Noname manuscript No.
(will be inserted by the editor)

Online and Transparent Self-Adaptation of Stream Parallel Patterns

Adriano Vogel[1,2] · **Gabriele Mencagli**[2] · **Dalvan Griebler**[1,3] · **Marco Danelutto**[2] · **Luiz Gustavo Fernandes**[1]

Received: date / Accepted: date

Abstract A representative part of real-world parallel applications is becoming more dynamic and long-running, demanding online (at run-time) adaptations. Stream processing is one of the application scenarios that demand continuous processing of data items arriving in real-time. However, it is challenging for humans to monitor and manually self-optimize complex and long-running parallel executions continuously. Moreover, although high-level and structured parallel programming aims to facilitate parallelism, several issues still need to be addressed for improving abstractions. This paper extends self-adaptiveness for supporting autonomous and online changes of the parallel pattern compositions. Online self-adaptation is achieved with an online profiler that characterizes the applications, combined with a new self-adaptive strategy and a model for smooth transitions on reconfigurations. The solution provides a new abstraction layer that enables application programmers to define non-functional requirements instead of hand-tuning complex configurations. Hence, we contribute with additional abstractions and flexible self-adaptation for responsiveness at run-time. The proposed solution is evaluated with different structures and processing characteristics on robust applications. The results show that it is possible to provide additional abstractions, flexibility, and responsiveness while achieving performance comparable to the best static configuration executions.

Keywords Parallel Programming · Self-Adaptive Parallelism · Parallelism Abstractions · Stream Processing

E-mail: adriano.vogel@acad.pucrs.br
[1]School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Brazil
[2]Department of Computer Science, University of Pisa (UNIPI), Italy
[3]Laboratory of Advanced Research on Cloud Computing (LARCC), Três de Maio Faculty (SETREM), Brazil

1 Introduction

Large amounts of data are being generated due to the proliferation of devices (e.g., sensors, cameras) to sense the external world. In this scenario, parallel computing is relevant for processing fast enough the high amount of data being generated [15, 22]. Moreover, the continuous data arrival requires stream processing applications to run for long or infinite periods, where such long executions are often subject to fluctuations in the environment (e.g., resource availability) and at the application level (input rate, workload) [27]. Hence, self-adapting entities (e.g., degree of parallelism, cores and their frequencies) during the execution is important for achieving responsiveness [5, 18, 26, 6].

From a programming perspective, structured parallel programming facilitates the task of parallelizing applications. Programmers can instantiate high-level pattern constructors and combine them in compositions [3, 25, 2]. In this scenario, online changing the pattern composition configurations* has been proposed for abstracting complexities from application programmers and increasing the adaptation space to provide flexibility [24]. This can reduce the burden on application programmers in such a way that configurations are adapted transparently [23]. However, the configuration space is usually large, which us challenging to find optimal configurations at run-time.

Moreover, more generic strategies for self-adaptive decision-making are needed, and dynamic changes can have detrimental effects on the Quality of Services (QoS). Hence, in previous work [27], we contributed with mechanisms for online self-adaptiveness of parallel patterns. The solution was integrated with a C++ programming framework (FastFlow [1]) and experimentally evaluated. In this extended version, we provide the following contributions:

– An autonomous self-adaptive strategy that avoids suboptimal configurations, which encompasses a lightweight online profiler of the application stages and an optimized decision-making for accuracy. The new strategy also supports latency as a new Service Level Objective (SLO). SLO refers to a metric of interest and its proper value to be enforced [14, 12].
– A model for smooth transitions between the parallel pattern configurations. A smooth transition is important because changing the configurations can have a critical impact on the QoS of applications (see Section 3.3).
– Extended validation of the proposed solution, including new scenarios and applications. Noteworthy, we provide a custom version of the PARSEC's Ferret application to regulate the Input Rate (IR) and support user-defined SLOs (throughput, latency).

This paper is organized as follows. Section 2 shows the motivational context of this work. Then, Section 3 presents the proposed solution and Section 4 provides an experimental evaluation. Moreover, Section 5 discusses related approaches and Section 6 concludes this paper.

*The term composition refers to the application topology (a.k.a. stream graph, graph topology). Here, the terms composition and configuration are used interchangeably.

2 Problem statement and motivation

The use of high-level parallel programming methodologies is a potential alternative to provide coding abstractions for application programmers [3], which can reduce the application programmers' burden. The main goal of high-level parallel programming is to reduce programming efforts while ensuring reasonable performance and portability. In this vein, pattern-based parallel programming provides composable recurrent structures instantiated by users/programmers, who can combine the patterns creating different configurations. In the scenario of stream processing applications running in multicore machines, there are pattern-based parallel programming frameworks. From the industry, a relevant example is Intel Threading Building Blocks (TBB) [28] and from academia, there are frameworks like FastFlow [1], GrPPI [8], and SPar [10].

