Selecting Execution-Time Server Parameters for Real-Time Stream Processing Systems

HaiTao Mei, Ian Gray and Andy Wellings

University of York, UK
(hm857, Ian.Gray, Andy.Wellings)@york.ac.uk

Abstract. This paper considers the integration of the stream processing programming model into a hard real-time system. It proposes the use of execution-time servers as a mechanism for giving good response times to the stream processing components without compromising the guarantees given to the hard real-time components. Attention is focused on selecting the parameters of the servers so that any spare processor utilisation is dedicated to the stream processing activities at a high priority. A server parameter selection algorithm is presented and evaluated.

Keywords: Real-Time Stream Processing, Server Parameter Selection

1 Introduction

In real-time systems, tasks are often classified as being hard or soft. Hard real-time tasks are those where it is absolutely imperative that responses occur within the specified deadline. Soft real-time tasks are those where response times are important but the system will still function correctly if deadlines are occasionally missed [2].

Due to increased computational demands, modern real-time systems now execute on multiprocessor platforms, and potentially as part of a cluster-based architecture. The stream processing programming model [14] is a programming model that aims to facilitate the construction of concurrent programs to exploit the parallelism available on these architectures. The paradigm is particularly suited to applications that have components that must process large volumes of data; so-called Big Data applications [8]. Although many of these applications have real-time requirements [1] most stream processing architectures are not targeted towards these requirements.

In this paper we address the integration of a stream processing paradigm into a real-time environment that also has hard real-time components. Processing streaming data is often computationally intensive and it is difficult to predict the volume and the cost of processing the data. Hence, we consider the components processing the data to be soft real-time, and thus primarily requiring good response times.

Execution-time servers [2] are a technique that is used in the real-time community to ensure that soft tasks that might demand unbounded CPU time still exhibit good response times. Servers limit the impact of soft tasks on hard tasks, so that the hard tasks can be guaranteed to meet their deadlines. However, the algorithms needed to determine the number of servers required and their parameters have a time complexity that is exponential [2]. Furthermore they have not previously been used in a stream processing environment. This paper provides three main contributions:

1. A proposal for using execution-time servers for processing real-time streaming data in soft real-time.
2. A fast $O(n^2)$ algorithm for selecting the (sub-optimal) number of servers and their parameters, which maximises the processor utilisation and gives good response times to stream processing activities.
3. Experimental evaluation showing the efficacy of the algorithm.

This paper is structured as follows. Section 2 summarises related work in the area of execution-time servers and the selection of the their parameters. This is followed by an overview of the architecture of the York Stream Processing Framework (SPRY) described in Section 3, which provides the context for this work. Section 4 describes how to generate the servers for handling real-time stream processing tasks, and the proposed heuristic that selects parameters for the generated servers. Section 5 evaluates our heuristic, and finally we give our conclusions. Throughout this paper we use the following notations: C = execution time, D = deadline, T = period, U = utilisation, and P = priority.

2 Related Work of Execution-Time Servers

In real-time systems that consist of both hard and soft components it is essential that the resources required by hard real-time components are reserved so that their deadlines are guaranteed. The term *spare capacity* is used to indicate any resources that are not required by the hard real-time tasks, and so therefore are available for the execution of soft real-time tasks [4]. The problems tackled in [4] are to identify the amount of spare capacity, and to make that capacity available at run-time for the execution of soft tasks. The former is determined using schedulability analysis techniques, and the later is supported by execution-time servers (sometimes called aperiodic or sporadic servers).

Execution-time servers are logical periodic tasks that have a set period, priority and capacity. The capacity is the amount of execution time that the server is allowed to execute during each period. Once the capacity is used, the server must wait for it to be replenished. When a server has capacity it is able to perform computation on behalf of its allocated tasks. The server is logical as it may not exist at run-time but be represented by some underlying Operating System resource reservation protocol.

There are many different types of execution time servers, see [11] for a review. The POSIX standard supports the sporadic server [7, 13]. 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 it 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 consumed is replenished at a time one server period on from its point of consumption.

The deferrable server [7, 13] allows a new logical thread to be introduced at a particular priority level. If the server's period and capacity are appropriately configured, all the periodic tasks in the system remain schedulable even if the server fully 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.

