Patterns for Distributed Real-Time Stream Processing

Pablo Basanta Val
Norberto Fernández García
Luis Sánchez-Fernández
Jesus Arias-Fisteus

Abstract—In recent years, big data systems have become an active area of research and development. Stream processing is one of the potential application scenarios of big data systems where the goal is to process a continuous, high velocity flow of information items. High frequency trading (HFT) in stock markets or trending topic detection in Twitter are some examples of stream processing applications. In some cases (like, for instance, in HFT), these applications have end-to-end quality-of-service requirements and may benefit from the usage of real-time techniques. Taking this into account, the present article analyzes, from the point of view of real-time systems, a set of patterns that can be used when implementing a stream processing application. For each pattern, we discuss its advantages and disadvantages, as well as its impact in application performance, measured as response time, maximum input frequency and changes in utilization demands due to the pattern.

Index Terms—Real-time patterns, stream processing, big data

1 Introduction

Big-data systems are one of the technologies marking an important momentum [1], [2], [3], [4], [5]. They are characterized by the need of processing huge collections of data that are difficult to process using traditional techniques and, thus, require specific processing tools. As indicated in [8], a variety of data intensive applications can be developed with a big data infrastructure, including applications like the Hadron Collider, the solar observatory of NASA, or Facebook. Many of them need to deal with some of the five V's of big data systems, which refer to great volumes of data, ranging from Terabytes to Exabytes, high velocity to process incoming data, variety in incoming data, and, more recently, veracity and value requirements.

Around it, several new frameworks have being created, including Hadoop [9], Storm [10], Spark [11] and Samza [12], which target different applications needs. Though many of these frameworks are still maturing, they are contributing new computational paradigms such as map-reduce [37] and distributed-stream processing [2], [4], [7], [8]. These frameworks may prop up or give support to the development of big data applications that reside in a local cluster hosted in the Internet.

Another characteristic of big data applications is that they often need to meet real-time requirements [13], [14]. For instance, the Hadron collider produces a 300 Gb/s stream that has to be filtered to 300 Mb/s for storage and later processing. This is also the case of high frequency trading systems, which have sub-second deadlines that have to be met. Lastly, another case of big data analytics that faces temporal restrictions is the trending topic detection algorithm used in Twitter to show a dynamic list with the most popular hashtags. The Twitter trending topic detection application is the case of a big data scenario characterized by velocity, with operational deadlines of up to 2 seconds to output an updated list of trending topics.

Among the different technologies [2] supporting big data applications Hadoop [9] and Storm [10] stand out. The former is targeted at batch computing, that can take minutes, days or even months to compute. The latter is targeted at online computing made on machines that have to process data with sub-second deadlines by means of parallel and distributed computing techniques in order to increase maximum input frequencies in application nodes. However, from the perspective of a real-time system those technologies are not the most efficient infrastructures to meet deadlines, because they lack the support required to assign the different pieces of a distributed application to a certain computational node [12], [13].

Furthermore, from the perspective of a real-time system, there are important problems such as defining what a real-time big data system is and what a proper infrastructure to support the system is. Although some pioneering research initiatives provide partial approaches [12], [13] [31], [32], [63], they are far from producing commercial solutions. The real-time community has not yet defined the bounds and limitations of these new real-time systems. Some pioneering work trying to go in that direction includes [14], [15], aiming at making map-reduce a real-time facility, [16], [17], [18], [19], [20] developing theoretical models for parallel and distributed infrastructures that speed up big data applications, and [21], [22], [23], specific studies that relate the cost of an application to its performance. However, there is still no mature drawing of what a real-time big data system is.

In the particular case of big data applications aimed at stream processing, one of the multiple difficulties in their development is the lack of a general catalog of computational patterns relating the differences in cost of these patterns to their response times [36]. Although some work along that line exists [13], [21], [22], mainly focused on high-performance computing, they are far from producing operational catalogs that can be revamped from a real-time systems perspective. For instance, [21], [22] proposed a series of patterns for data stream processing. However, these patterns are more focused on high-performance than on real-time predictability. From their perspective, our work adapts previous models from the high-performance computing scene to the real-time scene. In addition, some authors [13] have used a real-time version of Storm to analyze different implementation strategies from the point of view of their impact on application response times. These strategies are addressed on our work in a more proper way by analyzing their individual performance in small benchmarks that determine the impact that a certain type of pattern has on the response time of the application and the maximum input frequency of the system.

The work could also be applied to a number of previous distributed stream technologies: Stream [47], [48], StreaMIT [49], Aurora [50], Flextream [51] and the novel Spark [11] and Storm [10] initiatives. These technologies may profit from the patterns described in order to meet their application deadlines in an efficient way. Currently, all of them are focused on efficient (high-frequency) response times, but they have set meeting application deadlines aside. From this perspective, they may benefit from the scheduling model and the catalog of patterns described in this article.

The rest of this article is focused on developing the catalog of patterns for real-time distributed stream processing applications. Section 2 presents the state-of-the-art, where different patterns are introduced and their relation with this work are analyzed; it also describes different techniques that can be useful to develop real-time stream processing systems. Section 3 describes the stream model used in the evaluation of these patterns, as well as the scheduling model. Section 4 describes the patterns catalog. Section 5 evaluates the performance of each pattern. Section 6 concludes the paper and describes our ongoing work.

2 State-of-the-Art

2.1 Programming Patterns for Big data

There is a set of programming patterns useful to develop big data applications [13]. In a technological plane, map-reduce [37] offers different patterns for different processing stages including summarizations, filters and data organization, join, and input and output design patterns. Some authors [46] have illustrated the performance one may expect from some map-reduce policies. They show how the use of incremental policies and data placement may be beneficial from the point of view of the application.

Another type of big data infrastructure refers to data bases. In this arena, there are some technologies such as Cassandra [39] that offer a catalog of patterns that may impact on performance. For instance, the type of patterns defined in [39] are focused on avoiding some types of configurations that may result in low performance, such as storing an entity in a single column or mismanaged atomic updates. The performance results reported in [39] refer to batch processing applications meanwhile our contribution is focused on stream processing. Due to this, the analysis of the state of the art is focused on distributed stream processing and real-time computing.

2.1.1 Distributed Stream Processing Patterns

The first set of patterns refers to distributed stream processing and some current technologies like Storm [38]. References like [38] explain the different type of semantics, useful for application development. Unfortunately, it does not provide a basic catalog of patterns that can be used as a departing point to develop applications that have to meet deadlines. This is the contribution of our article, which provides this basic catalog of patterns to develop big data applications for stream processing.

There is a large tradition in distributed stream processing before the launch of Apache Storm, in academic and commercial fields. In the academic arena, there are a number of systems that support different stream abstractions such as Stream [47], [48], StreamMIT [49], Aurora [50], Flextream [51], River [52], Cayuga [53] and Naiad [54]. None of those academic approaches seems to be focused on meeting deadlines, although some of them could be easily extended with the infrastructure proposed in our article to support end-to-end deadlines. The main problem in all of them seems to be that they were not designed to meet end-to-end deadlines. Commercial systems include Apache Samza [7], Storm [8], and Spark Streaming [9]. Most of those commercial systems have been oriented towards low-latency but they have set aside the definition of end-to-end deadlines in streams.

In the specific case of stream computing [28], [29], [31], [32], there is a set of techniques that may have an impact on the application utilization. In [22] a catalog of ten patterns is proposed. These patterns are analyzed in the context of general-purpose applications (not with applications that have to meet deadlines). For each pattern, the authors analyzed its impact on the graph of the application (which may be changed or unchanged), the application semantics, and their on-line and off-line behavior. Our article includes a shorter catalog with six patterns; four share commonalties with [22] and two are specific for the real-time domain. In order to illustrate and evaluate these patterns, a scalable real-time trending topic detection application running on a real-time version of Apache Storm [13] has been developed. This short catalog refers to the minimum number of patterns required to develop this type of applications.

The relationship with [22] is analyzed in Table 1. In the definition of a useful catalog, our catalog removed those optimizations (reordering and redundancy elimination) that have not been used to develop the trending topics stream application. From the perspective of a real-time version of a distributed stream processing infrastructure, there are very useful techniques: operator separator, fusion and fission, which have been reinterpreted from the perspective of a deadline-based application. In addition, the strategy defined in placement has been reinterpreted from the perspective of worst-case computations, using the pattern to allocate stages to nodes in an efficient way.

TABLE 1
Comparing this Work with Existing Techniques and Optimizations

Technique optimization name	This work	Changes in DAG
Operator reordering	N-I (assumed perfectly ordered streams).	Y
Redundancy elimination	N-I (assumed efficient design)	Y
Operator separation	Adapted to meet deadlines	Y
Fusion	Adapted to meet deadlines	Y
Fission	Adapted to meet deadlines	Y
Placement	Re-interpreted from the point-of-view of a deadline based system	N
Load balancing	N-I	N
State sharing	N-I	N
Batching	N-I	N
Algorithm selection	N-I	N
	Single stage (goal of avoiding unnecessary distribution and parallelism)	**Y**
	Frequency modifier (goal of reducing stream computational demands)	**Y**

N I: refers to non integrated.

In addition, our catalog added two new patterns: single stage, useful to join all the scattered stages of a stream into a single stage, and frequency modification, which is useful to reduce unnecessary activations in a stage. The comparison also draws on changes carried out on the structure that defines the stream processing stages (called DAG: directed acyclic graph).

2.2 Real-Time Parallel Computing

Currently, the real-time community is extending the classical computational models to take advantage of the support given by multi-core infrastructures to the development of parallel applications. All these pieces of work have an important impact on the infrastructure required to run big data applications because, typically, big data applications cannot be hosted in a single node, but comprise several interconnected nodes.