In TBB, the application programmer/user can enable parallel execution by creating a parallel pipeline, declaring each function as a filter. Moreover, the programmer is responsible for defining whether a stage is parallel or sequential. In FastFlow, the user can also create pipelines with the application's routines and replicate (run in parallel) specific routines/stages using the Farm pattern.

From a runtime system's perspective, TBB creates tasks that are scheduled to a pool of threads, where dynamic scheduling controls thread oversubscription by avoiding context switching and time-sliced execution. But, TBB suffers from other problems related to the scheduling overhead with fine-grained tasks and I/O blocking operations within tasks. FastFlow, on the other hand, avoids these issues with a runtime where nodes are fixedly mapped onto threads, and the runtime can statically merge the nodes without changing the user functions. Nevertheless, FastFlow has a rigid execution model that may not be suitable for stream processing that are more irregular and dynamic applications. This model may increase the demand for resources without guaranteeing performance gains. Hence, we argue that there is a need to support adaptation for both of them.

In previous work [27], we evinced that stream processing applications that compute data in real-time require the application programmers to create a configuration of sequential or replicated stages. However, maintaining such a configuration for the entire execution can be limited because it has a high impact on QoS. Moreover, we have seen that it is not intuitive for application programmers to define these configurations. For instance, it was demonstrated that a video stream processing application that executes under fluctuations in the data IR or environment, the best configuration combining replicated, sequential, and merged stages varies with different scenarios that occur at run-time and from one programming framework to another.

The complexity of creating stage configurations increases when facing robust applications with several stages that can run sequentially or in parallel. For instance, Figure 1 shows the structure of the Ferret [21] application from the PARSEC benchmark suite, where we can note that the four middle stages were implemented as thread pools that can run in parallel. However, the profitability of running them in parallel can vary from how intensive and balanced

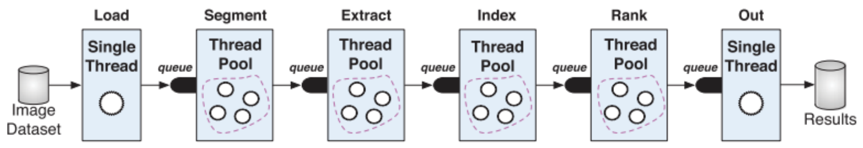


Fig. 1: PARSEC's Ferret pipeline structure. Extracted from [7].

the stages are. Additionally, the expected performance and QoS should also be considered for deciding whether sequential or parallel executions are applied.

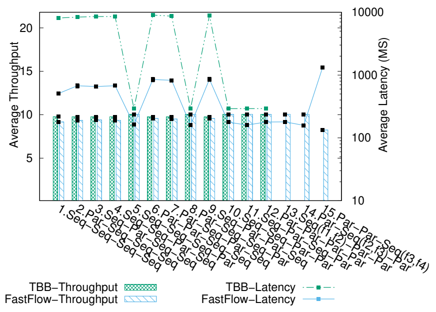
Ferret has a robust program structure, and it is known that it can be modeled with different shapes by composing and nesting parallel patterns [7], which we generically call *configurations*. A demonstrative example is provided in Figure 2 showing different implementations of the Ferret application with the frameworks TBB and FastFlow, where the setup described in Section 4. These results are from a new streaming version of the Ferret that computes data at a given IR and provides stream processing performance metrics like throughput and latency. For instance, in Figure 2a the data arrives at a fixed IR of 10 items per second, where 10 is a suitable throughput (items/s) for sustaining the IR. Latency is another relevant metric that corresponds to the time taken to compute a given item, where low latency is a constraint for many applications [25].

Here we show performance results from 15 relevant configurations, which are described in Section 4.2.2. Each configuration is a different scenario where stages are sequential, parallel, or merged. For instance, configuration 12 corresponds to the native implementation using Pthreads where all stages are parallel[†]. Notable, although configuration 12 demands more resources with 4 parallel stages, it is not the best performing latency.

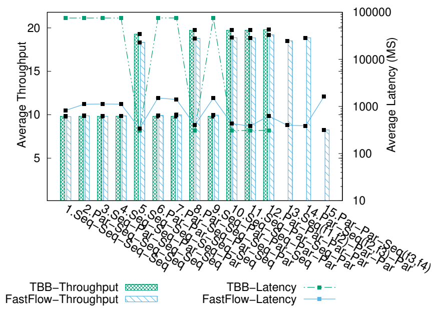
In fact, the best performance is when running in parallel the last stage, which is the most intensive one. In TBB, a reasonable latency is only achieved when the last stage is defined as parallel. Although defining such a configuration is easy, it is not intuitive and has a significant impact on the QoS. The problem is that defining such a configuration is up to the application programmers who are not experts in performance. Consequently, a potential misconfiguration would compromise the overall performance. Moreover, the first two stages are light because, in FastFlow, a configuration with these two stages merged (reducing resources consumption) showed no performance impact. It is important to note that there are no results from configurations 13, 14, and 15 with TBB because merging functions would require major changes to the application business logic code, which negatively impacts coding productivity.

