallel computer architecture, real-time system models, and stream processing. A brief history of stream processing which describes several typical stream processing has been given in Section 2.3.3. Section 2.3.4 discusses the stream processing classifications, including lazy or eager evaluation, and control-parallel data-parallel scheme. From

an efficiency viewpoint, as discussed in Section 2.3.4, lazy evaluation and data parallelism should be employed as much as possible, unless there is a necessity to use the other approaches, for example to facilitate distributed communication.

This chapter has also reviewed several typical stream processing techniques for both frameworks, and programming languages in different classifications, including Java 8 Streams, StreamIt, Storm, Spark Streaming. In addition, this chapter also reviews a real-time version of Storm, and JUNIPER project, which were designed to address real-time constraints.

StreamIt and Java 8 Streams support stream processing at the language level, but StreamIt is not a general purpose programming language. Storm and Spark Streaming are distributed live streaming data processing frameworks. The former one is designed for live streaming data, while the later one groups live streaming data into micro batches, and reuses the Spark batching processing runtime. However, none of these techniques fully considers the real-time constraints, although the real-time version of Storm and JUNIPER makes their first step toward addressing soft real-time constraints.

This chapter observed that none of the current stream processing techniques support real-time stream processing that can be integrated into a real-time system that also has hard real-time activities.

In summary, the real-time stream processing has the following challenges:

- Common stream processing frameworks are designed for time-sharing systems, their programming interface provides no support for capturing the real-time properties, such as priorities, deadlines.

- In addition, most of their runtimes are non real-time, for example, Java 8 Streams, Storm, and Spark run on standard JVMs. Porting these techniques directly into a real-time runtime, e.g., a real-time JVM, may cause unexpected problems. For example, as described in [71], processing parallel Java 8 Streams directly within a RTSJ real-time thread may suffer priority inversion problems.

- None of the the current techniques addresses the issue of performing stream processing in a real-time system so that its deadline can me met, whilst guaranteeing all the hard real-time activities in the system still meet their deadlines.

- None of the current stream processing techniques can *guarantee* the worst-case latency of each data item's processing in a data stream, or the response time of a batched data's processing.

The following chapters will address these challenges by providing a real-time stream processing architecture for multiprocessor platforms, with corresponding scheduling techniques, integration approaches so that the stream processing can be integrated into a real-time system that also has hard real-time activities. Then response time analysis equations are derived to guarantee that the real-time requirements of the stream processing are met.

Chapter 3

The Real-Time Stream Processing Infrastructure

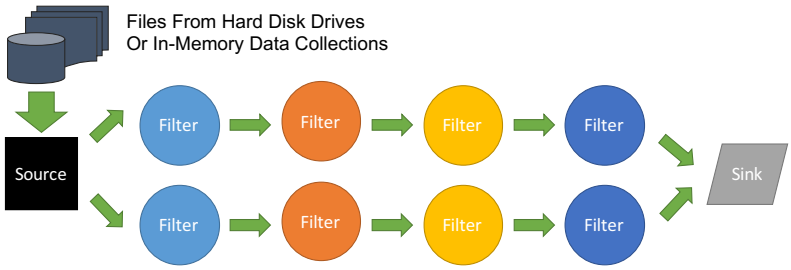
To address the issues discussed in Section 2.4, the goal of this chapter is to propose a real-time stream processing infrastructure with which both batched and live streaming data sources can be processed within the deadline or the latency requirements, while maintaining existing guarantees to the other hard real-time activities in the same system.

This chapter defines a real-time stream processing task model and the architecture of an infrastructure that supports that model. Applications that perform stream processing and run on this architecture must follow the pipeline software design pattern [81], as illustrated by Figure 3.1.

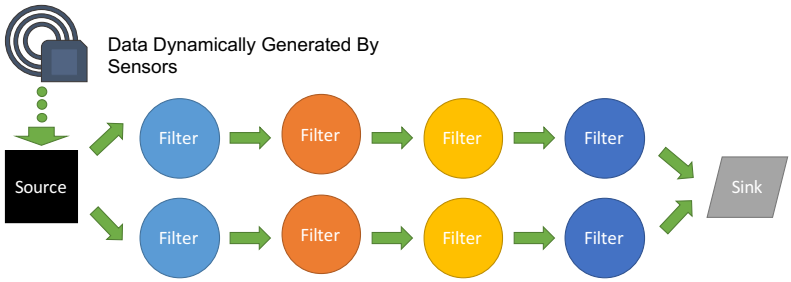
In the previous chapter, it was shown that the data source of stream processing can be from batched data or live streaming data; hence this pattern can be specialised into: processing a batched data source as illustrated by Figure 3.2a; and processing a live streaming data source as shown in Figure 3.2b.

Hence the infrastructure must provide an appropriate API for user applications, and be capable of supporting these two use cases. We assume the existence of a real-time stream processing task which encapsulates all the ac-

Figure 3.1: The pipeline software design pattern



(a) Processing batched data sources.



(b) Processing live streaming data sources.

Figure 3.2: Stream processing from different data sources.

tivities associated with reading and processing the data.

- For a batched data source, the task uses a real-time batch stream processing infrastructure as shown in Figure 3.3.

- For a live streaming data source, each data item can be processed by the proposed approach shown in Figure 3.4. The batcher uses *real-time micro-batching*. This means that it stores individual data items into a collection, and returns the collection sporadically when either the maximum batch size is reached or the timeout expires. See Section 3.4.2 for more details. Each of the returned collections can then be treated as a static data source and processed using the existing real-time batch stream processing infrastructure.

The reason for using real-time micro-batching when processing live streaming data sources is because it allows the data items to be processed more efficiently, compared to processing individually [96]. Processing items individually is inefficient because of infrastructure costs such as maintaining the tracking for each individual data item considering the failure recovery in a distributed context, or requiring execution-time servers with smaller periods

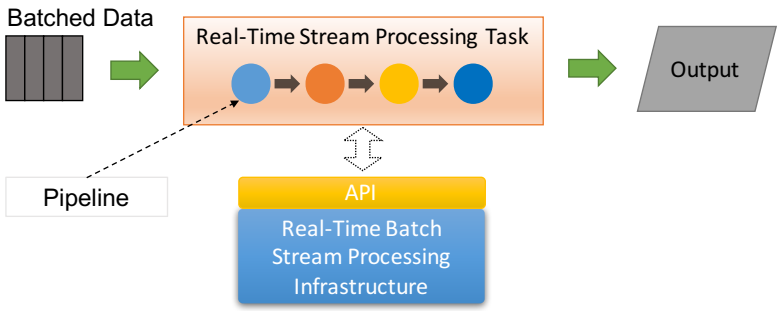


Figure 3.3: Real-time stream processing for a batched data source.

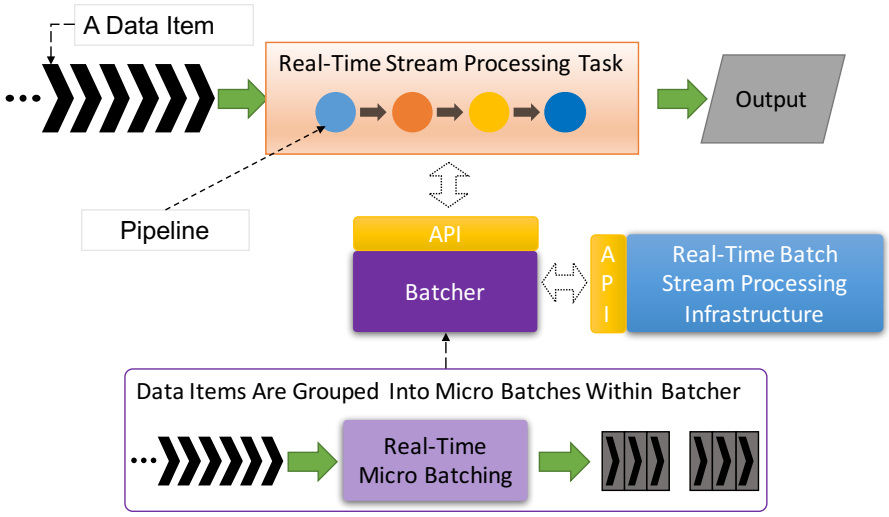


Figure 3.4: Processing a sequence of data items in real-time using the real-time micro-batching approach.

in this thesis (see the server parameter selection algorithm described in Section 4.3.1), therefore introducing extra context switch overheads. In addition, with a variable batch size and a timeout, real-time micro-batching allows the processing latency of each data item to be guaranteed. Section 4.4.1 shows how to determine the micro batch size and timeout values, and this is exemplified in the case study of Section 5.6.

The details of the architecture of the real-time batch stream processing infrastructure, and the real-time micro-batching architecture are described in the following sections.

There are many different approaches to defining and describing software architectures along with their design principles and rationales (see [51] for

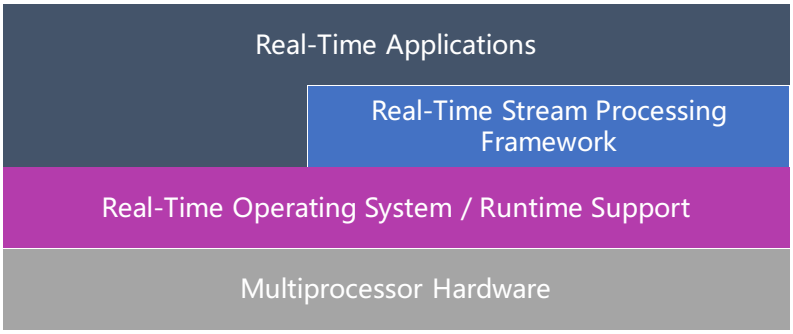


Figure 3.5: Real-time stream processing system overview

a review). Here we follow the principle that proposes "separating different aspects into separate views" [51] to describe the architecture of our proposed real-time stream processing infrastructure, using text descriptions alongside UML2 [79] diagrams. In particular, we make use of component diagrams and sequence diagrams to show the main components of the architecture and how they interact with each other and the end user.

This chapter is structured as follows. Section 3.1 gives the context of the infrastructure in a system. Section 3.2 describes the system model supported by the infrastructure and the assumptions on the underlying platform. The proposed real-time stream processing task model that is required by the infrastructure is described in Section 3.3. Section 3.4 describes the architecture specification of the proposed real-time stream processing infrastructure, along with component diagrams, and gives the implementation requirements. Finally, Section 3.5 summarises the contents of this chapter.

3.1 System Architecture

This section describes the system context for the proposed real-time stream processing. The system context can be illustrated by Figure 3.5, and contains the following layers:

- **Hardware**
  The lowest layer is the hardware layer. Typically, it is a physical multiprocessor machine, with cache coherent shared memory.

- **Real-Time Operating Systems and Runtime**
  The second layer runs a real-time operating system, for example, VxWorks, or Linux with a real-time kernel [14]. For instance, when the

applications are developed using the Ada programming language, they are running directly on the top of these operating systems. However, when it is required, such as developing using Java with the Real-Time Specification for Java (RTSJ), a real-time runtime also runs in this layer, such as a real-time Java Virtual Machine (JVM), e.g., JamaicaVM [9], which is running on the top of the operating system.

- **Real-Time Applications and The Real-Time Stream Processing framework**
  This layer contains applications that contain both hard real-time and soft real-time tasks. In addition, there are also several real-time stream processing applications, which use the real-time stream processing framework that implements the proposed architecture. The real-time stream processing framework processes both batched data and live streaming data processing in parallel to meet the real-time requirements, whilst maintaining the existing guarantees of any other hard real-time activities.

Note that, the design or implementation of the hardware, or the real-time operating system and runtimes layer is out of the scope of this thesis. Additionally, considering the deployment, the operating system and runtime supports are not necessarily required if the real-time stream processing infrastructure can be implemented directly on the top of bare metal.

3.2 System Model Supported by the Infrastructure

From the point view of real-time literature, the following real-time system models are supported by the work presented in this thesis:

- **Preemptive Fixed Priority**
  Rationale – as described in Section 2.2.1, priority-based scheduling is the dominant approach and the one supported by all real-time operating systems, and the preemption scheme makes higher priority tasks more responsive [38].

- **Fully Partitioned Scheduling**
  Rationale – as described in Section 2.2.2, schedulability analysis for such systems is more mature [38].

- **Sporadic Task Model**
  Rationale – this is the default model supported by schedulability analysis literature.

- **Mixed Hard Real-Time and Soft Real-Time Applications**
  Rationale – most real-time systems have either hard real-time, or soft real-time, or both hard and soft real-time applications [38]. The challenge of this work is to integrate soft/hard real-time streaming work with hard real-time activities.

- **Sporadic Live Streaming Data**
  Rationale – from the point of view of schedulability analysis in the real-time literature, the analysis for data with periodic or sporadic arrival is more mature [33]. Also in most systems data is going to arrive sporadically so it is not possible to simply claim it is periodic.

- **Hard and Soft Real-Time Stream Processing Activity**
  Rationale – similar to common real-time applications, both hard real-time and soft real-time stream processing are required to be supported in the most common real-time literature.

  Additionally, real-world systems are not entirely hard or soft because hard real-time components have to be carefully developed and analysed, and designers should try to minimise them as much as possible, so it is not realistic to simply claim the entire system is hard real-time.

- **Multiple Simultaneous Streaming Workloads**
  Rationale – this work is primarily focussed on embedded systems. Typically there is the request to process multiple streams from different data sources. Currently this thesis focuses on a single stream on a single multiprocessor machine, and considers the challenges of multiple streams in distributed systems as future work.

3.3 Real-Time Stream Processing Task Model

This section defines a predictable and analysable real-time stream processing task model, supported by the proposed infrastructure.

The proposed structure of a real-time stream processing task is illustrated as Figure 3.6. As shown in the figure, a real-time stream processing task

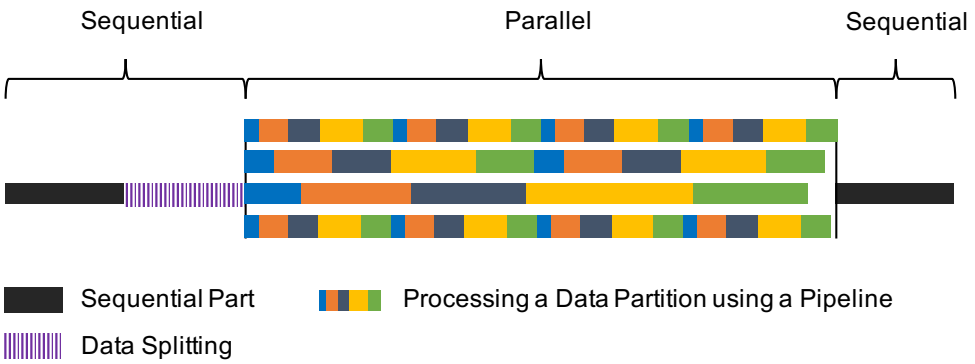


Figure 3.6: The structure of the real-time stream processing task

contains three phases: sequential execution before the parallel processing, the parallel data processing itself, and then sequential execution after the processing is complete.

The sequential code before and after the parallel processing is typically processed by the same processor, and this is the assumption used for our analysis model described in Chapter 5 for simplification. Arguably, they can be executed by different processor, more discussion on this is given in Section 4.2.1.

Data splitting (partitioning the input data ready for processing) must be performed sequentially, so that its analysis can be simplified, and avoid analytical pessimism (as described in Section 5.4.3). Additionally, data splitting is required to be completed before the parallel processing, because splitting on the fly can be interfered with by higher priority tasks in the same processor, therefore, delaying the processing in the other processors. In addition, splitting before the parallel processing also simplifies the analysis.

The proposed real-time stream processing task structure uses data parallelism, as illustrated by Figure 3.6, each processor performs the pipeline operations on separate partitions of the input data. The reason is that this thesis focuses on UMA platforms, as discussed in Section 2.3.4 and 2.3.7, data parallelism is sufficient and efficient. Additionally, this structure is independent of whether the pipeline is evaluated eagerly or lazily.

Multiple real-time streaming tasks are obtained by creating multiple instances of the proposed real-time stream processing task model.

The proposed real-time stream processing task supports (i.e., can be run in) the sporadic task model. The whole procedure as illustrated by Figure 3.6,

i.e., the structure of the real-time stream processing task, can be released (i.e., invoked) periodically or sporadically.

3.4 Architecture and Specification of the Real-Time Stream Processing Infrastructure

This section describes the architecture for the real-time stream processing infrastructure, with which the real-time stream processing task model described in Section 3.3 can be supported.

This section begins with the description of the architecture for real-time batched data processing, which is then extended to support real-time live streaming data processing. In addition, the implementation requirements for the proposed architecture are also given.

3.4.1 Supporting Real-Time Batched Stream Processing

The proposed real-time batch stream processing infrastructure supports the real-time stream processing task model proposed in Section 3.3 with a batched data source as its input. The proposed architecture for this infrastructure can be illustrated by Figure 3.7. Any instance of this infrastructure is used as a part of a real-time stream processing task that follows the model defined in Section 3.3. The main purpose of the proposed infrastructure is to allow batched data to be processed in parallel and in real-time, within a real-time stream processing task. Therefore, as a part of a real-time stream processing task, the proposed infrastructure is periodically (or sporadically) invoked with a batch to process.

The proposed infrastructure requires several configuration parameters, such as, priorities, execution-time servers, etc., to be configured. However, the details of how to generate the configuration parameters for a real-time stream processing task are described in the next chapter, i.e., Chapter 4.

Architecture Specifications

The proposed infrastructure is a subsystem, an instance of this infrastructure maintains the following components:

- A *Data Partitioner* – splits a batch into partitions to be executed by the workers. It provides the *Split Batch* interface, which splits a given

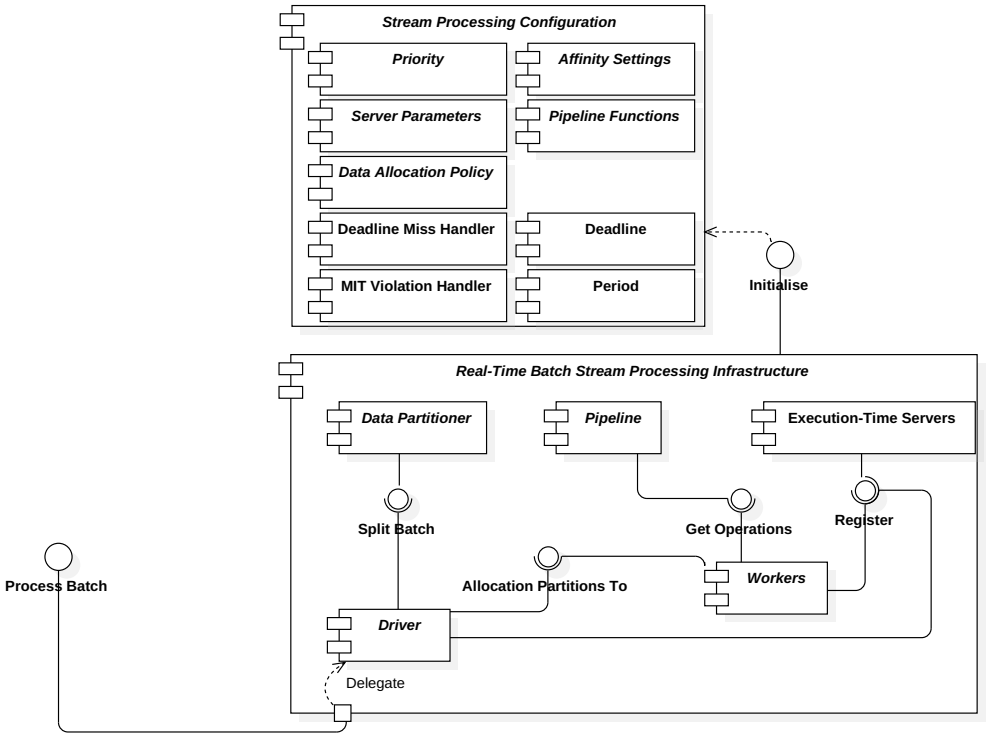


Figure 3.7: Real-time batched stream processing infrastructure component diagram.

batch into partitions.

- A *Pipeline* – which represents the processing logic, typically it contains one or more filters, and a sink. It provides the *Get Operations* interface, which returns all the operations recorded in the pipeline.

- Multiple *Workers* – there is one worker per processor. Each worker processes the allocated partitions at the given priority, with the processing logic defined by the pipeline. The worker provides the *Allocation Partitions To* interface, which allows one or more partitions to be allocated to the worker.

- Multiple *Execution-Time Servers* – providing the *Register* interface, which allows each worker to be registered to its corresponding execution-time server. The worker executes only when its server has capacity, otherwise the worker has to be suspended or transferred to the background priority.

- A *Driver* – which performs the splitting of the batch using the data partitioner and allocates the data partitions to workers for parallel processing. The driver implements the *Process Batch* interface. Once the *Process Batch* interface is invoked, the driver is executed with the given batched data.

For the initialisation, the real-time stream processing infrastructure requires the following parameters:

- *Priority* – the priority at which the workers and the driver execute.
- *Server Parameters* – the server parameters, for example, period, capacity, etc., for each execution-time server running in different processors.
- *Affinity* – which allow the fully-partitioned scheduling scheme to be supported, by pining each schedulable instance to the allocated processor.
- *Pipeline Functions* – describes the processing pipeline.
- *Data Allocation Policy* – describes how the partitions are allocated to each worker and their orders.

The affinity settings are based on the processors given to the application, and pipeline structure is defined by the users. The execution-time server parameters, priority, and data allocation policy are determined in Section 4.3.

In addition, the following additional parameters are required for a hard real-time stream processing task:

- *Deadline* – the deadline for the real-time stream processing.
- *Deadline Miss Handler* – the handler for the deadline miss.
- *Period* – describes the period, or the minimum inter-arrival time (MIT) of the invocation of the *Process Batch* interface.
- *MIT Violation Handler* – the handler for the invocation MIT violation.

Implementation Requirements

An implementation of this architecture must conform to the following requirements:

- The worker is required to be implemented as a schedulable instance, e.g., an OS thread, in order to exploit the parallelism provided by the underlying hardware platform.
- The driver can be either implemented as a function, or a schedulable instance. In the former case, the functionality of the driver is executed by the caller which itself is a schedulable instance. In the later case, the driver task itself executes its functionality.
- Each worker is required to be created before any processing occurs. For example, the workers can be created when the infrastructure is initialised. This avoids any delay introduced by worker creation during parallel processing, which would invalidate the worst-case response time analysis.
- The driver is required to perform the splitting, and allocate the data partitions to each worker before the any parallel data processing occurs. Data splitting and partition allocations finishes before the processing as discussed in Section 3.3.
- The data splitting is required to be performed sequentially, as discussed in Section 3.3.
- The data partitions are pre-allocated to each worker according to the allocation policy. Work-stealing is not allowed. The reason of using a static allocation is given at the end of this section.
- Each worker takes data partitions from its allocations using FIFO order immediately, once the data partition allocations for all workers are finished. Once a data partition is acquired, the worker processes this partition immediately with the pipeline, then takes another partition immediately after the current processing finishes. When a worker finishes all the allocated partitions, it is suspended or sleeps.
- The sequential code after the parallel processing executes immediately after all the parallel data processing is completed.
- Execution-time servers are required to be used to serve all the execution of the real-time batch stream processing task, including the execution of the sequential code before the processing, splitting and allocation, parallel processing, and the sequential code after the processing. Specifically:

  - Each worker is required to register to its corresponding execution-time server before its execution.
  - The driver (if is implemented as a schedulable instance) and the caller (if any) are required to register to the corresponding execution-time server before its execution.

  The reason for using execution-time servers is given at the end of this section. In addition, the start time of the execution-time server is the same as the stream processing task. However, for the worker processor (i.e., the processor that only executes the data processing), the first release of the server is delayed with the worst-case response time of sequential execution before the parallel processing. This enhances the schedulability of stream processing tasks (see Section 5.4).

- The given priority is the priority that will be assigned to the execution-time server, i.e., the priority at which the client is executing when the server has capacity. When the execution-time server has capacity, its client worker executes at the server's priority, otherwise, the worker is suspended or transferred to the background priority.

- All the involved schedulable instances in this infrastructure, including the caller (if any), are required to be configured with corresponding processor affinity settings, so that a fully-partitioned scheduling scheme is obtained.

- For hard real-time stream processing:

  - the deadline miss handler is required to be released when there is any deadline miss occurs;
  - the MIT violation handler is required to be released when there is any two invocations of the *Process Batch* interface within the period.

Note that, in the case where there is no caller, for example, the release of real-time batched data processing is controlled by a hardware timer, the driver is required be implemented as a schedulable instance.

Once the *Process Batch* is invoked with a batch as the input, the processing is performed under the coordination of the driver as illustrated by Figure 3.8. The purpose of this figure is not for precisely describing the implementation,

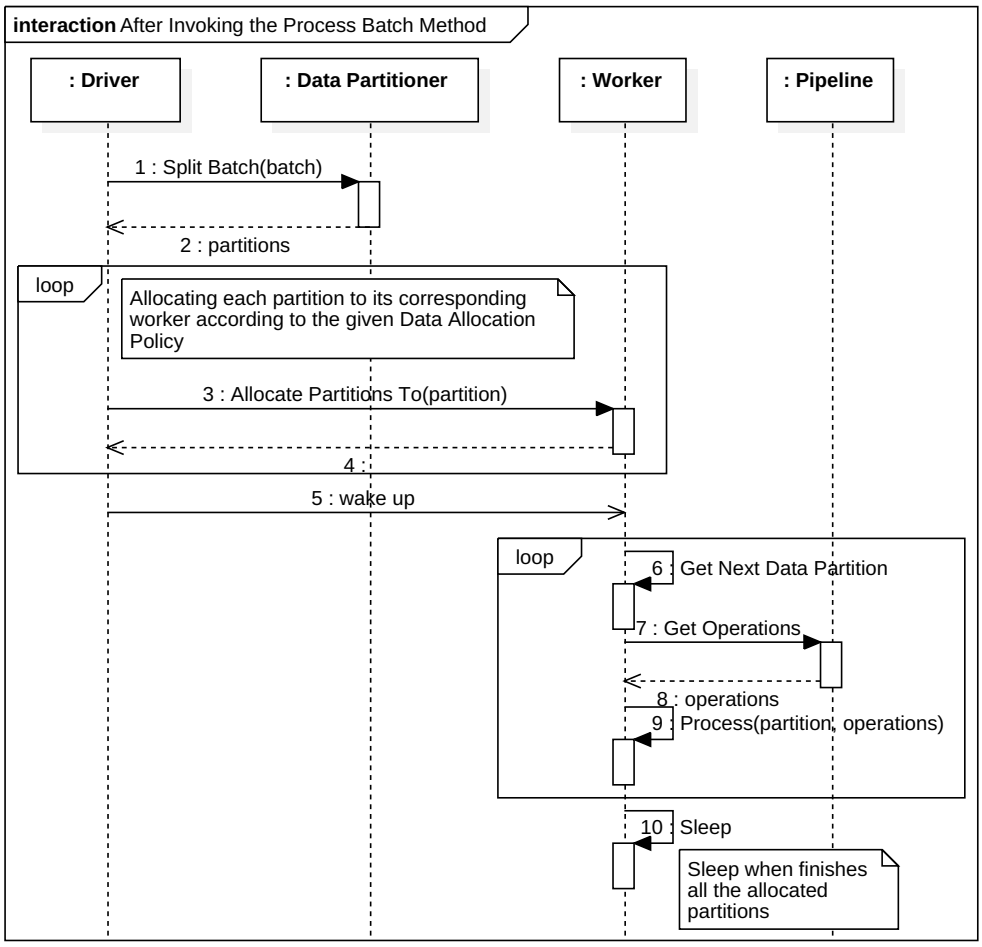


Figure 3.8: The behaviour of the driver after the *Process Batch* is invoked with a batch.

but demonstrating the overall execution of the driver. The driver first splits the input batch into partitions, then allocates each partition to different workers according to the data allocation policy. Then the workers are woke up and start to process allocated partitions with the pipeline. When a worker finishes its processing, it is suspended or sleeps until the next release of the stream processing task.

The Role of Execution-Time Servers

Typically stream processing is computationally-intensive. Additionally, when the stream processing task is soft real-time, the unpredictability of data volumes makes the corresponding CPU demand unpredictable. In any case,

streams are required to be processed within their hard or soft deadlines, whilst the hard real-time activities in the same system must remain schedulable.

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.

This thesis proposes that real-time stream processing can be executed under execution-time servers, so that the stream processing can meet its time constraints, while maintaining the existing guarantees for the hard real-time components.

Pre-Allocation of Data Partitions

Considering performing the worst-case response time analysis on the stream processing, it is observed that in a data-parallel model, the data partitions are required to be allocated to different processors with a static allocation approach, instead of using a dynamic work-stealing algorithm, in order to perform a sufficient worst-case response time analysis (RTA).

Performing timing analysis on the execution with a work-stealing algorithm is difficult, because the execution is dynamically determined by the work-stealing. Additionally, when using RTA, the worst-case situation can be too pessimistic.

For example, there are two processors: $Proc_1$ and $Proc_2$, but $Proc_2$ has a quite small computation capacity compared to $Proc_1$. The data splits into 4 parts: p_1 to p_4. Initially, each processor takes one partition, e.g., $Proc_1$ gets p_1 and $Proc_2$ gets p_2. Each processor takes another partition after finishing its current processing, according to the work-stealing algorithm. The worst case for $Proc_1$ is that it executes p_1, p_3, and p_4, while the worst case for $Proc_2$ is that it executes p_2, p_3, and p_4. However, these two situations cannot both occur, therefore introducing pessimism. For a larger number of processors and workloads the pessimism would be too great.

3.4.2 Supporting The Real-Time Live Streaming Data Processing

The sporadic real-time stream processing task model described in Section 3.3 implements real-time batched data processing. This section extends the sup-

port for batched data to live streaming data using real-time micro-batching.

The data items arrive sporadically from a live streaming data source. With the real-time micro-batching approach, a live streaming data source can be mapped to a sequence of micro batches, which are generated sporadically. Therefore, the processing of the live streaming data source can be transferred to an instance of the real-time stream processing task model, which was proposed in Section 3.3. This also enables the response time analysis equations derived for the real-time batch stream processing to be reused.

The real-time micro-batching approach described in this section allows each data item in a data flow to meet its processing latency requirement, whilst the hard real-time tasks in the same system remain schedulable. The details of the configuration of the real-time micro-batching approach is given in Chapter 4.

Real-Time Micro-Batching

In order to meet the latency requirement for each data item in a data flow, when using real-time micro-batching the size of each micro batch is determined by two factors:

- *Time* – Individual data items of the live streaming data source 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.

- *Input data volume* – Incoming data is buffered up to an application-defined maximum amount and once the buffer is full the batch is processed.