Among the list of available approaches, some remarkable work are [17], [18], [19], [20], [35], which collectively deal with the needs of some of the most popular computing paradigms. In [17] the authors considered the multi-core scenario and proposed a parallel synchronous model inspired by the primitives of OpenMPI. Each application is composed of segments, which can be parallel and/or sequential. The model includes task composition, which divides the input into different flows, as well as the use of partitioned deadline monotonic approaches, which enable establishing global utilization patterns using different allocation techniques. Reference [20] provides a parallel execution model for cyber-physical systems with low-level scheduling techniques similar to those proposed in this article. However, while their work is more focused on a lower level programming model (parallel tasks), our work is focused on a higher-level computation model based on streams. Finally, [35] offers techniques for system-on-chip infrastructures to meet deadlines. The relevant difference with [35] is in the application domain that in [35] seems to be targeted to low-level tasks, while our approach is for stream processing.

The computing model described in this article, which has been extended from [13], [14], is slightly different from [17]. It is based on the stream model offered by Storm, where each stage of the application may have a different input frequency in order to increase its end-to-end performance. The patterns proposed in this article are also interesting for [17], as they use their computational models to process different streams, like those coming from heavy sources like Twitter.

Another set of work ([18], [19], [23], [60], [61], [62]) are related to the scheduling of parallel tasks on multi-core processors. Their computational models are mainly based on the fork-join tasks and the authors describe end-to-end interactions as a set of sequential and parallel steps. The authors also provide a set of scheduling algorithms in charge of partitioning the system using fork-join primitives.

In addition, some authors [15], [16] have worked on the definition of a real-time version of map-reduce from a practical perspective. For instance, [15] has analyzed the problem using a constraint satisfaction problem and has introduced several heuristic strategies for this formulation. By using their constraint model, an off-line setting for map-reduce jobs is formulated, which is later on applied in the on-line scenario. Likewise, [17] deals with the heterogeneity in nodes and the differences in map-reduce tasks, proposing new scheduling algorithms that take into account these costs.

The distributed stream processing model described in our article has roots in distributed real-time Java [55], [56], which have been adapted to the internals of Apache Storm to produce a real-time version of Storm. The idea of the scheduling framework is to provide a simple computational model able run in an Apache Storm engine. It is also based on [57], taking the idea of independent releases (that have no release jitter) and on [58] for the idea of a formulism for a partitioner based on utilization for a multiprocessor system. Another advantage with respect to previous approaches is that it is easier to understand and to compute. Other approaches need the use of iterative and computationally expensive techniques based on heuristics to obtain response times (e.g., using [59]) that have to take jitter into account. Also, the implementation of the techniques as part of an Apache Storm stack is easier with the proposed framework than using the parallel techniques ([60], [61]), which do not map so easily to the programming model of Apache Storm.

The patterns proposed for streams are also feasible for a map-reduce infrastructure, but their adaptation needs to be explored in a different piece of work.

3 Computational Model

In the real-time context, a real-time stream is defined as a continuous sequence of data or items whose processing has some real-time requirements like a deadline from the input to the output. This model is a simplification taken from [14], where the authors produced a programming model based on a generalized version of Storm. It is also compatible with the computational model of Storm described in [33], [34], which is focused on the definition of different scheduling policies for Storm.

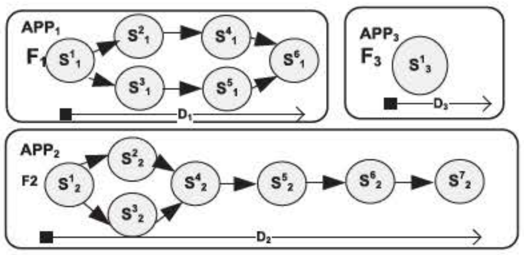


Fig. 1. Three applications with different deadlines.

3.1 Stream Model

The proposed model defines a bundle as a set of applications, where each application is characterized under the constraints of a directed acyclic execution graph. The example included in Fig. 1 shows different types of applications with different deadlines.

Formally, an application bundle (*Bundle*) is defined as a set of *n*- applications: APP_1 to APP_n that may run concurrently,

$$Bundle \overset{\text{def}}{=} (APP_1, \ldots,\ APP_n) \tag{1}$$

Likewise each application (APP_i) is characterized with its maximum input frequency (F_i), a computational end-to-end deadline (D_i), and a set of stages (S_i) arranged as a directed acyclic graph.

$$APP_i \overset{\text{def}}{=} (F_i, D_i,\ S_i) \tag{2}$$

For each application, there is a graph (S_i) composed of a finite number ($stg(i)$) of stages. Each stage of the graph has a maximum execution time (C_i^j) defining the maximum demanded execution time for each stage of an application:

$$S_i = \begin{pmatrix} C_i^1 \\ \ldots \\ C_i^{stg(i)} \end{pmatrix} \tag{3}$$

The invocation of stages has been enforced to use a maximum inter-arrival invocation frequency pattern to control the arrival of different items of a stream. This constraint avoids the problem of jitter activation, described in [57] and [59], thus producing simpler scheduling frameworks.

3.2 Computational Infrastructure

The type of cluster used to deploy this model is based on an architecture that supports a large number of regular processing units called CPUs (see Fig. 2). Each execution unit is connected to others. In addition, it is assumed that the infrastructure runs a fixed priority scheduling system in charge of running each stage of the application in a proper execution unit. This type of support is enough to use multi-processor real-time scheduling techniques [25], [26], [27] in combination with other scheduling policies.

Formally, there are *m*-interconnected machines each one providing an execution unit (called CPU in Fig. 2) that may host different stages of an application:

$$Cluster \overset{\text{def}}{=} (\pi_1, \ldots, \pi_m) \tag{4}$$

Each machine in the cluster supports a preemptive priority based scheduler and contributes its utilization ($U_i \leq 1.0$) to the total available utilization (U_{avail}),

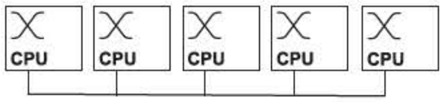


Fig. 2. Execution infrastructure arranged as a cluster.

$$U_{avail} = \sum_{i\ 0}^{m} U_i \tag{5}$$

3.3 Fully Described Real-Time System

To be fully characterized, all stages of all applications should be assigned to a machine of the cluster (π_i^j), a priority (P_i^j) assigned for its execution, and have a maximum frequency (F_i^j).

$$\left(F_i^j, P_i^j, \pi_i^j\right) \tag{6}$$

Once the application is characterized, we calculate global application utilization (U_{app}) needs as partial contributions of all the stages in all the applications, each one contributing its maximum utilization needs (U_{APPi}),

$$U_{bundle} = \sum_{i\ 1}^{n} \sum_{j\ 1}^{stg(i)} \left(C_i^j \cdot F_i^j\right) \tag{7}$$

3.3.1 *Schedulability Analysis: Response Time and Utilization*

After allocating (using a worst-fit policy to be compatible with [58]) each stage to one element of the cluster, the end-to-end response times may be calculated as arbitrary sequential or parallel stages that contribute worst-case response time. For sequential stages, worst-case computations are calculated as a summation of the different individual contributions,

$$rt_i^{seq(i)} = \sum_{j\ 1}^{seq(i)} (rt_i^j\ ) \tag{8}$$

Likewise, parallel stages derive worst-case computations as the maximum cost in all parallel stages,

$$rt_i^{par(i)} = \max\left(rt_{\text{i}}^{j}\right) \quad with\ j \text{ in } 1\ ..\ par(i) \tag{9}$$

To derive the interference with several tasks (supporting stages) running on the processing unit (CPU), response time analysis (RTA) [27] combined with periodic enforcers may be used. They offer the following recursive equations to calculate the worst-case computation time (*rt*) of a task-set:

$$rt_i = C_i\ + \sum_{j\ \ 1}^{i\ 1} \lceil rt_i \cdot F_j \rceil C_j \tag{10}$$

RTA enables to compute the response time of a task set using the demanded computation time (*C*) of all tasks and its maximum activation frequency (*F*) in an incremental way by adding partial contributions of all other tasks (or applications). Blockings (*B*) allow not only the modeling of local interference but also of the effects of a general purpose network. Therefore, our previous equation for response time is transformed into:

$$rt_i = C_i \ + B_i + \sum_{j \ 1}^{i \ 1} \lceil rt_i \cdot F_j \rceil C_j \qquad (11)$$

Under a constrained scenario, with $B + 1/F = D$ in all stages, it is also possible to define a sufficient (but not necessary) bound to the number of computational units (m) necessary in a cluster to support an application (see [55] and [58]). It is done as a partial demand of the stages which contribute to the total utilization of the bundle (U_{bundle}),

$$U_{bundle} = \sum_{\forall i,j} \left(C_i^j \cdot F_i^j \right) \ < \ (m - 1/2) \cdot (U_i) \qquad (12)$$

One of the advantages of Eq. 12 is that it may be used to offer online admission control using a global partitioner.

3.4 Performance Indicators for Patterns

The following performance indicators have to be taken into account in the evaluation of patterns:

- *Application utilization (U_{app}).* One of the evaluated indicators is the total utilization carried out at each stage of the application as well as the whole application. The goal of this indicator is to determine the local and global costs involved in processing an application. It is also useful to concretize the performance one may expect from this type of infrastructure. The use of different patterns is expected to have an impact on the utilization of the application. For instance, some may require extra computations, which adds extra overhead to the application.
- *Maximum input frequency for a given application.* For some patterns one may also determine properties related with the maximum input frequency and the parameters of the application. This is important, because many applications need to process inputs with a frequency a single computational unit is unable to handle, especially in stream-based applications with high-frequency input data. In general, the higher the input frequency the application may deal with, the better the system is. It is expected that the use of different patterns enable the possibility of increasing the input frequency of an application (e.g., by shifting part of the data from one machine to another).
- *Overhead of the proposed pattern and tradeoffs.* Many patterns have on impact in the application in terms of overhead. Some patterns are able to reduce the utilization of the application while others increase it. Unfortunately, not all applications benefit from all patterns. Therefore, for each pattern, a careful evaluation is needed to determine a range of application scenarios in which the use of the pattern is just slightly beneficial or even counterproductive.
- *Response-time of the proposed pattern.* It refers to the maximum time required for processing an item of a stream. The calculation of the worst-case response time requires solving out an equation in the evaluation scenario.
- *Expandability of the proposed pattern.* Expandability (EXP) refers to the maximum number of times an application can increase its current input frequency. For a given application, we define the maximum expandability (*EXP*) in frequency as a minimum local expandability in all stages of an application.
- *Utilization of an application ($U_{APP(i)}$)).* It is defined as the contribution of each different stages of the application to the utilization demanded by the application.