A relevant implication from the Figure 2 is that determining the best configuration can be complex and error-prone, which is not a suitable responsi-

[†]The Pthreads version is not structured pattern-based, such results are not shown here for the sake of visual clarity. The performance of FastFlow and TBB is comparable with the native implementation. The reader interested in comparison can refer to reference [7].



(a) Input Rate 10 items/s.



(b) Input Rate 20 items/s.

Fig. 2: Example on a PARSEC's Ferret configuring the stages with thread pool. Latency is on logarithmic scale

bility for the application programmers. Moreover, the best configuration to be used can vary at run-time because of the usual fluctuations on stream processing applications [18,27]. For instance, the input rate can change due to network fluctuations or variations in the number of devices producing data. Resource availability can also fluctuate in shared/dynamic environments like Clouds. Consequently, stream processing applications are expected to support flexible adaptations at run-time. Considering that several aspects correlate in a nonlinear manner, the user/programmer should not be expected to on the fly hand-tune the configurations. In Section 3 we present the proposed solution for providing abstractions and dynamic self-adaptation at run-time.

3 Proposed solution

Here we describe the solution that we envision for providing additional abstractions to application programmers and to enable flexible online self-adaptation at run-time. In our perspective, a feasible solution is to support users/programmers to set only high-level goals like throughput or latency SLOs, which can be offered as parameters or attributes [12,9].

3.1 General design goals and requirements

An effective approach for dynamic parallel pattern compositions is expected to meet the following goals:

– Abstract from application programmers the demand to find the best configuration of parallel patterns. The programmers should set SLOs instead of hand-tuning configurations. This can be achieved with autonomous strategies that can provide a suitable QoS (achieves the SLO consuming fewer resources).

- Enable dynamic and flexible reconfigurations at run-time for avoiding the need to recompile and rerun long-running stream processing applications.
- Respond at run-time to changing conditions, where dynamic reconfigurations can enable self-adaptiveness.

There are also important **requirements** for ensuring QoS and efficiency:

- No application downtime, the adaptation should not interrupt the data processing and output provisioning.
- Smooth transition on reconfigurations. Change from one composition to another can be necessary. However, this is expected to be stable without intrusiveness and overheads like latency glitch and throughput spikes [24].
- A suitable solution is expected to be lightweight and execute without demanding a significant extra amount of resources.
- Efficiency: an optimal configuration is the one that meets user/programmer goals and that requires fewer computing resources. Consuming fewer resources increases the system efficiency, reduces energy consumption, and costs less (i.e., pay-per-use environments).

3.2 Decision-making strategy

A decision-making strategy is the core of a self-adaptive strategy responsible for deciding the best actions to be enforced. However, assumptions are necessary for designing a flexible and generalizable decision-making strategy. The rationale for such assumptions is to abstract technicalities that have to be implemented for each specific scenario. In Section 3.4 we show an example of implementation in a C++ programming framework. The main necessary assumptions are:

- Runtime system's mechanisms are available for applying dynamic changes to configurations and for changing from one configuration to another.
- The strategy receives alternative configurations to be considered at runtime. Such configurations could be defined by a user or by an expert system.
- The strategy receives information for making decisions, which can be provided by external monitoring entities.
- The data to be processed comes at a given IR and the strategy is alerted in case the IR changes.

The designed self-adaptive decision-making for the pattern composition configurations is described here at a high-level abstracting lower implementation details. This description is expected to be sufficient for the reproducibility of the proposed solution. The decision-making has the following steps:

1. **Online profiling step**: Lightweight instrumentation gathers execution statistics from each stage, which help in characterizing how intensive and balanced they are. The profiling step measures the actual processing capacity of each stage and ranks them by less to more intensive. Moreover, a

given stage is tagged if its average service time (time spent computing the tasks) is at least 20% higher than all other service time of stages. This step is executed at the beginning of the execution with the first configuration provided. It can be repeated at any time, such as when a given application enters a new processing phase. For increasing the profiling accuracy, it is recommended that the first configuration executes all stages sequentially.