One of the main problems with the use of execution-time servers is the calculation of their parameters so that the spare capacity can be effectively used. An investigation into server parameter selection for fixed priority preemptive systems is given by Davis and Burns [3] in the context of hierarchical scheduling. Their work proposed using servers to serve applications that have multiple real-time tasks, with the goal of low overall system utilisation. The paper provided a set of algorithms that determine the optimal value for one server parameter. Davis and Burns illustrate how to use exhaustive search to find optimal servers, and show that the general problem is intractable.

An alternative approach to execution-time servers is to use a slack stealing algorithm [6]. However, this approach has to be used in conjunction with an on-line scheduler and has high overheads.

In this paper, we use deferrable servers because they have a simple run-time representation. However, the type of server is not essential to our approach.

3 The Architecture of the York Stream Processing Framework (SPRY)

A stream processing system consists of a collection of modules that compute in parallel and communicate via channels [14]. Modules can be either *source capturing* (that pass data from a source into the system), *filters* (that perform atomic operations on the data) or *sinks* (that either consume the data or pass it out of the system). Real-time stream processing systems are stream processing systems that have time constraints associated with the processing of data elements as they flow through the system from source to sink. In general, the data sources of stream processing systems can be classified into two types [9]: batched and streaming. A batched data source is where the data is already present in memory, and its content and size will not change during processing. A streaming data source represents data that arrives dynamically, its content and size will change with time.

The York Stream Processing Framework (SPRY) was developed to support a streaming data paradigm for both batched and streaming data sources in real-time, which targets shared memory multiprocessor platforms.

Initially, SPRY integrates Java 8 Streams and the Real-time Specification for Java (RTSJ) to support real-time batched data processing in parallel [9]. The Java 8 Stream API enables pipelined or parallelised processing of data sources with concise code. This work replaces the default Java 8 Streams processing infrastructure with our proposed real-time ForkJoin thread pool, which is illustrated in Figure 1. In a real-time ForkJoin thread pool, one real-time worker thread is created for each processor with a priority, and an optional execution-time server.

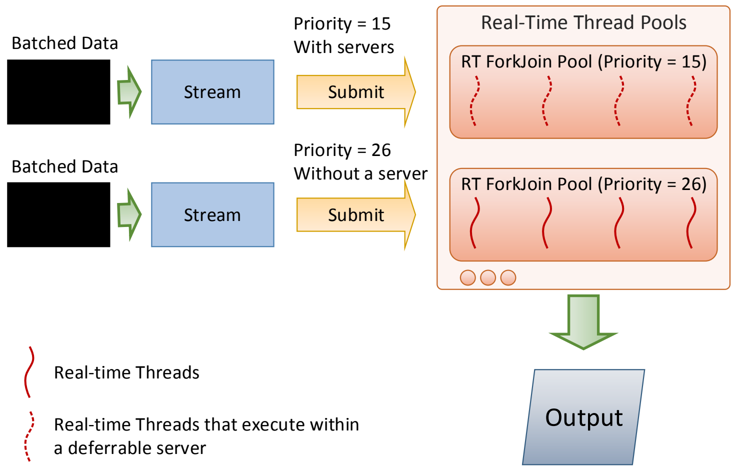


Fig. 1: Overview of the Real-time Stream Framework.

Then, the handling of streaming data in real-time is supported in SPRY by employing our proposed real-time micro batching approach [10]. In a data flow, data items are grouped into micro batches so that each micro batch is treated as a static data source and processed using Java 8 Streams with our real-time ForkJoin thread pool. The real-time streaming facility in SPRY is illustrated by Figure 2.

One goal of SPRY is to process data elements as soon as possible, but also to limit the processing impact on other hard real-time activities in the same

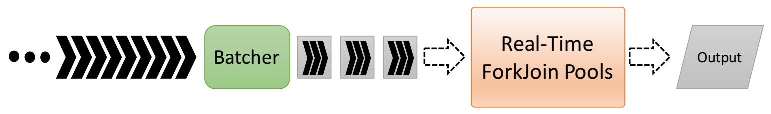


Fig. 2: The overview of real-time processing streaming data.

system by using execution-time servers (e.g. deferrable severs [7]). Currently, SPRY targets fully-partitioned systems, where a task is not allowed to migrate to any other processor once it has been allocated. SPRY pins each worker thread in the real-time ForkJoin thread pool to different processors to support this scheduling scheme.

We assume that real-time activity has already been partitioned via some task allocation approach that takes into account locality of streaming data and its interaction with hard real-time activities. On each processor, the response time of processing each partition of a batched data source can be analysed using current analysis techniques for deferrable servers [3].