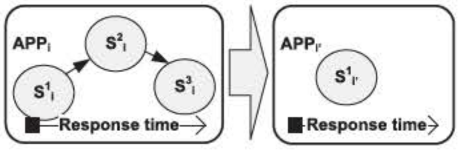


Fig. 3. Single stage application pattern.

4 Catalog of Patterns

This section describes the catalog of six patterns for real-time application processing systems we propose. Each one is presented in a similar style as those described in the original catalog [22]. Each pattern has a dedicated section that includes a high-level description of the pattern and its formalization in terms of the scheduling framework described in the previous section.

4.1 Single Stage Application Pattern

The first pattern in the list is the single stage application pattern (see Fig. 3). In this pattern all stages of an application are transformed into a single stage, which contains all the execution logic of the application. This type of pattern is beneficial from the perspective of reducing communication overhead, typically associated to the reception and transmission of messages. This pattern minimizes communication overhead.

On the other hand, this pattern has a harmful behavior from the perspective of parallelism, because it limits the concurrent execution of application stages. Being focused on its logic, single-stage application applications cannot benefit either from the effect of pipelines, the parallelism of the tasks, or the parallelism in the input data.

The pattern is also useful for improving performance because it reduces the penalties due to communications (e.g., serialization issues, and message transmissions).

The utilization of all stages ($U_{APP(i)}$) of an application (APP_i) is calculated as follows:

$$U_{APP_i} = \sum_{j \ 1}^{stg(i)} \left(C^j_{APP_i} \cdot F^j_{APP_i} \right) \qquad (13)$$

After regrouping all stages into a single stage application (named i'), the global utilization associated to the single-stage application gets reduced because the cost associated to each stage decreases (no serialization is required for the communications),

$$U_{APP_{i'}} = \left(C^1_{APP_{i'}} \cdot F^1_{APP_{i'}} \right) \qquad (14)$$

In this way, the utilization saved by the pattern (U_{saved}) is defined as the difference between the two previous utilizations,

$$U_{saved} \ = \ U_{APP_i} - U_{APP_{i'}} \qquad (15)$$

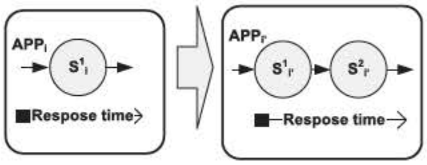


Fig. 4. Operator separation: Pattern illustrated.

Each stage has a maximum available utilization ($U_{avail(APP(i))}$) that can be used to expand local hosted stages,

$$EXP_{APP_i} \leq \min\left\{ \frac{\mathrm{U}_{\mathrm{avail(1,APP(i))}}}{\mathrm{C}^{1}_{\mathrm{APP(i)}} \cdot \mathrm{F}^{1}_{\mathrm{APP(i)}}} \quad , \cdots, \quad \frac{\mathrm{U}_{\mathrm{avail(stg(i),APP(i))}}}{\mathrm{C}^{\mathrm{stg(i)}}_{\mathrm{APP(i)}} \cdot \mathrm{F}^{\mathrm{stg(i)}}_{\mathrm{STR(i)}}} \right\} \quad (16)$$

Because of the single stage pattern, the maximum expandability (EXP) transforms into,

$$EXP_{APP_{i'}} \leq \frac{\mathrm{U}_{\mathrm{avail(1,APP(i'))}}}{\mathrm{C}^{1}_{\mathrm{APP(i')}} \cdot \mathrm{F}^{1}_{\mathrm{APP(i')}}} \quad (17)$$

In principle, the main concern with this pattern is that it limits the maximum processing frequency of the application at the cost of removing also some benefits. Nevertheless, it can be useful for some applications with a reduced utilization demand.

4.2 Operator Separation Pattern

The catalog continues with the operator separation pattern (Fig. 4). This pattern suggests separating a stage into two, in such a way that the factor of use of each one of them is individually reduced. The division of work increases application response time, due to the extra transmission costs. However, on the other hand and in exchange for those costs of transmission, the application increases its parallelism.

As shown in Fig. 11, the pattern splits one stage into two (from $\mathrm{S}^1_{\mathrm{i}}$ to: $\mathrm{S}^1_{\mathrm{i'}} \rightarrow \mathrm{S}^2_{\mathrm{i'}}$). Therefore, there is an increase in work demanded to process the application ($\mathrm{U}_{\mathrm{extra}}$), which is described as follows:

$$U_{extra} = U_{APP_{i'}} - U_{APP_i} = \left(C^1_{i'} + C^2_{i'} + C^1_i\right) \cdot F^1_{APP_{i'}} \quad (18)$$

Potentially, there is also an increment in the input frequency because one stage is split into two, increasing parallelism. With respect to maximum expandability (EXP), the system designer has to remove this expandability constraint referred to the initial application (APP_i),

$$EXP_{APP_i} \leq \frac{\mathrm{U}_{\mathrm{avail(1,APP(i))}}}{\mathrm{C}^{1}_{\mathrm{APP(i)}} \cdot \mathrm{F}^{1}_{\mathrm{APP(i)}}} \quad (19)$$

In addition, two new constraints associated to the division phase have to be considered,

$$EXP_{APP_{i'}} \leq \frac{\mathrm{U}_{\mathrm{avail(1,APP(i'))}}}{\mathrm{C}^{1}_{\mathrm{APP(i')}} \cdot \mathrm{F}^{1}_{\mathrm{APP(i')}}}$$

and

$$EXP_{APP_{i'}} \leq \frac{\mathrm{U}_{\mathrm{avail(2,APP(i'))}}}{\mathrm{C}^{2}_{\mathrm{APP(i')}} \cdot \mathrm{F}^{2}_{\mathrm{APP(i')}}} \quad (20)$$

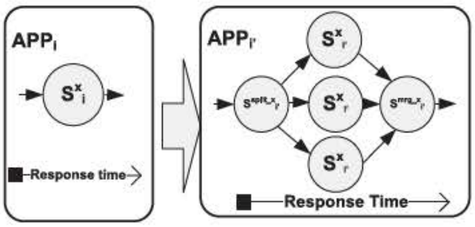


Fig. 5. Fission pattern.

4.3 Fission Pattern

This pattern deals with parallelism. Its mission is to divide the computation graph into several branches, which are computed in parallel to be later merged (see Fig. 5). Potentially, the pattern is useful to increase the degree of parallelism of the application when the input frequency of the data is very high and cannot be processed with a single computational unit. The advantage is that this type of pattern enables higher input frequencies and reduces end-to-end response-times. However, this is done at the cost of adding extra overhead associated to distributed communication.

The first impact of the pattern is an increase in the utilization because it requires several stages. A first stage is in charge of splitting the application into different branches (z), each one running part of the application, to be merged later at a final stage.

The extra computation (U_{extra}) needed to implement the pattern is the following:

$$U_{extra} = U_{APP_{i'}} - U_{APP_i} = \left(C^{split\_x}_{i'} + C^{mrg\_x}_{i'}\right) \cdot F_i \quad (21)$$

In addition, the constraints referring to the maximum frequency of the application should not consider the constraints associated to the stage that is going to be fissioned,

$$EXP_{APP_i} \leq \frac{\mathrm{U}_{\mathrm{avail(x,APP(i))}}}{\mathrm{C}^{\mathrm{x}}_{\mathrm{APP(i)}} \cdot \mathrm{F}^{\mathrm{x}}_{\mathrm{APP(i)}}} \quad (22)$$

Instead, they should consider the new constraints introduced by the splitter (split), the merger (merge) and each parallel branch generated for the fission pattern:

$$EXP^{split\_x}_{APP'_i} \leq \frac{\mathrm{U}_{\mathrm{avail(split\_x,\ APP(i'))}}}{\mathrm{C}^{\mathrm{split\_x}}_{\mathrm{APP(i')}} \cdot \mathrm{F}^{\mathrm{split\_x}}_{\mathrm{APP(i')}}} \quad (23)$$

$$EXP^{mrg\_x}_{APP'_i} \leq \frac{\mathrm{U}_{\mathrm{avail(mrg\_x,APP(i'))}}}{\mathrm{C}^{\mathrm{mrg\_x}}_{\mathrm{APP(i')}} \cdot \mathrm{F}^{\mathrm{mrg\_x}}_{\mathrm{APP(i')}}} \quad (24)$$

$$EXP^{x}_{APP'_i} \leq \frac{\mathrm{U}_{\mathrm{avail(x,APP(i'))}}}{\mathrm{C}^{\mathrm{x}}_{\mathrm{APP(i')}} \cdot \mathrm{F}^{\mathrm{x}}_{\mathrm{APP(i')}}} \ \forall\, q \ in \ [1, z] \quad (25)$$

4.4 Frequency Modification Pattern

In some cases applications can modify their data rates by means of a filtering operation, reducing the input rate at the next stage in the application (see Fig. 6). In this type of cases the operation frequency can be changed to reduce the number of activations. With this approach the maximum utilization required by the application is reduced.

Formally, the pattern changes the arrival of items to the processor reducing the input frequency of the application.

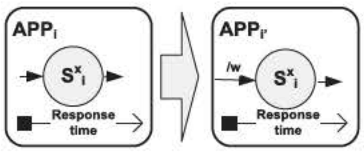


Fig. 6. Frequency modification.