Architecture Specifications

This section proposes an architecture that supports the real-time micro-batching approach, this architecture is named *batcher*, and is illustrated by Figure 3.9. The batcher is a subsystem, an instance of which maintains the following components:

- A *Buffer* – which is used to store the incoming data items from a live streaming data source. The buffer has an application-defined maximum

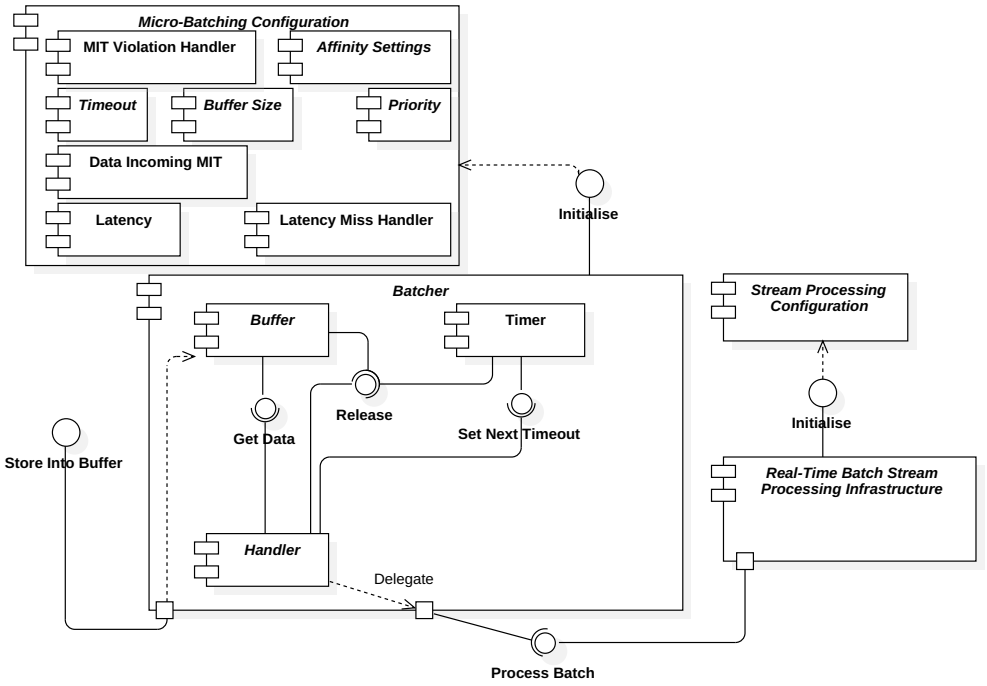


Figure 3.9: The real-time live streaming data processing infrastructure component diagram.

size. The buffer provides the *Get Data* interface, which retrieves all the data out of the buffer.

- A *Timer* – which maintains the timeout, releases the handler when the timeout expired. Additionally, it allows the next timeout to be set via the *Set Next Timeout* interface.

- A *Handler* – which is released via its *Release* interface when either the timeout expired, or the buffer reaches the maximum size. Once the handler is released, it turns the data items in the buffer to a micro batch, e.g., a collection, then invokes the *Process Batch* interface provided by the real-time batch stream processing infrastructure to process the micro batch.

- An *Interface* – named *Store Into Buffer*, which is implemented by the buffer. It allows the data item to be stored into the buffer, when the buffer is full, the buffer releases the handler.

To initialise a real-time micro batching instance using the batcher, the parameters below are required:

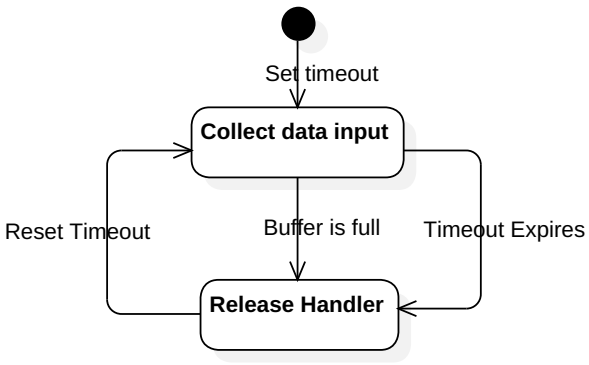


Figure 3.10: The Real-Time Micro-Batching approach.

- *Priority* – the priority that will be assigned to the execution-time server, i.e., the priority at which the handler is executing when the server has capacity.
- *Handler Affinity* – indicates to which processor the handler is assigned.
- *Timeout* – specifies the timeout interval for the timer.
- *Buffer Size* – defines the maximum buffer size.

For any hard real-time usage, the following additional parameters are required:

- *Data Incoming MIT* – describes the possible MIT of the incoming data items.
- *MIT Violation Handler* – the handler, which is released when any two data items arrive within the MIT.
- *Latency* – the latency for the processing of each data item.
- *Latency Miss Handler* – the latency miss handler.

The execution-time server parameters, priority, the maximum buffer size and timeout value are determined in Section 4.4. The affinity settings are determined in the following requirements.

Implementation Requirements

An implementation of the batcher is subject to the following requirements:

- The handler is required to be implemented as a schedulable instance to handle the timer's timeout event and the buffer is full event, therefore decoupling the batcher subsystem and real-time batch stream processing infrastructure subsystem.
- The handler is required to be created before the batcher starts executing, to avoid introducing any delay in the real-time micro-batching.
- The handler is allocated to the processor according to the application configurations. Processor affinities are used to forbid any migration to support the fully-partitioned scheme.
- The handler is required to register to the corresponding execution-time server, so that it makes the execution of the handler's functionality part of an instance of the proposed real-time stream processing task model. When the execution-time server has capacity, the handler executes at the server's priority, otherwise, the handler is suspended or transferred to the background priority.
- The priority of the batcher must be the same as the priority given to the real-time batch stream processing infrastructure, so that the handler executes at the same priority as the real-time stream processing task's priority. The reason for this is that the handler is a part of a real-time stream processing task.
- The release and the behaviour of the handler is illustrated Figure 3.10 by and described below,
  - The interface *Store Into Buffer* provides the functionality that stores the data item into the buffer, once the buffer is full, the handler is required to be released immediately.
  - The timer maintains the next timeout, once the timeout expired, the handler is released immediately.
  - Once the handler is running after release, it retrieves all the data items from the buffer, and turns them into a splittable collection, i.e., a micro batch. Then invokes the real-time batch stream processing infrastructure immediately with the micro batch.
  - Reset the next timeout for the timer immediately when the handler is released.

- The data items are required to be stored into the buffer once they arrived at the system. This is assumed by the configuration approach and worst-case processing latency analysis described in Chapter 4 and 5.

- For any stream processing task with hard real-time constraints:

  - the latency miss handler is required to be released when there is any data processing misses the latency requirement;
  - the MIT violation handler is required to be released when there is any two data of the *Process Batch* arrives within the MIT;
  - the deadline of the real-time batch stream processing infrastructure equals to the minimum possible inter-arrival time of the releases of micro batches, and the corresponding deadline miss handler is required to be given to the batch stream processing infrastructure.

  Note that, the invocation MIT and the invocation MIT violation handler of the batch stream processing infrastructure are not required. The reason is that the data incoming is monitored by the `Batcher`, if the data incoming MIT violation does not occur, any two micro batches cannot be released within the invocation MIT of the batch stream processing infrastructure.

3.5 Summary

This chapter first described the goals and philosophy of the proposed real-time stream processing system. This is followed by the system context described in Section 3.1, and the supported system models in Section 3.2. The proposed real-time stream task model was then introduced in Section 3.3. The proposed real-time stream processing task model employs a data parallel processing model, with sequential code executing before and after the parallel processing. From the point view of real-time literature, the whole real-time stream processing task uses a sporadic task model, i.e., the whole processing is released either periodically or sporadically with a minimum inter-arrival time.

This chapter then proposed an architecture for the real-time batch stream processing infrastructure that inputs a batched data source in Section 3.4.1, so that the sporadic real-time stream processing task model can be supported. In addition, the real-time micro-batching approach was proposed in Section 3.4.2

to support real-time live streaming data processing. The proposed real-time stream processing infrastructure enables both batched data and live streaming data sources to be processed within the deadline (or latency requirements), whilst guaranteeing that the hard real-time tasks in the same system will meet their deadlines. The implementation requirements of the architecture have also been described in these two sections.

The proposed architecture for the real-time stream processing infrastructure is concerned primarily with UMA architectures, and fully-partitioned systems. However, the underlying approach is also appropriate for NUMA architectures, and globally scheduled or semi-partitioned systems can also be supported with affinity settings.

The major difficulty is how to configure the instance of the proposed real-time stream processing infrastructure in a real-time system, so that the batched data can be processed within its deadline (or each data item is processed within the latency requirements in a live streaming data source), whilst guaranteeing that the hard real-time tasks remain schedulable. These challenges will be addressed in Chapter 4 that describes how a real-time stream processing task is configured, with the response time analysis derived in Chapter 5 that guarantees that the deadlines of the stream processing, or the latency requirements of the data items within the live streaming data source are met.

Chapter 6 describes a prototype implementation of the specification called The York Real-Time Stream Processing Framework, or SPRY. SPRY uses Java 8 Streams, in conjunction with the Real-time Specification for Java (RTSJ).

Chapter 4

Scheduling and Integration

A real-time stream processing task model has been described in Chapter 3, along with an architecture of the presented real-time stream processing infrastructure to support that model. This chapter addresses the issue of how to integrate a real-time stream processing activity into a system that also has hard real-time tasks.

This chapter assumes that the logical software structure of the system has already been developed and this has resulted in a set of tasks whose basic real-time characteristics (e.g. worst-case execution times) are known.

This chapter focusses on how this architecture is mapped to the physical platform, and how the real-time stream processing activity is configured so that the data can be processed within the deadline, whilst guaranteeing that the hard real-time tasks in the same system will meet their deadlines.

The approach consists of two top level activities:

- Allocation of tasks (including the real-time stream processing task).
- Configuration and analysis of the real-time stream processing task so that
  - for a batched data source, the batch can be processed within its deadline and the worst-case response time can be analysed;
  - for a live streaming data source, each data item can be processed within the latency requirements, and the worst-case latency can be analysed.

The analysis to be used on such configurations is described in Chapter 5. The chapter aims to achieve the best performance for both hard and soft tasks

with the available platform, rather than to minimise the resource use of the platform.

This chapter is structured as follows. This chapter first states some assumptions and introduces the notations used in (Section 4.1). This is followed by the description of task allocation in Section 4.2. Section 4.3 explains how to configure a real-time stream processing task that inputs from a batched data source so that the data can be processed within the deadline, while the hard real-time activities remain schedulable. This section also describes the proposed server parameter selection algorithm, and data allocation policy. Section 4.4 describes how to determine the maximum micro batch size, and the timeout value of a real-time micro-batching instance for a live streaming data source (i.e., the *Batcher* proposed in Section 3.4.2), so that each data item can be processed within the latency requirements while the whole system is schedulable. Finally, Section 4.5 summarises the chapter and discusses the overall approach.

4.1 Assumptions and Notations

This section describes the assumptions that this work is based on, and introduces the notations used in this chapter.

4.1.1 Assumptions

This work is based on the following assumptions as claimed in the system model described in Section 3.2:

- The system is fully-partitioned and scheduled pre-emptively using fixed priorities.
- Hard real-time tasks arrive either periodically with a fixed interval of time, or sporadically within a minimum inter-arrival time. Hence we support the sporadic task model common in the real-time scheduling literature.
- No software resources are shared (i.e., no synchronisation or mutual exclusion) between parallel data processing tasks.
- The hard real-time tasks have deadlines less than or equal to their minimum inter-arrival times, because this is the most common scheme in the sporadic task model [38].

- The I/O interrupt handlers in the same system are modelled as high-priority sporadic real-time tasks. These tasks will not affect our real-time stream processing task model, therefore their schedulability analysis is independent of the analysis of our real-time stream processing tasks. More complicated I/O interrupt handling is subject to future work.

4.1.2 Notation

The notation used in scheduling and configuration of the proposed real-time stream processing system, along with the notation for both real-time tasks and execution-time servers, and the real-time stream processing tasks are described as follows.

- A task is represented by τ_i, with a unique priority i, and has its relative deadline D_i, worst-case execution time C_i, and period T_i.
- The worst-case response time of the task τ_i is R_i, which is the longest time from when the task arrives to when it completes its execution.
- Given a batched data source, its processing is periodic, or sporadic with a minimum inter-arrival time. The real-time stream processing task that inputs from batched data has a unique priority i, a deadline D_i, a worst-case execution time C_i for executing all the code including data processing, and a period T_i.
- Given a live streaming data source, the items arrive sporadically with a minimum inter-arrival time (MIT^{item}), the worst-case execution time for processing each item is C^{item}, and the deadline, i.e., the latency, for processing each item is D^{item}.
- An execution-time server has a unique priority S, a capacity C_S, and replenishment period T_S.
- Ignoring interference from higher priority activities, if a server's capacity is replenished at time t, there will be a gap from t until the point in time when its capacity can begin to be consumed by the arrival of the served task. The difference between the maximum and minimum possible values for this is a server's *jitter* – J_S.
- When a periodic task is executed under a server, the task is defined as *bound* when the task's period is an exact multiple of its server's period

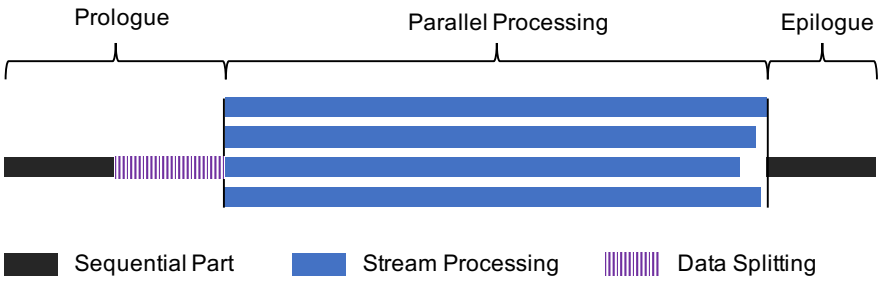


Figure 4.1: The structure of the stream processing task.

and each release of a bound task coincides with each replenishment of the capacity of the server [47].

4.2 Allocation of Tasks

In a fixed-priority pre-emptive fully partitioned system, task allocation is an NP-Hard problem [38]. Several heuristics have been proposed for task allocation and many of these are summarised by Davis and Burns [49]. A simple algorithm is to use 'best-fit' to allocate task into processors, then for each processor to use deadline monotonic priority assignment for tasks. Typically the goal of such an allocation strategy is to reduce the required number of processors, while the 'worst-fit' allocates tasks into processors more evenly.

However, the overall approach taken in this chapter is independent of the heuristic used. All that is required is an allocation of the hard real-time tasks that is schedulable.

4.2.1 Real-Time Stream Processing Task Model for Analysis

In order to exploit the spare capacity of the physical platform, the stream processing task contains multiple threads of control executing in parallel as described in Section 3.3.

However, from the point view of the scheduling and the schedulability analysis, the structure of the stream processing task can be simply illustrated by Figure 4.1.

The execution of a stream processing task contains the following three phases:

- Prologue: Sequential initialisation occurs, followed by the splitting of the

batch into partitions and the allocation of partitions to parallel threads for processing. In many stream processing systems the splitting itself can occur in parallel, for example, the parallel splitting of Java 8 streams (see Section 2.3.7). For real-time systems, more predictable splitting can be obtained by doing the splitting sequentially as discussed in Section 3.3.

- Processing: The data partitions are processed according to the application's requirements.

- Epilogue: The results of the processing are combined and reduced if necessary. For simplicity, this phase is assumed to be performed sequentially by the same processor that executes the prologue.

Note that, when the epilogue is executed by another processor, the analysis response time (described in Section 5.5) for the epilogue is required to use the execution-time server running on that processor. Additionally, the current server generation algorithm (see Section 4.3.1) examines every possible data processing window (i.e., the time interval between when the prologue finishes its execution, and the latest time when the epilogue has to start its execution), and finds the maximum possible computation time that can be guaranteed within the data processing window, from all the processors. When the prologue and epilogue execute in the same processor, the data processing window is determined by the server used in this processor. However, if they execute in different processors, the data processing window is determined by two servers: the server that executes the prologue, and the server that executes the epilogue. This requires the server generation algorithm to check every combination of those two servers, to find out the maximum computation time that can be guaranteed by examining every possible data processing window.

For the schedulability analysis, the real-time stream processing task is considered to be *periodically* released, although it is a sporadic model (defined in Section 3.3), i.e., it can be released either periodically or sporadically. This is because that from the point view of scheduling and providing schedulability guarantees for both hard real-time tasks and the stream processing task, it is required to consider the worst-case. Therefore, the sporadic stream processing task is treated as periodic, with a period equals to the possible minimum inter-arrival time.

Similarly, when processing a live streaming data source using the proposed

real-time micro batching approach (described in Section 3.4.2), the processing of micro batches, i.e., the release of the corresponding real-time stream processing task, is also released periodically in the worst-case. As the data items arrive continuously with MIT^{item} in the worst-case, the micro batch is released periodically, therefore, the real-time stream processing task is periodically released.

4.3 Configuration and Analysis of the Real-Time Stream Processing Task for Batched Data

The architecture that has been proposed in Chapter 3 supports real-time stream processing for both batched data and live streaming data sources. This section focuses on the real-time stream processing for a batched data source.

The approach given in Section 3.4.1 is to use execution-time servers to perform the stream processing to meet the deadline, whilst bounding the impact of the processing so that the hard real-time tasks in the same system remain schedulable. This is turn will influence the selection of the execution-time server parameters.

This section defines an approach to configure and analyse a real-time stream processing task that inputs from a batched data source, to achieve the goal that the real-time stream processing activity has enough computation resources to complete its processing in order to meet its deadline, while the hard real-time tasks remain schedulable.

The approach explains how to generate execution-time servers, and determines the data allocation policy for a real-time stream processing task, and test its schedulability. More specifically, the execution-time server generation algorithm (described in Section 4.3.1) selects the priority, period, and capacity for each server on each processor. In order to instantiate a real-time stream processing task for a batched data source using the *real-time batch stream processing infrastructure* proposed in Section 3.4.1, these parameters and the data allocation policy are required to be used for its configuration. The proposed approach is described as follows.

Given a real-time stream processing task τ_i for a batched data source, with period T_i, and deadline D_i:

1. Generate execution-time servers for each processor using the algorithm described in Section 4.3.1.

2. Perform the data partitioning, and allocate the data partitions to each processor, more specifically, the generated execution-time servers for each processor, using the approach described in Section 4.3.2.

3. Analyse the worst-case response time of τ_i, i.e., R_i, using the analysis equations derived in Section 5.4.

The real-time stream processing task τ_i is schedulable when its worst-case response time is within its deadline, i.e., $R_i \leq D_i$. Otherwise, it is not schedulable.

4.3.1 Server Parameter Selection

This section describes how to generate execution-time servers to execute the real-time stream processing task. The real-time stream processing task is required to meet its deadline (in our case, its period) when being executed under the server. All the other hard real-time periodic or sporadic activities in the system must also remain schedulable. We make the real-time stream processing task a bound task in order to enhance its schedulability and to reduce the server capacity requirements [46].

Overall Principle

With the approach adopted by this thesis, the data can only be processed after the prologue completes its execution, and the epilogue has to finish its execution before D_i. The length of the data processing window between the prologue and epilogue determines how long the remaining processors can perform their data processing.

This data processing window can be illustrated by Figure 4.2. The data processing window is the time window between the response time of the prologue, i.e., $R_{Prologue}$ in the figure, and the latest time when the epilogue has to start its execution to meet the deadline. To determine the latest time when the epilogue starts we consider three cases, as shown in Figure 4.3:

1. Only the epilogue executes during its last server period, as shown in Figure 4.3a.
   In this case, the latest time t when the epilogue has to start its execution is when the server's last period starts. If each processor completes its

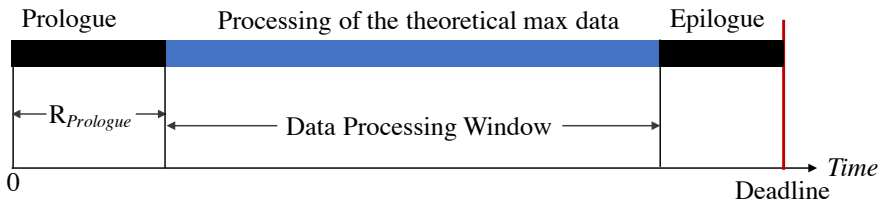
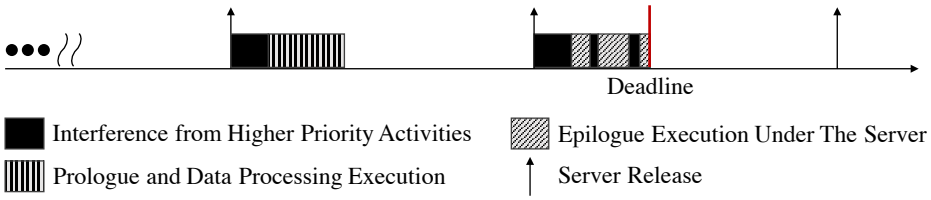
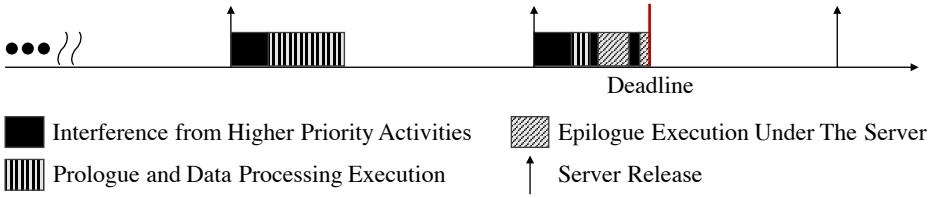


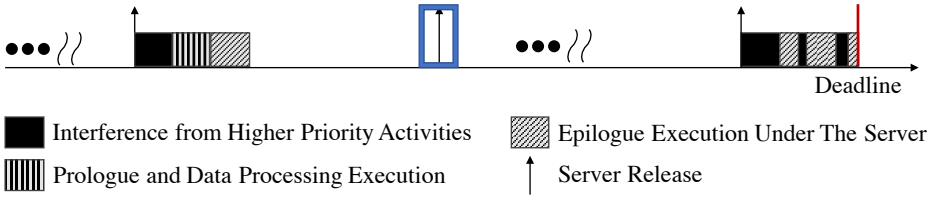
Figure 4.2: The data processing window shown in the prologue processor. The data of a real-time stream processing task can only be processed within this data processing window. The solid box represents execution-time server running, but the preemption or server capacity replenishment is not shown.



(a) Only the epilogue executing at the last server period.



(b) Part of the prologue and data processing, plus the epilogue executing at the last server period.



(c) Part of the prologue and data processing, plus the epilogue executing shares a server period, and the epilogue executes at more server periods.

Figure 4.3: The latest time when the epilogue starts.

data processing phase before t, the epilogue will meet the deadline. Introducing a *bound* task with the WCET of $C_{Epilogue}$, and calculates the worst-case response time R of this task using the techniques presented in Section 5.2. Then $t = Deadline - R$.

2. The epilogue finishes at its last server period, but part of the data processing (this might also including the prologue execution) also executes within this server period, as illustrated by Figure 4.3b.
   In this case, the latest time t is when the epilogue starts its execution, then receives as much as interference as possible, and just meet the deadline. t can be calculated using the approach presented in the previous case.

3. The epilogue executes within several server periods, and it shares a server period with the data processing (might also including prologue execution), as illustrated by Figure 4.3c.
   In this case, the first part of the epilogue has to finish by the time t' when the server is replenished just after the epilogue starts, as shown in the rectangle in Figure 4.3c.

   Given current server S, and the maximum computation time that can be guaranteed before the deadline, C_{MAX}, can be calculated using a binary search with analysis techniques presented in Section 5.2. Then, the WCET of the part of the epilogue in server's last period, $C^{Last}_{Epilogue}$, can be calculated by:

$$C^{Last}_{Epilogue} = C_{MAX} - \left\lfloor \frac{Deadline}{T_S} \right\rfloor C_S$$

   The WCET of the first part of the epilogue, $C^{First}_{Epilogue}$, can be calculated as follows:

$$C^{First}_{Epilogue} = (C_{Epilogue} - C^{Last}_{Epilogue})\%C_S$$

   Then t' can be calculated by:

$$t' = \frac{C_{MAX} - (C_{Epilogue} - C^{First}_{Epilogue})}{C_S} T_S$$

   Finally, the latest time, t, when the epilogue has to start its execution is the time when the first part of the epilogue starts to execute, receive as much as interference as possible, and just finishes at t'. Using the similar approach described in the previous case, the worst-case response time, R, of the first part of the epilogue can be calculated, and $t = t' - R$.

The maximum length of the data processing windows is from the latest time when the prologue finishes to the latest time when the epilogue has to start its execution, i.e., $t - R_{Prologue}$.

After these time values have been determined, the server generation algorithm examines every possible data processing window to find the maximum possible computation time that can be guaranteed within the data processing window, from all the processors.

The size of the data processing window varies with the execution-time server used in the prologue processor, as the response time of the prologue can be different when using different servers (similar for the latest time when the epilogue has to start).

Therefore, the proposed algorithm first generates execution-time servers for the prologue processor. For each generated server, it then calculates the maximum data processing window length, then generates execute-time servers for the remaining processors so that the computation time that can be guaranteed from these processors are maximised. Finally, a combination of a prologue server with the corresponding servers in the remaining processors, that guarantees to deliver maximum possible computation time for the real-time stream processing task can be obtained.

Server Generation Algorithm

Given a real-time stream processing task τ_i, with period T_i, and deadline D_i, the following algorithm generates the servers that can deliver the maximum capacity within the stream processing task's deadline using bound servers. The reason is that this algorithm checks all the possible bound servers on each processor, it always returns the combination of servers which delivers the maximum possible computation time.

For the processor that executes the prologue, the proposed server parameter selection algorithm is given by Figure 4.4 using pseudo code. The intuition behind the algorithm is that, the algorithm first generates prologue servers with all the possible periods. For each prologue server, the data processing window can be calculated, the algorithm then generates a set of servers with all the possible periods for each of the remaining processors. Finally, the combination of servers for all the processors, which delivers the maximum possible computation time can be found.

Note that, there might be multiple possible combinations of servers in each

processor that can deliver the maximum computation time for a real-time stream processing task. Users can choose an arbitrary combination according to their preferences, for example, higher priority servers make the stream processing more responsive.

```
1 Max_C = 0; /* The maximum possible computation time that can be
    guaranteed from all the processors */
2 Result = {}; /* The corresponding servers on each processor */
3 Order the hard real-time activities using deadline monotonic priority
    assignment [30], and check schedulability;
4 Calculate exact divisors of T_i as the potential periods for the server;
5 forEach(period T_S in periods){
6   Create a server S with deadline D_S = T_S;
7   Find the base priority for the server S using deadline monotonic
      assignment;
8   Use a binary search between 0 and T_S to determine the maximum
      capacity C_S for S at its priority level with the system
      remaining schedulable;
9   Use the Max_C_From_All_Processors(Server S) subroutine to
      calculate the maximum possible computation time C_G^All that can
      be guaranteed for τ_i from all the processors, along with the
      corresponding servers;
10  if(Max_C < C_G^All){
11    Max_C = C_G^All;
12    Result = servers;
13  }
14 }
15 return Result;
```

Note that:

- In line 4, exact divisors ensure the server has maximum schedulability [46].
- In line 8, the schedulability of each real-time activity can be analysed using the techniques described in Section 5.1.
- In line 7, when S has the same deadline as another hard real-time activity at priority j, then S is required to be examined at both priority $j+1$ and $j-1$ to determine its maximum schedulable capacity in line 8. If S can deliver the same capacity when running $j+1$ or $j-1$, choose either one.
- In line 9, the subroutine is described in Figure 4.5. With an execution-time server S running on the prologue processor, this subroutine calculates the sum of the maximum possible computation time that can be guaranteed for the real-time stream processing task, from all processors. Additionally, the corresponding execution-time servers will also be recorded.

Figure 4.4: Server generation algorithm.

```
1 Max_C_From_All_Processors(Server S){
2   MaxFromAll = 0; /* Records the maximum possible computation time
        that can be guaranteed from all the processors, with S */
3   Servers = {}; /* Records the corresponding servers on each
        processor, including server S */
4   Calculate the maximum computation time C_G that can be guaranteed
        before D_i with S, employing the response time analysis
        equation that is described in Section 5.2;
5   Calculating the maximum possible data processing window
        (DataProcessingWindow), with the approach described in this
        section.
6   forEach(Processor P in all processors){
7     if(P is the prologue processor){
8       MaxFromAll += C_G;
9       Servers.add(S);
10    }
11    else{
12      MaxFromP = 0; /* Records the max C guaranteed from P */
13      ServerP = null; /* Records the corresponding servers on P */
14      Calculate exact divisors of T_i as the potential periods for
            the server;
15      forEach(period T_S' in periods){
16        Create a server S' with deadline D_S' = T_S';
17        Find the base priority for the server S' using deadline
              monotonic assignment;
18        Use a binary search between 0 and T_S' to determine the
              maximum capacity C_S' for S' at its priority level with
              the system remaining schedulable;
19        Calculate the maximum computation time C'_G that can be
              guaranteed with the data processing window with S',
              which is equivalent to determining the maximum
              computation time that can be guaranteed before a
              deadline of D' = DataProcessingWindow using the
              techniques described in Section 5.2;
20        if(C'_G > MaxFromP){
21          MaxFromP = C'_G;
22          ServerP = S';
23        }
24      }
25      MaxFromAll += MaxFromP;
26      Servers.add(ServerP);
27    }
28  }
29  return MaxFromAll and Servers;
30 }
```

Figure 4.5: The subroutine used by the server generation algorithm.

4.3.2 Pre-Allocation of Partitioned Data to Execution-Time Servers

As illustrated in Figure 4.1, the real-time stream processing task consists of the prologue, multiple processing threads, and the epilogue. The servers generated provide the processor resource for the execution of all these stages. As each server's capacity depends on the utilisation of the hard real-time tasks assigned to that processor, the processing threads do not progress at the same rate, hence the allocation of data partitions to each processor must be carefully managed in order to reduce the overall response time of the stream processing task.

In a system where one processor is heavily-loaded, spreading the processing load evenly between the servers will not minimise the overall response time of the real-time stream processing task. Furthermore, as discussed in Section 3.4.1, a dynamic allocation of data items to servers is not appropriate due to the pessimism.

As introduced in Section 4.2, the data partitions are processed by multiple schedulable instances running in different processors. In each processor, the data processing is served by one corresponding execution-time server, which is generated using the algorithm proposed in Section 4.3.1. In this section, we assume that each partition can be processed in isolation in each batched data source (or micro batches generated using the proposed real-time micro-batching).

After splitting, for each partition, the processor that it is dispatched to is determined using the following approach:

1. Find the processors, which are able to provide enough capacity for processing partition p. The capacity that can be provided for data processing can be obtained during performing the execution-time server generation algorithm, which is described in Section 4.3.1.

2. Calculate the time when the partition's processing can be completed in each processor found in Step 1, using the analysis techniques described in Section 5.4.

3. Allocate p to the processor, which is the earliest one to finish the processing of p. The partition p is not allocated to the processor which has the lowest utilisation because low utilisation does not necessarily guarantee processing can be completed earliest.

Note that, the failure in Step 1 when trying to allocate a partition means that the stream processing task with this data source is not scheduable, because the epilogue is not able to finish its execution before the stream processing task's deadline in the worst-case.

Supporting Micro-Batching for the Live Streaming Data

The data items in a live streaming data source are latency sensitive, therefore, the partition allocation order is dependent on their arrival order. Hence, it is required to order the partitions in a micro batch according to their arrival order before performing the above approach.

4.4 Configuration and Analysis of the Real-Time Stream Processing Task for Live Streaming Data

As indicated in Section 3.4.2, the proposed architecture supports real-time stream processing for a live streaming data source using real-time micro-batching. This section describes how to configure an instance of the *Batcher*, so that each data item in a live streaming data source can be processed within its latency requirements, and hard real-time tasks in the same system remain schedulable.