2. **Evaluation**: Assesses if the defined SLO is satisfied. If positive, goes to step 6 with the current configuration. If not, goes to step 3. The decision-making strategy infers that two values are significantly different when they contrast higher than 20% (a threshold). In [26] this parameter was ascertained as a suitable one for stream processing applications.
3. **Shortlisting configurations**: Previous work [27] applied experimental runs with all configurations. In practice, this can affect QoS because bad configurations could be used. The new proposed strategy aims to reduce the number of experimental runs and still cover the suitable parts of the configuration space. This is possible by using the profiling step's information to search and shortlist the potentially optimal configurations. If more than one configuration can be optimal, the strategy goes to step 4, or if only one is suitable, sets this one as active and go to step 6. Also, a configuration with the most replicated stages is set if the SLO is not being achieved and there are no bottlenecks or optimal configurations.
4. **Trial phase**: Activates each suitable configuration candidate for a given time interval and gathers statistics. The rationale for executing each shortlisted configuration is to evaluate which configurations in practice perform better for the specific application, workloads, and environments. This measures each configuration's actual processing capacity. Moreover, the time interval is a relevant parameter that expert users can customize. However, the default value from empirical results for testing each configuration is 5 seconds. The previous implementation from [27] tested all configurations for only 1 second because this solution did not profile and shortlisted the best one. It tried all configurations available. In practice, we have seen that one second as time interval is too low and subject to unpredictable variations during the training step. In the current optimized version, 5 seconds is the proper time interval because a suboptimal configuration will not be tested as these do not pass the shortlisting step.
5. **Selects the best configuration**: This phase evaluates which configurations from step 5 achieved the desired SLO. If no configuration achieved the goal, enforces the one with the best value. On the other hand, if more than one is optimal, select the one with light stages merged and fewer replicated stages. This decision is to enforce the most optimal configuration that maintains QoS and at the same time consumes fewer resources.
6. **Stable phase**: Stabilizes in a configuration and periodically evaluates if the SLO is being satisfied. In practice, every 10 seconds, the current status is verified. This comparison uses the same threshold from step 2 to avoid the instability caused by fluctuations. Steps 3 to 5 are repeated if the data

gathered indicates changes or if the SLO is violated. In this case, it is searched for additional bottlenecks and potentially optimal configurations.

It is important to note that the decision-making strategy is not employing an exhaustive search. In fact, up to 20 alternative configurations are supported. However, only the suitable ones are to be activated and tested at run-time. Moreover, Section 3.3 addresses the relevant aspect of how to achieve safety and stability when transitioning from one configuration to another.

3.3 Transitioning between configurations

When reconfigurations are necessary at run-time, it should be smooth without compromising the QoS. One solution is to employ a draining phase that flushes all the tasks from the configuration to be stopped before activating the new one. From a theoretical standpoint, a flush is relevant for avoiding that two configurations run simultaneously, which would cause unpredictable performance variability or losses (throughput spikes and latency glitches [24]).

One may think that the draining phase is a trivial problem solved by simply waiting for a random time. However, we have seen that choosing for how long to wait for the draining to complete is a non-trivial value in practice. On the one hand, not waiting for enough causes performance and resource fluctuation, influencing the training step and QoS. On the other hand, waiting for too long on reconfigurations can also hurt QoS and the designed goal of avoiding application downtime. Consequently, we tackled this challenge by developing an autonomous model that automatically estimates how long to wait. Such a model is mainly expected to find a balance value being accurate, generic, and lightweight. The draining time estimation is inspired by adaptive self-clocking from Jacobson/Karels scheme [16] for estimating the TCP retransmission timeouts, our samples/entities subject to variance on parallel applications are:

Number of items buffered: This aspect refers to buffer sizes used in the runtime system and the number of computing elements (*e.g.,* nodes) that use buffers for communicating in a given composition. For a generalization purpose, we assume that the runtime system provides mechanisms for collecting this value or provides parameters for limiting the buffer's sizes.

Computations' service time: Considering that applications have significant contrasts in terms of grain and tasks computational intensiveness, the service time is expected to be a broad metric and flexible for different applications. A monitor can gather data and feed the model with the information of the average service time of the tasks being processed at a given moment, which corresponds to a given active configuration.

Processing Capacity: Refers to the computation capacity of the active configuration to process the tasks buffered and finalize the draining phase. We have discussed in Section 2 that each configuration and programming framework has specific processing capacities in terms of the number of nodes and the mapping to threads. Consequently, only considering the service time and the number of buffered tasks would be suboptimal because the actual computational capacity of each configuration varies. Generally, the processing capacity considers the number of computing elements that compute a given application's business logic code. Additional nodes/elements that do not process business logic code necessary in the programming framework should not be included in the processing capacity.

From the provided description, it is possible to note that the model is not simple and must consider the variability of service time, runtime parameters, and processing capacity. Moreover, the model must continuously measure and accurately estimate the time to drain. Considering the potential overhead of the machinery to collect and process data at run-time, in Section 4.3, we characterize the transitioning between configurations using this model.

3.4 Implementation

C++ frameworks and libraries available were considered for implementing the proposed solution. There are industry and academic solutions such as Intel TBB [28], FastFlow [1], and SPar [10]. Considering the support for performing adaptations at run-time, TBB has mechanisms only for dynamic task distribution and load balancing, where other mechanisms have to be implemented by hand. Considering that we are interested in higher-level abstractions, FastFlow is more flexible by supporting dynamic adaptation on several aspects like the parallelism degree and communication queues' concurrency modes [6, 25]. Thus, FastFlow was used for implementing the proposed solution.