4 Selecting Server Parameters for Real-Time Stream Processing in SPRY

In this section, the server parameter selection problem for the SPRY framework is considered. The overall problem is how to find one or multiple deferrable servers against a task set, so that the system utilisation is maximised, and all the hard real-time tasks in the task set remain schedulable. System utilisation is given by the following equation:

$$U_{System} = \sum_{\forall \tau_i \in System} \frac{C_i}{T_i}$$

Typically, deferrable servers require three parameters: priority, period, and capacity. However, in order to make the problem tractable, we fix one of the parameters. Server priority is fixed to the highest priority in the system. The reason for this is that soft real-time tasks must provide a fast response time without being unduly interfered with by hard real-time activities. Selecting a high priority but then limiting the amount of allowed execution time makes the latency of each item in the stream as small as possible whilst maintaining hard real-time guarantees.

Arguably, very long-running batched data processing tasks could be executed at background priority to simply use all available idle time. However, running at high priority introduces benefits when the processing time is relatively short, or partial results are time sensitive.

4.1 Execution-Time Servers Generation

Given a hard real-time periodic task set where each task has a unique priority (e.g., assigned by the deadline monotonic priority assignment algorithm); the proposed approach is described as follows:

1. Put all tasks in the task set into a queue.
2. Get the task from the queue with highest priority as τ_p, the priority of which is p. Remove task τ_p from the queue.

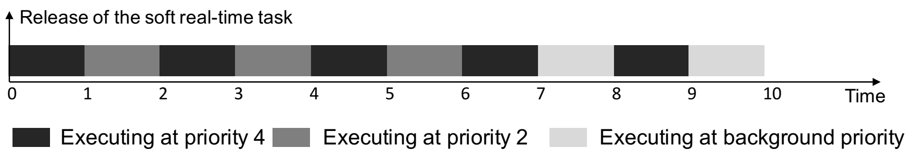


Fig. 3: Execution of the soft real-time task.

3. Create a deferrable server at priority $S = p + 1$.
4. Find a combination of its replenishment period T_S and capacity C_S, so that $\frac{T_S}{C_S}$ is as high as possible, and all the lower priority (i.e., less than or equal to p) tasks in the tasks remain schedulable. This can be done using the proposed heuristic that will be proposed in Section 4.2.
5. Go to step 2 when the queue is not empty, otherwise exit.

For example, consider a task set which consists of two tasks: τ_5 ($C_5 = 3$, $D_5 = 5$, $T_5 = 10$), and τ_3 ($C_3 = 2$, $D_3 = 10$, $T_3 = 10$). The priority 5 is the highest priority. Using the proposed algorithm, two deferrable servers can be found. They are S_1 (with priority = 6, $C_{S_1} = 2$, $T_{S_1} = 10$), and S_2 (with priority = 4, $C_{S_2} = 3$, $T_{S_2} = 10$). After adding these two servers to the system, τ_5 and τ_3 are still schedulable, with R_5=5 and $R_3 = 10$ which can be calculated using the response time analysis (RTA) equations [2].

A set of deferrable servers at different priority levels have been found using the proposed algorithm. At runtime, if the set is not empty, the soft real-time task will be executed by the highest priority deferrable server which has capacity left. When the current capacity is exhausted, the soft real-time task will be transferred to another deferrable server which has the highest priority amongst all servers that have capacity left (if any). Otherwise, the soft real-time task has to be suspended or, more typically, executed at the background priority. The soft real-time task will be transferred to a higher priority server (compared to its current executing priority), when the server's capacity is replenished.

For example, we have two deferrable servers: S_1 (priority = 4, $C_{S_1} = 1$, $T_{S_1} = 2$), S_2 (priority = 2, $C_{S_2} = 3$, $T_{S_2} = 10$), and one soft real-time task which requests 10 time units of execution. They are all released at time 0 and the execution is illustrated by Figure 3.

4.2 Generating Parameters for each Server

The priority of a server is determined using the algorithm proposed in Section 4.1. The period and capacity are yet to be calculated. However, once the period and the priority of a server can be determined, we can employ the optimal server capacity allocation algorithm that was proposed in [3] to calculate the capacity. The proposed capacity searching algorithm uses a binary search to search the capacity between 0 and the period of the server for the maximum capacity C for which all the tasks in the system remain schedulable.

Therefore, the problem then becomes how to choose the period for the server at each priority level.