In order to optimise the processing of the data and to ensure the required response time for processing each data item is met, it is necessary to determine the maximum size of the micro batch for which the latency, L, of processing every item of data can be met.

Recall that, when using the real-time micro-batching, the processing of the live streaming data source is performed by a real-time stream processing task, as described in Section 3.4.2. Therefore, the latency of a data item in the micro batch depends on the period of the real-time stream processing task. The period of the real-time stream processing task for a live streaming data source itself will depend on the size of the micro batch. Hence it is necessary to examine various micro batch sizes to determine the maximum size that can be processed.

The proposed activities to determine the maximum micro batch size and timeout value can be illustrated in Figure 4.6, and are discussed in the following subsections.

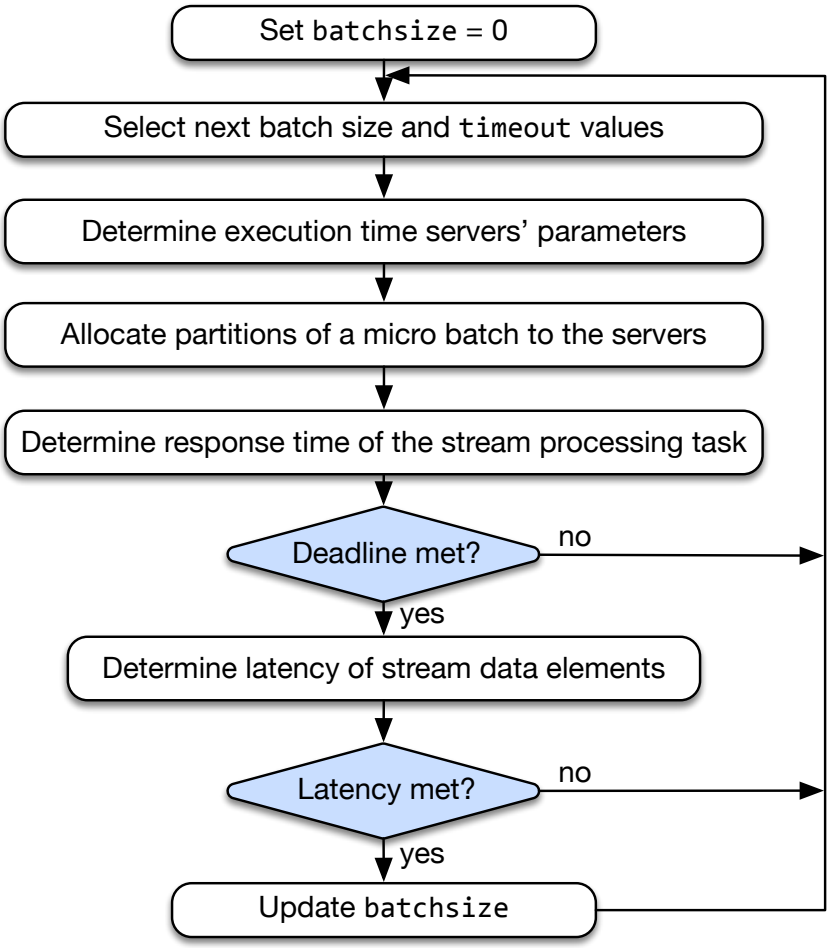


Figure 4.6: Configuring the stream processing task.

Considering the configuration of an instance of real-time micro-batching infrastructure for a live streaming data source. The buffer size and the timeout for the *Batcher* can be determined by the above approach. Recall that, the *real-time batch stream processing infrastructure* is used for processing each generated micro batch. Therefore, for the scheduling and configuring the processing of each micro batch, it uses the same approach proposed for the batched data source processing in Section 4.3. Again, the priority, period, and capacity for each server in each processor are determined using the execution-time server generation algorithm described in Section 4.3.1; while the data allocation policy is determined using the approach described in Section 4.3.2. Note that, the *Handler* is treated as a part of the prologue.

4.4.1 Determining Micro Batch Size and Timeout Value

The value of the micro batch size will be in the range of $1..\frac{L}{MIT^{item}}+1$. This is because the waiting time of the first data item in any batch that reaches that size will be L, therefore any processing of it will certainly miss its deadline. Hence the loop depicted in Figure 4.6 is bounded. We do not prescribe the exact algorithm for searching in the range, but for small ranges a simple `for` loop is sufficient. To check each batch size:

- Select an unchecked batch size. The period of the real-time stream processing task can be determined as $T = (n-1) \times MIT^{item}$. We use $n-1$ to account for the fact that an item may arrive at time 0.
- Given the allocation and the stream processing task's period, the execution-time servers (and their parameters) for each processor can be generated using the algorithm proposed in Section 4.3.1.
- The processing of each partition in the micro batch can then be allocated to a server using the approach proposed in Section 4.3.2.
- The schedulability of the stream processing task is then checked along with the latency of each data item in the micro batch using the techniques described in Section 5.4. The latency for any item is its waiting time plus its processing response time. With the worst-case data arrival, the waiting time for the x^{th} data item is $(n-x) \times MIT^{item}$, where $1 \leq x \leq n$. Therefore, the latency of this data item is $(n-x) \times MIT^{item}$ plus its processing response time. The live streaming data processing is schedulable if the following conditions can be met:
  - The latency of each item can meet the given time constraints.
  - The response time of the stream processing is less than or equal to its period.

Using this approach, the maximum micro batch size can be determined.

4.4.1.1 Determining the Timeout for Micro Batching

In a real system, data does not always arrive at its maximum allowable rate (MIT^{item}). Therefore the micro batch will not always be completely filled, and a timeout is required to release the micro batch early to avoid any deadline miss.

Given the maximum size for the micro batch is N, the worst-case is that the 1^{st} to $(N-1)^{th}$ data items arrive with the MIT^{item}, but the N^{th} item does not arrive at all. This is because all but one of the data items must still be processed and the first item has to wait the longest. This is almost identical to the batch with N data times for which we have already calculated the period, T, of the stream processing task in Subsection 4.4.1.

Employing T as the timeout, if the full batch is schedulable, all partially-filled micro batches are certainly schedulable for the following reasons:

1. It has at most N items, therefore the response time of processing this micro batch is no bigger than the response time of processing the full micro batch.

2. For any i^{th} item, where $1 \leq i \leq N$, the item is allocated to the same processor compared to the full micro batch (according to the allocation algorithm). Therefore, the item has the same processing response time.

3. For any i^{th} item, where $1 \leq i \leq N$, the waiting time of this item is less than or equal to the one in the full micro batch, as the data items do not always arrive with MIT^{item}. Therefore, the latency of this item will not be any larger.

Hence, given any schedulable maximum micro batch size N, then any micro batch that is released with a time out $T = (N-1) \times MIT^{item}$ is certainly schedulable.

Limited Buffer Size

The application might have limited buffer size, which is less than N, where N is the maximum possible micro batch size as calculated using above approach described in Section 4.4.1.

In this case, we can still employ the approach described in Section 4.4.1 to determine the maximum batch size, by replacing the upper bound of the loop, i.e., $\frac{L}{MIT^{item}} + 1$, with the application buffer size. Suppose that, after the loop stops, the maximum schedulable batch size is determined as B. In the worst-case, the period of processing micro batches is $T' = (B-1) \times MIT^{item}$.

However, in this case, the timeout value can be bigger than T' to increase the efficiency. Suppose the maximum timeout value is T''. For a micro batch is released because the timeout has expired, the worst-case situation is that: the

first $B-1$ items arrives from 0 with MIT^{item} as the inter-arrival time; while the last data item just arrives at time T''. By checking each timeout value in the range of $T'...T$, T'' can be determined, where $T = (N-1) \times MIT^{item}$.

Lastly, in the case where the data item arrives relatively slow, i.e., a half-full micro batch is released when the timeout is expired. This situation is also schedulable, the proof scheme described in Section 4.4.1.1 can be used to prove this.

4.5 Summary and Discussion

This chapter has presented the overall approach to the introduction of real-time stream processing tasks into real-time systems that also has hard real-time activities. Both batched and live streaming data sources are supported.

A key principle of the proposed approach is to use execution-time servers, in order that the real-time stream processing task can be executed so that the real-time requirements can be met, whilst maintaining the existing guarantees to hard real-time activities. As exhaustive server parameter selection has an exponential time complexity [46], an $O(n^3)$ execution-time server generation algorithm has also been proposed, where n is the number of tasks. However, it is still fast, for example, it takes 0.1 seconds to generate servers for the 4 cores multiprocessor system that has 128 hard real-time tasks. In addition, this chapter also proposed a data allocation approach that splits a batch into partitions, and allocates the partitions to difference execution-time servers regarding to their response time, therefore making the response time of the whole stream processing as short as possible.

The configuration of the real-time stream processing task for batched data source has been described in Section 4.3. The real-time live streaming data processing is supported by using the micro-batching approach. This chapter also has described how to configure the micro-batching approach for processing a live streaming data source in real-time systems in Section 4.4, so that each data time is processed within the latency reqirements.

4.5.1 Discussion

This chapter has made three main assumptions. The first is that the worst-case processing time of a batch's (or micro batch's) partition is not data sensitive. If it is, then the pre-allocation of partitions to servers might not be appropriate

and a more dynamic allocation might be required. This is the subject of future work.

Secondly, we have assumed that the deadline of the real-time stream processing task is equal to its period, which is equal to the worst-case response time needed to process a batched data source (or a micro batch generated from live streaming data sources), such that the data is processed within the deadline (or the individual data items meet their latency requirements). Our approach and analysis techniques allow the deadline of the stream processing task to be less than its period. This allows us to optimise the above approach to provide a more responsive system. For example, we can introduce an artificial deadline for the stream processing task, and slightly decrease that artificial deadline until the task can not be scheduled. Then the minimum possible worst-case response time can be obtained from the analysis of the task with the deadline just before the unschedulable artificial deadline. Similarly, for a live streaming data source, an artificial latency can be introduced to optimise the latency of each item when processing a live streaming data source.

Finally, this chapter targets a *single* stream processing task, however, the proposed approach can be extended to support multiple real-time streaming tasks. One possibility is to order all the stream processing tasks using deadline monotonic priority assignment, and schedule each real-time stream processing using the current approach. Note however that the current execution-time server generation algorithm (see Section 4.3.1) is a greedy algorithm, which searches maximum possible capacity for each candidate server in each processor. A particular stream processing task might not require the entire capacity generated by the algorithm. Therefore, it is required to return the extra capacity left for the current stream processing to the system. Details of this approach are the subject of future work.

Chapter 5

Schedulability Analysis

The previous chapter explained how to integrate real-time stream processing with hard real-time activities so as to not affect the timing properties of those activities. This chapter describes the response time analysis (RTA) for the real-time stream processing task, which is described in Section 3.3. From the point of view of analysis, the execution of the real-time stream processing task is shown in Figure 4.1, and is broken down into three parts. The prologue, which is sequential and prepares the parallel stream processing, the parallel stream processing itself, and the epilogue, which is sequential and executes on the same processor as the prologue.

Related RTA techniques are reviewed in Section 5.1. In addition, in order to analyse the worst-case response time of a real-time stream processing task, the RTA of a periodic/sporadic task which executes under an execution-time server is summarised in Section 5.2. Blocking introduced by accessing shared resources is analysed in Section 5.3, the RTA equations in above two sections are refined.

In this chapter, we use a deferrable execution-time server, due to the simplicity of its implementation. The worst-case response time analysis for a task executing under a deferrable server is a modified version of the original analysis [47].

This analysis is applied to the three components of the real-time stream processing task in Section 5.4 to determine the complete response time. The analysis targets a fixed-priority pre-emptive system with a fully-partitioned scheduling scheme. In addition, as discussed in the previous chapter, the I/O handling for a live streaming data source's collection is modelled as a sporadic hard real-time task, and its analysis can be done using RTA in Section 5.1.

Moreover, as the size of a batched data source can be so large that the I/O handling raises a long blocking time, the analysis that accommodates the blocking is given in Section 5.3.

An example of how to configure and perform schedulability analysis for a real-time stream processing task that inputs from a batched data source is given in Section 5.5, while a case study of real-time live streaming data processing, including configuration, worst-case latency analysis, etc., is given in Section 5.6. Finally, Section 5.7 summaries the contents of this chapter.

5.1 Worst-Case Response Time Analysis (RTA)

Given a real-time activity τ_i running at priority i, the worst-case response time can be calculated (without blocking) using the following equation [29]:

$$R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j \tag{5.1}$$

where $hp(i)$ is the set of activities of higher priority than i. J_j is the release jitter of the higher priority activity relative to τ_i, for a periodic or sporadic task $J_j = 0$. However, if the high priority task is a deferrable server, then it may have a release jitter due to the 'double hit' phenomenon, as explained in the following section.

5.1.1 The Double Hit

A deferrable server's capacity can be consumed at any time within its period. In certain situations, it may block a lower priority task for a long time interval - longer than the actual capacity of the server. For example, a served task may arrive exactly C_S time units before the replenishment point. It will then use the current full capacity plus another full capacity in the next period. This effect is called the *double hit* [34], as illustrated in Figure 5.1.

Therefore, this requires extra consideration when analysing the interference from a deferrable server on lower priority tasks. Bernat and Burns [34] accommodate the *double hit* problem in the RTA by introducing a jitter, i.e., $J_j = T_j - C_j$ in the equation when j is a deferrable server.

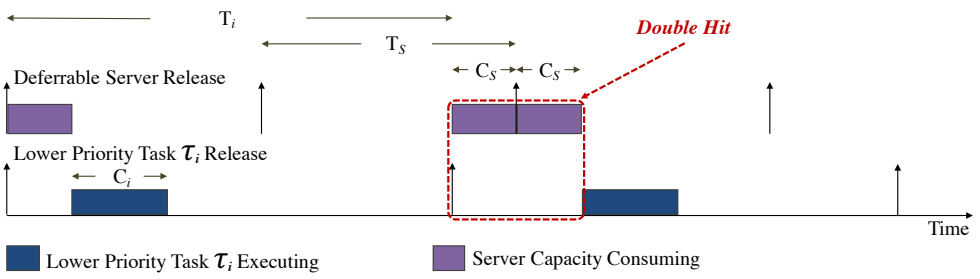


Figure 5.1: A task receives *double hit* from a higher priority deferrable server.

5.1.2 Refining the Double Hit Analysis

Bernat and Burns [34] treat the jitter as a constant that has to be applied to all lower priority tasks. However, this thesis observes that the *double hit* does not occur in every situation.

Theorem 1. *For any periodic activity that is released at the same time as a higher priority deferrable server, when calculating the interference from the server, double hits cannot occur if the server's period is an exact divisor of the current activity's period.*

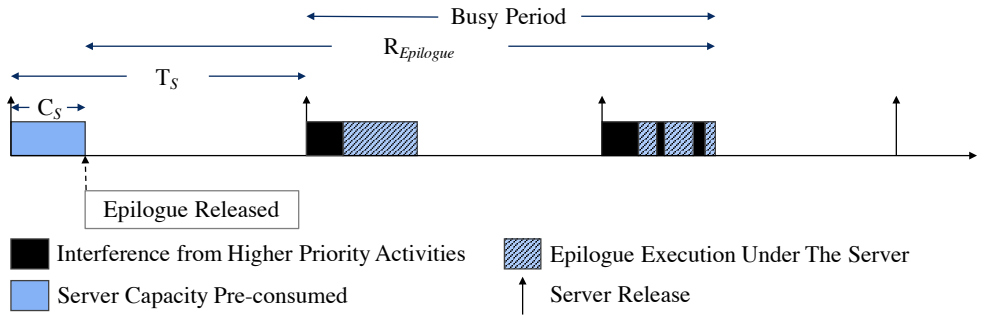
Proof. It has been proved by Bernat and Burns [34] that the worst-case response time for a lower priority task occurs when the *double hit* occurs, i.e., when the server S starts consuming its capacity exactly C_S time units before a replenishment occurs, and this instant is also the release time of the lower priority task. However, if the task is periodic, releasing together with the server that has a period of an exact divisor of the task's period, whenever the task is released it must be at the server's replenishment point. This violates the necessary condition for the *double hit*, therefore, *double hit* cannot occur. ■

Therefore, the equation is refined as follows.

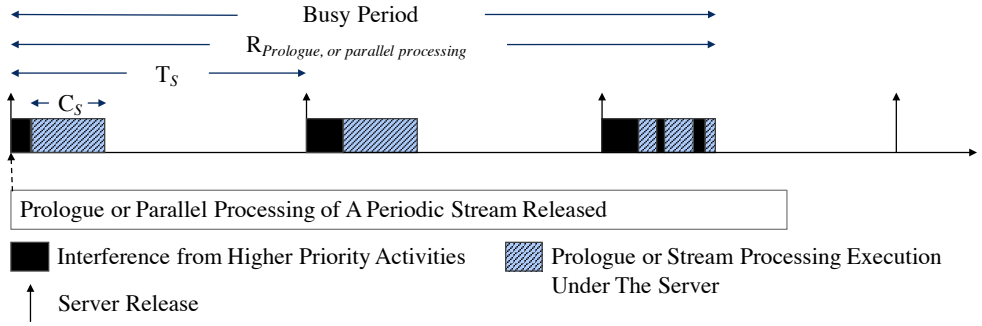
$$R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i + J_j^i}{T_j} \right\rceil C_j \tag{5.2}$$

where $hp(i)$ is the set of activities of higher priority than i. J_j^i is the release jitter of the higher priority relative to τ_i, for a periodic or sporadic task $J_j^i = 0$. For a deferrable server,

$$J_j^i = \begin{cases} 0, & \text{if } \tau_i \text{ is periodic, and } T_j \text{ is an exact} \\ & \text{divisor of } T_i \\ T_j - C_j, & \text{otherwise} \end{cases} \tag{5.3}$$



(a) Critical Instance for the epilogue task executing under a deferrable server.



(b) Critical Instance for a prologue or parallel processing of stream's data task executing under a deferrable server.

Figure 5.2: The critical instance and busy period.

Finally, applying the release jitter of the task itself, the worst-case response time of a task with release jitter is represented by $R_i^{Final} = R_i + J_i$. In general, for periodic activities $J_i = 0$ because they do not suffer release jitter [38].

5.2 RTA for a Task Executing under a Deferrable Server

The analysis presented in this chapter considers the scenario where a periodic or sporadic task is executed by a deferrable server at a unique priority, whilst all the other hard real-time activities run at their unique priorities. The execution of the task (e.g., the real-time stream processing task in a processor) can receive interference from both higher priority periodic or sporadic tasks.

When analysing the worst-case response time of a task that executes under a deferrable server, the critical instances for bounded tasks and unbounded tasks are different [47]:

- *For an unbounded client task, the critical instance occurs when:*

    1. The task is released at the time when the server's capacity is just consumed, and this consumption occurred as early as possible in that server's period. The epilogue is an example of an unbound task. The critical instance can be illustrated by Figure 5.2a. In this case, when the task requests capacity from the server, the server has none and therefore the task has to wait for the server's next replenishment.
    2. Once capacity of the task's server is replenished at the start of its next period, its consumption (i.e., the task's execution under this server) is delayed for as long as possible due to interference from higher priority activities.

- *For a bounded client task, the critical instance occurs when:*

    1. The task is released at the same time as the server is replenished; this is illustrated by Figure 5.2b. Due to the fact that the task is bound, it will be released at the same time as the server releases eventually. The prologue of a periodic real-time stream processing task is an example of a bound task.
    2. The task's execution under this server is delayed for as long as possible due to interference from higher priority activities.

The details and the worst-case response time for the whole real-time stream processing task is described in Section 5.4.

5.2.1 Analysis

Consider a periodic task τ_i executing under a deferrable server S. The original recursive equation for the worst-case response time analysis of task τ_i is given by Davis and Burns [47]:

$$w_i^{n+1} = L_i(w_i^n) + \left(\left\lceil \frac{L_i(w_i^n)}{C_S} \right\rceil - 1 \right) (T_S - C_S) + \sum_{\forall j \in hp(S)} \left\lceil \frac{max\left(0, w_i^n - \left(\left\lceil \frac{L_i(w_i^n)}{C_S} \right\rceil - 1 \right) T_S \right) + J_j}{T_j} \right\rceil C_j \tag{5.4}$$

where

$$L_i(w) = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w + J_j}{T_j} \right\rceil C_j \tag{5.5}$$

Recurrence starts with the value of:

$$w_i^0 = C_i + \left(\left\lceil \frac{C_i}{C_S} \right\rceil - 1 \right) (T_S - C_S)$$

The idea of this response time analysis uses the concept of *busy periods* and *loads* [47]. The Equation 5.4 consists of the following parts:

- The load:

  $$L_i(w)$$

  which is equals to the total length of execution-time server's execution.

- The total length of gaps in complete server periods, not including the final period:

  $$\left(\left\lceil \frac{L_i(w_i^n)}{C_S} \right\rceil - 1 \right) (T_S - C_S)$$

- The interference from higher priority activities in the server's final period:

  $$\sum_{\forall j \in hp(S)} \left\lceil \frac{max\left(0, w_i^n - \left(\left\lceil \frac{L_i(w_i^n)}{C_S} \right\rceil - 1 \right) T_S \right) + J_j}{T_j} \right\rceil C_j$$

Using a recurrence relation, the length of the busy period, i.e., w_i in Equation 5.4, can be solved. The length of the busy period is the task's response time (plus its release jitter when having one).

The equation makes two assumptions. Firstly it assumes that the server is executing more than one task. Secondly it assumes that the server may cause the 'double hit' phenomenon. In this thesis, there is the only task using the server, the load, i.e., $L_i(w)$ on the server's busy period is constantly:

$$L_i(w) = C_i \tag{5.6}$$

Then, taking into account the variation in the release jitter caused by the deferrable server, the final recursive equation can be simplified as follows:

$$\begin{aligned} w_i^{n+1} = {} & C_i + \left(\left\lceil \frac{C_i}{C_S} \right\rceil - 1 \right) (T_S - C_S) \\ & + \sum_{\forall j \in hp(S)} \left\lceil \frac{max\left(0, w_i^n - \left(\left\lceil \frac{C_i}{C_S} \right\rceil - 1 \right) T_S \right) + J_j^S}{T_j} \right\rceil C_j \end{aligned} \tag{5.7}$$

where, $hp(S)$ is the set of higher priority activities compared to server S, and J_j^S is the release jitter of the higher priority activity j. For a periodic or sporadic task or server, $J_j^S = 0$ [47]. Again, for a deferrable server:

$$J_j^S = \begin{cases} 0, & \text{if } T_j \text{ is an exact divisor of } T_S \\ T_j - C_j, & \text{otherwise} \end{cases} \tag{5.8}$$

The recurrence starts with the value of:

$$w_i^0 = C_i + \left(\left\lceil \frac{C_i}{C_S} \right\rceil - 1 \right) (T_S - C_S)$$

and ends either when $w_i^{n+1} = w_i^n$ or $w_i^{n+1} > D_i - J_i$. In the former case, the response time of the task is given by $w_i^{n+1} + J_i$. In the later case, the task is not schedulable. The jitter J_i is the release jitter of the task relative to the release of the server. It is zero for bound tasks, otherwise, it is $(T_S - C_S)$. Recall that, in general periodic activities do not suffer release jitter [38], therefore it is not considered in the final response time representation.

5.3 Blocking

The stream processing task might access a shared resource with other hard real-time tasks (which might be executed by an execution-time server) in the prologue or the epilogue. Moreover, the hard real-time tasks themselves might access shared resources. We assume the parallel processing does not access shared resource.

This section refines the analysis equations so that the blocking introduced by accessing a shared resource is accommodated. This section is mainly based on the work presented in [47]. However, the analysis equations provided by the original assumes all the tasks are executed by execution-time server. In addition, this section assumes:

1. There is a set of shared resources G. Any task τ_i might access a resource r, for at most an execution time $b_{r,i}$. The length of this critical section is less than the capacity of its server S, i.e., $b_{r,i} < C_S$. $b_{r,i}$ will be much smaller than C_S for a well-designed application [47].

2. While a task τ_i accesses a shared resource, then the priority of it (and its server if there is one) is increased to a ceiling priority, which is higher than any task (or server) that shares this resource with τ_i.

3. If a task is executed under a server, and that server's capacity is just exhausted when it accesses a resource, the server continues to execute this task at the ceiling priority until the critical section is completed.

4. The server's capacity in the next release is *not* reduced by the amount of the overrun, to reduce the implementation complexity.

5. Each server only executes a single task.

6. There are no nested resource access.

The longest time that server S might overrun is given by:

$$B_S^{Task} = \max_{\forall r \in G}(b_{r,i} | i \in S) \tag{5.9}$$

This value is the longest critical section within the task, which is executed by server S.

The longest time that a task (or a server executing a task) can be blocked due to lower priority activity executing at a priority higher than it (i.e., the ceiling priority) is given by:

$$B_i = \max_{\forall j \in lp(i)} (b_{r,j} | r \in global(i,j)) \tag{5.10}$$

where $global(i,j)$ represents the set of global resources shared between activity i or activities with a priority higher than i, and j.

5.3.1 RTA with Blocking

For the critical instance, the busy period of a task is increased due to the following factors:

- The longest blocking time (i.e. B_i) due to lower priority activity executing at a priority higher than it due to operations of the synchronisation protocol, e.g., the priority ceiling protocol.

- Each execution of each higher priority server j is increased by the longest time that server j might overrun, i.e., B_j^{Task}.

- If current activity is a server, then its client task overruns with B_i^{Task}.

Therefore, the RTA equation is refined as follows.

$$R_i = C_i + B_i^{Task} + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i + J_j^i}{T_j} \right\rceil \left(C_j + B_j^{Task} \right) \tag{5.11}$$

where $hp(i)$ is the set of activities of higher priority than i. J_j^i is the release jitter of the higher priority relative to τ_i, for a periodic or sporadic task $J_j^i = 0$. For a deferrable server,

$$J_j^i = \begin{cases} 0, & \text{if } \tau_i \text{ is periodic, and } T_j \text{ is an exact} \\ & \text{divisor of } T_i \\ T_j - C_j, & \text{otherwise} \end{cases} \tag{5.12}$$

$B_i^{Task} = 0$ if i is not a server, and $B_j^{Task} = 0$ if j is not a server.

5.3.2 RTA for a Task Executing under a Deferrable Server with Blocking

Similar to the approach presented in Section 5.3.1, the busy period of the task that executes under a server S is increased by:

- The longest blocking time (i.e. B_S) due to lower priority activity (compared to S) executing at a priority higher than it due to operations of the synchronisation protocol, e.g., the priority ceiling protocol.
- Each execution of each higher priority server j is increased by the longest time that server j might overrun, i.e., B_j^{Task}.

$$w_i^{n+1} = C_i + \left(\left\lceil \frac{C_i}{C_S} \right\rceil - 1 \right) (T_S - C_S) + B_S + \sum_{\forall j \in hp(S)} \left\lceil \frac{max\left(0, w_i^n - \left(\left\lceil \frac{C_i}{C_S} \right\rceil - 1 \right) T_S \right) + J_j^S}{T_j} \right\rceil \left(C_j + B_j^{Task} \right) \tag{5.13}$$

where, $hp(S)$ is the set of higher priority activities compared to server S, and J_j^S is the release jitter of the higher priority activity j. For a periodic or sporadic task or server, $J_j^S = 0$ [47]. Again, for a deferrable server:

$$J_j^S = \begin{cases} 0, & \text{if } T_j \text{ is an exact divisor of } T_S \\ T_j - C_j, & \text{otherwise} \end{cases} \tag{5.14}$$

Additionally, $B_j^{Task} = 0$ if j is not a server.

5.4 RTA for the Real-Time Stream Processing Task

In the previous chapter, servers have been generated with periods that are exact divisors of the real-time stream processing task's period.

Specifically, for a periodic real-time stream processing task, the prologue and the parallel processing in the worker processor are equivalent to *bound* tasks, from the point of the analysis. The reason is the release of the prologue coincides with each replenishment of the capacity of its server, because the server's period is an exact divisor of the whole stream processing's period.

Theorem 2. *For the release of the parallel processing in a worker processor, a release r that occurs earlier than the expected time t (i.e., the time when the previous release occurred plus T_{Stream}), is equivalent to a* bound *task, and the analysis is still valid.*

Proof. For the analysis, it is assumed that r arrives at t, and the server also replenishes at t. The worst-case response time of this release is R, the latest time when this parallel processing execution is completed is $t + R$. However, if r arrives at t', where $t' < t$. The worst-case scenario is where the parallel processing can not execute until time t, which indicates the latest time when the parallel processing is still $t + R$. Therefore, the analysis is still valid. ■

However, for the epilogue, the server's capacity may have been consumed by the parallel processing in this processor. Therefore, the critical instance for the epilogue is modelled as an *unbound* task. In addition, for the prologue or the parallel processing of a sporadic stream processing task, they are subject to *unbound* tasks in the analysis [47].

5.4.1 Analysis

The execution of a real-time stream processing task τ_i (as described in Section 3.3) can be divided into the following phases:

1. Sequential execution of the task before the data splitting (prologue),
2. Splitting (prologue) - the data splitting before its processing,
3. Processing - the parallel stream processing, and
4. Sequential after the parallel processing (epilogue).

The worst-case execution time of the phase 1, phase 2, and phase 4 are represented by C_i^1, C_i^2, and C_i^4. The The worst-case execution time required for processing an data partition in phase 3 is C_i^{item}.

Consider a periodic real-time stream processing task τ_i, which is executed by processor P_{τ_i}. The parallel processing uses processors P_0 to P_{n-1}, including P_{τ_i}, where $n \geq 1$. In a fully partitioned system, the prologue (phases 1 and 2) are performed on processor P_{τ_i}, then all the allocated processors are used for the parallel processing, and finally the epilogue is executed on processor P_{τ_i}. Phase 4 only starts after all parallel sections of phase 3 are complete.

In order to analyse the worst-case response time of the real-time stream processing task τ_i, we have to consider its entire execution. The prologue can be analysed as a whole because its constituent phases are executed sequentially by the same processor (in our current implementation we make this restriction). In addition, each of its releases coincides with the server's replenishment, i.e., *bound* task. Employing Equation 5.7 with jitter ($J_i = 0$), load ($C_i = C_i^1 + C_i^2$), and the generated server; the worst-case response time R_i^2 of executing the prologue can be calculated.

The parallel processing of the data partitions starts once the splitting is finished. According to the pre-allocation principles described in Section 4.3.2, we can calculate the worst-case execution time for the data partitions that were allocated to each processor. For example, n partitions were allocated to a processor P_i, then the worst-case execution time for data processing in P_i can be calculated by:

$$C_{P_i}^3 = C_i^{item} \times n$$

Then, the next step is to calculate the time when the parallel data processing in each processor completes:

- For processor P_{τ_i} the prologue and the allocated data processing can be treated as a whole. Therefore the worst-case response time $R_i^{3,P_{\tau_i}}$ of this whole execution in this processor can be calculated by using the Equation 5.7 with jitter = 0, load = $C_i^1 + C_i^2 + C_{P_{\tau_i}}^3$, and the generated server for this processor.

- For each of the other processors $P_i, P_i \neq P_{\tau_i}$, the processing is released at R_i^2. In addition, because τ_i is a *bound* task, therefore, the worst-case response time R_i^{3,P_i} for the stream processing in this processor can be

calculated using the Equation 5.7 with jitter = 0, load = $C^3_{P_i}$, and the generated server for this processor.