Abstracting specific and complex implementation technicalities, the proposed self-adaptive strategy was implemented in FastFlow in the form of a ready-to-use C++ header-only library. The solution works by default in FastFlow's blocking mode. Figure 3 provides a representation of the implementation. The *Manager* is the entity that implements the self-adaptive strategy and is embedded in the data source and uses the runtime system's mechanisms for applying changes in an autonomous mode. Another entity is *Monitor* implemented as another embedded entity within the Sink stage that gathers data and feds the *Manager*. Figure 3 represents a scenario where a pipeline with 3 stages is active and others configuration declared remain inactive. Moreover, the lower part of Figure 3 demonstrates the achievable flexibility because several other configurations can be created and activated at run-time if necessary.

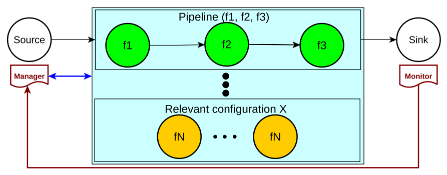


Fig. 3: Implementation in FastFlow.

In the current implementation, the applications' business logic code is reused by alternative configurations. For using the proposed solution, the application programmers have to include self-adaptive strategy headers and add two extra code lines for calling the *Manager* and *Monitor*. Moreover, the higher accuracy of the profiling step demands on each application stage that a timer is initialized and that the *Monitor* is called.

The programmer can declare custom configurations using the FastFlow interface. For instance, the three staged pipeline used in Figure 3 can be declared and added with two C++ code lines, and other compositions can be declared and included with similar coding productivity. The characteristics of the configurations (e.g., buffer sizes, stages are sequential, parallel, or merged) also have to be defined at a higher parametric description. The application programmer can be assisted with tools for designing additional configurations and coding, such as RPL [17] and SPar's compiler [10].

4 Evaluation

4.1 Experimental setup

A multicore machine equipped with an Intel Xeon processor 2.40 GHz (12 cores- 24 threads) and 32 GB of memory was used for running experiments. The operating system is Ubuntu Server 16.04 and G++ compiler (7.5.0) with -O3 flag. The FastFlow runtime system's buffer sizes were set to 1.

The strategy is characterized in a scenario simulating IR changes. The performance is evaluated with static configuration executions using the same configurations as a baseline. We call static configuration the executions where a given configuration is compiled and maintained during the entire execution. The execution's correctness was ascertained by hashing the outputs. In Section 4.2 we describe the applications and configurations tested. Then, in Section 4.3 the decision-making is characterized with the different SLOs supported and application characteristics, Section 4.4 evaluates the performance of self-adaptive executions compared to baseline static executions, and Section 4.5 provides an overview of the results.

4.2 Applications and configurations

The evaluation of the proposed solution covers different applications and configurations. Considering that each application has a specific number of stages, workload pattern, and balance between stages, for each application we created a scenario of relevant configurations to be available for the self-adaptive strategy to use (or not) at run-time. In this evaluation, configurations using parallel stages use the default value of 2 replicas (parallelism degree) per stage.

4.2.1 Synthetic application

Synthetic is an application where 10,000 tasks with a total service time of 24 milliseconds (ms) are computed. This application has three functions where different configurations can be composed with a sequential or parallel stage. *Configuration 1* has the three sequential stages representing scenarios where the performance demand is not high, and the stages are balanced. *Configuration 2* has the first stage computing in parallel and stages 2 and 3 are sequential. Such a configuration can be suitable when the stages are not balanced, and the parallel stage is the bottleneck. *Configuration 3* has the second stage computing in parallel and the stages 1 and 3 sequential and *Configuration 4* has the third stage computing in parallel and the stages 1 and 2 sequential.

In *Configuration 5* stages 1 and 2 execute in parallel and the third stage is sequential, such a configuration is relevant when the performance demand is higher and the sequential stage is lighter than the others. *Configuration 6* has stages 2 and 3 execute in parallel and the first stage is sequential and in *Configuration 7* all stages execute in parallel, which can be relevant when the performance demand is higher and the stages are balanced. It is important to note that *Configuration 7* tends to consume more resources.

The configurations that are suitable vary from application characteristics and the performance demand. In this synthetic application, many other configurations could have been declared and made available for the self-adaptive strategy. However, these 7 are representative enough for evaluating the accuracy and performance of the proposed solution. Additionally, this synthetic application allows flexible customizations of load balancing between the stages. Two application versions were created for evaluating the self-adaptive strategy: one where the stages are balanced and the other that has unbalanced stages. In the balanced version, if we attribute a total computing weight of 6 each stage would have a weight of 2, meaning they are perfectly balanced. With the balanced stages, the optimal configurations are 1 and 7. On the other hand, the unbalanced version has also a total stages' weight of 6. However, the first stage has a weight of 1, the second weight of 3, and the third stage has a weight of 2. In this case, the major bottleneck is the second stage and if the performance demand is high the third stage can become the second bottleneck.