Difficulties in Server Parameter Selection

Searching for optimal periods for servers at different priority levels has an exponential time complexity. For a task set of n tasks with periods between 0 and p, the complexity of an exhaustive search is n^p. For example, if we try to find servers against a task set that contains 100 tasks where a potential period of each server is from 0 to 10, it will require 10^{100} runs of the binary capacity search. If each run takes 1 nanosecond, the whole search will take about 3×10^{83} years.

It is therefore necessary to employ a discontinuous landscape search algorithm to search towards the optimal server parameters.

Heuristic for Server Parameter Selection

In order to mitigate the time complexity of the server parameter selection problem, we propose a fast and simple heuristic for determining the potential period and capacity for the servers. We use a discontinuous greedy search approach. In the algorithm below, *task* refers to the hard real-time tasks of the system, and *server* to the execution-time servers used to execute the streaming jobs. The algorithm is described as follows:

```
Sort servers as highest priority first;
Calculate the exact divisors of the deadline of each hard task in the
    task set, and add them into the list: potentialPeriods. The list is
    ordered by longest period first, and contains no duplicate periods;
foreach(Server s in servers){
   ServerUtilisation = 0;
   foreach(Period t in potentialPeriods){
      Binary search capacities between 0 and the period of s for the
          maximum capacity C, such that all the hard tasks in the system
          remain schedulable;
      if(a schedulable capacity found){
         if(C/t > ServerUtilisation){
            assign the C as the capacity of server: s;
            assign the t as the period of server: s;
            ServerUtilisation = C/t;
         }
      }
   }
}
```

Using this algorithm, a proposed set of deferrable servers that are described in Section 4.1 can be found, which is sub-optimal. Given a task set that consists of n hard real-time tasks, the time complexity of this heuristic search approach is $O(n^2)$. This is because the RTA [2] for the schedulability test that is invoked in the `foreach` loop within the heuristic has a $O(n)$ complexity. Even still, the algorithm is very fast. For example, on a personal laptop computer, it only takes 25 seconds to find servers for a task set of 100 tasks with maximum period 100. The performance of this algorithm is evaluated in Section 5.

5 Evaluation of Server Parameter Selection Algorithm

This section presents the results of empirical investigations into our heuristic for server parameter selection.

The experiment evaluates the performance of our heuristic against schedulable task sets with total utilisations of 30%, 40%, 60%, and 80%. At each utilisation level the task set size varies from 1 to 100 and the total utilisation is normally distributed into each task using the algorithm proposed in [5].

Synthetic task sets are generated for the experiments. For each task within a task set, the period is a unique randomly generated number between 10 and 1000 and the execution-time is calculated using $C_i = U_i \times T_i$, where U_i and T_i represent the utilisation and the period of the task. The schedulability of each generated task set is validated using response time analysis proposed in [2]. Within each task set, the deadline of each task is equal to its period. Experiments are performed 100 times and the results are shown in Figure 4.

The results are presented in box plot graphs of the system utilisation achieved after having applied the determined server parameters. This is given by:

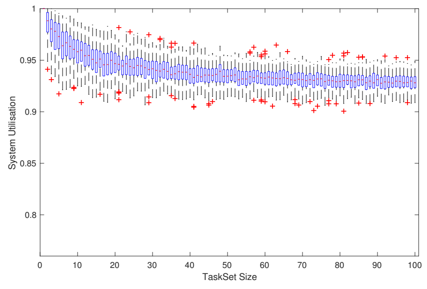
$$\sum_{\forall S \in Servers} \frac{C_S}{T_S} + \sum_{\forall \tau_i \in Taskset} \frac{C_i}{T_i}$$

System utilisation is shown on the y-axis, whilst the size of the task set is on the x-axis.

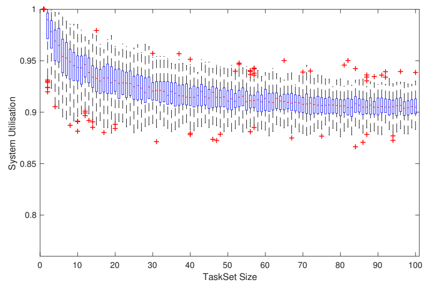
Overall, our heuristic improves the system utilisation significantly. When the utilisation of the task set is 30%, the system utilisation is improved to above 94% on average for arbitrary task set size. The system utilisation achieves 100% when the task set has only one task, this is because the algorithm simply creates a server with the same period as the task and using all available slack (i.e., $D_i - C_i$, $D_i = T_i$) of the task as its capacity.