It is also necessary to consider the response time of each individual data item. For any data item, the processor that processes it and its processing order is determined by the pre-allocation algorithm described in Section 4.3.2. The response time of this item can be calculated by removing the workload of processing all items after this item in this processor and then repeating the above steps.

The response time of the parallel data processing phase is the maximum of all involved processors:

$$R^3_i = \max(R^{3,P_{\tau_i}}_i, \max(R^{3,P_i}_i + R^2_i)), where P_i \neq P_{\tau_i}$$

Finally we consider the epilogue of the real-time stream processing task τ_i (phase 4). The worst-case situation is that is when it is released (after the barrier synchronisation detailed above) the last of the current server's capacity has just been consumed. Therefore, the worst-case response time R^4_i for phase 4 can be calculated using the Equation 5.7 with the generated server S, jitter = $T_S - C_S$, and load = C^4_i.

Finally, the worst-case response time of τ_i is calculated by:

$$R_i = R^3_i + R^4_i$$

Note that, for a sporadic real-time stream processing task, the analysis of prologue, parallel processing and epilogue is subject to an *unbound* task.

5.4.2 Blocking

When a real-time stream processing task accesses shared resources in its prologue or epilogue, the above analysis is required to use the refined blocking RTAs presented in Section 5.3.

However, this thesis requires that a task should not lock a resource in the prologue, and release the lock in epilogue. The reason is that, the higher priority tasks typically have a short period, a long enough interval of resource locking will certainly result in a deadline miss for the higher priority task.

5.4.3 Mitigating Analysis Pessimism

The described analysis is a sufficient schedulability test, however, the analysis is pessimistic for the RTA of the *unbound* task, for example, the epilogue. The

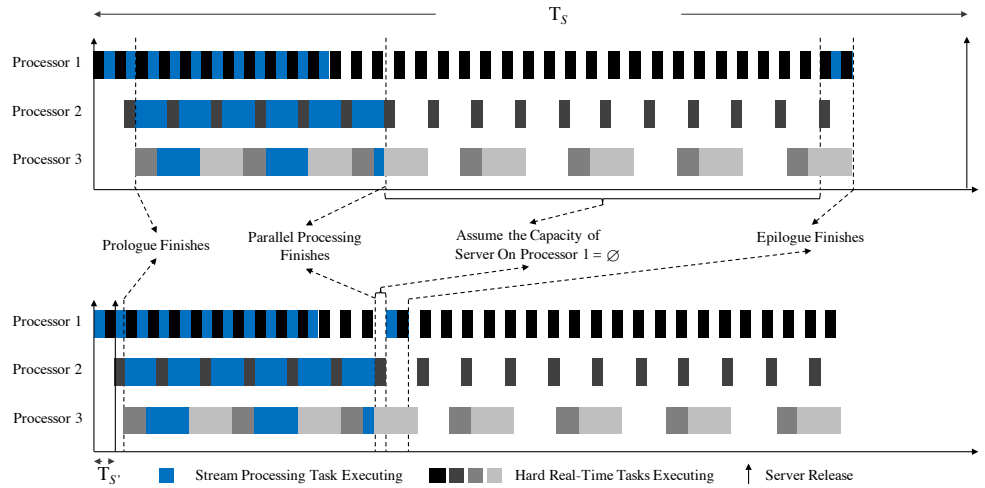


Figure 5.3: The pessimism of stream RTA for two different servers. The top chart shows a server with long period and the lower chart a server with shorter period and therefore less pessimism. Note that, only first two server's releases are shown in the figure.

analysed worst-case is that at the point of the epilogue's release its server has just been exhausted. The analysis therefore has to wait for server replenishment before processing time can be guaranteed. This waiting time is affected by the period of the server. For example, as illustrated in Figure 7.2, the gap is significant when the generated server has a longer period.

The solution to this problem, is to make the observation that when performing the RTA for the epilogue's processor, all candidate servers were already checked during the system's schedulability analysis.

Theorem 3. *If any candidate servers were observed to make the real-time stream processing task meet its deadline, then the servers generated by the algorithm are also guaranteed to make the stream processing task schedulable regardless of whether the pessimistic RTA of the epilogue fails to guarantee its deadline.*

Proof. From the point view of scheduling, if a candidate set of servers from different processors can make the real-time stream processing task meet its deadline, then any other server set that gives the same or more available computation time before the deadline of the stream processing task can also guarantee the task to meet its deadline. As the servers that were generated by our server generation algorithm can provide the maximum available com-

putation time before the deadline of the stream processing task, therefore, the stream processing task can certainly meet the deadline. ■

5.5 An Example of RTA for a Batch Real-Time Stream Processing Task

This section provides a fully-worked example of how to calculate the worst-case response time of a real-time stream processing task that inputs from a batched data source.

Consider a fully partitioned system that has 3 processors. There are 4 hard real-time periodic tasks, which are described in Table 5.1, and one periodic real-time stream processing task τ_i in this system.

The real-time stream processing task is released on processor P_0, with a period of 800 time units and a deadline of 780 time units after each of its releases. It can utilise all the processors for the parallel data processing. The worst-case execution time (WCET) for the sequential code before the data processing in τ_i is 18 time units, the splitting requires 1 time unit, and the sequential code after the parallel data processing is 11 time units. The data has 12 partitions, and the worst-case computation time required for processing each partition is 30 time units. Both the prologue and epilogue execute on processor P_0.

Table 5.1: Real-time Tasks Characteristics

Name	Priority	C	T	D	Processor
τ_1	11	10	20	20	P_0
τ_2	9	10	40	40	P_1
τ_3	5	20	100	50	P_2
τ_4	3	40	100	100	P_2

5.5.1 Execution-Time Server Generation for the Real-Time Stream Processing Task

Using the algorithm proposed in Section 4.3.1, considering that processor P_0 is the processor that executes the prologue and epilogue; the following servers that are given in Table 5.2 are examined, along with the maximum possible

computation time that can be guaranteed for the real-time stream processing task from all the processors.

Table 5.2: Possible Deferrable Servers For Processor P_0. DPW Represents the Data Processing Window.

Priority	Max C	T	DPW	Max C in DPW From All Processors
10	400.000	800	710.0	1190.0
10	200.000	400	710.0	1190.0
10	100.000	200	710.0	1190.0
10	80.000	160	710.0	1190.0
10	50.000	100	710.0	1190.0
10	40.000	80	710.0	1190.0
10	20.000	50	700.0	1090.0
10	20.000	40	710.0	1190.0
10	12.000	32	700.0	1070.0
10	10.000	25	695.0	1085.0
12	10.000	20	730.0	1210.0 (MAX)
10	10.000	20	710.0	1190.0
12	5.000	16	699.0	1024.0
12	5.000	10	725.0	1205.0
12	3.000	8	709.0	1092.0
12	2.500	5	722.5	1202.5
12	2.000	4	722.0	1202.0
12	1.000	2	721.0	1201.0
12	0.500	1	720.5	1200.5

Calculating the Maximum Possible Computation Time that can be Guaranteed from all Processors

For example, consider the candidate server S' in Table 5.2, which has a period of 800 time units, the maximum possible capacity can be determined to be 400 time units using binary search. Server S' can guarantee 390 time units computation time for the maximum, before the real-time stream processing task's deadline. Subtracting the computation time required for the prologue and epilogue, the maximum possible computation time that can be guaranteed

for data processing from P_0 is $390 - (18 + 1) - 11 = 360$ time units.

In addition, using server S', the response time of the prologue can be calculated to be 39 (see details of the calculation in the next subsection), the latest time when the epilogue has to start is calculated to be 749 time units, where 11 is the WCET of the epilogue. Therefore, the data processing window between the prologue and epilogue is $DataProcessingWindow = 749 - 39 = 710$ time units.

For the remaining processors, i.e., P_1 and P_2, the maximum possible computation time that can be guaranteed within the data processing window is 540, and 290 time units. For example, considering processor P_1, all the candidate servers, and corresponding maximum computation time that can be guaranteed using each server within the data processing window are given in Table 5.3. Note that, there might be multiple servers can provide the maximum computation time within the data processing window. The selection of servers is described in the following subsection.

Therefore, with S' running in the prologue processor, i.e., P_0, the total computation time that can be guaranteed from all the processors for the data processing is $360 + 540 + 290 = 1190$ time units.

In addition, a Java implementation of the server generation algorithm is available at [18].

The Selected Execution-Time Servers

As can be seen, when generating servers for each processor, there might be multiple servers that can guarantee the maximum computation time for the real-time processing task. In this example, we select the servers with a long period for each processor. This is done for efficiency, because the server with a long period requires fewer context switches. The selected servers for each processor are given in Table 5.4.

5.5.2 Calculating the Worst-Case Response Time

Firstly, we calculate the response time of the prologue. Employing Equation 5.7 with a $J_i = 0$, a load of $18 + 1 = 19$ (the execution time of the prologue thread), and the server S_0. The worst-case response time R_i^2 of executing phase 1 and 2 (i.e., the prologue) is calculated as the following equation. Note that, the server runs at the highest priority, therefore, there is no interference from hard real-time tasks.

Table 5.3: Possible Deferrable Servers for Processor P_1. DPW represents the Data Processing Window.

Priority	Max C	T	Max C in DPW
8	600.000	800	540.0
8	300.000	400	540.0
8	150.000	200	540.0
8	120.000	160	540.0
8	70.000	100	499.0
8	60.000	80	540.0
8	30.000	50	429.0
10	30.000	40	540.0
8	30.000	40	540.0
10	15.000	32	345.0
10	15.000	25	434.0
10	15.000	20	540.0
10	10.000	16	450.0
10	7.500	10	540.0
10	6.000	8	540.0
10	3.750	5	540.0
10	3.000	4	540.0
10	1.500	2	540.0
10	0.750	1	540.0

$$w = 19 + \left(\left\lceil \frac{19}{10} \right\rceil - 1 \right) (20 - 10) = 29$$

Therefore, the worst-case response time of the prologue is 29.

According to the principle proposed in Section 4.3.2: 3 partitions are allocated to processor P_0; 6 partitions are allocated to P_1; and 3 partitions are allocated to P_2.

Then we calculate the time when each processor finishes its processing of the allocated partitions:

- For processor P_0:

  The worst-case response time R_i^{3,P_0} of the whole execution of the pro-

Table 5.4: Selected Deferrable Servers

Name	Priority	C	T	D	Processor	U
S_0	12	10	20	20	P_0	0.500
S_1	10	30	40	40	P_1	0.750
S_2	6	20	50	50	P_2	0.400

logue and the data processing is calculated to be 209, by using the Equation 5.7 with the $J_i = 0$, a load of $18 + 1 + 3 \times 30 = 109$, and the server S_0.

- For processor P_1:
  The worst-case response time R_i^{3,P_1} for the data processing in this processor is 230, after calculating using the Equation 5.7 with the $J_i = 0$, a load of $6 \times 30 = 180$, and the server S_1.

- For processor P_2: The worst-case response time R_i^{3,P_2} for the data processing in this processor is 210, after calculating using the Equation 5.7 with the $J_i = 0$, a load of $3 \times 30 = 90$, and the server S_2.

Therefore, the parallel data processing finishes at time:

$$\begin{aligned} R_i^3 &= Max(209, Max(230 + 39, 210 + 39)) \\ &= 259 \end{aligned}$$

The last step is to calculate when the epilogue finishes its execution. The worst-case response time R_i^4 for phase 4 is calculated to be 31, using the Equation 5.7 with the $J_i = 20 - 10 = 10$, a load of 11, and the server S_0.

Finally, the worst-case response time of τ_i (the real-time stream processing task) is calculated: $R_i = 259 + 31 = 290$, therefore, the task is schedulable.

A visualisation of the execution this real-time stream processing task's in its worst-case, along with all the hard real-time tasks in the system can be illustrated by Figure 5.4. As can be seen, the prologue finishes at 29 ms, the parallel data processing finishes at 259 ms, and the real-time stream processing task finishes at 290 ms. The gap between 259 ms and 269 ms represents the pessimism that occurs from assuming the capacity of server S_0 to be zero at the end of data processing.

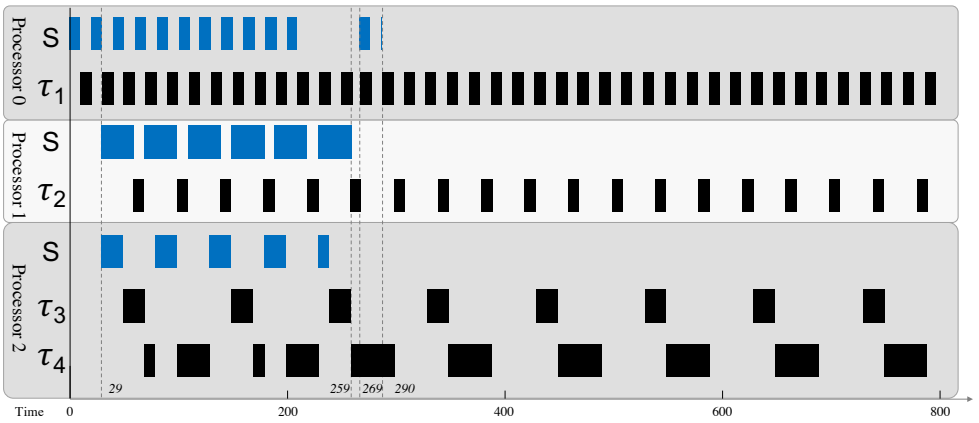


Figure 5.4: The worst-case execution of the stream processing. S represents the real-time stream processing task.

5.6 A Case Study of Real-Time Live Streaming Data Processing

This section describes the thesis's motivating case study of an aircraft which hosts several hard real-time tasks for its defence system, but that also has to process live streaming data sources in real-time to meet a time constraint.

5.6.1 Overview

As introduced in Section 1.1.2, this case study considers an aircraft, which is equipped with a synthetic aperture radar (SAR). This aircraft has a mission to generate images of a series of target areas using SAR, whilst its defence systems aim to guarantee its safety during the flight at a maximum speed of 2160 km/h [1], as shown in Figure 5.5 (a replication of Figure 1.2 for convenience of presentation). The minimum distance between any two target areas is 15 meters.

Each image of a target area must be generated within 480 ms after the echoes return. In order to meet the resolution requirement of the imagery of a target area, the worst-case execution time of generating an image from the raw echoes is 40 ms. Specifically, the SAR uses the spotlight imaging mode [75]. In this mode, the radar beam is steered as the aircraft moves, so that it illuminates the same target area over a period of time, as illustrated by

[1] This speed is chosen for the simplification of the calculation, and it is also close to the super-cruise speed of the F-22 Raptor [78]

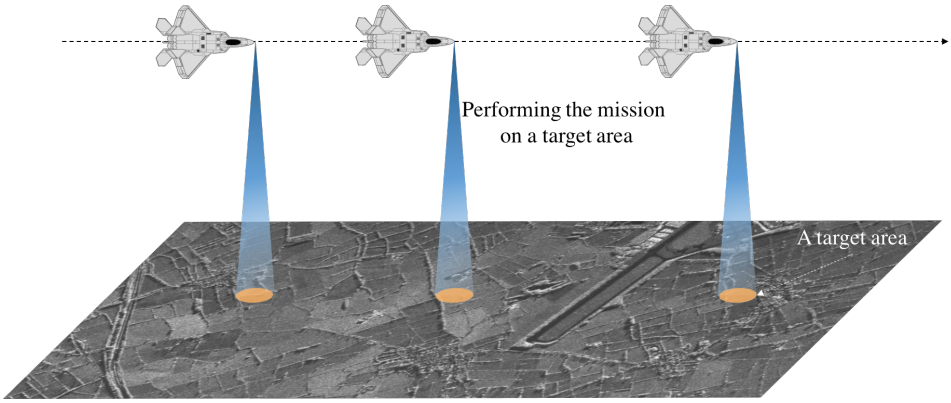


Figure 5.5: The mission of the generating images of target areas using SAR.

Figure 5.6. All the echoes from a target area are stored along the aircraft flies through the spotlight synthetic aperture, when illuminating the target area. Once the aircraft leaves the synthetic aperture, all the recorded echoes from the target area are summed, and as an input to generate the image of this target area.

The mission control computer in the aircraft is a 4 processor SMP system. The defence system is taken from the Generic Avionics Platform (GAP), which is similar to existing U.S. Navy / Marine Corps aircraft [68]. It aims to sufficiently detail the complexity and timing constraints in the mission control software that is typically found in aircraft. The tasks are allocated to different processors using a worst-fit allocation scheme according to their utilisation so that the load is more evenly distributed across different processors. The priority of each task remains unchanged from [68]. All the hard real-time tasks in the defence system are described by Table 5.5. The cost of accessing shared resources is not provided, therefore the resource sharing among tasks in the defence system is ignored in this study.

5.6.2 Mission Modelling

The mission of generating images of a series of target areas can be modelled as a real-time stream processing task, which inputs from a live streaming data source. The minimum inter-arrival time (MIT^{item}) of the image generation is 25 ms (15m ÷ 2160km/h). The WCET of processing each input, C^{item}, is 40 ms. The deadline (or latency) for processing each input, D^{item}, is 480 ms. In this example, the prologue (e.g., buffer manipulation etc.) and epilogue (e.g.,

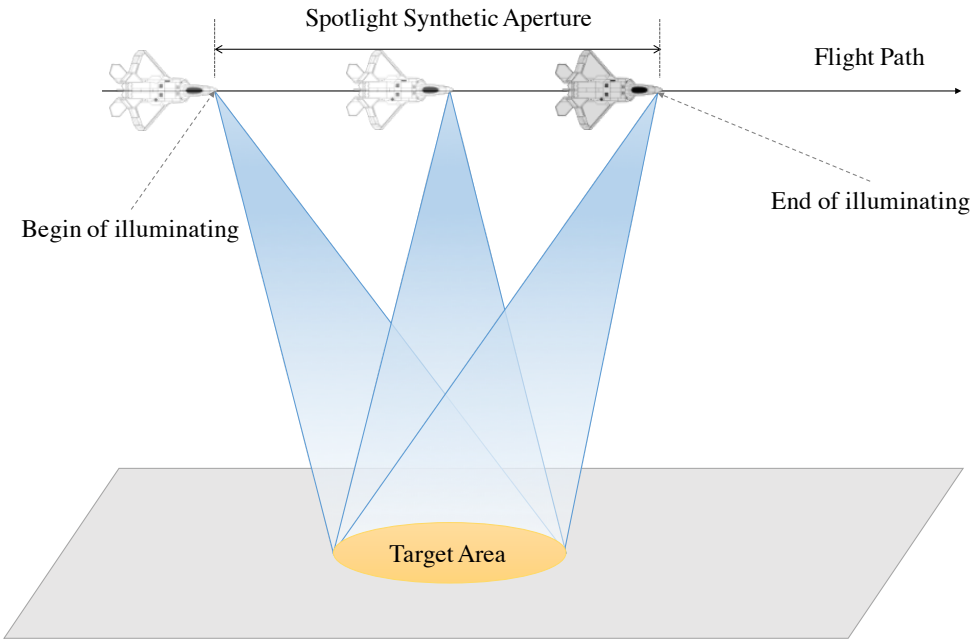


Figure 5.6: Illustration of a SAR operating on a target area, using the spotlight mode.

memory deallocation) of the real-time stream processing task are executed in processor 0, whilst the parallel processing uses all processors. The WCET of the prologue is 10 ms (including 1 ms of splitting), and the epilogue WCET is 2 ms.

Employing our real-time micro-batching approach, the maximum batch size is calculated using the approach described in Section 4.4.1. The candidate batch sizes, i.e., $1, 2, 3...19$ are examined one by one. Finally, the maximum schedulable batch size is determined to be 17 (because batches of size 18 may cause deadline misses).

With the given batch size of 17, the image generation of target areas can be modelled as a periodic real-time stream processing task, with the above prologue and epilogue. The period and the deadline is 400 ms, which is calculated by $(17 - 1) \times 25$ ms (25 is the MIT of the batch).

The following section details and exemplifies the schedulability analysis of the system.

Table 5.5: Hard real-time tasks in the system. *Proc* is the assigned processor ID.

Name	Priority	C	T & D	U	*Proc*
Weapon Release	98	3	200	1.5	0
Rader Tracking Filter	84	2	25	8.0	1
RWR Contact Mgmt	72	5	25	20.0	2
Data Bus Poll Device	68	1	40	2.5	3
Weapon Aiming	64	3	50	6.0	0
Radar Target Update	60	5	50	10.0	3
Nav Update	56	8	59	13.6	0
Display Graphic	40	9	80	11.3	1
Display Hook Update	36	2	80	2.5	3
Tracking Target Update	32	5	100	5	3
Nav Steering Cmds	24	3	200	1.5	1
Display Stores Update	20	1	200	0.5	2
Display Key Set	16	1	200	0.5	3
Display Stat Update	12	3	200	1.5	2
BET E Status Update	8	1	1000	0.1	3
Nav Status	4	1	1000	0.1	3

5.6.3 Schedulability Analysis

Considering processor P_0 that executes the prologue and epilogue, execution-time servers are generated for processor P_0 using the server generation algorithm proposed in Section 4.3.1, and these servers are given in Table 5.6. In addition, with each candidate prologue server, the maximum possible computation time that can be guaranteed from all the processors is also calculated and given in Table 5.6. The details of calculating the data processing window between the prologue and epilogue, and the maximum computation time that can be guaranteed within the data processing window can be found in Section 5.5.1. Again, the proposed Java implementation of the server generation algorithm is available at [18].

In processor P_0, the server with period of 400 ms can make the system to provide the maximum capacity before the real-time stream processing task's deadline D_i (400 ms) in the worst-case. In this example, the corresponding selected servers are described in Table 5.7. Note that, in the remaining

Table 5.6: Possible Deferrable Servers For Processor P_0. DPW Represents the Data Processing Window.

Priority	Max C	T	DPW	Max C in DPW From All Processors
55	314.000	400	360.0	1147.0 (MAX)
55	153.000	200	360.0	1139.0
55	75.000	100	360.0	1133.0
55	55.000	80	360.0	1108.0
99	21.000	50	388.0	1071.7
99	21.000	40	388.0	1113.7
99	14.000	25	388.0	1127.7
99	12.000	20	388.0	1143.7

processors, when multiple servers can guarantee the stream processing task schedulable, only the server with a period that is longer or equal to than 100 are selected, for the visualisation the analysis (see Figure 5.7), and to further demonstrate that our server generation algorithm is flexible.

Table 5.7: Generated Deferrable Servers

Name	Priority	C	T	D	Processor	U
S_0	55	314	400	400	0	0.785
S_1	23	317	400	400	1	0.793
S_2	71	156	200	200	2	0.780
S_3	35	78	100	100	3	0.780

The inputs in the batch, i.e., the returned radar pulses for each target, in the worst-case micro batch are partitioned to different processors using the approach proposed in Section 4.3.2 before the parallel processing starts. Their allocations are described in Table 5.8.

5.6.3.1 Periodic Stream Processing Task Response Time Analysis

This analysis uses the techniques described in Section 5.4. Firstly, employing Equation 5.7 with $J_i = 0$ ms, a load of 10 ms (the execution time of the prologue), and the server S_0. The worst-case response time R_i^2 of the prologue

Table 5.8: Input Partitioning

Processor	Allocated Data Items (Arrival Index)
0	2, 4, 8, 12, 16
1	3, 7, 10, 14
2	1, 5, 9, 13
3	0, 6, 11, 15

is calculated as the following recurrence:

$$
\begin{aligned}
w_i^0 =& 10 + \left(\left\lceil \frac{10}{314} \right\rceil - 1 \right) (400 - 314) \\
=& 10 \\
w_i^1 =& 10 + \left(\left\lceil \frac{10}{314} \right\rceil - 1 \right) (400 - 314) \\
&+ \left\lceil \frac{max\left(0, 10 - \left(\left\lceil \frac{10}{314} \right\rceil - 1 \right) 400 \right) + 0}{200} \right\rceil 3 \\
&+ \left\lceil \frac{max\left(0, 10 - \left(\left\lceil \frac{10}{314} \right\rceil - 1 \right) 400 \right) + 0}{50} \right\rceil 3 \\
&+ \left\lceil \frac{max\left(0, 10 - \left(\left\lceil \frac{10}{314} \right\rceil - 1 \right) 400 \right) + 0}{59} \right\rceil 8 \\
=& 24 \\
w_i^2 =& 10 + \left(\left\lceil \frac{10}{314} \right\rceil - 1 \right) (400 - 314) \\
&+ \left\lceil \frac{max\left(0, 24 - \left(\left\lceil \frac{10}{314} \right\rceil - 1 \right) 400 \right) + 0}{200} \right\rceil 3 \\
&+ \left\lceil \frac{max\left(0, 24 - \left(\left\lceil \frac{10}{314} \right\rceil - 1 \right) 400 \right) + 0}{50} \right\rceil 3
\end{aligned}
$$

$$+ \left\lceil \frac{max\left(0, 24 - \left(\left\lceil \frac{10}{314} \right\rceil - 1\right) 400\right) + 0}{59} \right\rceil 8$$

$=24$

Therefore, the worst-case response time of the prologue is 24 ms.

Then we use Equation 5.7 to calculate the time when each processor finishes its processing of the allocated partitions:

- For processor P_0:
  The worst-case response time R_i^{3,P_0} of the execution of the prologue and data processing is 274 ms. ($J_i = 0$ ms, load $= 10 + 5 \times 40 = 210$ ms, server S_0)

- For processor P_1:
  R_i^{3,P_1} for the data processing in this processor is 211 ms ($J_i = 0$ ms, load $= 4 \times 40 = 160$ ms, server S_1)

- For processor P_2:
  R_i^{3,P_2} for the data processing in this processor is 209 ms ($J_i = 0$ ms, load $= 4 \times 40 = 160$ ms, server S_2)

- For processor P_3:
  R_i^{3,P_3} for the data processing in this processor is 212 ms ($J_i = 0$ ms, load $= 4 \times 40 = 160$ ms, server S_3)

Therefore, the parallel data processing is complete at 288 ms. This can be calculated using the following equation:

$$\begin{aligned} R_i^3 &= Max(274, Max(211 + 24, 209 + 24, 212 + 24)) \\ &= 274 \end{aligned}$$

The last step is to calculate when the epilogue is finished. The worst-case response time R_i^4 for the epilogue is calculated to be 102 ms using Equation 5.7 with $J_i = 400 - 314 = 86$ ms, load $= 2$ ms, and server S_0.

Finally, the worst-case response time of the stream processing task is calculated: $R_i = 274 + 102 = 376$ ms.

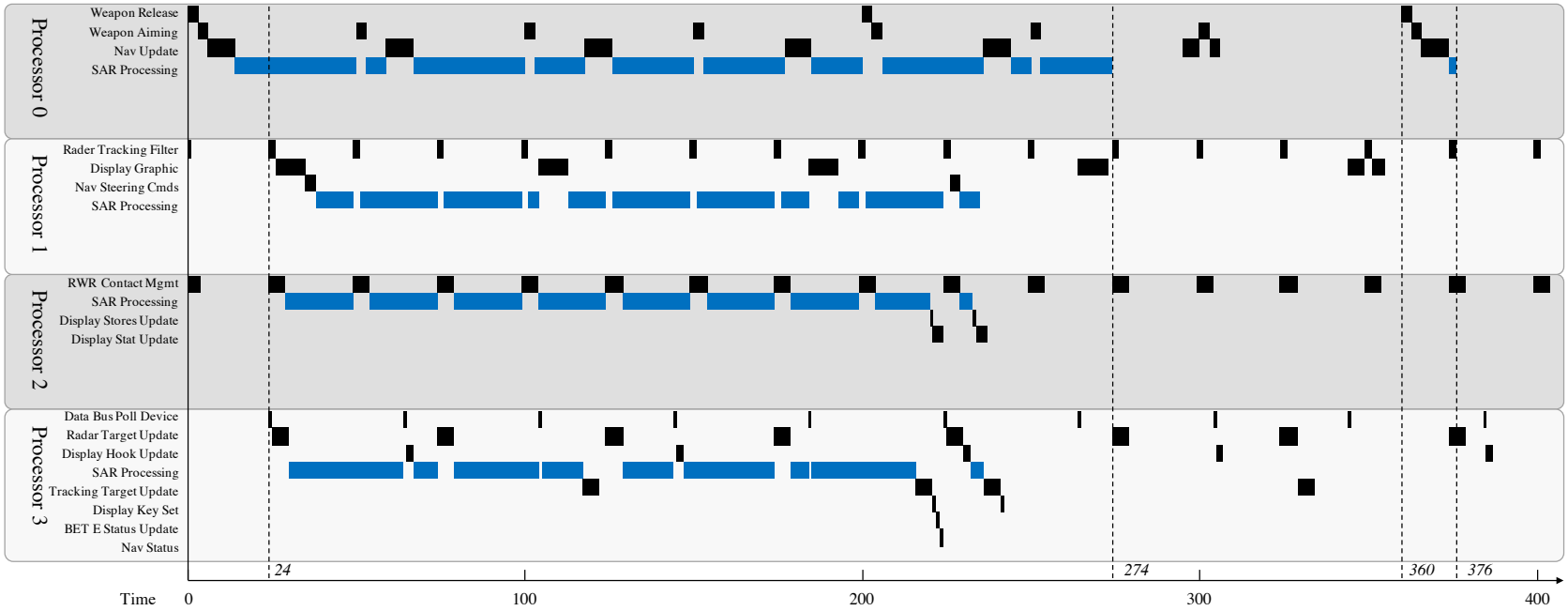


Figure 5.7: The worst-case execution of the stream processing.

A visualisation of this real-time stream processing task's worst-case response time and all the hard real-time tasks in the system is illustrated in Figure 5.7. As can be seen, the prologue finishes at 24 ms, the parallel data processing finishes at 274 ms, and the whole real-time stream processing task finishes at 376 ms. The gap between 274 ms and 360 ms represents the pessimism that occurs from assuming the capacity of server S_0 to be zero at the end of data processing.

5.6.3.2 Latency Analysis

The latency of each data item, i.e. the image generation of each target area, is calculated using the approach described in Section 5.4 and given in Table 5.9. For example, when calculating the latency of the first item in a full batch, the

Table 5.9: Worst-Case Latency of Image Generation.

Item	0	1	2	3	4	5	6	7	8
L	473	449	425	407	418	399	384	360	369
Item	9	10	11	12	13	14	15	16	
L	349	329	305	323	308	285	261	274	

waiting time is $25 \times (17-1) = 400$ ms. The first item is processed in processor P_3, and it is the first data item to be processed. The processing response time of the first data item can be calculated to be 73 ms using Equation 5.7 with $J_i = 0$ ms, a load of the WCET of processing of the item (i.e., 40 ms), and server S_3. Therefore, the latency of the first item is $400 + 73 = 473$ ms.

The response time of the micro batch is less than its period, and the latency of each data item is less than the given constraint, therefore, the real-time live streaming data processing mission is schedulable.

5.7 Summary

This chapter has first summarised related RTA techniques in Section 5.1, and the RTA of a periodic/sporadic task that executes under an execution-time server in Section 5.2.

As a supplement to the original jitter analysis work [47], this thesis notes that an identified worst-case scenario for interference from a higher priority deferrable server called a *double hit* (also known as *back-to-back hit*, described

in Section 5.1.1) does not occur in every situation. Therefore the jitter-based analysis approach for calculating the interference from deferrable servers can be refined, as described in Sections 5.1 and 5.2. This observation can maximise the server's capacity in the server generation algorithm, which is described in Section 4.3.1.