Typically, the change in frequency affects the utilization of the application, which gets reduced as its input frequency decreases. It increases also the maximum achievable frequency for an application.

The savings in utilization (U_{saved}) demand for a simple stage are the following:

$$U_{saved} = U_{APP_i} - U_{APP_{i'}} = \left(C_i^x \cdot F_i^x\right) - \left(C_i^x \cdot \frac{F_i^x}{w}\right) \quad (26)$$

In addition, the constraint for the maximum frequency gets increased by a factor proportional to the reductions in frequency:

$$EXP_{APP_{i'}} \leq w \cdot EXP_{APP_i} \quad (27)$$

4.5 Fusion Pattern

The fusion pattern (Fig. 7) is similar to the single stage pattern seen previously, and shares many of its problems and advantages. Basically, this pattern groups two adjacent stages to create a new one. The rationale of this pattern is that, by grouping two stages the latencies are reduced, because communication costs are minimized. Furthermore, by having several of these elements in the same machine the individual utilization of the node can be increased.

The fusion pattern tends to reduce the amount of required resources. The utilization saved (U_{saved}) is:

$$U_{saved} = U_{APP_i} - U_{APP_{i'}} = \left(C_i^x + C_i^{x+1}\right) \cdot F_i^x - C_{i'}^x \cdot F_{i'}^x \quad (28)$$

Potentially, it also decreases maximum application frequencies, because merging items may be a heavy process. The system introduces this single constraint on the maximum frequency,

$$EXP_{APP_i'} \leq \frac{\mathrm{U}_{\mathrm{avail(x,APP(i'))}}}{\mathrm{C}^{\mathrm{x}}_{\mathrm{APP(i')}} \cdot \mathrm{F}^{\mathrm{x}}_{\mathrm{APP(i')}}} \quad (29)$$

It also removes the two sequential stages constraints,

$$EXP_{APP_i} \leq \frac{\mathrm{U}_{\mathrm{avail(x,APP(i))}}}{\mathrm{C}^{\mathrm{x}}_{\mathrm{APP(i)}} \cdot \mathrm{F}^{\mathrm{x}}_{\mathrm{APP(i)}}} \quad (30)$$

and

$$EXP_{APP_i} \leq \frac{\mathrm{U}_{\mathrm{avail(x+1,APP(i))}}}{\mathrm{C}^{\mathrm{x+1}}_{\mathrm{APP(i)}} \cdot \mathrm{F}^{\mathrm{x+1}}_{\mathrm{APP(i)}}} \quad (31)$$

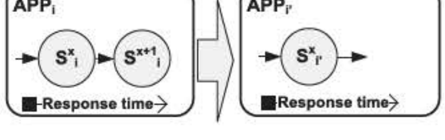


Fig. 7. Fusion pattern.

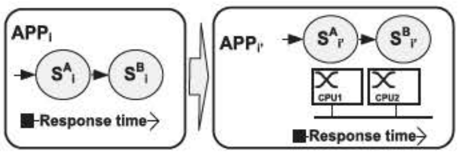


Fig. 8. Placement pattern illustrated and evaluation scenario.

4.6 Placement Pattern

The last pattern of the catalog is in charge of location. In particular, it enables the application to use information about temporal properties of the application and the cluster infrastructure to allocate different processing stages of the application (see Fig. 8). In order to choose the right location for each stage, this pattern takes into consideration irregularities of the system. In the terminology used in this article, an irregularity refers to a change in the conditions of the system in charge of processing the application. In the particular case of this scenario, it refers to differences in the performance of processors and the networks that interconnect them. The greater this asymmetry, the greater the costs of execution in this type of systems, because they have to be dimensioned for the worst-case.

The logical suggestion is to give explicit instructions on where to place the different stages that compose a stream processing application. The goal is to avoid sending over the network portions of state that are in the same computational unit. The cost of this type of strategy is the need of additional information about the type of application that is being developed. This information has to be combined with the infrastructure scheme to produce a proper configuration.

To model irregularities, the concept of asymmetry utilization (U_{asym}) measures the percentage of CPU time lost due to the lack of an ideal infrastructure. It is defined as the difference between the ideal and the available utilization,

$$U_{asym} = U_{real} - U_{avail} \quad (32)$$

Each computational unit contributes to increase global asymmetry (U_{extra}),

$$U_{extra} = \sum_{i\ 0}^{m} U_{asym}(i) \quad (33)$$

Asymmetry reduces with the maximum frequency of an application, which gets reduced by the asymmetry coefficients in all nodes where the infrastructure presents asymmetry:

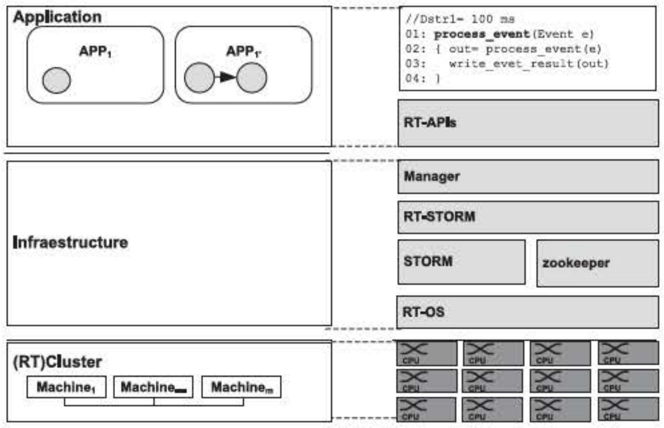
$$EXP_{APP_i} < EXP_{APP_i}.U_{asym\ (i,j)}\ with\ j\ in\ 1\ ..stg(i) \quad (34)$$

5 Evaluation of the Patterns

This section includes the evaluation of patterns previously formalized in Section 4. For each pattern, this section describes the main performance changes and the main characteristics of the applications.

5.1 Evaluation Infrastructure

The evaluation was based on an application that processes application micro-blogging messages coming from Twitter. The application is able to generate a trending topics list in less than one second for different input data frequencies ranging from 0 Hz to 10 kHz. The used infrastructure showed that the patterns validate an application processing 10^7 millions of



```
//Dstr1- 100 ms
process_event(Event e)
{ out= process_event(e)
  write_evet_result(out)
}
```

Fig. 9. Architecture used for the evaluation of applications.

TABLE 2
Test Application Characterization, Software-Stack and Infrastructure

Test Application Characteristics	
Application Processing (worst-case) Costs	From 0.3 to 100 ms
Input Frequencies	From 1 to 10 kHz
End-to-end Deadlines	Strict end-to-end deadlines from 100 ms to 10 s.
Messages: Characterization and worst-case costs	Strings from 10 bytes to 10k Bytes Communication costs from 0.3 to 2 X ms (network + infrastructure overhead).
Software Stack	
RT-Storm	rt-storm-0.9.3.10
O.S.	rt-linux kernel.3.2
Infrastructure	
Nodes	40
Cores per CPU	1
Core speed	1,197 Mhz
Memory	8 Gbytes
Network	1 Gigabit/s

data per second in a cluster with 40 machines. In this scenario, it may offer sub-second end-to-end performance using Eq. 12 to determine the number of machines required to meet a deadline. This application processes strings with variable length to carry out variable size computations. The application benchmarked receives strings, processes them and sends results forward. This type of application is common in many information systems in charge of counting word occurrences or detecting patterns in text (see [41]).

In the case of the trending topics application we use as case study, the main difference between a stream based algorithm and an algorithm based on static data is their programming model. With static data, the algorithm has access to all the tweets. It needs to query the ones it needs (e.g., those tweets that were published in the last 30 minutes) and construct the word counting tables from them. In the stream based case, new tweets arrive dynamically and their words need to be counted as soon as they arrive. In addition, mechanisms are needed to subtract the words of the oldest tweets from the counting table (e.g., those older than 30 minutes).

The software stack used for the evaluation refers to a COTS infrastructure with a reduced version of Apache Storm 0.9.3 installed (see Fig. 9). Although the costs are referred to Apache Storm, the proposed pattern catalog is general enough to be implemented in other frameworks (e.g., Spark) or ad-hoc infrastructures supporting the proposed communication facilities.

In addition, notice that all benchmarks are referred to a single application in isolation. This type of approach can be also useful for computational nodes sharing data coming from different inputs, especially in those strategies based on CPU reservation techniques that guarantee a minimum available time [24], [27]. Our test refers to a scenario with machines connected with a network (see Table 2). The minimum latency of this stack is 300 microseconds and accounts for serialization overheads which are included as a communication cost. The measure takes into account the time required to switch the context and part of the execution of processing overheads. Due to the use of a middleware such as Storm, one has to deal with this type of worst-case communication overheads. Low-level approaches based on modifying the operating system reduce those times remarkably. However, many of the facilities offered by Storm (such as automatic acknowledgement of messages and other recovery mechanisms) would not be available for programming. The application also benefits from many of the mechanisms included to synchronize streams available in Apache Storm such as its *bolt* and *spot* programming model.

In addition to the trending topic application there is also a synthetic benchmark derived from the trending topics application. In the synthetic benchmark, input frequencies moves from 1 to 10 kHz and the cost of processing data moves from 0.3 to 100 ms. In addition, the communication model introduces communication blockings that are in the range of 0.3 to 3 ms.

5.2 Single Stage Application Pattern

5.2.1 *Profitability*

To show the benefit of the single stage application pattern, the case of a simple application with a set of n-sequential stages (Fig. 10) is analyzed. This application is running on a unique computational unit, which executes all the stages in the pipeline. The application has a total application cost of 0.8 ms, which may be split into several computational units. This value is representative for the characteristics of the application we want to model (see Table 2), which is part of the benchmark.

In addition, the messages required to communicate each node have maximum response time costs of 0.7 ms. The potential maximum frequency of the input data ranges from 0.1 Hz to 100 kHz. This type of application is compatible with the application requirements described in Table 2.