The mean system utilisation decreases slightly when the task set utilisation increases, this is because the remaining space for the server is decreased. For example, given a task set with $\tau_1(D_1 = T_1 = 30, C_1 = 5)$, and $\tau_2(D_2 = T_2 = 20, C_2 = 5)$, we can have a server $(T_S = 4, C_S = 1.857)$ resulting in system utilisation of 88.1%. If C_1 and C_2 are set to 10 then no server can be found and system utilisation remains 83.3%.

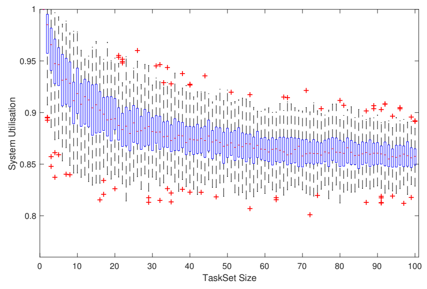
In addition, the mean system utilisation also decreases when the size of the task set increases. The reason is that the more tasks, the lower the chance that they display harmonic periods. A task set with fully harmonic periods enables 100% utilisation [12]. For example, if we have two tasks with harmonic periods: $\tau_1(D_1 = T_1 = 40)$, $\tau_2(D_2 = T_2 = 20)$, and the utilisation of each is 25%, we can find a server $(T_S = 8, C_S = 4)$ which makes system utilisation 100%.



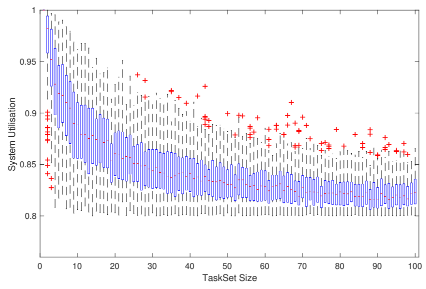
(a) 30% total utilisation.



(b) 40% total utilisation.



(c) 60% total utilisation.



(d) 80% total utilisation.

Fig. 4: System utilisation after using servers for task sets with deadline equal to period.

However, if we reconfigure T_1 to 30, maximum system utilisation can be shown through exhaustive searching to be only 89.3%. As discussed previously, exhaustive searching is only feasible for the very smallest task sets.

It can be observed that as the utilisation of the hard real-time tasks increases, the total utilisation achievable by this approach varies increasingly. This is because our approach only uses a single execution server for the task set with the deadline equals to the period, which must be placed at a single priority level. As the utilisation increases, it is harder to find a space in the schedule to efficiently use the available CPU time.

6 Conclusions

In this paper, we have investigated how to use execution-time servers for real-time stream processing in fixed priority-based pre-emptive multiprocessor systems. The proposed approach is to use a set of deferrable servers to execute the soft real-time stream processing tasks at as high a priority as possible, whilst still guaranteeing the hard real-time tasks in the system remain schedulable. A heuristic has been proposed for server parameter selection that works for tasks with their deadline less than or equals to their periods. The experiments show that significant utilisation can be made available for handling stream processing tasks when using our heuristic.

References

1. N. C. Audsley, Y. Chan, I. Gray, and A. J. Wellings. Real-Time Big Data: the JUNIPER Approach. 2014.
2. A. Burns and A. Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001.
3. 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.
4. R. I. Davis. *On exploiting spare capacity in hard real-time systems*. PhD thesis, University of York, 1995.
5. 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.
6. R. I. Davis, K. W. Tindell, and A. Burns. Scheduling slack time in fixed priority pre-emptive systems. In *Proceedings of the Real-Time Systems Symposium, 1993.*, pages 222–231. IEEE, 1993.
7. 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.
8. J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers. Big data: The next frontier for innovation, competition, and productivity. 2011.
9. 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.
10. 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.
11. L. Sha and J. B. Goodenough. Real-time scheduling theory and Ada. Technical report, DTIC Document, 1989.
12. L. Sha, M. H. Klein, and J. B. Goodenough. Rate monotonic analysis for real-time systems. In *Foundations of Real-Time Computing: Scheduling and Resource Management*, pages 129–155. Springer, 1991.
13. B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic Task Scheduling for Hard Real-Time Systems. *Real-Time Systems*, 1:27–69, 1989.
14. R. Stephens. A Survey of Stream Processing. *Acta Informatica*, 34:491–541, 1997.