In addition, blocking due to accessing to shared resources are accommodated into these analysis equations in Section 5.3.

The worst-case response time analysis of the real-time stream processing task is described in Section 5.4, by performing the worst-case response time analysis for each execution phase of the real-time stream processing task. Additionally, this section has also noticed the pessimism in the response time analysis for the epilogue, and a solution to this problem has been given in Section 5.4.3. This chapter has also explained how to analyse a real-time stream processing task that inputs from a batched data source or a live streaming data source using these techniques.

Specifically, an example of how to configure and schedule a real-time stream processing task that inputs from a batched data source to meet the deadline, whilst maintaining the existing guarantees for hard real-time activities, has been given in Section 5.5. This example has described the execution-time server generation, data allocation, and worst-case response time analysis for the real-time stream processing task used this example; a case study based on a modified Generic Avionics Platform to demonstrate the overall approach of real-time live streaming data processing with the proposed real-time micro-batching approach has been described in Section 5.6. This case study has explained how the maximum micro batch size and the timeout value are determined, followed by the worst-case latency analysis for each data item.

Chapter 6

SPRY - The York Real-Time Stream Processing Framework

The architecture of the real-time stream processing was presented in Chapter 3, this chapter describes the York Real-Time Stream Processing Framework (SPRY), which is an implementation of the proposed architecture using Java and the Real-Time Specification for Java (RTSJ).

The presented architecture could be implemented using other programming languages, such as Ada, however, Java and RTSJ were adopted. The reasons are given in Section 6.1.

The real-time batch stream processing infrastructure's implementation is discussed in Section 6.2, while the implementation of the real-time micro-batching architecture is described in Section 6.3. Section 6.4 accounts for the overheads of SPRYEngine in the analysis. Section 6.5 describes the case study (see Section 5.6) using SPRY. Section 6.6 summarises the contents of this chapter.

The source code for SPRY can be found in [18].

6.1 Use of Java and the RTSJ

Java is a well-established programming language, which is widely used in industry [11]. Large-scale commercial applications are developed in Java or in programming languages built on the top of Java, as described in Chapter 2.

Even though C++ or Ada is a more commonly used language in real-time systems, these languages have no built-in stream processing support, which is similar to Java 8 streams.

Furthermore, over the last decade the Java platform has been augmented with real-time facilities by the Real-Time Specification for Java (RTSJ) [17], which mitigates against the Java limitations for real-time systems.

Java code is compiled to byte-code, which is architecture-neutral and executed by the Java virtual machine (JVM). This supports the portability of Java programs. As an interpreted language, the performance of Java might be insufficient for high performance computing. However, Just-in-Time (JIT) technology compiles code to native binaries just before it is used within a Java program execution so that the performance will be improved. In addition, for more predictable code execution, Java code can be compiled to native binaries to achieve high performance using ahead-of-time compilation techniques, which is supported by implementations of the RTSJ.

The real-time support that is required by the proposed real-time stream processing architecture, such as, the preemptive priority-based scheduling, execution-time servers, affinity settings, etc., are all provided by RTSJ.

The Java 8 Stream [13] framework is adopted when implementing the pipeline, and the real-time stream processing infrastructure that evaluates the pipeline in real-time. The Java 8 Stream framework supports data parallelism (see Section 2.3.4) that is the scheme used by the proposed real-time stream processing architecture, and enables efficient bulk data processing.

As has been discussed in Section 2.3.4, for a single node multiprocessor platform, data parallelism is a more efficient choice than control parallelism because it requires no synchronisation during data processing so that the processors can be utilised more effectively than with control parallelism. In addition, data parallelism is also widely used in modern stream processing applications. Section 6.1.1 compares Java 8 streams with StreamIt [24],which uses a control parallelism model for pipelines by default. StreamIt has been chosen for the comparison as it a language that was designed explicitly for stream process applications. Additionally, as StreamIt is designed for high performance stream processing applications [87] (and a data parallelism model can be defined by the user). We also compare the efficiency of the Java 8 Stream framework with StreamIt when both of them use the data parallelism model.

The experiments are performed on an AMD Opteron 32-core processor

platform, with a 64-bit Linux operating system (kernel version 4.4.0). The Java SE 8 and StreamIt version 2.1.1 [6] are used in this section.

6.1.1 Data Parallelism versus Control Parallelism

This experiment considers a benchmark that simulates the synthetic aperture radar (SAR) image generation using the stream processing. This benchmark is based on the SAR benchmark in the HPEC Challenge benchmark suites provided by the MIT Lincoln Laboratory [8], which is originally written in MatLab.

This experiment implements this benchmark using both Java 8 streams and StreamIt, and the structure of the stream processing in this benchmark can be illustrated by Figure 6.1. As shown in the figure, the pipeline contains 8 filters, each of them is fitted into a Java 8 stream pipeline and a StreamIt pipeline. As the StreamIt compiler compiles the code into native binaries, typically the compiled program has a better performance than the interpreted program [74]. Therefore, in order to focus on the efficiency of different frameworks, in the Java benchmark, the functionality of each filter is implemented using the same C code. The C code is compiled using g++ from the GNU Compiler Collection (GCC), i.e., the same backend of the StreamIt compiler, with the O3 optimisation option. Then these functionalities are accessed by Java 8 streams using the Java native interface (JNI).

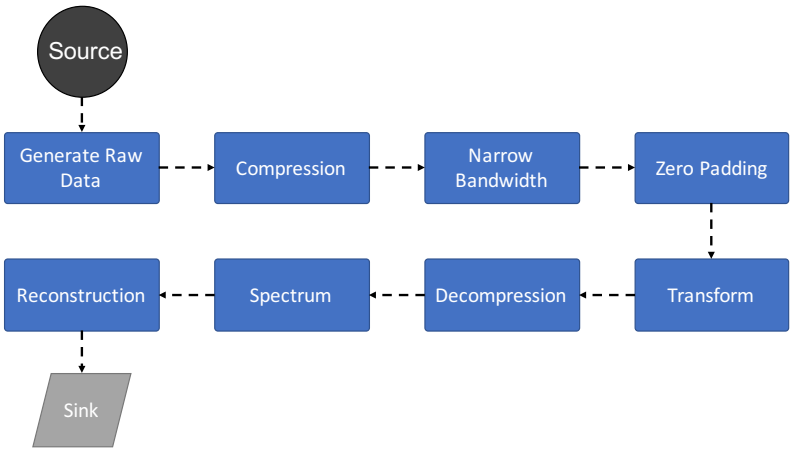


Figure 6.1: The pipeline structure of the SAR benchmark. The benchmark inputs from an integer source. After reading an input, it generates radar echoes, and digitally reconstructs the SAR image.

In this experiment, the benchmark is configured with the parallelism equal to 1, 2, 4, 8, 16. The experiment is performed with these configuration 30 times, and 64 inputs, which requires the experiment to run for more than 24 hours. In addition, each input requires a relatively long computation time, this mitigates the percentage of communication cost in the experiment for StreamIt. The response times of the stream processing (i.e., the time interval from when the stream processing starts to when all the data items have been processed) are measured, and shown in Figure 6.2. Note that, the results have a relatively small variation (see coefficient of variation in Table 6.1), therefore, the worst-case response time is used to represents the experiment result, and variations are not shown.

Table 6.1: Variations in the SAR Stream Processing Response Times. Coefficient of Variation (CV) Represents the Standard Deviation/the Mean Response Time.

Java 8 Streams SAR					
Processors	1	2	4	8	16
SD	280.47	166.57	200.45	83.73	105.63
CV	0.0007	0.0008	0.0020	0.0017	0.0041
StreamIt SAR					
Processors	1	2	4	8	16
SD	991.10	702.77	378.97	1088.31	919.48
CV	0.0019	0.0023	0.0021	0.0059	0.0050

As we can see, overall, the response times of the Java 8 streams are smaller than the StreamIt's response times. In addition, the response time of Java 8 streams decreases as the number of allocated processors increases, in the whole experiment. However, the response time of the StreamIt benchmark first decreases as the number of allocated processors increases, but the increment stops when there are 8 or 16 processors.

The reason the response time of the StreamIt benchmark does not scale down when there are 8 or 16 processors is because the pipeline contains fewer filters compared to the allocated processors; and by default, StreamIt employs a control-parallel model, i.e., allocates each filter to different processors. In this experiment, the pipeline only contains 8 filters in total, therefore, the response time of the StreamIt benchmark can not decrease after 8 processors because

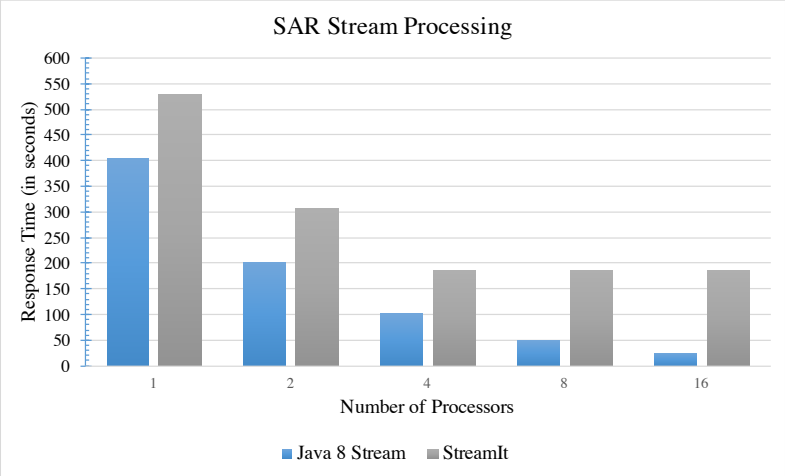


Figure 6.2: The observed worst-case response times of the SAR benchmark executing with different parallelism.

the remaining processors will be idle. However, all the processors are utilised by the data-parallel model, which is used by Java 8 streams. Therefore, the response time of the Java 8 streams scales down as the number of processors increases.

In more details, as shown in the results, the response time of the StreamIt benchmark increases when moving from the experiment with 4 processors to the experiment with 8 processors. Recall that, the StreamIt compiler merges filters when there are less processors. Therefore, allocating these filters to 8 processors requires more inter-filter communications and coordination, compared to using 4 processors. The overhead introduced by the communications increases the response time.

Moreover, the other factors that impacts the scalability of the control-parallel pipeline, i.e., the StreamIt benchmark in this experiment, are:

- In this experiment, each filter in the StreamIt pipeline requires different amount of computation time for processing an input (see the computation time required for each filter to process an input is estimated by the StreamIt compiler, and shown in Figure 6.3). In this situation, different processors process the input at different rates. This results in one or more processors being idle during the stream processing, as discussed in Section 2.3.4.

- The inputs have to be processed one by one. As described in Sec-

```
/* Part of the output generated by the StreamIt compiler */
...
Work Estimates:
    Reconstruction__43                12150009 (36%)
    Transform__28                     7200009  (21%)
    Spectrum__38                      7140009  (21%)
    GenerateRawData__8                5070009  (15%)
    Narrowbandwidth__18               1440009  (4%)
    Decompression__33                 150009   (0%)
    Compression__13                   60009    (0%)
    ZeroPadding__23                   30009    (0%)
    Source__3                         6        (0%)
    Sink__46                          3        (0%)
Building stream config...
...
```

Figure 6.3: The percentage of the computation time required by each filter in the SAR benchmark.

tion 2.3.4, when the first processor is processing the first input, the remaining processors have to wait and be idle.

In summary, for a single node multiprocessor platform, assuming there is enough data, the control parallelism requires the code to be more parallelised as the parallelism increases, therefore introducing extra complexities. However, the data parallelism is easier to handle this case. They are identical when there are enough inputs, the control parallelism uses all the processors and all the processors are well balanced.

More formally, the statistical significance of the experimental results can be demonstrated using an ANOVA analysis. This is given in Appendix A of this thesis. The analysis demonstrates at the 95% confidence level that for this application data parallelism has a better performance than the control parallelism.

6.1.2 Infrastructure Overheads

The goal of this experiment is to evaluate the efficiency of Java 8 streams and StreamIt, when both of them use the data parallelism mode.

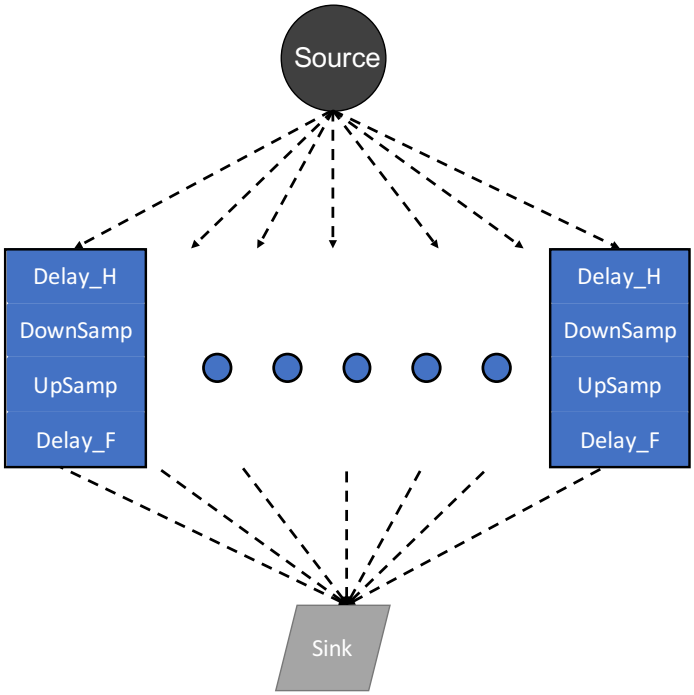


Figure 6.4: The StreamIt version of filter bank benchmark, which contains multiple configurable branches. It performs multi-rate signal processing. On each branch, a delay, filter, and downsample is performed, followed by an upsample, delay, and filter.

The benchmark used in this experiment is based on the C version of the filter bank benchmark [23] provided by StreamIt. The original StreamIt version of the filter bank benchmark has 8 branches, each of which is a pipeline that contains 6 filters. These $8 \times 6 = 48$ filters are allocated to different processors by the StreamIt compiler, rather than these 8 branches. In order to evaluate the efficiency of StreamIt with the data-parallel model, this experiment creates a new filter bank benchmark based on the C code, rather than using the StreamIt version of the benchmark. In the new benchmark, all the filters are merged to be a new filter, which will be fitted into each branch, so that the StreamIt version of the filter bank benchmark uses its data parallelism mode.

The StreamIt version of new benchmark, which with a parallelism of 4 is illustrated by Figure 6.4. Note that, the source and sink in the StreamIt benchmark are allocated with two dedicated processors, so that each branch will have an entire processor to utilise during the experiment.

Again, in the Java benchmark, the functionality of the filter bank is im-

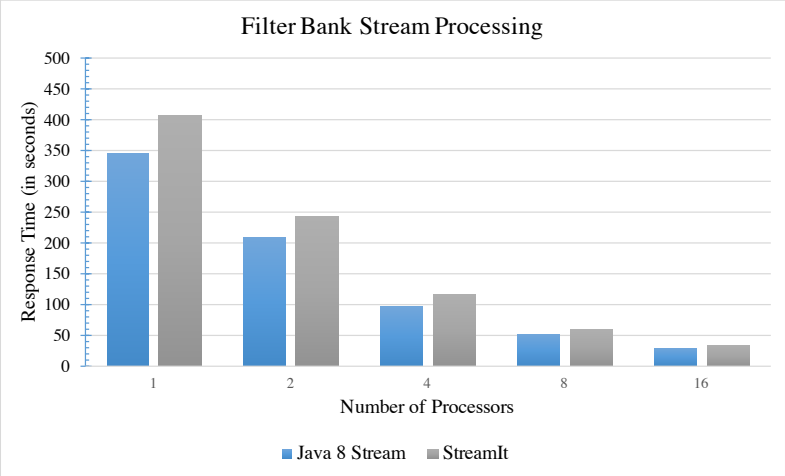


Figure 6.5: The observed worst-case response times of the filter bank benchmark.

plemented using the same C code, compiled using the same backend of the StreamIt compiler, and accessed via JNI.

The experiment is performed 30 times with the parallelism equals to 1, 2, 4, 8, 16. The inputs are 131072 items, which allows the experiment to run for 24 hours. The response time of the stream processing in both Java 8 streams and StreamIt benchmarks are measured, because the variation in the result is small (see CV in Table 6.2).The worst-case response time are shown in Figure 6.5.

Table 6.2: Variations in the Filter Bank Stream Processing Response Times. Coefficient of Variation (CV) Represents the Standard Deviation/the Mean Response Time.

Java 8 Streams Filter Bank					
Processors	1	2	4	8	16
SD	151.57	5759.41	263.08	163.48	292.25
CV	0.0004	0.0306	0.0027	0.0032	0.0098
StreamIt Filter Bank					
Processors	1	2	4	8	16
SD	180.22	5364.92	1236.12	448.69	175.19
CV	0.0004	0.0239	0.0109	0.0075	0.0051

As can be seen the response times of Java 8 streams are shorter than the

response times of StreamIt, but the scalability of them are similar. The time required for the application's startup time is not included. The response time is measured from the time when the application starts its stream processing, to the time when all the input has been processed. After the application has been started, Java uses optimised JVM code, while StreamIt uses its libraries that introduce overheads. Therefore, the response times of Java streams are smaller.

Note that, the startup time of a Java application is typically longer than StreamIt. This is because Java requires the entire JVM to be started, while StreamIt only requires the main function of a C program to be loaded. However, the startup time is not typically considered in the schedulability analysis, therefore, it is not considered in this evaluation.

Overall the Java 8 streams are more efficient than StreamIt, even though the execution of the Java 8 streams are interpreted. Again, the statistical significance of the experimental results can be demonstrated using an ANOVA analysis. This is given in Appendix A of this thesis. It shows that for this experiment the Java 8 stream has a better performance at the 95% confidence level.

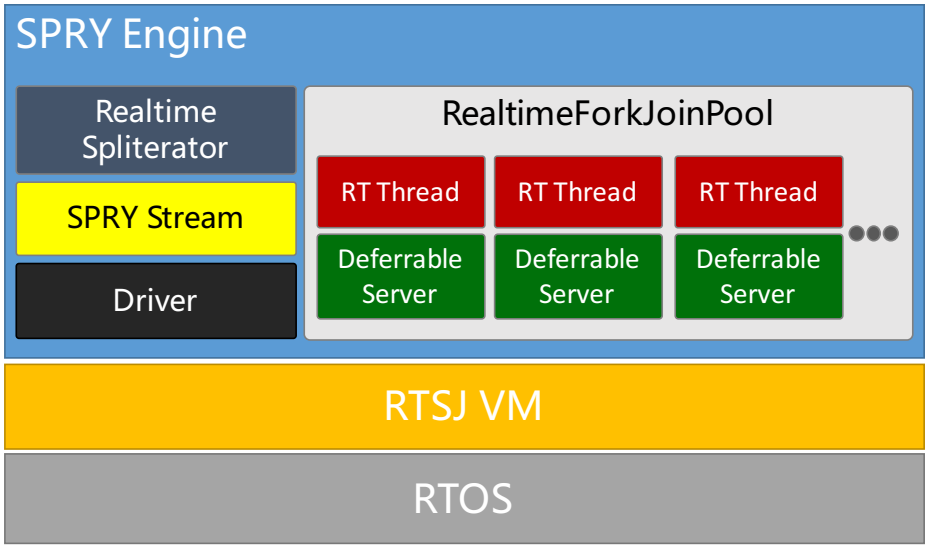
6.2 SPRYEngine – the Real-Time Batch Stream Processing Infrastructure Implementation

This section describes the implementation of the real-time batch stream processing infrastructure, which was presented in Section 3.4.1. The implementation of the real-time batch stream processing infrastructure is called *SPRYEngine*.

SPRYEngine is built on the top of a RTSJ virtual machine, which is running on a real-time operating system. The SPRYEngine, and the classes it uses are illustrated by Figure 6.6a. The mapping from the components in the architecture (see Figure 3.7) to the classes in SPRYEngine is given in Figure 6.6b. For example, the workers component is implemented by real-time worker threads in a `RealtimeForkJoinPool`, which is based on the Java `ForkJoinPool` library.

The discussion of the implementation of SPRYEngine is structured as follows.

1. Section 6.2.1 considers the integration of Java 8 streams with the RTSJ,



(a) The architecture of SPRY Engine.

Component	Implementation Class	Library Used
The Real-Time Batch Stream Processing Infrastructure		
Driver	Driver	ForkJoinTask
Data Partitioner	RealtimeSpliterator	Spliterator
Pipeline	SPRYStream	Java 8 Stream
Execution-Time Server	DeferrableServer	PGP
Workers	RealtimeForkJoinPool	ForkJoinPool
Remaining Configuration Parameters		
Data Allocation Policy	-	HashMap
Deadline	-	RelativeTime
Period	-	RelativeTime
Deadline Miss Handler	-	AsyncEventHandler
MIT Violation Handler	-	AsyncEventHandler

(b) Real-Time Batch Stream Processing Infrastructure Implementation. PGP Represents `javax.realtime.ProcessingGroupParameters`.

Figure 6.6: The implementation of SPRYEngine and corresponding components in the real-time stream processing architecture.

so that a Java 8 stream can be processed in real-time (i.e., at a given priory, and its execution is bounded by execution-time servers). This section mainly describes the implementation of `RealtimeForkJoinPool` and `DeferrableServer` in SPRYEngine.

2. Section 6.2.2 describes how to allocate data partitions to threads according to an allocation policy, with the help of `RealtimeSpliterator` for data partitioning.

3. The implementation of `Driver` is decribed in Section 6.2.3.

4. Section 6.2.4 describes the implementation of `SPRYStreams`, which allow the pipeline of SPRYEngine to be configurable.

5. The implementations of deadline miss and MIT violation detection, and their handlers are given in Section 6.2.5.

6. Section 6.2.6 describes the procedure when the method (called `processBatch`) of SPRYEngine is invoked, this is corresponds to the sequence illustrated in Figure 3.8.

7. Finally, an example of using SPRYEngine to create a real-time stream processing task is given in Section 6.2.7, it demonstrates the initialisation of SPRYEngine.

6.2.1 Real-Time Streams

The implementation of the SPRYEngine utilises the Java 8 stream processing libraries. However, these have been designed to address real-time concerns. The main problem is that, as described in Section 2.3.7, Java 8 streams are evaluated by the `ForkJoinPool`, which uses standard Java threads rather than real-time threads.

Therefore, the first step is to focus on modifying the behaviour of the `ForkJoinPool` so that the worker threads are real-time threads rather than standard threads.

Difficulties In Creating a Real-Time Thread Pool

The `ForkJoinPool` has been designed so that the programmer has some control over its configuration; in particular the number of worker threads. It also allows the application to provide its own factory (i.e., an interface or a method for creating new instances of a class) 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 customised ForkJoinPool 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 first consider a delegation approach. With this approach, each 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;
  }
  public void run(){
    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, we conclude that integrating the RTSJ with the ForkJoinPool requires the source code to be modified.

The RealtimeForkJoinPool

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. Specifically, a `RealtimeForkJoinPool` contains one worker thread per processor, because the work load involves no blocking. In addition, each worker thread of a pool is executed under the control of an execution-time server, as discussed in Section 3.4.1. The real-time worker threads are obtained by patching the code of `ForkJoinWorkerThread`, so that it directly extends the RTSJ `RealtimeThread`.

When the constructor `RealtimeForkJoinPool` is invoked, the worker threads are first created. Then, each worker thread is assigned with the given priority, registered to the corresponding execution-time server, and pined to a processor using RTSJ `AffinitySet` within the worker thread's constructor.

In addition, we have suggested changes to JSR 282[1] to allow a Java thread to execute at a real-time priority, therefore constructing real-time ForkJoin thread pools without modifying the library source code, which have now been adopted.

Deferrable (Execution-Time) Servers

The framework is independent of the server technologies, it uses the approach suggested in [92] to allow a range of servers to be associated with it. However, our current analysis only consider deferrable servers, hence, only the implementation of the deferrable server is considered in this thesis.

[1]The JCP Expert Group has released a new version of the RTSJ (Version 2.0) in early 2017. This version is compatible with Java 8.

In RTSJ, a server can be effectively generated when assigning processing group parameters (PGP) to one or more aperiodic real-time threads. The parameter defines the server's start time, capacity, and period. According to the implementation requirements described in Section 3.4.1, the driver thread has the same priority with the worker threads in a `RealtimeForkJoinPool`. Therefore, all the threads that are assigned to any PGP in this framework have the same priority. Hence, a deferrable server can be obtained.

Processing a Java 8 Stream in Real-Time

So far, the proposed `RealtimeForkJoinPool` allows a Java 8 stream to be evaluated at a given priority, and executed under deferrable servers. An example of performing a real-time stream processing that counts how many words in a batched data source is given below.

```
PriorityParameters priority; /* The priority */
BitSet affinities; /* All the allocated processors */
ProcessingGroup[] servers; /* Execution-Time Servers */
ArrayList<String> data; /* A batched data source */
long count;

RealtimeForkJoinPool rtPool = new RealtimeForkJoinPool(priority,
    affinities, servers);

final Runnable sp = new Runnable() {
   @Override
   public void run() {
      count = data.parallelStream().flatMap(line ->
          Stream.of(line.split("\\W+"))).count();
   }
};

rtPool.submit(sp);
```

This example first creates a `RealtimeForkJoinPool`, then submits the stream processing pipeline to it via a `Runnable` instance. The data is split, allocated to different worker threads by the Java 8 Stream framework. Then the data partitions are processed by each worker thread at the given priority, when the corresponding server has capacity.

6.2.2 RealtimeSpliterator and Pre-Allocating Data Partitions to Worker Threads

As reviewed in Section 2.3.7, the input of a Java 8 stream is partitioned by the `Spliterator`, which initially maintains references to all the data items in the input. Once the `trySplit` method is invoked, a spliterator splits itself into two parts from the middle. By default, the data splitting occurs dynamically within the whole procedure of the Java 8 stream evaluation. Additionally, the first splitting occurs when the terminal operation of a Java 8 stream is invoked.

In order to support the data pre-allocation scheme required by SPRY-Engine, first we introduced the `RealtimeSpliterator`, which implements the `Spliterator` interface. It splits out one (or more according to the granularity) data items from the head of the input, and keeps the remaining data items.

The code of the Java 8 Stream terminal operation is required to be patched, so that the default spliterator is replaced by our `RealtimeSpliterator` before the first splitting occurs. A terminal operation with the help of the `RealtimeSpliterator`, splits the input into partitions. These partitions are then pushed into each local queue of the worker threads in a `RealtimeForkJoinPool`. All the worker threads are woken up once all the partitions have been allocated. For example, the `compute` method of the `ForEachTask` that implements the stream's `forEach` operation is modified using the following code fragement.

```
spliterator = new RealtimeSpliterator<S>(spliterator);
/*... some code omitted ...*/
ArrayList<Spliterator<S>> partitions = new ArrayList<>();
partitions.add(leftSplit);
Spliterator<S> tempSpltr = null;
while ((tempSpltr = spliterator.trySplit()) != null){
   partitions.add(tempSpltr);
}
ForEachTask<S, T> partitionToPush = null;
/* push partitions into queue */
for (int i = 0; i < partitions.size(); i++) {
   Spliterator<S> s = partitions.get(i);
   partitionToPush = new ForEachTask<>(task, s);
   partitionToPush.push();
}
```

```
/* wake up the other workers */
task.notifyWorkers();
```

In order to support the above procedure, the `ForkJoinPool` requires code patching, so that the `ForkJoinPool` allows a partition (which has been encapsulated into a `ForkJoinTask`) to be pushed into its target thread's queue via a push method, according to a given data allocation policy that is implemented using Java `HashMap`. Additionally, by default, the `ForkJoinPool` wakes up the worker thread once any data partition is pushed into its queue. However, according to the our real-time stream processing task model, the data processing occurs only when all data allocation has been completed. Therefore, the `ForkJoinPool` is modified so that it avoids waking up the worker thread when pushing a data partition into its queue, and provides a method that allows all the workers to be woken up when all data partitions have been allocated. Recall that, the worker thread accesses the data partitions using a work-stealing algorithm (see Section 2.3.7) once it has been woken up. Therefore, the work-stealing algorithm is replaced so that each worker thread only takes data partitions from its own queue.

6.2.3 The Driver

The `Driver` class is implemented as a `ForkJoinTask`. When the `processBatch` method is invoked, the `Driver` will be submitted to a `RealtimeForkJoinPool` to perform the data partitioning and start the parallel processing of a Java 8 stream.

According to the real-time stream processing task model, the `Driver` should execute in the prologue processor. Therefore, the `RealtimeForkJoinPool` is required to be patched to allow the prologue processor to be configurable, and to ensure that only the worker thread on the prologue processor performs the `Driver`'s functionality. Specifically, `Driver` is submitted to a `RealtimeForkJoinPool`'s shared queue for execution. The code of `ForkJoinPool` is patched so that, only the worker thread on the prologue processor can access the shared queue, i.e., can take the `Driver` to execute.

Additionally, in the implementation, we also consider that a real-time thread might execute some sequential code, before invoking the `processBatch` method of the SPRYEngine, and then execute some sequential work after that. In this case, the prologue processor is the processor that executes

this real-time thread. Therefore, this real-time thread is registered to the corresponding execution-time server, when it first invokes the `processBatch` method.

6.2.4 The SPRYStream Pipeline

Recall that the proposed real-time stream processing infrastructure allows the pipeline of the stream processing to be configured. The reason is that this ensures the processing of the micro batch (described in the following sections) uses Java 8 streams, with the help of SPRYEngine.

However, the Java 8 Stream pipeline (e.g., `.map().filter().forEach()`) cannot be created outside of the context of a Java 8 stream, and a Java 8 stream can only be created from a single source of input data, and the source can not be changed once a stream has been created. In addition, a Java 8 stream is activated once it has been created. These features conflict with the SPRYEngines requirements, and therefore it is necessary to develop our own version of a stream, called a `SPRYStream`. The `SPRYStream`'s API is compatible with the existing Java 8 Stream API [73].

`SPRYStream` was defined as an interface that extends the Java `Stream` interface, but also allow their processing pipeline to be reused over different input collections (i.e., to apply to multiple batched data sources) via providing deferred terminal operations. When a normal terminal operation is invoked, the `SPRYStream` evaluates as same as a Java 8 stream. If a deferred terminal operation is invoked, the `SPRYStream` does not evaluate until the `processData` method is invoked. In addition, `SPRYStream` provides an method called `attachData` to allow a data source to be attached before the evaluation. All the deferred terminal operations of `SPRYStream` are given below, where `SPRYBaseStream` defines the `processData`, `attachData` methods.

```
public interface SPRYStream<T> extends Stream<T>, SPRYBaseStream<T> {
   public void forEachDeferred(Consumer<? super T> action);
   public void forEachOrderedDeferred(Consumer<? super T> action);
   public void toArrayDeferred();
   public <A> void toArrayDeferred(IntFunction<A[]> generator);
   public void reduceDeferred(T identity, BinaryOperator<T>
       accumulator);
   public void reduceDeferred(BinaryOperator<T> accumulator);
   public <U> void reduceDeferred(U identity, BiFunction<U, ? super
       T, U> accumulator, BinaryOperator<U> combiner);
```

```
    public <R> void collectDeferred(Supplier<R> supplier,
        BiConsumer<R, ? super T> accumulator, BiConsumer<R, R>
        combiner);
    public <R, A> void collectDeferred(Collector<? super T, A, R>
        collector);
    public void minDeferred(Comparator<? super T> comparator);
    public void maxDeferred(Comparator<? super T> comparator);
    public void countDeferred();
    public void anyMatchDeferred(Predicate<? super T> predicate);
    public void allMatchDeferred(Predicate<? super T> predicate);
    public void noneMatchDeferred(Predicate<? super T> predicate);
    public void findFirstDeferred();
    public void findAnyDeferred();
    public void iteratorDeferred();
    public Spliterator<T> spliteratorDeferred();
}
```

The `processData` method takes a reference to a batched data source to be processed, and optionally a callback which is called to present the result. In our implementation, this callback delegates the SPRYEngine's `SetCallback` method. If the `processData` method is invoked when there is not a terminal operation, an `NoTerminalOperationException` will be thrown. If the `processData` method is invoked when there is not a data source, it perform the stream processing on an empty collection, as with existing Java streams if they are created on an empty collection.