4.2.2 Ferret

Ferret is a stream-parallel benchmark that searches for similarities on data items like audio, images, and video [21, 7]. For the evaluation, we modified the original ferret version to a streaming version. This streaming version computes data items at a fixed speed instead of reading the data as fast as possible from the disks, simulating a scenario where the data comes in real-time from the network at a given speed to be computed. The streaming version also covers the instrumentation to collect stream processing metrics like throughput and latency, instead of the execution time. We used the PARSEC native as the input set, which is a representative workload.

Ferret can be modeled with several configurations. In this evaluation, we created 15 alternative configurations for challenging the self-adaptive strategy to find the best ones at run-time considering different scenarios. The configurations from 1 to 12 explore possible combinations of sequential and parallel stages, whereas configurations 13, 14, and 15 cover the merging of sequential stages. The merging can be relevant for cases where the stages are unbalanced and the lighter can be merged. Importantly, the self-adaptive strategy has a profiling step for characterizing each application and its workload.

4.2.3 Person Recognition

The Person Recognition is a stream processing application [11] where we used a customized version that has three functions to detect and verify people in video streams. It receives a video input and applies a denoising step for improving the quality. Then, it detects and marks the faces with a red circle. These faces are compared with the training set of faces. The experiments were run using as input a 30 seconds video with a resolution of 260 pixels.

In the Person Recognition, we used 5 alternative configurations from reference [27] that cover sequential, parallel, and merged stages. In *Configuration 1* all application functions are merged in a sequential stage (1S.). *Configuration 2* separates the functions into two stages (Pipe-2S.), whereas *Configuration 3* runs with one more stage(Pipe-3S.). Considering that some applications or performance goals are not suitable for sequential stages, *Configuration 4* shows an example of a pipeline with a parallel stage (P.S.1) running all functions, which in FastFlow is a Farm parallel pattern. Considering that functions can be decomposed into multiple parallel stages, *Configuration 5* provides a variation of *Configuration 4* where two parallel stages (P.S.2) are employed, which can be useful for applications that are not embarrassingly parallel.

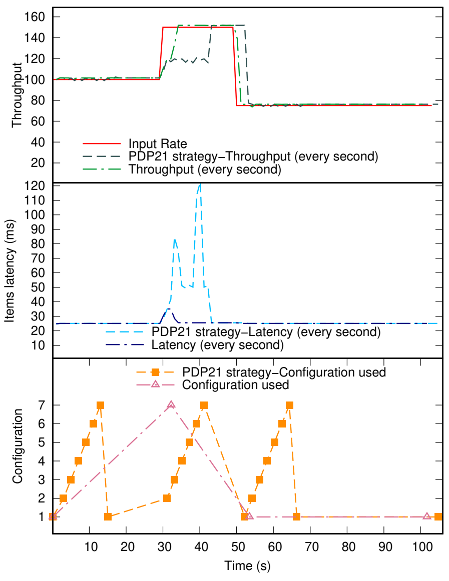
4.3 Self-adaptive strategy characterization

This section characterizes the decision-making of the self-adaptive strategy.

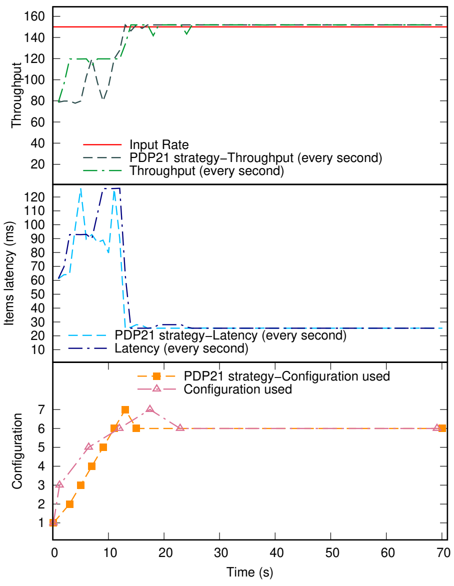
4.3.1 Decision-making with throughput SLO

The first results to characterize the solution are from the synthetic application. The proposed solution is compared to the previous one called PDP21 [27]. Figure 4a shows the results of balanced application stages where the defined SLO is to have a throughput (items/s) outcome equal to the IR. Representative for stream processing scenarios, there are two changes in the IR as fluctuations that can occur at run-time.