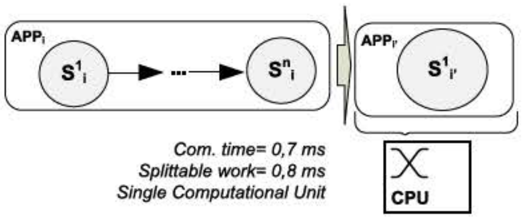


Fig. 10. Single stage pattern evaluation scenario.

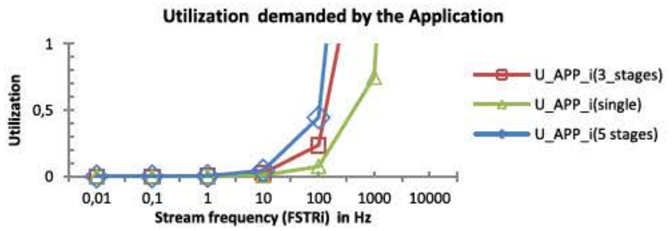


Fig. 11. Single stage pattern benefits (1/2).

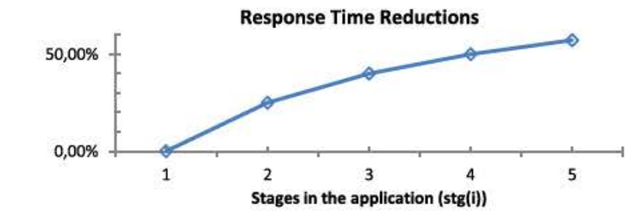


Fig. 12. Single stage benefits (2/2): Impact on response time.

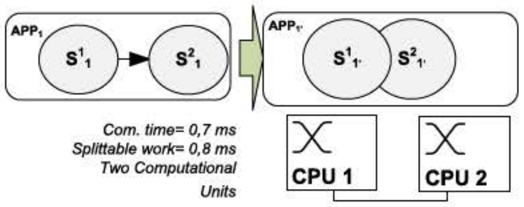


Fig. 13. Single stage unprofitable scenario.

In this scenario, if the different application stages are grouped, computation time will be saved. This is mainly due to the serialization/deserialization costs of communications, which disappear, resulting in shorter response times. This is shown in Fig. 11, which analyzes the utilization of the application for different input frequencies when different types of setups are used (1, 3 or 5 processing stages). Utilization time grows when the number of stages that compose the application diminishes, because communication between two adjacent stages is not longer required. Thus, the performance is degraded as the number of stages that compose an application is increased. In the example, the maximum input application frequency is 1 kHz for the single stage setup.

Complementing the previous results, Fig. 12 shows the reductions in response times that can be obtained by grouping the different stages that compose an application. The figure shows that the benefits of this pattern on the cost of the application are bigger for higher number of stages because their associated communication overheads are higher. For example, in the analyzed scenario, response times will be reduced to 58 percent when the application is composed by 5 stages.

5.2.2 *Unprofitability*

Unfortunately, a monolithic application does not turn out to be the best for all cases, especially when the application requires more than one computational unit. In that case, applications cannot be processed properly and, thus, some mechanism to harness parallelism is necessary. The main source of unprofitability is that maximum frequencies achievable for the application get typically reduced when the application runs on single model (reducing the maximum speed required for the processing of an application).

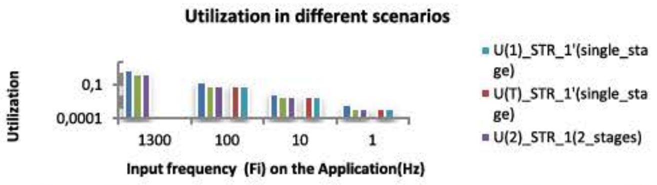


Fig. 14. Single stage unprofitable scenario: Application utilization. U(1) refers to the utilization taken in the first CPU for the application allocation, U(2) to the second CPU and U(T) (U$_1$ + U2) to the total utilization that takes into account both CPUs.

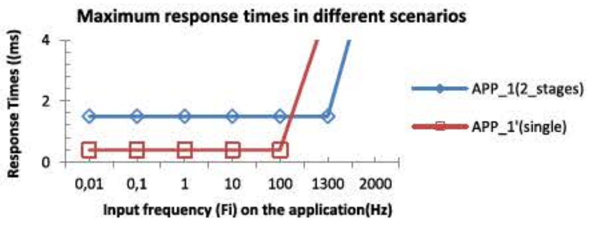


Fig. 15. Single stage unprofitable scenario. Dependency of the maximum response time with the input frequency of the application.

Reusing the previous example, Fig. 13 explores what happens when a new computational unit is added to the infrastructure. In this case, the monolithic application (S_1^1) will not be able to use two computational units, but if the monolithic application were split into two pieces (namely $S_{1'}^1$ and $S_{1'}^2$) its processing speed would increase in scenarios with two or more processors.

Complementing the scenario described in Figs. 13 and 14 shows the utilization of each one of the parts that compose the application and its contribution to the utilization in each computational unit. These results are obtained for different operation frequencies ranging from 1 Hz to frequencies higher than 1 kHz. The first thing to note is that it is not possible to work with a single computational unit at frequencies higher than 1.3 kHz. From that frequency on, it is not possible to continue and one must resort to two computational units and to a proper application partitioning that assigns different application stages to different computational units, thus incurring in overheads associated to unit-to-unit intra communications. Fig. 14 shows the results in terms of utilization of the different computational units (U(1),U(2)) described in the scenario of Fig. 7 and the global utilization for the application ($U(T) = U(1) + U(2)$). Results are described for the original scenario, which consists of 2-stages, and the proposed one with a single stage pattern.

Fig. 15 shows the evolution of response time for different frequencies. These results indicate how response time increases due to the cost of communication. The maximum overhead is constant as the frequency of the application increases, until the system is no longer feasible. Results show also the frequencies at which the system disappeared for the different setups (single stage, two stages). From that frequency on, the system is not feasible with that setup.

5.3 Operator Separation Pattern

5.3.1 *Profitability*

In order to illustrate a scenario where this type of policy has positive effects, the evaluation assumes that the computation of each datum takes 2 ms and its transmission requires 0.7 ms (see Fig. 16). In this context, the idea of distributing a

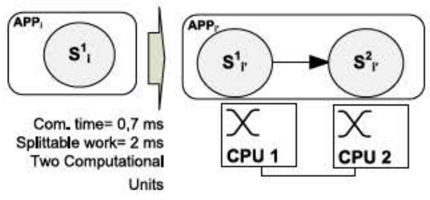


Fig. 16. Operation separation pattern scenario.

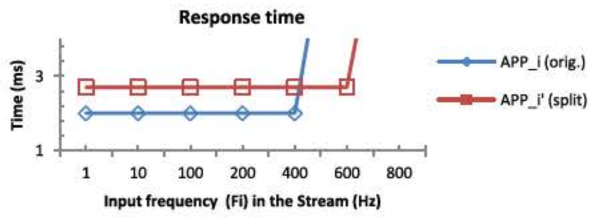


Fig. 17. Impact on the response time of the application. Both for the original configuration and the operator separation pattern.

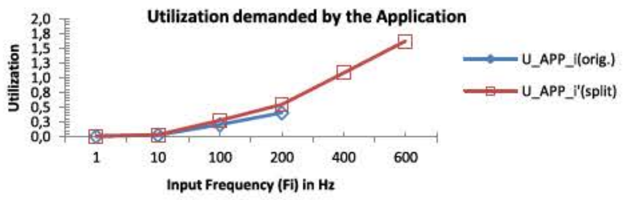


Fig. 18. Impact on the utilization of the application. The results refer to the total utilization costs of the application in all the processors of the cluster.

stage into two helps to reduce the utilization requested from the application locally in each node, enabling higher frequencies at the input of the application.

Fig. 17 shows the relationship between the response time and input frequency of each application before and after applying the pattern. Without the operator separation pattern, response time is lower, until a maximum input frequency reaches 400 Hz. From this frequency on, the system is no longer feasible and its response time increases dramatically. The configuration with two sequential stages may reach 600 Hz, although its response time is higher (2.7 versus 2 ms).

With respect to CPU utilizations, as shown in Fig. 18, it increases with the input frequency until the system is unfeasible. Results also show how the two computational units take more time than the single computational unit configuration. This difference in time is attributed to the cost of sending the information from one stage of the application to the next.

There is another crucial factor to take into account when working with this pattern: the ratio between the amount of work split and the cost of the extra communication required to transmit this information. From the point of view of total system utilization, the divided application is always more expensive than a non- divided equivalent. Empirically, this fact is shown in Fig. 19. From the ratio of equality, where the cost of communication equals that of data processing, an effective unloading is observed as application cost dominates transmission costs. Even for low ratios, like those shown in Fig. 19, the appreciable gap may be high and close to 40 percent.

5.3.2 *Unprofitability*

Fig. 19 also helps to show the limits from which partitioning stops being profitable, in terms of maximum CPU

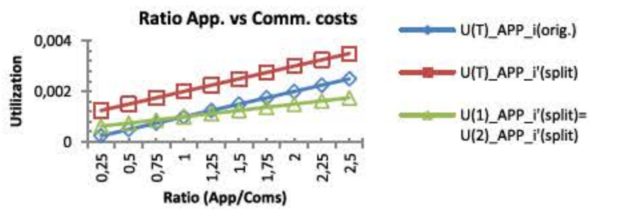


Fig. 19. Impact on the utilization of the application. In the experiment, *orig* refers to the *non split* scenario utilization total demanded utilization, and U(1) an U(2) to the individual utilization on the split scenario.

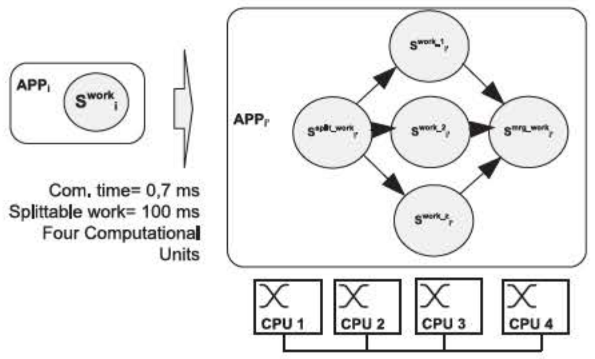


Fig. 20. Fission pattern evaluation scenario.