A `SPRYReferencePipeline` implements the `SPRYStream` interface, and represents a `SPRYStream` of Java objects. In addition, we have implemented the equivalent classes for Java's primitive types.

In a `SPRYStream` pipeline, 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 `SPRYStreams`.

A `SPRYStream` pipeline can either be initialised when passed to the constructor of the SPRYEngine, or by a functional interface named `Reference-`

`PipelineInitialiser`, which is required by the constructor. By employing functional interfaces, the SPRYEngine is able to take the advantage of Java's lambda expressions to make code more concise. See the example in Section 6.2.7.

6.2.5 Detecting Deadline Miss and MIT Violation

Considering the SPRYEngine might be used for hard real-time stream processing, any deadline miss and invocation minimum inter-arrival time (MIT) violation are required to be detected, and their handlers should be released. Invoking the MIT violation means that the time span of any two invocations of the SPRYEngine's `processBatch` method is less than the given MIT (i.e., the period) of the SPRYEngine.

The times of the most two recent invocations of the `processBatch` method that is provided by the SPRYEngine are recorded. The MIT violation handler is released if the time interval between these two times is less than the given period of the stream processing task. Additionally, a RTSJ `OneShotTimer` can be created to monitor the deadline miss. Specifically, the timer fires at the absolute time of the next deadline, and is canceled if the data processing has been completed within the deadline. SPRYEngine provides methods that allow the deadline, period, deadline miss handler, and MIT violation handler to be configured.

6.2.6 The `processBatch` Method of SPRYEngine

Figure 3.8 showed the required behaviour of the `Driver` after the `Process Batch` is invoked. This section summarises the sequence of actions undertaken by the SPRYEngine to meet these requirements.

1. SPRYEngine records the time of this invocation. Then,
   (a) calculates the time span between the most recent two invocations, and fires the MIT violation handler if the invocation MIT is violated;
   (b) invokes the `RealtimeForkJoinPool.submit` method, with the `Driver` (it will be pushed into the `RealtimeForkJoinPool`'s shared queue);
   (c) starts a timer that will fire and release the deadline miss handler when the next deadline expires.

(d) waits until all the partitions have been processed.

2. The `RealtimeForkJoinPool` wakes up the worker thread that is running on the prologue processor. Once that worker thread wakes up, it takes the `Driver` from the shared queue, and execute it. Therefore, the `SPRYStream.processData` method is invoked, with the given data.

3. The `SPRYStream` uses the `RealtimeSpliterator` to partition the data, and pushs all the partitions into their corresponding worker threads via a push method provided by the `RealtimeForkJoinPool`.

4. The `RealtimeForkJoinPool` pushes each partition to a worker thread's local queue according to the given data allocation policy, and wakes up all the worker threads.

5. The worker thread takes a partition from its local queue, processes it with the `SPRYStream` pipeline, then tries to take another one. Note that, the execution of the worker thread is controlled by the execution-time server it registered to.

6. SPRYEngine gets the time when all the data has been processed, and cancel the deadline monitoring timer when the processing meets the deadline.

6.2.7 Initialising a SPRYEngine Instance

This example demonstrates how to initialise an instance of SPRYEngine to perform a real-time stream processing that counts how many words in a batched data source.

```
PriorityParameters priority; /* The priority */
BitSet affinities; /* All the allocated processors */
ProcessingGroup[] servers; /* Execution-Time Servers */
ArrayList<String> data; /* A batched data source */
DataAllocationPolicy dap; /* Data allocation policy */
int prologueProcessor; /* The prologue processor */
long count;

SPRYEngine<String> spry = new SPRYEngine<>(
   priority,
   /* The processing pipeline */
```

```
    p -> p.flatMap(line ->
        Stream.of(line.split("\\W+"))).countDeferred(),
    dap, affinities, prologueProcessor, servers);

/* set the deadline and its miss handler */
spry.setDeadlineMissHandler(deadline, deadlineMissHandler);
/* set the period and its violation handler */
spry.setMITViolateHandler(period, MITViolateHandler);
/* set the call back to get the result */
spry.setCallback(r -> count = r);

/*Within a real-time thread, perform the real-time stream process for
    the data by invoking the SPRYEngine */
spry.processBatch(data);
```

The SPRYEngine uses a `SPRYStream` to allow the pipeline to be given within the constructor. A `RealtimeForkJoinPool` is used by the SPRYEngine to process the data with multiple processors, according to the data allocation policy. The execution-time servers are used to bound the impact of the real-time stream processing to other real-time activities. Finally, the deadline miss and the invoking MIT violation are also detected by the SPRYEngine.

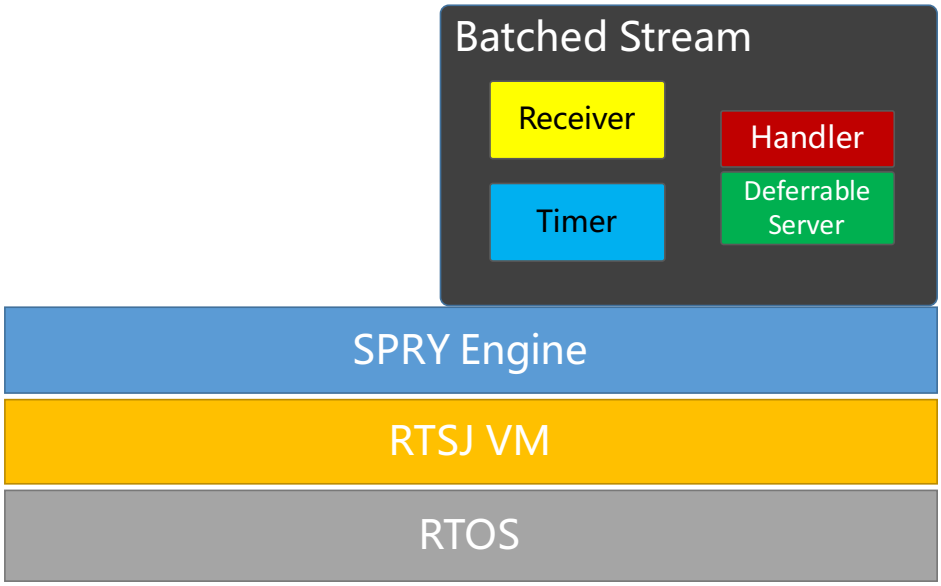
6.3 BatchedStream – the Real-Time Micro-Batching Implementation

The real-time batch stream processing infrastructure implementation has been discussed in the previous section. This section describes the implementation of the `Batcher` architecture proposed in Section 3.4.2, as a new framework called *BatchedStream*.

The structure of the BatchedStream can be illustrated by Figure 6.7a. The mapping from the components in the real-time micro-batching architecture (see Figure 3.9) to the classes in BatchedStream is given in Figure 6.7b. Note that, the implementation of execution-time server, the deadline for micro batch (i.e., for the SPRYEngine), and the deadline miss handler has been described in Section 6.2.

The discussion of the implementation of BatchedStream is structured as follows.

1. The implementation of `Receiver` is described in Section 6.3.1.



(a) The architecture of Batched Streams.

Component	Implementation Class	Library Used
The Real-Time Micro-Batching Infrastructure		
Buffer	Receiver	ArrayList
Timer	Timer	PeriodicTimer
Handler	Handler	AsyncEventHandler
Remaining Configuration Parameters		
Latency	-	RelativeTime
Data Incoming MIT	-	RelativeTime
Latency Miss Handler	-	AsyncEventHandler
MIT Violation Handler	-	AsyncEventHandler

(b) Real-Time Micro-Batching Infrastructure Implementation.

Figure 6.7: The implementation of Batched Streams and corresponding components in the architecture.

2. Section 6.3.2 describes the implementation of `Timer`.
3. Section 6.3.3 describes the implementation of `Handler`, and discusses how to use execution-time servers to execute `Handler`.
4. How to detect latency miss and data arriving MIT violation is described in Section 6.3.4.
5. Section 6.3.5 discusses the parameters that are required by the construc-

tor of BatchedStream.

6. Finally, an example of how to initialise a BatchedStream for real-time live streaming data processing is given in Section 6.3.6.

6.3.1 Receiver

`Receiver` is the implementation of the `Buffer` component in the real-time micro-batching architecture. It provides an method called `store` that allows the data to be stored into a memory area, which is implemented using Java `ArrayList`. It notifies the `Handler` when it has received enough data items. Applications can use a dedicated real-time thread to receive data from a live streaming data source, e.g., a TCP/IP socket, and store the data via invoking the `store` method that is provided by the `Receiver`. The receiver also provides an method called `retrieve`, which allows the stored data items to be encapsulated into a collection and returned.

`Receiver` is implemented as an abstract class, so that users can implement their own receivers, which are attached to difference live streaming data sources.

6.3.2 Timer

`Timer` implements the `Timer` component in the real-time micro-batching architecture, using RTSJ's `PeriodicTimer` class. It manages when the next timeout occurs. It releases the `Handler` when the next timeout expires; the next fire time is automatically reset.

6.3.3 Handler

`Handler` implements the `Handler` in the real-time micro-batching architecture, using the RTSJ `AsyncEventHandler` class.

Once released, it retrieves data from the receiver as a micro batch, then performs the real-time stream processing over the micro batch with the help of SPRYEngine (i.e., via invoking SPRYEngine's `processBatch` method), and resets the next timeout.

The execution of `Handler` is a part of the prologue, therefore, it has to execute on prologue processor, using RTSJ `AffinitySet`. In addition, it is required to be registered to the execution-time server, which is running on the prologue processor.

6.3.4 Detecting Latency Miss and Data Incoming MIT Violation

BatchedStream might be used in real-time live streaming data processing with a hard latency requirement, therefore, the latency miss and the data incoming MIT violation are required to be detected, and their handlers should be released.

Similar to the approach described in Section 6.2.5, the data incoming MIT violation detection is implemented within the `store` method, which is provided by `Receiver` and allows the data to be stored into the buffer.

For the latency measurement, the implementation is similar to the approach presented in Section 6.2.5, for each item, create a RTSJ `OneShotTimer` to monitor the latency miss. Note that, in the intermediate operations in `SPRYStream` that might filter data items out, such as the `filter` operation, timers associated with items that are filtered out are canceled.

In addition, as the SPRYEngine processes each micro batch. Therefore, the deadline of the SPRYEngine instance is required to be configured with a value equals to the possible MIT of the micro batch releases. However, the invocation MIT and the invocation MIT violation handler of the SPRYEngine are not required. The reason is that the data incoming is monitored by the BatchedStream, if the data incoming MIT violation does not occur, any two micro batches cannot be released within the SPRYEngine's invocation MIT.

The BatchedStream provides methods that allow the these time values and handlers to be configured.

6.3.5 The Constructor Parameters of BatchedStream

SPRYEngine is used as the processing infrastructure for micro batches generated by BatchedStream. The initialisation of the SPRYEngine is transparent to users.

Thus, the `SPRYStream` pipeline, execution-time servers, priority, allocated processors, data allocation policy when processing each micro batch, and the prologue processor are required by the constructor of BatchedStream.

Additionally, a callback can be passed into BatchedStream, so that the result for every micro batch can be further processed (e.g., to accumulate). This can also be passed in via a new method named `setCallback`, which is a delegation of the SPRYEngine's callback method, which is then delegated by

the callback of SPRYStream.

6.3.6 Initialising a BatchedStream Instance

An example of initialising a BatchedStream to perform real-time stream processing for a real-time live streaming data source is given below.

```
/* The maximum possible micro batch size */
Receiver receiver = new StringSocketReceiver(BufferSize, "localhost",
    1989);
/* create and configure the BatchedStream's parameters, omitted...*/
BatchedStream<String> bs = new BatchedStream<>(
   receiver ,/* The Receiver */
   timeout, /* The timeout of micro-batching */
   priority, /* The priority */
   /* The stream processing pipeline */
   p ->
       p.map(x->x.toUpperCase()).forEachDeferred(x->System.out.println(x)),
   affinities, /* All the allocated processors */
   dap, /* Data allocation policy for micro batch processing */
   prologueProcessor, /* Indicates the prologue processor */
   servers /* Execution-Time Servers */);

/* set the deadline for micro batch and its miss handler */
bs.setBatchProcessingDeadlineMissHandler(microBatchDeadlineMissHandler);
/* set the data incoming MIT and its violation handler */
bs.setDataIncomingMITViolationHandler(dataMIT, MITViolationHandler);
/* set the latency for each item and the latency miss handler */
bs.setLatencyMissHandler(latency, latencyMissHandler);
bs.start();
```

A receiver is created to receive data items into a buffer, which has a maximum size. Note that, the data collection is done by a real-time thread, which is maintained by the receiver in this example.

When the constructor of the BatchedStream is invoked, an instance of Timer and Handler is created. The BatchedStream then passes the reference of Handler to the Receiver, so that it releases the instance of Handler when it has received enough data items.

Once the start method of the BatchedStream is invoked, the receiver starts to receive data from the given live streaming data source (i.e., from a TCP/IP socket), and the timer is started to maintain the timeout of the micro

batch's release. The data incoming MIT, the processing of each micro batch, and the latency of each data item are monitored by the BatchedStream.

6.4 Accounting for the Overheads of SPRY in the Analysis

This section describes how to account for the overhead introduced by the SPRY.

The overheads of SPRYEngine is required to be accounted when performing the response time analysis for a real-time stream processing task, which inputs a batched data source. The SPRYEngine is created before use, this section describes how to account the overheads in the whole stream processing after invoking the SPRYEngine's `processBatch` method.

In general, system overheads can be classified as synchronous and asynchronous [91]. Synchronous overheads are incurred by an application when it invokes a call on the system's infrastructure. This is accounted for by adding the WCET of the system's code that is executed by the caller to the WCET of the application code. Asynchronous overheads are incurred by threads internal to the system and from the handling of interrupts.

The synchronous overheads, which are introduced by the infrastructure invocations, and contain the following parts:

1. The overhead of when invoking the `processBatch` method, which submits the `Driver` to the `RealtimeForkJoinPool`, and wakes up a worker thread.

2. The overhead introduced by the worker thread running on the prologue processor (before the stream processing), when it tries to take the `Driver`, record times, invoke the `SPRYStream.processData` method, perform the data partitioning using `RealtimeSpliterator`, make inquiries to the data allocation policy, push partitions into each worker thread's local queue, and wake up all the rest worker threads.

3. The overhead introduced by the worker threads running on the remaining processors. Each of these threads introduces overhead when it takes each partition from its local queue, and processes it with the `ReusableStream` pipeline.

4. The overhead introduced by the worker thread running on the prologue processor, after the stream processing, and when it invokes any given callback.

When applying the analysis, part 1 and part 2 should be added to the prologue in the real-time stream processing task model, part 3 should be added to the parallel data processing on the remaining processors, and part 4 is added to the epilogue.

For a BatchedStream, data is collected by a dedicated real-time thread, therefore, the only overhead required to be considered is the one introduced by the `Handler`. Apart from the invocation of the SPRYEngine's `processBatch` method, all the remaining functionalities are performed by the `Handler` can be added to the prologue and epilogue accordingly.

The asynchronous overheads include interrupts for timers, and execution-time servers. In addition, there are also overheads introduced by the garbage collector (GC). In RTSJ, GC can be classified as work-based or time-based. In the former case, each time an application requests to allocate an object, GC performs amount of work, which is determined by the request rate. In the later case, a real-time thread is created for GC, and it runs at a given priority, and periodically with a budge in each period [91]. SPRYEngine uses JamaicaVM [9], which uses a work-based approach. Therefore, the WCET required by the garbage collector for allocating new memory areas for all the instances created by the SPRYEngine during the stream processing, can be added to the prologue in the analysis.

Typically, the WCET of the overheads can be determined by either ahead-of-time analysis of the code, or estimating the upper bound by measurement [38]. This is subject to future work.

Further more, Java 8 streams provide several stateful intermediate operations, such as `sort`, which might order the first arrival item to the last position, therefore violating the analysis. The proposed solution is executing the stateful operations in the prologue, and the computation required of the execution is added to the prologue.

6.5 Representation of the Case Study

This section discusses the representation of the case study presented in Section 5.6. The goal is to describe how to represent a task with given real-time

properties using SPRY, such as the periodic task, rather than describing the implementation of the functionally of those hard real-time tasks in the Generic Avionics Platform (GAP) [68], for example, the weapon aiming task.

The discussion of the implementation of BatchedStream is structured as follows.

1. The representation of hard real-time tasks in the defence system is described in Section 6.5.1.

2. The representation of the SAR image generation mission is given in Section 6.5.2.

6.5.1 Representation of GAP Hard Real-Time Tasks

Each hard real-time task in the defence system are periodic, and they are represented using RTSJ `RealtimeThread`, and `PeriodicParameters` to characterise its period, deadline, first release time. The priority is configured using `PriorityParameters`, the affinity of the thread is configured using `Affinity`. The code that can be used to create a real-time task is given below.

```
public static RealtimeThread create(int cpu, long period, Runnable
    func, int prio, AbsoluteTime firstRelease, String name) {
  RelativeTime D, T;
  D = T = new RelativeTime(period, 0);
  PriorityParameters priority = new PriorityParameters(prio);
  RealtimeThread thread = new RealtimeThread(priority, null) {
     @Override
     public void run() {
        while(true){
           waitForNextPeriod();
           func.run();/* Implementation of the functionality */
        }
     }
  };
  /* The release parameters */
  PeriodicParameters periodicParameters = new
      PeriodicParameters(null, T, null, D, null, new
      AsyncEventHandler(){
     @Override
     public void handleAsyncEvent() {}
  });
```

```
    thread.setReleaseParameters(periodicParameters);

    /* set affinity */
    BitSet processor = new BitSet();
    processor.set(cpu);
    Affinity.set(Affinity.generate(processor), thread);
    return thread;
}
```

Note that, RTSJ only supports 28 real-time priorities. Therefore, in each processor, the priority of each hard real-time tasks is mapped into a unique priority, which is between 11 and 38.

6.5.2 Representation of The SAR Image Generation Task

The radar image generation mission task is represented using SPRY's Batched-Stream, the code is given as follows. SPRYStream is used to describe the radar image generation pipeline.

```
RealtimeReceiver receiver;/* Collecting radar signals */
DeferrableServer[] servers = createServers(startTime); /* Servers */
RelativeTime timeout = new RelativeTime(400, 0); /* Timeout */
/* Allocates processor 0, 1, 2, 3 */
BitSet affinities = new BitSet();
affinities.set(0);affinities.set(1);affinities.set(2);affinities.set(3);
/* Data Allocation Policy */
DataAllocationPolicy DAP = new CustmisedDataAllocationPolicy();
DAP.addPairs(0, 2, 4, 8, 12, 16);
DAP.addPairs(1, 3, 7, 10, 14);
DAP.addPairs(2, 1, 5, 9, 13);
DAP.addPairs(3, 0, 6, 11, 15);
/* The prologue processor */
int prologueProessor = 0;
/* Real-time Stream Processing */
BatchedStream<Integer> streaming = new BatchedStream<Integer>(
    receiver,
    timeout,
    priority,
    p -> p./* Processing pipeline */.forEachDeferred(/* Update Display
        etc. */),
    affinities, DAP, prologueProessor, servers
```

```
);
/* Set the latency miss handler */
AsyncEventHandler latencyMissHandler = new AsyncEventHandler(){
   @Override
   public void handleAsyncEvent() {/* handle latency miss */}
};
streaming.setLatencyMissHandler(new RelativeTime(480, 0),
    latencyMissHandler);
/* start */
streaming.start(startTime);
```

6.6 Summary

This chapter has described the SPRY framework, which is an implementation of the proposed real-time stream processing architecture in Chapter 3.

This chapter has first given the rationale for using Java and RTSJ in Section 6.1 as the programming language for the implementation. In addition, in order to support the decision, the efficiency of Java 8 streams and the StreamIt, and the efficiency of the data-parallel pipeline and control-parallel pipeline have also been evaluated. ANOVA has been applied on the results (see Appendix A) to support our decision to base our implementation of the Java 8 Stream framework.

Then the real-time batch stream processing infrastructure has been implemented as a new framework called SPRYEngine, which is described in Section 6.2. The details and involved difficulties in implementing, and an example of the SPRYEngine have been described in this section.

The real-time micro-batching architecture has been implemented as a new framework called BatchedStream, which is given in Section 6.3. An example of using the BatchedStream for real-time live streaming data processing has been given in this section, along with the implementation details.

Accounting for the overheads of SPRYEngine in the analysis is discussed in Section 6.4, and lastly, Section 6.5 describes the representation of the case study using SPRY.

Chapter 7

Evaluation

This thesis has presented an overall approach to the integration of the stream processing programming model with the traditional embedded system programming model. We have demonstrated how to use the approach in Chapter 4, with a case study described in Section 5.6. In Chapter 6, we have also shown that the presented approach is realisable in practice, by presenting a prototype implementation using RTSJ, which is called SPRY.

This chapter has two goals: to determine the extent to which the major constraints/assumptions of the presented approach has an effect on its efficiency; and to compare the presented approach to a traditional embedded approach, which does not employ the stream processing programming paradigm.

The presented approach is independent of the execution-time server technologies. However, the constraint has been made that only one single execution-time server is used for each processor. In addition, it is assumed that the single server can efficiently use the spare computation time in each processor, i.e., the time that is not used by hard real-time tasks. In Section 7.1, this assumption is tested by comparing the computation time that can be guaranteed from a single server and from multiple servers, with randomly generated hard real-time tasks, and random system requirements. It is demonstrated that little schedubalibity is lost because of the constraint of using a single server per processor.

Additionally, certain assumptions have been made to make the analysis tractable. As discussed in Section 5.4.3, our analysis approach is a sufficient but not an exact analysis, due to the pessimism introduced by the execution-time server replenishment gap. In Section 7.2, the results of the presented

analysis approach is compared to the results from simulations, and we demonstrate that the amount of pessimism is small, which indicates that only little schedulability is lost.

It has been demonstrated that the effectiveness of the presented approach is not undermined by the constraints/assumptions we have made. Together with the case study described in the thesis and the prototype implementation of the SPRY framework, we have provided the evidence that the presented approach is effective and feasible. However, it hasn't been demonstrated that our approach is superior to the traditional embedded system approach so far. In Section 7.3, a set of experiments that compare the schedulability of these two approaches have been performed. Several representative experiments and their results are given in this section, to provide the evidence that the presented approach is superior.

However, there is no silver bullet, any approach might have its limitations. In Section 7.4, we discuss the issue of current task allocation scheme and micro-batching, when the stream processing activity has a very high priority, and with tight deadline/latency requirements.

Finally, Section 7.5 summarises the chapter's findings.

Experiment Setup

The experiments consider scheduling a randomly selected [48] set of hard real-time tasks, and a stream processing task in a 16 cores fully partitioned system, which has a same amount of cores as the experiments performed in Section 6.1. In addition, representative experiments with more cores are also performed.

The size of the hard real-time task set is 128 in all experiments. The periods of hard real-time tasks are randomly generated between 1 and 1000 time units, which covers the range of the GAP [68] tasks' periods. As with the GAP tasks, each hard real-time task in the experiments has its period equal to the deadline. The experiment also investigates the difference when allocating hard real-time tasks to different cores using a worst-fit, best-fit, or random-fit algorithm. The best-fit algorithm used in this thesis allocates a task to the most busy core, where it is schedulable. The worst-fit algorithm always allocates a task to the most idle core. The random-fit algorithm allocates a task to a random core, where it is scheduable.

The period, and loads of the stream processing task, or the MIT and computation time required for processing each live streaming data item are

different in each experiment. The details of these parameters will be given in the following sections.

Each experiment is run 100 times, for each time, a set of hard real-time tasks are generated, and the utilisation is distributed to different tasks based on the approach presented in [48]. Then the schedulability of both hard real-time tasks and stream processing tasks are examined using response time analysis, and the total number of schedulable runs are recorded. Note that, the response time analysis for the traditional embedded approach is given in Appendix B. In addition, for the scheduling simulation, the worst-case is guaranteed to be caught if the simulation runs through the hyper-period of all the tasks. However, in reality, it requires thousands of years to complete the simulation. Therefore, in this section, the simulation window is 100 times of the period of the stream processing task, after which the results stay more constant.

7.1 Single or Multiple Execution-Time Servers

Given a real-time stream processing task, the goal of this section is to find out whether creating multiple execution-time servers for each processor introduces more guaranteed schedulable computation time than using just a single execution-time server.

The single server for each processor is generated using our presented server parameter selection algorithm. When generating multiple servers, the server is generated with the period starts from the smallest divisor of the stream processing task's period, and each server with the maximum schedulable capacity. The hard real-time task set size is 8, and the period and the deadline of the stream processing task is 800. The maximum computation time can be provided are given in Figure 7.1.

As can be seen from the figure, there is no significant difference between these two approaches, as both lines are on top of each other. Running the experiments with more hard real-time tasks, and different periods and deadlines for the stream processing task, gives similar results.

When using multiple servers, the prologue can be executed as soon as possible, thereby ensuring that the data processing begins as soon as possible. However, the server parameter selection algorithm has already considered the length of the data processing window by using as small a period (higher pri-

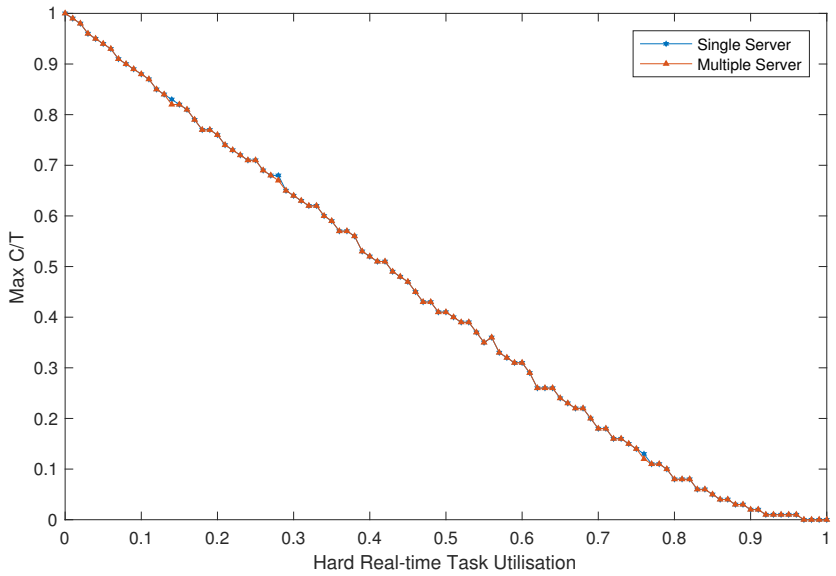


Figure 7.1: The maximum computation time can be provided by a single or multiple servers within a given period.

ority) server as necessary. Therefore, as shown in the figure, the improvement is not significant.

In summary, using one single server for each processor is sufficient for current approach, as multiple servers introduce more implementation overheads.

7.2 Accuracy of the Analysis

This section will evaluate the pessimism of the analysis by comparing our analysis to the results from a scheduling simulator. The simulation of the analysis uses the same set of hard real-time tasks, execution-time servers, etc. The overhead of the execution-time servers is set to be zero, as it has no impact on the results of this experiment. This is because overhead is accounted for by subtracting from the beginning of a server's capacity at each release. This is identical to adding some computation loads to the data processing.

The representative experiment considers the hard real-time task set contains 128 hard real-time tasks, while the stream processing task has a period of 800 time units, the deadline equals to the period, and the WCET for total data processing is 8000 time units, which contains 800 data partitions. The prologue and epilogue of the stream processing task is configured to be 80, as

they are relatively small.

The hard real-time tasks are allocated to different cores using best-fit, worst-fit, and random-fit. Then the total utilisation of the hard real-time task set is increased from 0 to 16, the schedulability of the whole system is tested and recorded, and shown in Figure 7.2. Note that, not shown in the figure, the system schedulability for all approaches before 4.5 is 100%, and 0% after 6.0.

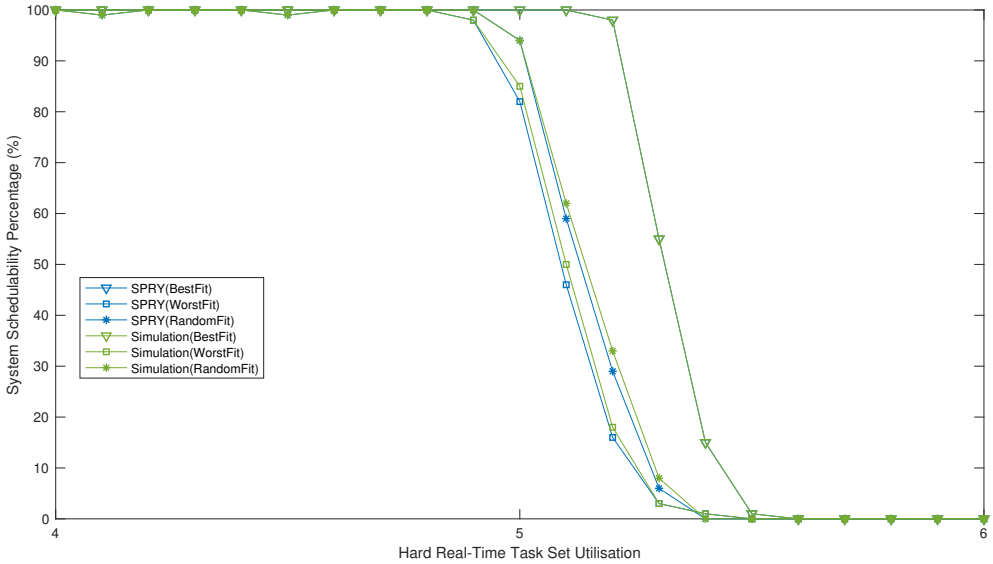


Figure 7.2: The accuracy of the presented analysis approach.

As can be seen, for each hard real-time task set allocation scheme, the simulation result is only slighter better than the analysis result. This indicates that the pessimism of our analysis is acceptable. Again, this section also runs experiments with different input parameters for the stream processing task, the results are similar.

7.3 Comparing to Traditional Embedded Approach

This section compares the schedulability of the SPRY approach against the schedulability of the applications that are more traditionally handled by embedded systems for streaming applications.

The traditional embedded approach splits the data source into partitions, then creates the corresponding prologue task, data processing tasks (one per partition), and the epilogue task. The period is equals to the stream processing

task's period, and the priority is determined using deadline monotonic priority assignment. When allocating tasks, the generated stream processing sub tasks and all the hard real-time tasks in the task set are considered together, with a first-fit, worst-fit, or random-fit allocation algorithm. If multiple generated stream processing sub tasks are allocated to the same core, they are merged into one task. In addition, the epilogue task is merged into the data task that finishes lastly. The priority for each task is determined using deadline monotonic priority assignment.