The PDP21 strategy started trying all configurations. Considering that the SLO was being achieved, the new strategy avoided the unnecessary training in step 2 of the decision strategy (see Section 3.2). Reacting to the IR change around the second 30, the new strategy accurately on one step went to configuration 7 that executes all stages in parallel, inferred by the profiling step that detected balanced stages. By contrast, the PDP21 strategy tested all configurations again, resulting in lower throughput and higher latency for several seconds due to testing suboptimal configurations. Then, after the second 50, the IR dropped, and the executions went back to configuration 1 that sustained the SLO.



(a) Balanced stages.

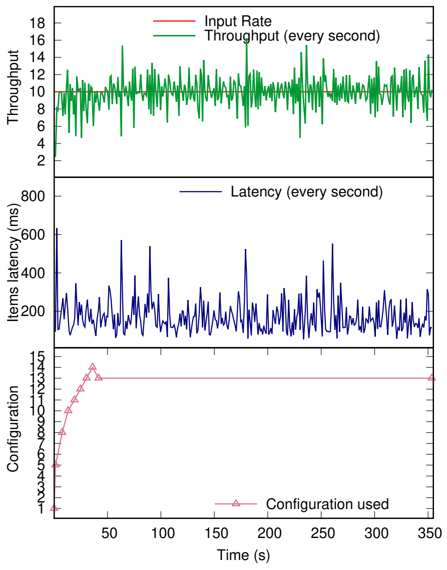


(b) Unbalanced stages.

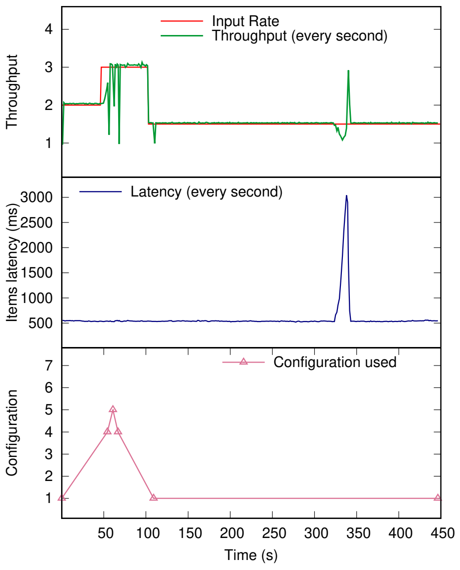
Fig. 4: Characterization with the synthetic application.

The outcome from Figure 4b with the unbalanced stages is distinct. The execution starts with the throughput (items/s) lower than the IR. Consequently, the new strategy searches for better configurations, which results in shortlisting and entering the trial phases with configurations 3, 5, 6, and 7. The rationale behind such as decision is that the profiling correctly detected the second stage as the bottleneck (Section 4.2.1) and shortlisted the configurations where the second stage is parallel as an attempt to overcome the bottleneck. Even during the trial phase, it is noticeable that the performance improved in terms of throughput and latency. Then, the strategy stabilized with configuration 6 that provided QoS and demands fewer threads than configuration 7.

The PDP21 strategy had to apply all configurations to find the best one. By contrast, the new strategy inferred the best configuration with fewer steps, which is very relevant for applications [18]. The strategies used different time intervals for testing each configuration, the new strategy uses the default value of five seconds, and PDP21 tests configurations for one second. Five seconds is expected to be a time interval suitable for a wider range of applications and can also be customized for specific application characteristics. Another relevant aspect evinced in figure 4 concerns the transitioning model. Notably, the transitions between configurations are smooth without throughput drops or latency glitches. The new strategy is also characterized with more realistic applications where we only show results from the new strategy for the sake of visual clarity.



(a) Ferret with IR 10.



(b) Person Recognition with a changing IR.

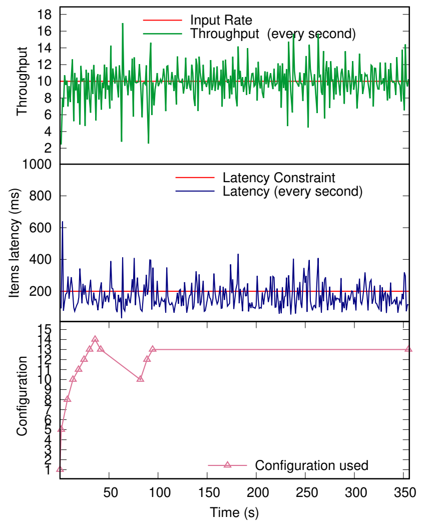
Fig. 5: Throughput (items/s) Characterization.

Figure 5 provides results with two applications. Figure 5a shows Ferret where is notable that the metrics collected in real-time present fluctuations due to the application's processing characteristics. Importantly, the self-adaptive strategy's profiling step detected the *Rank* stage as the bottleneck and shortlisted configurations where this stage executes in parallel. Then, after the trial phases, it stabilized with configuration 13 that presented a suitable performance, and that consumes fewer resources with the first stages merged.