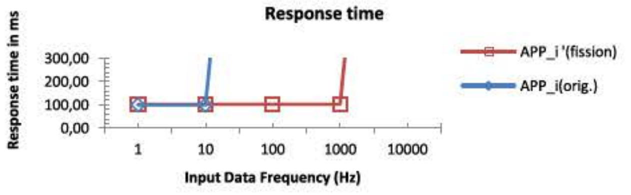


Fig. 21. Different application response time for fission pattern evaluation scenario.

utilization terms. Typically, those limits appear when applications are divided and the time necessary to execute the logic of the application is less than the time of transmission of the data from the first to the second node. In the empirical evidence of Fig. 19, that breaking point corresponds to a 1.0 ratio. In the benchmarked application, the worst-case happens at the origin, when there is an increase of four times the total cost which corresponds to a 0.25 ratio.

5.4 Fission Pattern

5.4.1 *Profitability*

An evaluation scenario has been developed to show the properties of this type of pattern (see Fig. 20). The initial scenario consists of a single application, whose load of 100 ms is distributed between the different processors available in the infrastructure. This load is distributed with a division process and is grouped with a merge process, whose cost is bounded by the communication costs (0.7 ms).

The results show the positive effect of this technique when the appropriate circumstances are given. They occur when the cost of processing data is higher than the cost of sending data from one node to another. They indicate the possibility of processing a greater number of input application items at a low-cost in terms of overhead (see Fig. 21). In the example, where communication costs are less than 0.7 ms, an additional 2 percent delay is added in exchange

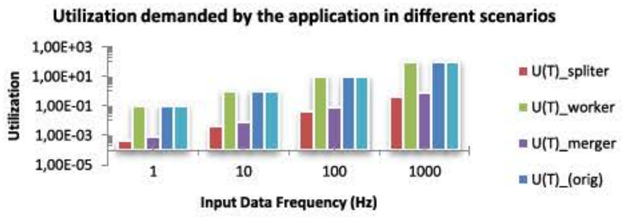


Fig. 22. Different utilizations for scenarios related to fission pattern scenario.

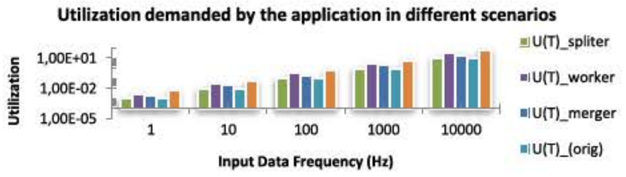


Fig. 23. Different utilizations for a scenario with communication costs of 2 ms and with 0.8 ms processing time.

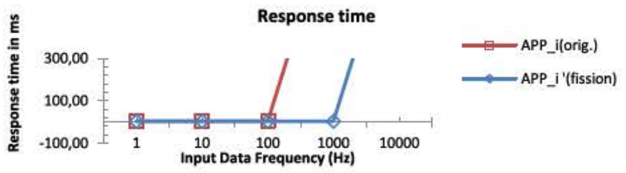


Fig. 24. Different response times for a scenario with communication costs of 2 ms and with 0.8 ms processing time.

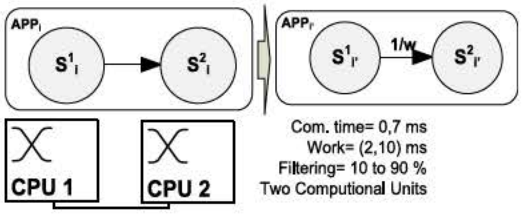


Fig. 25. Frequency modification illustrated.

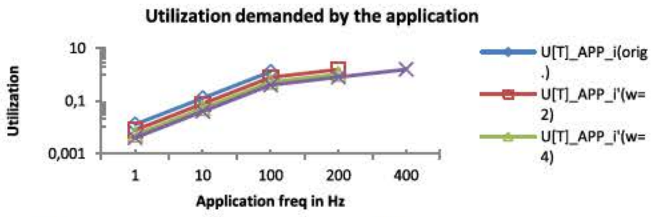


Fig. 26. Frequency modification pattern: Utilization results.

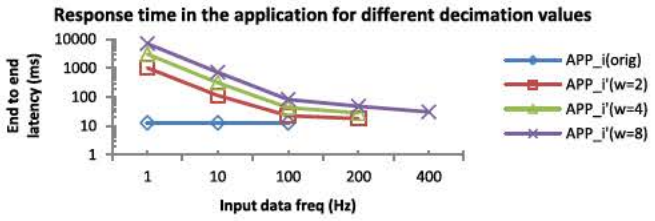


Fig. 27. Frequency modification pattern: Results.

of this delay, there is a potential increase in the maximum input frequency of more than 100 times.

These utilization results show that, for the given scenario, both original and fissioned approaches have close utilization figures. Results included in Fig. 22 show an overhead under 3 percent in terms of extra computation times required to process data.

This good behavior is in part due to the small cost of the splitting and merging operations. Fig. 22 shows that the time needed for splitting and merging is hundreds of times lower than the total time required to process the application. The configuration shows a very profitable scenario, where the work carried out $(U(T)_{worker})$ surpasses the work demanded by the splitter $(U(T)_{spliter})$ and the merger $(U(T)_{merger})$, with utilizations closer for the fissioned $(U(T)_{fission})$ and the initial $(U(T)_{orig})$ configurations.

5.4.2 Unprofitability

In order to illustrate a situation that hinders the performance of the fission pattern, the evaluation scenario is changed to one where the communication cost is greater than the application execution cost. In this type of application, the fission pattern introduces a high computational overhead, near 400 percent of the net time required to process the whole application. As shown in Fig. 23, the results of the scenario indicate that using a non-splitting policy may be better than splitting. That is because communication costs dominate the cost of processing the stream.

If the maximum input frequency of the application is analyzed, it can be seen that the distributed configuration produces shorter response times with higher frequency inputs (Fig. 24), increasing global utilization.

5.5 Frequency Modification Pattern

5.5.1 Profitability

In order to show the profitability of the pattern, a two-stage application has been set up (Fig. 25). The first stage feeds the second with information directly coming from its input, but with a reduced output frequency. In this scenario, the maximum demanded work is 2 ms for the first stage and 10 ms for the second, with communication costs always under 0.7 ms.

As aforementioned, this pattern has a positive effect in the utilization factor, which decreases. Fig. 26 shows this fact empirically. The results displayed show the system utilization before the optimization is applied and after applying it. This evaluation is defined for different frequencies and with different decimation values. The results show reductions of 40, 60 to 80 percent of the total utilization when the input frequency is decimated by 2, 4, and 8, respectively. They also show how the technique is able to support a greater input application frequency. In the cases studied, the maximum application frequency increases from 100 to 400 Hz.

5.5.2 Unprofitability

One of the pernicious effects associated to this pattern is that the maximum latency of the application increases (Fig. 27). The observed delays range from 1 to 7 seconds when the input frequency is low. In the cases of a higher frequency, the absolute delay diminishes remarkably, increasing the global utilization instead. For example, it can be seen in those results that when the frequency is increased in 100 Hz the delays introduced by the technique are of 10, 40, and 50 ms for different decimation factors.

Another interesting indicator to measure is the overhead introduced by the pattern in the application response times (Fig. 28). This increase in overhead diminishes as the

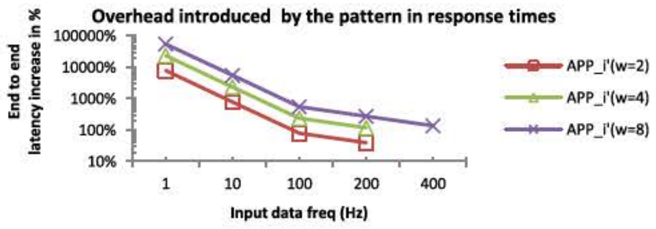


Fig. 28. Overhead due to the latency generated by the frequency modification pattern.

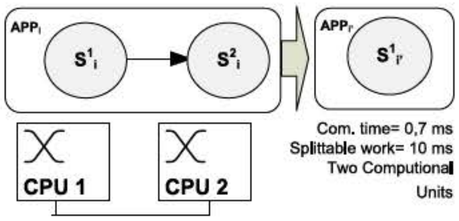


Fig. 29. Fusion pattern evaluation scenario.

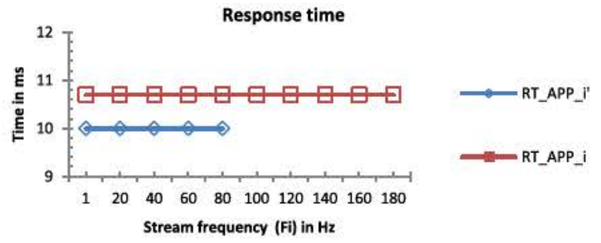


Fig. 30. Impact of the fusion pattern.

frequency increases, because the application waits for the arrival of an item of a stream to produce an output. From the perspective of the computational overload, this result implies that, as the input frequencies go up to around 1 Hz, response times increase by 78 percent for light decimating (1 of each 2), 236 percent for moderate (1 of each 4), and 556 percent for higher decimation (1 of each 8).

5.6 Fusion Pattern

5.6.1 Profitability and Unprofitability

In order to analyze a profitable and a non-profitable use-case, a new scenario has been created. It is shown in Fig. 29. Basically, it consists of a hardware infrastructure with two computational units that process an application whose total execution cost is 10 ms, whereas the communication cost is 0.7 ms. On this scenario, we analyze the variation of the response time with the input frequency of the application.