Note that, when the real-time stream processing task's utilisation is not greater than 100%, another alternative approach could be creating a single sporadic task for the stream processing. However, this approach is covered by the traditional embedded approach with a best-fit task allocation scheme, therefore, it is not covered in the experiment.

These experiments have a set of input parameters, such as the size of the hard real-time tasks set, the period and WCET of the processing task. However, it is difficult to evaluate every combination of values for these parameters in a limited time. This section selects several representative combinations of values for these parameters, and performs the evaluation.

The selected hard real-time task size is 128, and the number of cores are configured to be 16. In addition, an experiment with 128 cores that covers most multicore processors used in embedded systems is performed to demonstrate the scalability of SPRY. The details of the parameters for the real-time stream processing tasks are given in the following subsections.

The evaluation contains two parts:

- experiments for real-time batched data processing, which are discussed in Section 7.3.1.
- experiments for real-time live streaming data processing, which are discussed in Section 7.3.2

7.3.1 Batched Data Source Evaluation

This section considers a stream processing task with a period of 800 time units, a data processing of 4000 time units (i.e., a utilisation of 500%), and the deadline equals to the period. The prologue and the epilogue is configured to be 1% of the stream processing task's period, as they are not computationally intensive.

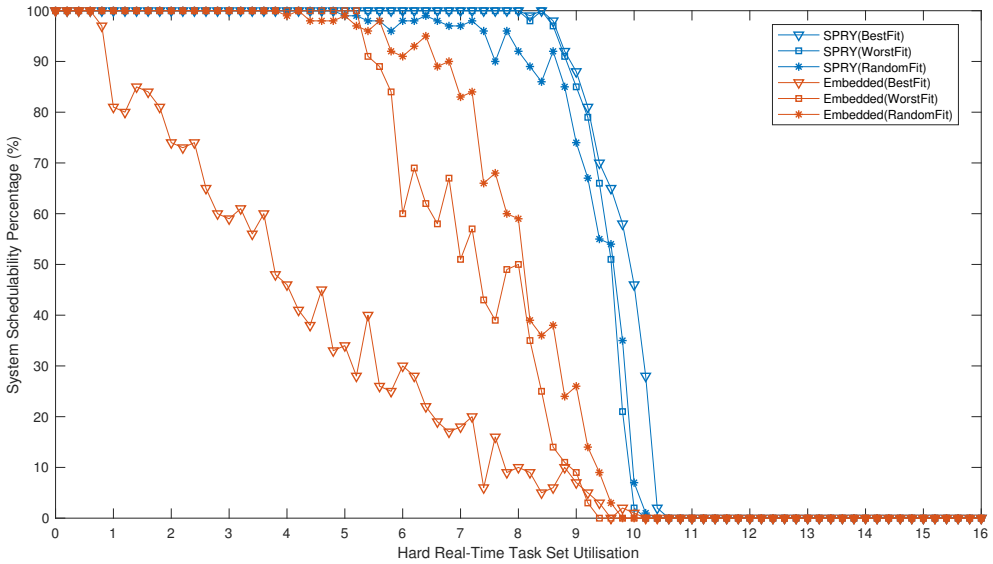


Figure 7.3: The schedulability of the system for a stream processing task with a period of 800, WCET of data processing of 4000, with 128 hard real-time tasks.

The results are illustrated by Figure 7.3. SPRY provides the best schedulability, when the hard real-time tasks are allocated with a best-fit algorithm. The reason is that, a best-fit allocation scheme for the hard real-time task allocation leaves more idle cores, compared to the worst-fit or random-fit. Creating execution-time server on an idle core can have the capacity equals to the period. The replenishment gap is zero, therefore reducing the pessimism in the analysis, which is described in Section 5.4.3. In addition, the server's capacity can be maximised as there is no other lower priority task to consider the 'double-hit' effect introduced by deferrable servers, see Section 5.1.1.

The performance of the traditional embedded approach is limited because the processing can not run at as high a priority as SPRY can. This approach performs even worse when using a best-fit allocation approach. The reason is that, when using the best-fit task allocation algorithm, the prologue task might be allocated with higher priority hard real-time tasks into the same core. The interference from these higher priority tasks increases the response time of the prologue, therefore, delaying the whole data processing. Using worst-fit allocates several higher priority tasks evenly in all the cores, while a random-fit allocation might results in the prologue task is allocated to a core with less higher priority tasks compared to the worst-fit. Therefore, the

random-fit performs better overall.

This section also runs experiments with different input parameters for the stream processing task, the results are similar.

7.3.1.1 Execution-Time Server Overhead

The experiments performed in the previous section assume zero system overheads. Of course, an implementation will have some overhead and this will reduce schedulability. However, for most part this overhead is identical in all the experiments. The exception is the overhead of supporting the servers. In this section we reduce the capacity of the server to reflect this overhead. A similar approach is performed in [47] to measure the impact of server implementation overhead. Note that, no overhead is added to the subtasks generated by the traditional embedded approach.

We have run the same experiment with overhead values of 2%, 5%, 10%, and 15% of its capacity, but the results are all similar. For example, with the overhead of 10% of the capacity, the result is shown in Figure 7.4. Typically, the implementation overhead is around $1\% \sim 2\%$, SPRY is still efficient with server implementation overhead.

Moreover, the experiment is re-run with overhead with absolute values of 1, 2, and 4 time units, similar results can be obtained. 4 milliseconds is a reasonable extreme big value for the overhead of an execution-time server [58].

This indicates that even with the addition of overheads to SPRY it can still provide a better performance than simpler solutions.

7.3.1.2 Scalability

This experiment introduces more cores, i.e., 128 cores in total, and considers the stream processing task with a period of 800, utilisation of 7000%. In addition, the hard real-time task set contains 1024 tasks in this experiment. The best system schedulability results of SPRY and the embedded approach are shown in Figure 7.5. As can be seen, SPRY can provide 100% system schedulability until the utilisation of hard real-time tasks is increased to 5500%, where the maximum available schedulable utilisation is 5800% (i.e., 12800%-7000%).

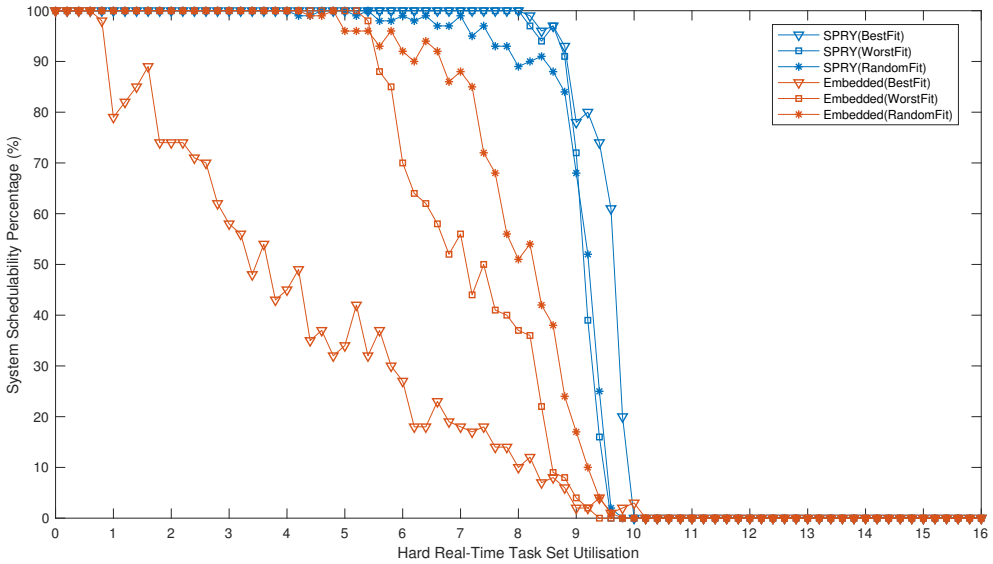


Figure 7.4: The schedulability of the system for a stream processing task with a period of 800, WCET of data processing of 4000, with 128 hard real-time tasks. The overhead of the server is 10% of its capacity.

7.3.1.3 Discussion

In summary, SPRY provides the best result when the hard real-time task are allocated with a best-fit algorithm, and overall it provides a better performance compared to the traditional embedded approach.

Arguably, when there is only one hard real-time task per core, the worst-fit algorithm can allocate them one per core, therefore, generating bound server to achieve utilisation of 100% for each core. However, this cannot be assumed to be a common practice in the real-world.

The performance of the traditional embedded approach is worse compared to SPRY because of the interference from higher priority tasks. When running the experiment with different stream period, e.g., 400 and 2000 time units, and different utilisation, such as 50%, 100%, and 1000%, SPRY still provides better results. The results are also similar, when the stream processing task is sporadic, or the deadline is less than its period.

Note however, when the utilisation of the stream processing task is relatively small, the difference between SPRY and the embedded approach decreases. The reason is that, the length of the whole execution is shorter, therefore, receiving less interference from the higher priority tasks. This conclusion can also be conducted by comparing Figure 7.5 and Figure 7.3 in this

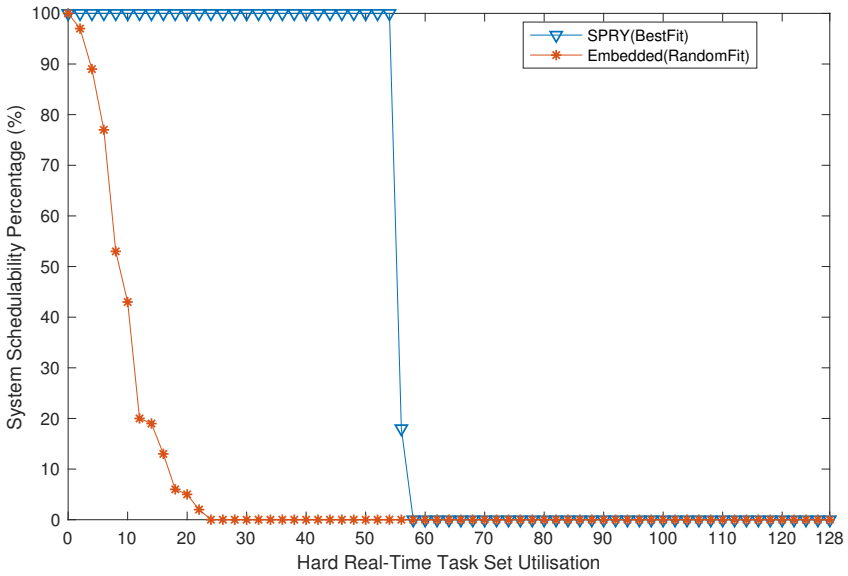


Figure 7.5: The schedulability of the system for a stream processing task with a period of 800, WCET of data processing of 56000, with 1024 hard real-time tasks.

section.

However, when the period of the stream processing task is relatively small, i.e., the priority of the stream processing task can be almost the highest in the whole system, the traditional embedded approach can perform better than SPRY occasionally (see Section 7.4.1).

7.3.2 Live Streaming Data Source Evaluation

This section considers the real-time processing of live streaming data sources, i.e., the data item is not splittable and requires a small processing time, using SPRY and the traditional embedded approach. Both of them uses the presented real-time micro-batching approach to group the data items into micro batches before processing. For the processing of micro batches, the prologue is configured to be zero as the micro batch splitting only requires passing references of data items to workers, epilogue values are configured using the same approach presented in the last section.

The experiment in this section uses input from data flows with different characteristics, such as different computation time required for processing each

data item, different arrival rates, and different latency requirements. The following data flows are investigated:

- The WCET for processing each data item of 10 time units, and MIT of 1 or 2 time units (i.e., the required utilisation is 1000% or 500%). The latency requirement is 30, 50, and 100 time units.

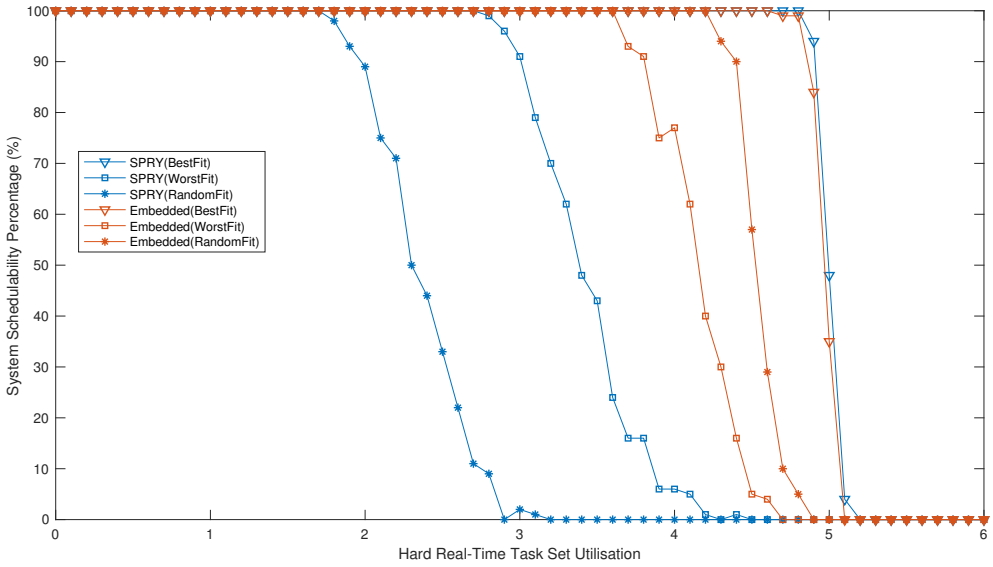
- The WCET for processing each item of 100 time units, MIT of 10 or 20 time units, latency requirement of 300, 500, and 1000 time units.

Note that, the MIT in this experiment is smaller than the required computation time for data items, because otherwise the micro batch size is always 1, which is a sporadic task that requires no parallel processing.

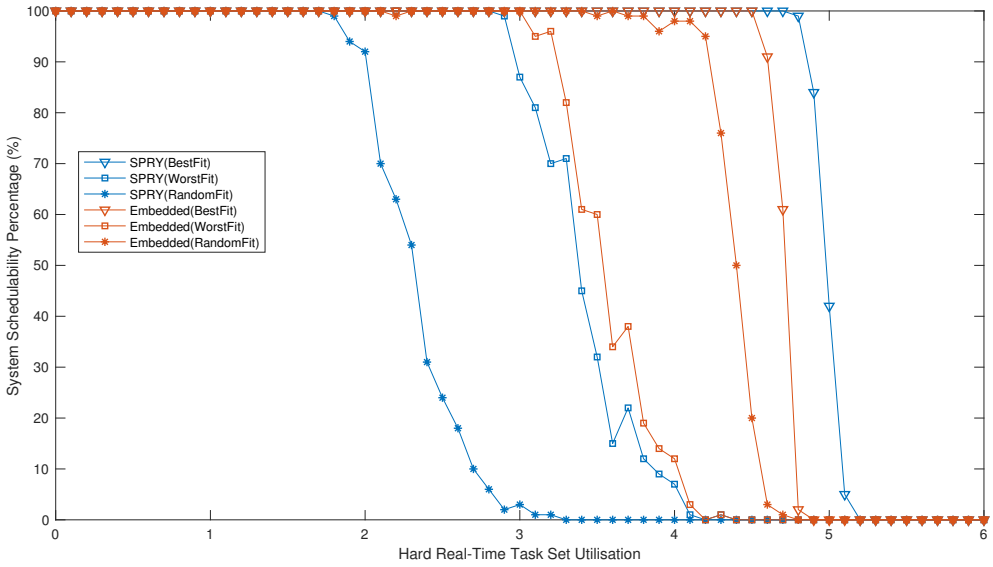
After running the experiments, the selected representative results are discussed as follows. The results of the experiment that input from a data flow (with a WCET for processing each data item of 10 time units, and MIT of 1 time units, and the latency requirement of 30 time units), and the experiment that input from a data flow (with a WCET for processing each data item of 100 time units, and MIT of 10 time units, and the latency requirement of 300 time units) are shown in Figure 7.6a and Figure 7.6b respectively.

As can be see from the figures, the SPRY and embedded approach with the best-fit provides a better schedulability in both experiments. In addition, the difference between SPRY and embedded approach increases in Figure 7.6b. This is because, the release period of the micro-batching in the second experiment is bigger than the first experiment, therefore, SPRY can potentially runs the stream processing at a higher priority compared to the embedded approach. For example, in the first experiment, the release period of a micro-batching approach with the maximum size of 11 is only 10 time units. This results in the subtasks generated by the embedded approach running almost at the highest priority, because the hard real-time task's period is normally distributed within 1 and 1000 time units. Therefore, the performance of SPRY and the embedded approach is very similar.

To test a looser latency requirement, we run the second experiment with a latency requirement of 1000 time units, and MIT of 10 time units. The results are shown in Figure 7.7. The difference among different approaches decreases as the latency requirement increases. The reason is that, increasing the latency results in extending the range of the maximum possible micro-batching size. This generates more options for the micro-batching, therefore, increasing the



(a) The schedulability of the system for a live streaming processing task with a MIT of 1 time units, WCET of processing each data item of 10 time units, latency of 30 time units, with 128 hard real-time tasks.



(b) The schedulability of the system for a live streaming processing task with a MIT of 10 time units, WCET of processing each data item of 100 time units, latency of 300 time units, with 128 hard real-time tasks.

Figure 7.6: The system's schedulability with different live streaming data sources.

chance to be schedulable.

Considering a relatively slow data flow, i.e., running the experiment with

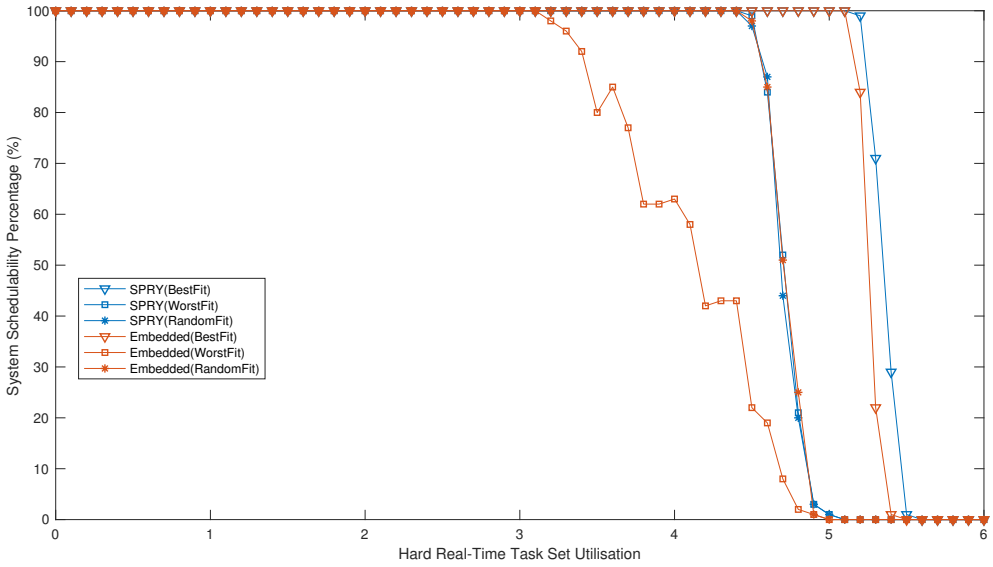


Figure 7.7: The schedulability of the system for a live streaming processing task with a MIT of 10 time units, WCET of processing each data item of 100 time units, latency of 1000 time units, with 128 hard real-time tasks.

a data flow with the MIT of data items of 2 time units, WCET for processing each item of 10 time units, and latency requirement of 30 time. Similar results can be obtained, and they are shown in Figure 7.8. The upper bound of the utilisation of the hard real-time task set is increased to around 10, as the requirement computation utilisation of the data flow is 500%.

7.3.2.1 Discussion

When processing a live streaming data source, the computation time required for processing each data item could be relatively small. SPRY still has a slight advantage compared to the embedded approach when the period of the micro-batching is small. However, the advantage increases when the period of the micro-batching increases, because SPRY can execute the stream processing at a higher priority by using execution-time servers with a small period.

7.4 Limitations

Throughout the evaluation given in this chapter, SPRY has consistently outperformed the traditional embedded system approach. However, there are some scenarios where SPRY does not perform well. This section considers two

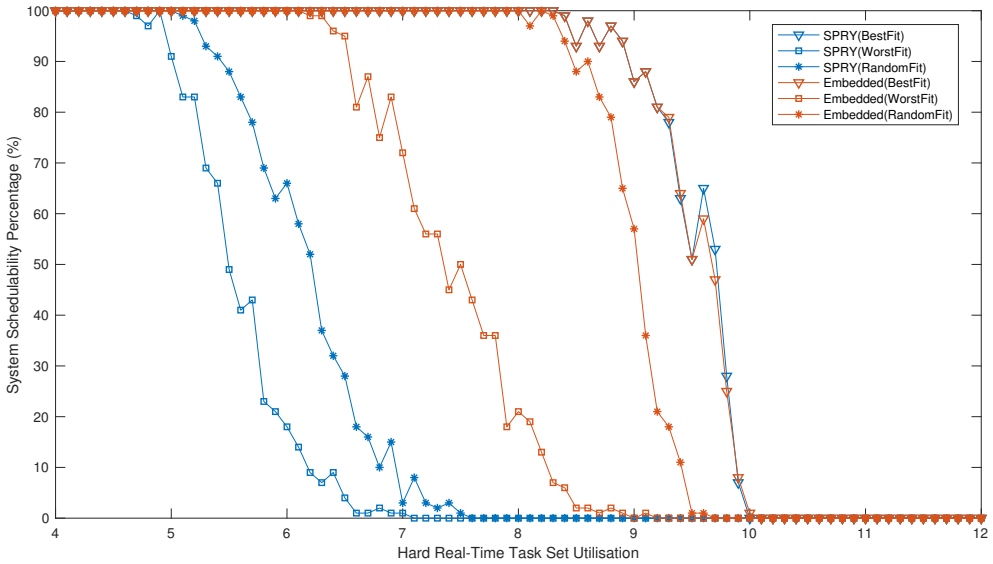


Figure 7.8: The schedulability of the system for a live streaming processing task with a MIT of 2 time units, WCET of processing each data item of 10 time units, latency of 30 time units, with 128 hard real-time tasks.

such cases: when the stream processing task in batch processing has a small period with a tight deadline, and when the latency requirement of live data processing is small. In the former case, the problem is caused by the limitation of the task allocation approach, and in the later it is due to the theoretical limitation of micro-batching.

7.4.1 Task Allocation Limitation

This sections discusses an extreme situation, where the stream processing task runs at almost the highest priority, and the traditional embedded approach might perform better than SPRY. Considering the experiment that inputs from a batched data source with a period of 15 time units, 16 data partitions (WCET of each is 10 time units), and the deadline of 10 time units. The prologue and epilogue are zero. The results are shown in Figure 7.9.

As can be seen that the embedded approach provides a better performance than SPRY after the utilisation of hard real-time tasks greater than 170%, because of the following reasons.

1. In this example, in order to process this batch within the deadline, 16 execution-time servers which can provide 10 time units' computation

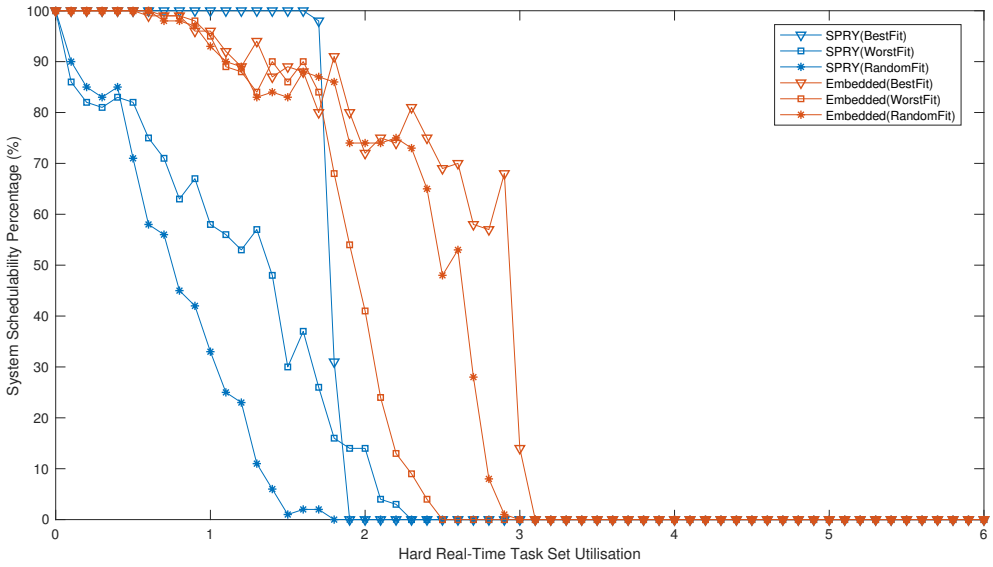


Figure 7.9: The schedulability of the system for a stream processing task with a period of 15 time units, 16 data partitions (WCET of each is 10 time units), and the deadline of 10 time units. The system also has 128 hard real-time tasks.

time within 10 time units are required, such as 16 servers (with $T_S = 15$, $C_S = 10$, running at the highest priority).

However, SPRY assumes the hard real-time tasks have been allocated, then generates a server per core. In this case, the hard real-time task are allocated into 2 cores with a best-fit, and the remaining 14 cores are idle. SPRY may generates 14 servers (with $T_S = 15$, $C_S = 15$, running at the highest priority). These servers are not be able to accommodate the processing of 16 data partitions within 10 time units.

2. Additionally, due to the 'double-hit' phenomenon introduced by the deferrable server, the capacity of the deferrable servers that are generated by SPRY is a smaller, compared to the WCET of those subtasks generated by the embedded approach. This makes the schedulability of SPRY even worse.

As SPRY is not restricted to a execution-time server technology, if the experiment is re-run by first creating periodic servers for SPRY, then allocating these servers, and finally allocating hard real-time tasks with a best-fit allocation, the results are shown in Figure 7.10. The schedulability of using

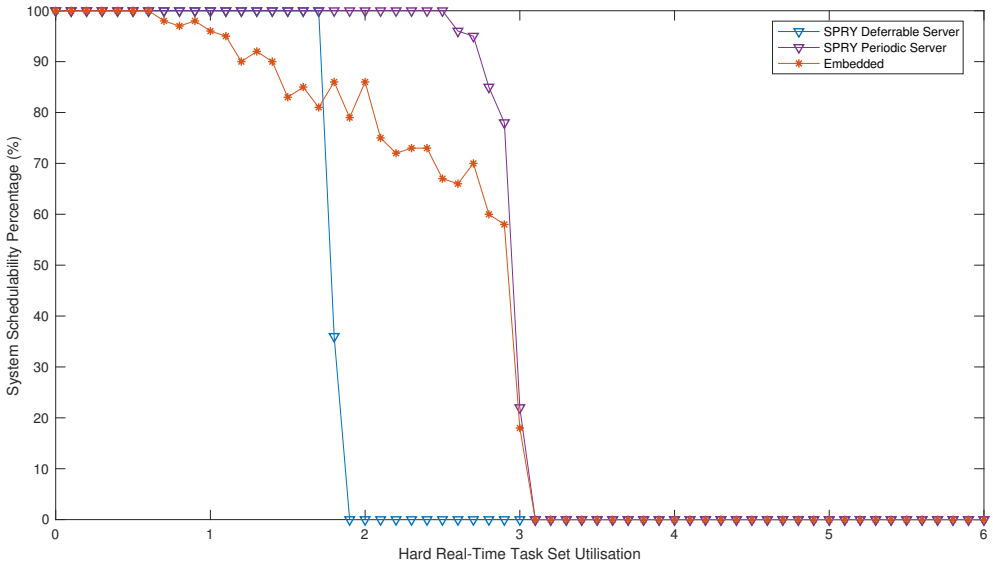


Figure 7.10: The schedulability of the system for a stream processing task with a period of 15 time units, 16 data partitions (WCET of each is 10 time units), and the deadline of 10 time units. The system also has 128 hard real-time tasks. Allocating servers before hard real-time tasks.

embedded approach is selected from the best result from best-fit, worst-fit and random-fit. As can be seen, the new approach provides the best system schedulability.

However, its difficult to determine which task allocation scheme for SPRY should be used to get the optimal performance. The task allocation in fully-partitioned systems has been proved to be NP-Hard [38], therefore, it is difficult to predict whether we should allocated hard real-time tasks ahead of SPRY servers, or not. To find a sub-optimal task allocation scheme for SPRY might require a discontinuous searching algorithm, such as simulated annealing [89], with heuristics. This is subject to the future work.

7.4.2 Limitations of Real-Time Micro-Batching

The real-time micro-batching approach can not schedule a live streaming data source, in which the latency requirement L is less than 2 times of the WCET of processing each item C^{item}.

Proof. When using real-time micro-batching,

1. The response time R^{batch} of processing of each micro-batching (even

though the size of which is 1) is greater or at least equal to C^{item}, i.e., $R^{batch} \geqslant C^{item}$.

2. According to the conditions required in Section 4.4.1, R^{batch} should be less than or equal to the interval of micro-batching timeout. This indicates the waiting time $Waiting^{item}$ for the first arrival item is at least R^{batch}, i.e., $Waiting^{item} \geqslant R^{batch}$.

3. For any data item, once the micro batch that contains this item is released for processing, the response time of processing this item $R^{item} \geqslant C^{item}$.

The latency of the first data item $L = Waiting^{item} + R^{item}$, where $Waiting^{item} \geqslant R^{batch} \geqslant C^{item}$ (1 and 2), and $R^{item} \geqslant C^{item}$ (3). Therefore, $L \geqslant 2 \times R^{item}$

Hence, the real-time micro-batching approach can not schedule a live streaming data source, for which the latency requirement $L < 2 \times C^{item}$. ■

However, the exception is that when MIT is greater than the C^{item}, the real-time micro-batching can still schedule it. In this case, the real-time micro-batching is equivalent to a sporadic task with a period of MIT, WCET of C^{item}.

7.5 Summary

The effectiveness and feasibility have been demonstrated by the examples presented elsewhere in the thesis and the prototype implementation. This chapter has tested our approach under the constraints/assumptions we made: using a single execution-time server per processor, and the critical instance in analysis. The experimental results in Section 7.1 has shown that using a single server per processor does not lose significant schedulability compared to using multiple servers. In addition, the analysis pessimism has been evaluated via comparing the results from the analysis with the simulation, in Section 7.2. The result shows that the amount of the analysis pessimism is small, therefore only little schedulability is lost.

In addition, the SPRY approach and its supporting analysis has been compared with traditional embedded approaches and its analysis for processing either batched and live streaming data sources in Section 7.3. The results

show that the presented SPRY approach provides a better schedulability in most cases.

As has been discussed in Section 7.4, when the stream processing activity has a very small period, i.e., itself running at the highest priority, the embedded approach occasionally provides a better results than SPRY, when there is a very tight deadline. A proposes solution has been given in Section 7.4.1, and with the proposed approach SPRY still provides a better result. In addition, the limitation of using micro-batching approach has also been discussed in Section 7.4. A possible solution to the limitation of the real-time micro-batching approach will be discussed in the next chapter, to direct the future work.

Chapter 8

Conclusions and Future Work

This thesis has presented an approach to integrate stream processing into real-time embedded systems, where the stream can be processed within the given time constraints, and all the remaining hard real-time tasks in the same system remains schedualable.