The obtained results, depicted in Fig. 30, show the aforementioned effects. At a first glance, the merged configuration reduces application response time, but also the maximum input frequencies offered by the system. Without applying the fusion pattern, the maximum response time of the application increases from 10 to 10.7 ms. However, it also increases the maximum frequency that can be processed by the system from 80 to 180 Hz.

5.7 Placement Pattern

5.7.1 Profitability

Like in previous cases, a scenario was developed to illustrate the potential benefits of this pattern (see Fig. 31). In

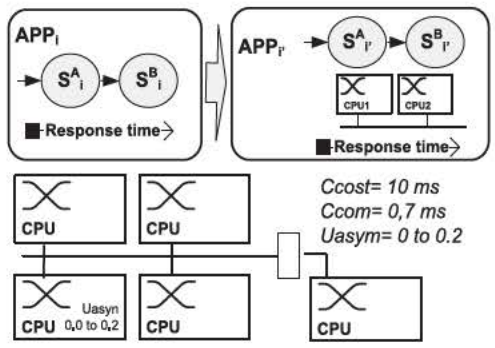


Fig. 31. Evaluation scenario.

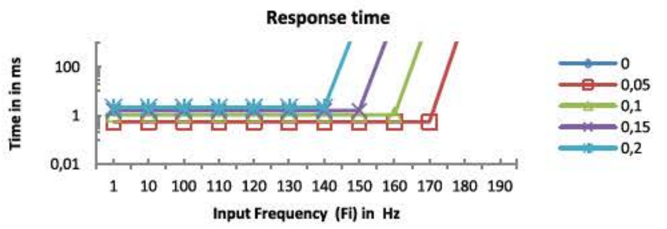


Fig. 32. Placement scenario pattern benefits. Illustrated for different asymmetry coefficients: 0, 0.05, 0.1, 0.15, and 0.2.

this case, the application consists of an application with two stages whose cost totalizes 10 ms. The overhead introduced by communication mechanisms is close to 0.7 ms. Therefore, with an asymmetry of 0 all CPUs provide similar performance. Likewise, with an asymmetry coefficient of 0.2, the difference between the best and the worst scenarios, difference between the use of one CPU and another one is around 20 percent of the utilization available for the application.

The results of the scenario illustrate how a proper placement of the application processing stages may be beneficial because it takes advantage of the knowledge from the topology of the infrastructure (see Fig. 32). In this particular scenario, choosing the type of processor may imply differences in response time from a minimal 5 percent for small asymmetry to 25 percent in the most asymmetrical case. Furthermore, the scenario also shows that, by properly choosing the computational unit to execute a processing stage, the maximum input frequency to which the system may be increased from 140 Hz to 180 Hz. The results shown in Fig. 32 illustrate how the placement pattern is beneficial for the response time of the application.

5.8 Discussion

Our proposed catalog defined six patterns, which can be considered a basic collection of design recommendations for developing real-time application processing applications. For each pattern, namely: *single stage application*, *operator separation*, *fission*, *frequency modifier*, *fusion*, and *placement pattern*, Table 3 summarizes its expected impact in terms of utilization demands, expected response time and maximum input frequency. The results of the table summarize the main performance indicators associated to each pattern and refer to the scheduling framework described in Sections 3 and 4, and the test-bed described in this section. Many of the results are also valid for other stream processing frameworks, but they have to be properly adapted for each scheduling framework and application combination.

TABLE 3
Dominant Performance Patterns Tradeoffs

Pattern	*Utilization Demanded*	*Max input frequency*	*Response time*
Single Stage Application	Decreases (+)	Decreases ( )	Decreases (less abstraction overhead) (+) Increases (lack of parallel execution) ( )
Separator Operator	Increases ( )	Increases (+)	Decreases (parallel execution) (+) Increases (extra communications overhead) ( )
Fission	Increases ( )	Increases (+)	Decreases (parallel execution) (+) Increases (extra communications overhead) ( )
Frequency Modifier	Decreases (+)	Increases (+)	Increases (adds latency) ( )
Fusion	Decreases (+)	Decreases ( )	Decreases (less overhead from infrastructure) (+)
Placement	Decreases (+)	Increases (+)	Decreases (avoids asymmetric nodes) (+)

(+ means desired characteristic and means undesired)

The single stage application pattern groups all stages of an application into a single stage, decreasing the overall costs in terms of demanded utilization. Typically, it also decreases the maximum achievable computational frequency for the application because it does not take advantage of the existence of a network and/or serialization processes communicating machines, thus limiting maximum operational frequencies. Regarding response time, the pattern is able to reduce the overhead due to existence of a network communication. However, it also suffers from the lack of a parallel infrastructure able to reduce the response time of applications.

The operator separation pattern increases the maximum input frequency of the application, by splitting work into several sequential stages. As a result, the overhead due to the application increases, requiring additional machines that have to serialize/deserialize further messages. Another positive consequence of the pattern is that it increases the maximum operational frequency because the load gets split among different nodes. The response time gets reduced by the fact of using parallel computation, but the application also suffers an increase in overhead associated to the extra serialization/deserialization work.

The fission pattern increases the maximum input frequency of an application by parallelizing heavy stages of an application, which are split to several nodes to be merged later. However, this is done at the cost of adding extra computational overhead that requires extra utilization. On the other hand, the response time gets reduced by use of a parallel infrastructure and only gets increased by serialization/deserialization mechanisms.

The frequency modifier pattern filters inputs and/or outputs, thus decreasing application costs. However, it may also increase response time due to unexpected latencies that might happen during filtering. The pattern has benefits in terms of utilization, which gets reduced, and maximum input frequency, which gets increased.

The fusion pattern groups processing stages by decreasing maximum input frequencies. A positive effect is that it decreases utilization costs and response time, because serialization and network overheads disappear.

Lastly, the placement pattern uses information about the cluster and the application to select in which computational unit each processing stage is going to be hosted. The main advantages of the pattern are that it is able to increase the maximum input frequency supported by the system reduce response times. Another positive aspect of the pattern is that demanded utilization is decreased.

6 Conclusions and Ongoing Work

Distributed application processors allow the implementation of big data applications that require a continuous application of information to be analyzed. Part of these applications (like, for instance, HFT systems) have end-to-end quality-of-service requirements and may benefit from the use of real-time techniques. Taking this into account, in this paper a catalog of six patterns to develop real-time stream processing applications has been defined. The pros and cons of the different patterns have been exposed, as well as an analysis of the impact in performance that can be expected due to their usage. Results show that these patterns may have an important impact on some critical performance indicators of applications such as maximum input frequency, total processing cost and local or global utilizations. All patterns have shown a great potential and, therefore, are in our opinion relevant for application developers.

Our ongoing work comprises three interrelated research lines. The first one is the adaptation of these patterns to the context of RDF application processors, analyzing the overhead of semantic-web technologies [42]. Our second active research line turns on the applicability of these patterns in industrial environments [30], [44] that could benefit from the use of distributed processors. Finally, we also plan to deploy these patterns in complex distributed applications to analyze their mutual inter-relationships in depth.

Acknowledgments

This work been partially supported by Distributed Java Infrastructure for Real-Time Big Data (CAS14/00118). It has been also partially funded by eMadrid (S2013/ICE-2715), HERMES-SMARTDRIVER (TIN2013-46801-C4-2-R) and AUDACity (TIN2016-77158-C4-1-R); and also by European Union's 7th Framework Program under Grant Agreement FP7-IC6-318763. We are also in debt with our anonymous reviewers that improved the quality of the article.

References

[1] P. Zikopoulos and C. Eaton, *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. New York, NY, USA: McGraw Hill, 2011.

[2] N. Marz and J. Warren, *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Greenwich, U.K.: Manning Publica tion Co., 2015.

[3] J. Forsyth and L. Boucher, "Why big data is not enough," *Res. World*, vol. 50, pp. 26 27, 2015.

[4] R. Cortés, et al., "Sport Trackers and Big Data: Studying user traces to identify opportunities and challenges," INRIA Paris, Paris, France, Tech. Rep. RR 8636, 2014.

[5] M. Chen, M. Shiwen, and L. Yunhao, "Big data: A survey," *Mobile Netw. Appl.*, vol. 19, no. 2, pp. 171 209, 2014.

[6] K. Gang Hoon, S. Trimi, and C. Ji Hyong, “Big data applications in the government sector,” *Commun. ACM*, vol. 57, no. 3, pp. 78 85, 2014.

[7] H. V. Jagadish, et al., “Big data and its technical challenges,” *Com mun. ACM*, vol. 57, no. 7, pp. 86 94, 2014.

[8] V. N. Gudivada, R. Baeza Yates, and V. V. Raghavan, “Big data: Promises and problems,” *Comput.*, vol. 48, no. 3, pp. 20 23, Mar. 2015.

[9] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The Hadoop distributed file system,” in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol.*, 2010, pp. 1 10.

[10] Storm, “Distributed and fault tolerant real time computation,” 2015. [Online]. Available: https://storm.incubator.apache.org/

[11] Spark, “Lightning fast cluster computing,” 2014. [Online]. Avail able: https://spark.apache.org

[12] Samza, “Apache Samza distributed computing framework,” 2015. [Online]. Available: http://samza.apache.org/

[13] I. Gray, Y. Chan, N. C. Audsley, and A. J. Wellings, “Architecture awareness for real time big data systems,” in *Proc. 21st Eur. MPI Users’ Group Meet.*, 2014, Art. no. 151.

[14] P. Basanta Val, N. Fernandez García, A. J. Wellings, and N. C. Audsley, “Improving the predictability of distributed stream pro cessors,” *Future Generation Comp. Syst.*, vol. 52, pp. 22 36, 2015.

[15] L. T. X. Phan, Z. Zhang, B. T. Loo, and I. Lee, “Real time MapRe duce scheduling,” University of Pennsylvania, Philadelphia, PA, USA, Tech. Rep. N. MS CIS 10 32, 2010.