The success criteria described in Section 1.4 have all been met and are discussed below.

SC1 *The definition of a generic architecture of a real-time stream processing infrastructure, which supports both batched data and live streaming data sources processing with real-time constraints, and is programming language independent.*

Chapter 3 has developed a real-time stream processing task model, and an architecture that supports this model, along with its implementation requirements. This architecture is based on UML, component diagrams, etc., and does not assume any specific programming language.

SC2 *A process for engineering real-time systems that have both hard real-time and hard or soft stream processing components, which focuses on how this architecture is to be mapped to the physical platform and how the stream processing activity for both batched data and live streaming data sources is configured.*

Chapter 4 has defined the process, and examples given in that chapter along with the case study described in Section 5.6 have demonstrated how to use our approach, so that not only the data (or live streaming) can be processed within the given time constraints, but also all the hard

real-time tasks in the same system can still meet their deadlines. This includes the server parameter selection, and the pre-allocation of data partitions, determining the maximum batch size for the micro-batching when processing a live streaming data source.

SC3 *Response time analysis to determine the schedulability of stream processing for a batched data source, and latency for a live streaming data source.*

Response-time analysis has been derived in Chapter 5, to guarantee that the real-time requirements are met. The correctness of the analysis has been demonstrated in the evaluation Chapter.

SC4 *A framework for integrating real-time stream processing activities with hard real-time components, and its implementation using the Real-Time Specification for Java (RTSJ).*

Chapter 6 has presented a prototype implementation of this architecture (SPRY) using a modified Java 8 streams library and RTSJ, which gives the evidence that this real-time stream processing architecture can be implemented in practice.

SC5 *An evaluation that demonstrates that the proposed model is as effective as a more typical real-time systems model that does not use the stream processing paradigm.*

Experiments in Chapter 7 have evaluated the presented approach and its supporting analysis against traditional embedded approaches and its analysis for processing either batched and live streaming data sources. The results show that our presented approach provides a better schedulability in most cases. In addition, in Section 7.1 and 7.2, our approach has been tested under the constraints (single server per processor, as described in Section 3.4) and assumptions (the critical instance of the epilogue task, in the analysis described in Section 5.4) we made, the results show that the effectiveness of the presented approach is not undermined by these constraints/assumptions.

These success criteria have been met, and demonstrated the thesis hypothesis (stated in Section 1.3):

Programming languages or existing frameworks' support for

stream processing is insufficient for addressing real-time requirements. However, a generic architecture of a real-time stream processing infrastructure can be created to support predictable and analysable processing of both batched and live streaming data sources, and can be used in high-integrity real-time embedded systems. Moreover, the architecture can be implemented as a framework using Java, with the Java Fork/Join framework and the Real-Time Specification for Java.

8.1 Key Findings

This section summarises key findings of the thesis. The findings are grouped under three headings: the stream processing task model, the use of execution-time servers, the use of Java and the RTSJ.

Stream Processing Task Model

The presented real-time stream processing task model was developed based on the findings below.

1. **Data Parallel Versus Control Parallel** – According to the processor allocation, a stream processing pipeline can be either mapped across different processors, i.e., control parallel, or duplicated on each processor, i.e., data parallel. Section 2.3.4 discussed how the main disadvantage of control parallel is that the amount of parallel execution is limited by the structure of the algorithm. Moving to a parallel architecture can require the programmer to redesign their system. In addition, as the computation time required by each filter is different, therefore, a filter might be the bottleneck of the whole processing. If there is enough data items to process then this is not a problem with data parallel, therefore, data parallel is adopted by our real-time stream processing task model.

2. **Lazy Evaluation Versus Eager Evaluation** – The pipeline can be evaluated lazily (i.e., the actual data processing starts when only a filter that triggers the processing is invoked) or eagerly (i.e, any filter in this model triggers the processing immediately). In our implementation, lazy evaluation is adopted as the implementation is based on the Java 8 Stream processing framework. Lazy evaluation also provides potential

optimisation opportunities, such as avoiding unnecessary evaluation as discussed in Section 2.3.4. However, lazy or eager evaluation only affects the way to obtain the worst-case execution time, which is an input parameter to our analysis framework.

The stream processing input data can be either batched data or live streaming data. Data parallel with lazy evaluation is particularly suitable for parallel processing of batched data with a large volume. For the live streaming data source, the individual data items are group into micro batches to exploit potential parallel processing architectures, as individual items are not splittable. This reduces the analysis difficulties, and makes execution-time servers feasible in practice.

Execution-Time Servers for Predictability

This thesis provides the first use of execution-time servers to perform stream processing activities. Stream processing can be computationally-intensive. Hard real-time stream processing work load is known a priori, however, the soft real-time stream processing activity might make an unpredictable CPU demand. Execution-time servers runs stream processing task at a higher priority, minimises the response time of the prologue before the parallel processing, therefore reducing the whole response time to meet the deadline. Servers also bound the CPU demand made by stream processing tasks, so that all the hard real-time tasks in the same system remain schedulable.

Experimental results indicate that most spare capacity can be reclaimed by using a single server per processor, with limited overheads (see Section 7.5).

In addition, making a stream processing task *bound* to its server (i.e., aligning their releases), enhances the schedulability by avoiding the analysis pessimism. Moreover, a *bound* task is also free of 'double-hit' (see Section 5.1.1) introduced by higher priority deferrable servers, therefore maximising the capacity that can be reclaimed by deferrable servers. This observation has been proved, and is a supplement to the original RTA (as described in Section 5.1).

Java and the RTSJ

The SPRY implementation is based on the Java 8 Stream processing framework, and the Real-Time Specification for Java (RTSJ). However, the default Java 8 stream framework is difficult to be used in a real-time environment:

- The default Java framework partitions input data dynamically, and these data partitions are dynamically taken by worker threads using a work-stealing algorithm. This makes the analysis extra difficult, and even very pessimism (see Section 2.3.7 and 3.4).

- The restrictions (see Section 2.3.7 and 6.2.1) of ForkJoin framework that is the underlying processing infrastructure for the Java 8 streams, introduces difficulties to process streams at real-time priorities. For example, the `ForkJoinWorkerThread` in a ForkJoin thread pool extends the standard Java thread, and therefore it is unable to create real-time threads. In addition, the ForkJoin pool checks if current thread is an instance of `ForkJoinWorkerThread`, if not, the ForkJoin pool transfers the processing of data partitions to a global default pool, which is hard-coded.

We modified the data partitioning and allocation algorithm by directly editing the source code of the ForkJoin pool, so that the processing is predictable and analysable. In addition, in order to integrate Java 8 streams to RTSJ, we also modified the source of the the library, so that the ForkJoin pool can run at real-time priorities. We have suggested changes to JSR 282[1] to circumvent this problem by allowing a Java thread to execute at a real-time priority, which have now been adopted.

8.2 Future Work

There are several possible areas of future research based on the work presented in this thesis.

Live Streaming Data Processing without Micro-Batching

As has been discussed in Section 7.4.2, the micro-batching approach can not schedule a live streaming data source with the latency requirement L, which is less than 2 times of the WCET of processing each item C^{item}. However, if we use the MIT of the data items to create servers/sporadic tasks, the period might be very small, and therefore not practical.

We currently propose an approach that allows each data item to be processed individually from a live streaming data source, moreover, our server

[1]The JCP Expert Group has released a new version of the RTSJ (Version 2.0) in early 2017. This version is compatible with Java 8.

generation algorithm and analysis equations can be re-used. Our preliminary work obtains a schedulable micro-batching timeout value, then allocates the item to its target processor once it arrives at the system. The allocation is determined by its arriving window, and execution-time servers are generated based on the timeout value.

In addition, note that, a dynamic approach where each workers tries to take items from a shared buffer, e.g., the approach used by Reactive Streams [16], has been investigated with preliminary experiments [72]. The result shows that this dynamic approach is not adaptable. This is because, for example, a worker that is running (typically not at the highest priority) on a busy core takes an item from a live streaming data source, the higher priority hard real-time tasks might pre-empt the processing of this item for an interval so that the deadline is missed.

Multiple Streams

As has been discussed in Section 4.5, the current approach targets a single stream processing task. However, multiple real-time stream processing tasks can be supported by extending the current approach. For the priority ordering of the stream processing tasks, a possible solution would be using the deadline/latency monotonic priority assignment. As the sever generation approach presented in Section 4.3.1 is a greedy algorithm, which searches the maximum possible capacity for each server in each processor. However, a particular stream processing task might not requires the entire capacity generated by the algorithm. As also discussed in Section 7.3, a best-fit allocation performs overall better. Therefore, the possible solution might be reduce the capacity from the execution-time server generated for the last processor, until the system is just schedulable. Then generate the execution-time servers for another stream processing task using the same approach.

Task Allocation

As has been discussed in Section 7.4, when the stream processing activity has a very small period (i.e., itself running at the highest priority) and a very tight deadline, the presented approach does not always provide a better result when the total utilisation of hard real-time tasks is high. This issues is caused by the task allocation algorithm, and a possible solution has been given in Section 7.4.1.

Its difficult to determine the optimal task allocation scheme for our approach, as task allocation in fully-partitioned systems has been proved to be NP-Hard [38]. However, a sub-optimal task allocation scheme might be found by employing a discontinuous searching algorithm, such as simulated annealing based on an existing work [89]. In addition, genetic algorithm might also be employed.

Supporting NUMA/Distributed/Network-on-Chip Platforms

The current analysis mainly considers SMP platforms, the analysis might be extended to support NUMA architectures. For a distributed system, it requires the current analysis takes data transmission in the network into account, and it could be based on an existing work [88].

Moreover, network-on-chip is designed for the many-core processors, in order to mitigate the bottleneck introduced by the bus in traditional CPUs. For real-time stream processing, messaging delays in a on-chip network requires to be taken into account for the analysis, possible analysis would be proposed by extending an existing work [59].

For a more complicated pipeline (or graph) of a stream processing task, which commonly appears in a distributed system, there are multiple synchronisation stages through the whole processing. Our current analysis could be extended to support a multi-stage pipeline, as has been discussed in Section 7.2 the amount of analysis pessimism is small. In addition, either the period of whole pipeline/graph processing could be used for the sever generation, or breaking the pipeline to multiple sub-pipelines, then treat them as multiple stream processing tasks, and generate servers for each one of them.

Global Scheduling

Our current architecture targets systems with fully-partitioned scheduling. Without pinning each worker to a processor, the architecture itself is possible to be implemented as a framework that executes on a globally scheduled system. However, this requires the analysis to be extended so that the stream processing task model on a globally scheduled system can be analysed.

Supporting GPU

General-purpose GPUs (GPGPUs) are often used to accelerate the processing of several stream processing workloads, such as image recognition. The presented approach could be used for real-time stream processing with GPUs. This might require additional work for the analysis, such as the analysis for copying data from system memory to the GPU memory. In addition, a real-time GPU scheduling framework, such as [53], might be required for the implementation.

Implementation in Other Programming Languages

In order to further demonstrate that the presented real-time stream processing architecture is generic, it is possible to implement this architecture using another programming language, such as Ada, or C with real-time POSIX. It allows the real-time stream processing paradigm to be introduced as a new functionality to existing embedded/real-time systems.

Data Allocation

The presented approach that describes how to configure a real-time stream processing task assumes that the worst-case processing time of a batch's (or micro batch's) partition is not data sensitive. If it is, then the pre-allocation of partitions to servers might not be appropriate and a more dynamic allocation might be required to improve the efficiency for the soft real-time case.

8.3 Closing Remarks

Modern real-time embedded systems often involve computational-intensive data processing algorithms to meet their application requirements, which increases the use of multiprocessor platforms. The stream processing programming model allows user to construct concurrent data processing programs to exploit the parallelism available on these architectures.

This thesis has proposed a generic real-time stream processing architecture, which allows parallel processing of both batched and live streaming data sources in a real-time system that also hosts hard real-time tasks, so that not only the data can be processed within the given time constraints, but also all the hard real-time tasks remain schedulable. An approach to configuring

applications for the architecture and the corresponding schedulability analysis has been developed.

This architecture and its analysis has been evaluated, and the result shows that the presented approach is effective. Together with the motivating case study and the prototype implementation of the SPRY framework, we have provided the evidence that the presented approach is feasible.

Appendices

Appendix A

Two-Way ANOVA Analysis of Benchmarking Results

Analysis of variance (ANOVA) is a general statistical technique, which separates the total variation in a set of measurements into the variation that is caused by the real differences among the alternatives being compared, and the variation introduced by the measurement noise [66].

The two-way ANOVA examines two different independent factors on one dependent variable, and determines both the main effect of contributions of each independent factor and if there is an interaction effect between them [21].

Using the approach proposed in [74], the goal of performing two-way analysis to the benchmarking results is to prove that both implementation framework, and the number of processors have an impact on the benchmark's response times, and also there is significant interaction between them, i.e., the Java 8 Stream framework and StreamIt have different efficiency.

The null hypothesis is made that both factors (implementation framework/language, and the number of processors) have no effect on the benchmark's response time, i.e., its efficiency. The sample size is 30, and the alpha value is 0.05, i.e., a statistical significance level of 95%. After performing the two way ANOVA analysis using MATLAB, the results are presented in Section A.1 and Section A.2.

The notations used by the following sections are summaries as follows:

- SS – the sum of squares due to each source.
- df – the degrees of freedom associated with each source.

- MS– the mean squares, which equals to $\frac{SS}{df}$.
- F – the ratio of the variance calculated among the means to the variance within the samples.
- Prob>F – the computed probability that the null hypothesis holds. If this value is close to zero, this casts doubt on the associated null hypothesis.

A.1 SAR Benchmarking Result Analysis

Performing the two-way ANOVA on the response times of the SAR benchmarks, the results are represented in Table A.1. MATLAB indicates that the probabilities are all zero.

The following F values are taken from the table [7] of F probability distribution for a given level of statistically significance.

- $F_{Framework}(4, (299 - (4 + 1 + 4))) = 2.3719$
- $F_{Processors}(1, (299 - (4 + 1 + 4))) = 3.8415$
- $F_{Interaction}(4, (299 - (4 + 1 + 4))) = 2.3719$

As we can see, $F_{Framework}$ and $F_{Processors}$ are much larger than the maximum value in the F distribution table. It indicates that the hypothesis is rejected, i.e., not only the implementation framework, but also the parallelism/processors have effect on the efficiency. In addition, $F_{Interaction}$ is also much later than the maximum value in the F distribution table, indicating that there is an interaction effect between them, i.e., Java 8 streams and StreamIt have different efficiency in this SAR stream processing benchmark.

A.2 Filter Bank Benchmarking Result Analysis

Performing the two-way ANOVA on the response times of the filter bank benchmarks, the results are represented in Table A.2. The result indicates that the probabilities are all very close to zero, such as $4.36603936546411 \times 10^{-184}$.

Employing the same approach, the F values are taken from the F distribution table as follows.

- $F_{Framework}(4, (299 - (4 + 1 + 4))) = 2.3719$

Table A.1: The two-way ANOVA results for SAR Benchmarks.

Source	SS	df	MS	F	Prob>F
Framework	222254081083.653	4	55563520270.9133	5022004.25177113	0
Processors	53240280623.6033	1	53240280623.6033	4812022.60680358	0
Interaction	2705453794.21343	4	676363448.553358	61131.8379004087	0
Total	278203024065.237	299			

Table A.2: The two-way ANOVA results for Filter Bank Benchmarks.

Source	SS	df	MS	F	Prob>F
Framework	292710497.646667	4	73177624.4116667	89994.9368915609	0
Processors	4010626.56333333	1	4010626.56333333	4932.32852753358	$4.36603936546411 \times 10^{-184}$
Interaction	2519635.75333321	4	629908.938333303	774.671432811610	$2.08424833416382 \times 10^{-153}$
Total	299476567.796667	299			

- $F_{Processors}(1, (299 - (4 + 1 + 4))) = 3.8415$
- $F_{Interaction}(4, (299 - (4 + 1 + 4))) = 2.3719$

As can be seen, the similar conclusion can be drawn that Java 8 streams and StreamIt have different efficiency in the filter bank stream processing benchmark.

Appendix B

Response Time Analysis for the Traditional Embedded Approach

The analysis for the traditional embedded approach is similar to the approach for SPRY. However, all the timing analysis is based on RTA (see Section 5.1), rather than RTA under execution-time servers.

The execution of a subtasks generated by the traditional embedded approach can be divided into the following phases:

1. Sequential before the data splitting, and the splitting (i.e., the prologue subtask),
2. The parallel stream processing, and
3. The epilogue subtask executing.

The worst-case execution time of the phase 1, and phase 3 are represented by C_i^1, and C_i^3. The The worst-case execution time required for processing an data partition in phase 2 is C_i^{item}.

The critical instance occurs when

- The prologue subtask is released at the same time, i.e., time 0, with all the hard real-time tasks in the prologue processor.
- Each remaining subtask and all the hard real-time tasks are release at time $R_{Prologue}$.

- The epilogue subtask is released at the same time with all the hard real-time tasks in the same processor, in the case where the epilogue subtask is not merged into another task.

Consider a stream processing task τ_i, where its prologue subtask is executed by processor P_{τ_i}. The parallel processing uses processors P_0 to P_{n-1}, including P_{τ_i}, where $n \geq 1$. In a fully partitioned system, the prologue (phases 1) are performed on processor P_{τ_i}, then all the allocated processors are used for the parallel processing, and finally the epilogue is executed on processor P_{τ_i}. Phase 3 only starts after all parallel sections of phase 2 are complete.

The prologue can be analysed as a whole using the analysis techniques described in Section 5.1 with jitter ($J_i = 0$), load ($C_i = C_i^1$), and the period of T_i, i.e., the stream processing task's period; the worst-case response time R_i^1 of executing the prologue can be calculated.

The parallel processing of the data partitions starts once the prologue is finished. According to the allocation scheme (i.e., worst-fit or best-fit), we can calculate the worst-case execution time for the data partitions that were allocated to each processor. For example, n partitions were allocated to a processor P_i, then the worst-case execution time for data processing in P_i can be calculated by:

$$C_{P_i}^2 = C_i^{item} \times n$$

Then, the next step is to calculate the time when the parallel data processing in each processor completes:

- For processor P_{τ_i} the prologue and the allocated data processing can be treated as a whole. Therefore the worst-case response time $R_i^{2,P_{\tau_i}}$ of this whole execution in this processor can be calculated by using the analysis techniques described in Section 5.1 with jitter = 0, load = $C_i^1 + C_{P_{\tau_i}}^2$, and a period of T_i.

- For each of the other processors $P_i, P_i \neq P_{\tau_i}$, the processing is released at R_i^1.The worst-case response time R_i^{2,P_i} for the stream processing in this processor can be calculated using the analysis techniques described in Section 5.1 with jitter = 0, load = $C_{P_i}^2$, and a period of T_i.

When considering the response time of each individual data item. For any data item, the response time of this item can be calculated by removing the workload of processing all items after this item in this processor and then repeating the above steps.

The response time of the parallel data processing phase is the maximum of all involved processors:

$$R_i^2 = \max(R_i^{2,P_{\tau_i}}, \max(R_i^{2,P_i} + R_i^1)), where P_i \neq P_{\tau_i}$$

Finally we consider the epilogue subtask (phase 3). In the case where the epilogue task is merged into another task, the response time of the stream processing is R_i^2. Otherwise, the worst-case situation is that is when it is released (after the barrier synchronisation detailed above), all the higher priority hard real-time tasks are release at the same time. Therefore, the worst-case response time R_i^3 for phase 3 can be calculated using the analysis techniques described in Section 5.1 with the jitter = 0, load = C_i^3, and period of T_i.

Finally, the worst-case response time of τ_i is calculated by:

$$R_i = R_i^2 + R_i^3$$

Note that, the assumed critical instance might not ever occur, therefore, introducing pessimism in the analysis. However, using the same approach presented in Section 7.2, the results of the analysis and the simulation are shown in Figure B.1. As can be seen, the amount of pessimism is small, the accuracy of the analysis is acceptable.

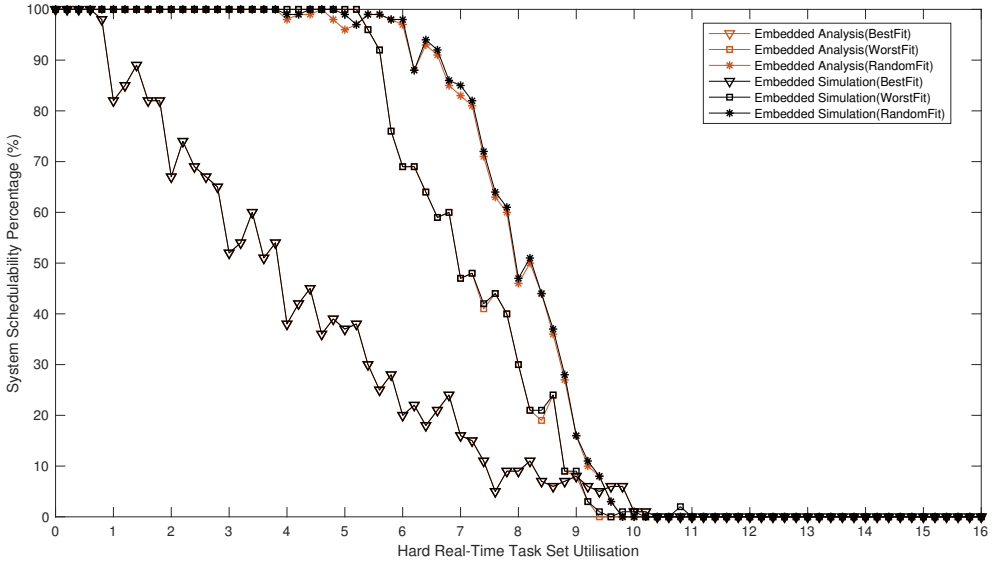


Figure B.1: The accuracy of the presented analysis approach for the traditional embedded approach.

Bibliography

[1] Apache Spark - Lightning-Fast Cluster Computing. http://spark.apache.org/. Accessed January 12, 2017.

[2] Apache Samza. http://samza.apache.org. Accessed January 12, 2017.

[3] Apache software foundation: Apache hadoop. http://hadoop.apache.org/. Accessed January 12, 2017.

[4] Apache Storm. http://storm.apache.org/. Accessed January 12, 2017.

[5] Borealis - distributed stream processing engine. http://cs.brown.edu/research/borealis/public/. Accessed January 12, 2017.

[6] Download StreamIt. http://groups.csail.mit.edu/cag/streamit/restricted/files.shtml. Accessed January 12, 2017.

[7] F-Distribution Tables. http://www.socr.ucla.edu/Applets.dir/F_Table.html. Accessed May 8, 2017.

[8] HPEC Challenge. http://www.omgwiki.org/hpec/files/hpec-challenge/index.html. Accessed April 30, 2017.

[9] JamaicaVM — aicas.com. https://www.aicas.com/cms/en/JamaicaVM. Accessed January 12, 2017.

[10] Java SE 8: Lambda Quick Start. http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html. Accessed January 24, 2018.

[11] Java Software — Oracle. https://www.oracle.com/java/index.html. Accessed May 16, 2017.

[12] JDK 8 Project. https://jdk8.java.net/. Accessed January 12, 2017.

[13] JEP 107: Bulk Data Operations for Collections. http://openjdk.java.net/jeps/107. Accessed January 12, 2017.

[14] Litmus-rt: Linux testbed for multiprocessor scheduling in real-time systems. http://www.litmus-rt.org. Accessed March 12, 2017.

[15] Parallel LINQ (PLINQ). https://msdn.microsoft.com/en-us/library/dd460688(v=vs.110).aspx. Accessed March 1, 2017.

[16] Reactive Streams. http://www.reactive-streams.org/. Accessed January 12, 2017.

[17] RTSJ Main Page. http://www.rtsj.org/. Accessed January 12, 2017.

[18] RTSYork/Real-Time-Stream-Processing. https://github.com/RTSYork/Real-Time-Stream-Processing. Accessed May 1, 2017.

[19] Spark Streaming — Apache Spark. http://spark.apache.org/streaming/. Accessed January 12, 2017.

[20] Spark streaming programming guide. https://spark.apache.org/docs/latest/streaming-programming-guide.html. Accessed February 02, 2017.

[21] Stats: Two-Way ANOVA. https://people.richland.edu/james/lecture/m170/ch13-2wy.html. Accessed May 8, 2017.

[22] Stream Processing in MATLAB. https://www.mathworks.com/discovery/stream-processing.html. Accessed May 8, 2017.

[23] StreamIt-Benchmarks. http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml. Accessed January 12, 2017.

[24] StreamIt-Research. http://groups.csail.mit.edu/cag/streamit/shtml/research.shtml. Accessed January 12, 2017.

[25] Understanding the Parallelism of a Storm Topology. http://storm.apache.org/releases/2.0.0-SNAPSHOT/Understanding-the-parallelism-of-a-Storm-topology.html. Accessed March 1, 2017.

[26] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and

architecture for data stream management. *The VLDB Journal—The International Journal on Very Large Data Bases*, 12(2):120–139, 2003.

[27] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[28] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. Stream: the stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 665–665. ACM, 2003.

[29] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[30] N. Audsley, A. Burns, M. Richardson, and A. Wellings. Hard real-time scheduling: The deadline monotonic approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software, Atlanta, GA, USA*, 1991.

[31] N. C. Audsley, Y. Chan, I. Gray, and A. J. Wellings. Real-Time Big Data: the JUNIPER Approach. 2014.

[32] M. AYOUB. *EMBEDDED AND REAL TIME SYSTEM DEVELOPMENT: A Software Engineering Perspective.* SPRINGER-VERLAG BERLIN AN, 2016.

[33] 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.

[34] G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, pages 68–78. IEEE, 1999.

[35] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010.

[36] I. Buck. Brook Specification v0.2. *Stanford University Press, graphics.stanford.edu/papers/brookspec-v0.2/brookspec-v0.2.pdf*, 2003.

[37] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 777–786. ACM, 2004.

[38] A. Burns and A. Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001.

[39] A. Burns and A. Wellings. *Analysable Real-Time Systems: Programmed in Ada*. CreateSpace Independent Publishing Platform, 2016.

[40] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Data Engineering*, page 28, 2015.

[41] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *ACM Sigplan Notices*, volume 45, pages 363–375. ACM, 2010.

[42] Y. Chan, A. Wellings, I. Gray, and N. Audsley. A Distributed Stream Library for Java 8. *IEEE Transactions on Big Data*, 2017.

[43] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. B. Zdonik. Scalable distributed stream processing. In *CIDR*, volume 3, pages 257–268, 2003.

[44] Y. Choi, Y. Lin, N. Chong, S. Mahlke, and T. Mudge. Stream compilation for real-time embedded multicore systems. In *Code generation and optimization, 2009. CGO 2009. International symposium on*, pages 210–220. IEEE, 2009.

[45] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Nsdi*, volume 10, page 20, 2010.

[46] R. Davis and A. Burns. An investigation into server parameter selection for hierarchical fixed priority pre-emptive systems. In *16th International Conference on Real-Time and Network Systems (RTNS 2008)*, 2008.

[47] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 10–pp. IEEE, 2005.

[48] R. I. Davis and A. Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Systems*, 47(1):1–40, 2011.

[49] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys (CSUR)*, 43(4):35, 2011.

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

[51] L. Dobrica and E. Niemela. A survey on software architecture analysis methods. *IEEE Transactions on software Engineering*, 28(7):638–653, 2002.

[52] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM international symposium on high performance distributed computing*, pages 810–818. ACM, 2010.

[53] G. A. Elliott, B. C. Ward, and J. H. Anderson. Gpusync: A framework for real-time gpu management. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, pages 33–44. IEEE, 2013.

[54] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[55] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *ACM SIGOPS Operating Systems Review*, 40(5):151–162, 2006.

[56] 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.

[57] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[58] R. Inam, J. Mäki-Turja, M. Sjödin, S. M. Ashjaei, and S. Afshar. Support for hierarchical scheduling in freertos. In *Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on*, pages 1–10. IEEE, 2011.

[59] L. S. Indrusiak. End-to-end schedulability tests for multiprocessor embedded systems based on networks-on-chip with priority-preemptive arbitration. *Journal of systems architecture*, 60(7):553–561, 2014.

[60] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, volume 41, pages 59–72. ACM, 2007.

[61] 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.

[62] S. M. Kuo, B. H. Lee, and W. Tian. *Real-time digital signal processing: fundamentals, implementations and applications*. John Wiley & Sons, 2013.

[63] 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.

[64] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[65] 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.

[66] D. J. Lilja. *Measuring computer performance: a practitioner's guide*. Cambridge university press, 2005.

[67] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[68] C. D. Locke, D. R. Vogel, and T. J. Mesler. Building a predictable avionics platform in Ada: a case study. In *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pages 181–189. IEEE, 1991.

[69] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[70] 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.

[71] H. 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.

[72] H. Mei, I. Gray, and A. Wellings. A java-based real-time reactive stream framework. In *Real-Time Distributed Computing (ISORC), 2016 IEEE 19th International Symposium on*, pages 204–211. IEEE, 2016.

[73] H. Mei, I. Gray, and A. Wellings. Real-time stream processing in java. In *Ada-Europe International Conference on Reliable Software Technologies*, pages 44–57. Springer, 2016.

[74] H. Mei and A. Wellings. Using JetBench to Evaluate the Efficiency of Multiprocessor Support for Parallel Processing. In *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*, page 47. ACM, 2014.

[75] A. Moreira, P. Prats-Iraola, M. Younis, G. Krieger, I. Hajnsek, and K. P. Papathanassiou. A Tutorial on Synthetic Aperture Radar. *IEEE Geoscience and remote sensing magazine*, 1(1):6–43, 2013.

[76] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177. IEEE, 2010.

[77] L. Newmeyer, D. Wilde, B. Nelson, and M. Wirthlin. Efficient processing of phased array radar in sense and avoid application using heterogeneous computing. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, pages 1–8. IEEE, 2016.

[78] R. O'Rourke. Air force F-22 fighter program: Background and issues for congress. DTIC Document, 2009.

[79] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified modeling language reference manual, the*. Pearson Higher Education, 2004.

[80] S. S. Sant'Anna. Final Operating System with Real-Time Support. Technical report, JUNIPER PRJECT, 2015.

[81] D. Spinellis. Notable design patterns for domain-specific languages. *Journal of systems and software*, 56(1):91–99, 2001.

[82] 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.

[83] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic Task Scheduling for Hard Real-Time Systems. *Real-Time Systems*, 1:27–69, 1989.

[84] R. Stephens. A Survey of Stream Processing. *Acta Informatica*, 34:491–541, 1997.

[85] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47, 2005.

[86] 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.

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

[88] K. Tindell and A. Burns. Guaranteed message latencies for distributed safety-critical hard real-time control networks. *Dept. of Computer Science, University of York*, 1994.

[89] K. W. Tindell, A. Burns, and A. J. Wellings. Allocating hard real-time tasks: an np-hard problem made easy. *Real-Time Systems*, 4(2):145–165, 1992.

[90] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[91] A. Wellings. *Concurrent and real-time programming in Java*. John Wiley & Sons, 2004.

[92] 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.

[93] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, volume 8, pages 1–14, 2008.

[94] M. Zaharia. An architecture for fast and general data processing on large clusters. Technical report, Technical Report No. UCB/EECS-2014-12, 3 Feb 2014. http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-12.html, 2014.

[95] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[96] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.