[16] Z. Tang, Z. Junqing, L Kenli, and L. Ruixuan, “A MapReduce task scheduling algorithm for deadline constraints,” *Cluster Comput.*, vol. 16, no. 4, pp. 651 662, 2013.

[17] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. D. Gill, “Parallel real time scheduling of DAGs,” *IEEE Trans. Parallel Dis trib. Syst.*, vol. 25, no. 12, pp. 3242 3252, Dec. 2014.

[18] A. Saifullah, J. Li, K. Agrawal, C. Lu, and C. D. Gill, “Multi core real time scheduling for generalized parallel task models,” *Real Time Syst.*, vol. 49, no. 4, pp. 404 435, 2013

[19] K. Lakshaman S. Kato, and R. Rajkumar, “Scheduling parallel real time tasks on multicar e processors,” in *Proc. 31st IEEE Real Time Syst. Symp.*, pp. 259 268, 2010.

[20] J. Kim, K. Lakschmanan, and R. Rajkumar, “Parallel scheduling for cyber physical systems: Analysis and case study on a self driv ing car,” in *Proc. ACM/IEEE International Conference on Cyber Phys ical Systems*, 2013, pp. 31 40.

[21] B. Theeten, I. Bedini, P. Cogan, A. Sala, and T. Cucinotta, “Towards the optimization of a parallel streaming engine for telco applications,” *Bell Labs Techn. J.*, vol. 18, no. 4, pp. 181 197, 2014.

[22] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm “A cata log of stream processing optimizations,” *ACM Comput. Surv.*, vol. 46, no. 4, Mar. 2014, Art. no. 46.

[23] R. Garibay Martínez et al. “Allocation of parallel real time tasks in distributed multi core architectures supported by an FTT SE network” in *Proc. Int. Conf. Archit. Comput. Syst.*, 2015, pp. 224 235.

[24] M. Fan, et al., “Enhanced fixed priority real time scheduling on multi core platforms by exploiting task period relationship.” *J. Syst. Softw.*, vol. 99, pp. 85 96, 2015.

[25] J. Carpenter, et al., “A categorization of real time multiprocessor scheduling problems and algorithms,” in *Handbook on Scheduling Algorithms, Methods, and Models*. Boca Raton, FL, USA: CRC Press, 2004, pp. 30 1.

[26] R. Davis and A. Burns, “A survey of hard real time scheduling for multiprocessor systems,” *ACM Comput. Surv.*, vol. 43, no. 4, 2011, Art. no. 35.

[27] L. Sha, et al., “Real time scheduling theory: A historical perspective,” *Real Time Syst.*, vol. 28, no. 2 3, pp. 101 155, 2004.

[28] B. Gedik, S. Schneider, M. Hirzel, and W. Kun Lung, “Elastic scal ing for data stream processing,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1447 1463, Jun. 2014.

[29] R. Mayer, B. Koldehofe, and K. Rothermel, “Meeting predictable buffer limits in the parallel execution of event processing oper ators,” in *Proc. IEEE Int. Conf. Big Data*, 2014, pp. 402 411

[30] P. Basanta Val and M. Garcia Valls, “A distributed real time Java centric architecture for industrial systems,” *IEEE Trans. Ind. Inf.*, vol. 10, no. 1, pp. 27 34, 2014.

[31] L. Golab and M. Tamer Ozsu, “Issues in data stream man agement,” *SIGMOD Rec.*, vol. 32, no. 2, pp. 5 14, 2003. [Online]. Available: http://doi.acm.org/10.1145/776985.77698

[32] M. Stonebraker, U. Çetintemel, and S. Zdonik, “The 8 require ments of real time stream processing,” *SIGMOD Rec.*, vol. 34, no. 4, pp. 42 47, 2005, Dec. 2005.

[33] L. Aniello, et al., “Adaptive online scheduling in Storm,” in *Proc. ACM Int. Conf. Distrib. Event Based Syst.*, 2013, pp. 207 218.

[34] T. Aniello, et al., “Cloud based data stream processing,” in *Proc. ACM Int. Conf. Distrib. Event Based Syst.*, 2014, pp. 238 245.

[35] L. Soares Indrusiak, “End to end schedulability tests for multi processor embedded systems based on networks on chip with priority preemptive arbitration,” *J. Syst. Archit. Embedded Syst. Des.*, vol. 60, no. 7, pp. 553 561, 2014.

[36] A. Jacobs, “The pathologies of big data,” *Commun. ACM*, vol. 52, no. 8, pp. 36 44, 2009.

[37] D. Miner and A. Shook, *Map Reduce Design Patterns: Building Effec tive Algorithms and Analytics for Hadoop and Other Systems*. Newton, MA, USA: O’Reilly Media, Inc., 2012.

[38] P. T. Goetz and B. O’Neill, *Storm Blueprints: Patterns for Distributed Real time Computation*. Birmingham, U.K.: Packt Publishing Ltd., 2014.

[39] S. Sharma, *Cassandra Design Patterns*. Birmingham, U.K.: Packt Publishing Ltd., 2014.

[40] H. Ching Hsien, “Intelligent big data processing,” *Future Genera tion Comput. Syst.*, vol. 36, pp. 14 16, 2014.

[41] N. Fernández García, J. Arias Fisteus, and L. Sánchez Fernández, “Comparative evaluation of link based approaches for candidate ranking in link to Wikipedia systems,” *J. Artif. Intell. Res.*, vol. 49, pp. 733 773, 2014.

[42] J. Arias Fisteus, N. Fernández García, L. Sánchez Fernández, and D. Fuentes Lorenzo, “Ztreamy: A middleware for publishing semantic streams on the Web,” *J. Web Sem.*, vol. 25, pp. 16 23, 2014.

[43] D. Mysore, S. Khupat, and S. Jain, “Big data architecture and patterns,” 2015. [Online]. Available: www.ibm.com

[44] H. Pérez and J. Gutiérrez, “Modeling the QoS parameters of DDS for event driven real time applications,” *J. Syst. Softw.*, vol. 104, pp. 126 140, 2015.

[45] J. C. S. dos Anjos, I. Carrera Izurieta, W. Kolberg, A. L. Tibola, L. Bezerra Arantes, and C. F. R. Geyer, “MRA++: Scheduling and data placement on MapReduce for heterogeneous environments,” *Future Generation Comp. Syst.*, vol. 42, pp. 22 35, 2015.

[46] L. Woo, K. Jin Soo, and M. Seungryoul, “Large scale incremental processing with MapReduce,” *Future Generation Comp. Syst.*, vol. 36, pp. 66 79, 2014.

[47] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design PAT TERNS: Elements of Reusable Object Oriented Software*. Boston, MA, USA: Addison Wesley Longman Publishing Co., Inc., 1995.

[48] A. Arasu, S. Babu, and J. Widom, “The CQL continuous query lan guage: Semantic foundations and query execution,” *VLDB J. Int. J. Very Large Data Bases*, vol. 15, no. 2, pp. 121 142, 2006.

[49] A. Arasu, et al., “STREAM: The Stanford data stream manage ment system,” Stanford InfoLab, Stanford, CA, USA, Rep. No. 2004 20, 2004.

[50] W. Thies, M. Karczmarek, and S. Amarasinghe. “StreamIt: A lan guage for streaming applications,” *Compiler Construction*. Berlin, Germany: Springer, 2002.

[51] D. Abadi, et al., “Aurora: A new model and architecture for data application management,” *VLDB J.*, vol. 12, no. 2, pp. 120 139, Aug. 2003.

[52] A. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke, “Flextream: Adaptive compilation of streaming applica tions for heterogeneous architectures,” in *Proc. 18th Int. Conf. Par allel Archit. Compilation Techn.*, 2009, pp. 214 223.

[53] L. Brenna, et al., “Cayuga: A high performance event processing engine,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2007, pp. 1100 1102.

[54] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, “Naiad: A timely dataflow system,” in *Proc. 24th ACM Symp. Operating Syst. Principles*, 2013, pp. 439 455.

[55] P. Basanta Val and M. Garcia Valls, “Towards a reconfiguration service for distributed real time Java,” presented at the REACTION 2012 Workshops, Puerto Rico, Dec. 4, 2012.

[56] P. Basanta Val and M. García Valls, “A simple distributed garbage collector for distributed real time Java,” *J. Supercomput.*, vol. 70, no. 3, pp. 1588 1616, 2014.

[57] J. Sun, “Fixed priority end to end scheduling in distributed real time systems,” University of Illinois at Urbana Champaign, Champaign, IL, USA, Tech. Rep. 97 1973, 1997.

[58] J. M. López, J. L. Díaz, and D. F. García, “Minimum and maximum utilization bounds for multiprocessor rate monotonic scheduling,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 7, pp. 642 653, Jul. 2004.

[59] J. Palencia and M. González Harbour, “Exploiting precedence relations in the schedulability analysis of distributed real time systems,” in *Proc. 20th IEEE Real Time Syst. Symp.*, 1999, Art. no. 328.

[60] C. Maia, M. Bertogna, L. Nogueira, and L. Pinho, “Response time analysis of synchronous parallel tasks in multiprocessor systems,” in *Proc. 22nd Int. Conf. Real Time Netw. Syst.*, Oct. 8 10, 2014, Art. no. 3.

[61] P. Axer, S. Quinton, M. Neukirchner, R. Ernst, B. Dobel, and H. Hartig, “Response time analysis of parallel fork join workloads with real time constraints,” in *Proc. 25th Euromicro Conf. Real Time Syst.*, 2013, pp. 215 224.

[62] H. Mei, I. Gray, and A. J. Wellings, “A Java based real time reac tive stream framework,” in *Proc. IEEE 19th Int. Symp. Real Time Distrib. Comput.*, 2016, pp. 204 211.

[63] M. T. Higuera Toledano, “Java technologies for cyber physical systems,” *IEEE Trans. Ind. Inf.*, vol. 13, no. 2, pp. 680 687, 2017.