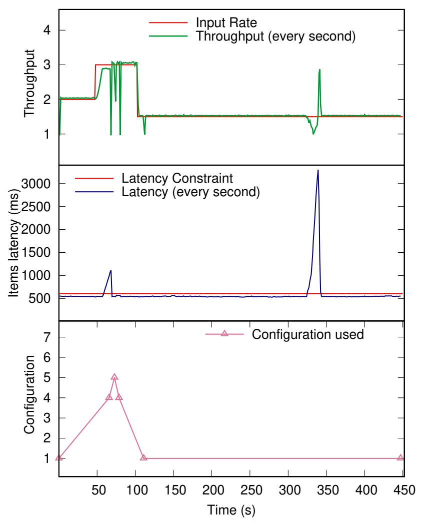
The results from the Person Recognition application emphasize the accuracy of the decision-making, which chooses the best configuration according to IR changes. In a scenario with SLO violations, configurations 4 and 5 were shortlisted and tried to achieve higher performance. Hence, the strategy applied configuration 4. Under a lower IR, the self-adaptive strategy returned to configuration 1 to increase efficiency by demanding fewer resources. Although the throughput reduced during some reconfigurations, the transitioning model showed accuracy because there was no application downtime.

4.3.2 Decision-making with latency SLO

Figure 6 shows results from the self-adaptive strategy with latency SLO that is a contraint. Figure 6a evinces Ferret with a SLO of 200 ms with fluctuations due to Ferret's characteristics. The strategy stabilizes with configuration 13, overcoming the bottleneck on stage *Rank*. Near the second 100, a significant application fluctuation increased the latency. Hence, the strategy detected an SLO violation and searched for a better configuration because some change could have occurred. The third pool stage (*Vec*) was detected as an additional bottleneck, where the strategy shortlisted and tried configurations 10, 12, and 13, where the two bottlenecks are executed in parallel. However, the strategy returned to configuration 13 that remained the most suitable configuration.



(a) Ferret SLO 400 ms.



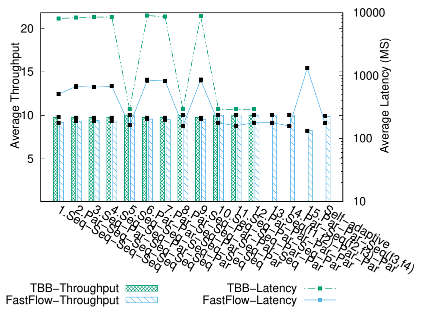
(b) Person Recognition SLO 600 ms.

Fig. 6: Latency Characterization.

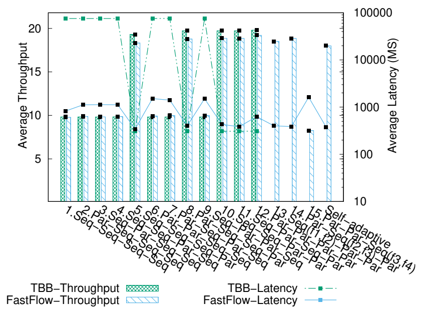
Figure 6b evinces a latency constraint of 600 ms, where a fluctuating IR varies from 1.5 to 3 FPS. A reconfiguration may be needed when the IR changes because not sustaining the IR increases the buffering and latency. This occurred after the second 50 when the IR increased. The active configuration did not sustain the IR, which increased the number of items buffered and the latency. Hence, the strategy detected the latency violation and self-adapted to configuration 4. After the second 300, there is a fluctuation (also seen in Figure 6b) that caused the throughput to decrease and the latency to increase. This fluctuation were not long enough for a reconfiguration because the latency SLO was being achieved when the self-adaptive entered the training step. Notably, the transition between configurations occurs without application downtime, which indicates that the model's estimation is accurate.

4.4 Performance evaluation

In this section, we compare the final performance of the self-adaptive executions to static ones using real-world applications. The results from the executions are an average of 10 runs and we also show the standard deviation, which is difficult to visualize in the figures because it is very low. Figure 7 shows results from Ferret, where the self-adaptive strategy was able to effectively adapt and find the best configuration (13) for achieving a performance competitive with the best static configurations. The best throughput in FastFlow, the runtime system of the self-adaptive solution, was with configuration 12 where the self-adaptive throughput was 6.3% lower. However, in the latency metric, the self-adaptive was 39.7% better than static FastFlow with configuration 12.



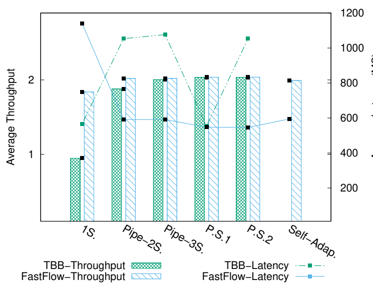
(a) Throughput with IR 10.



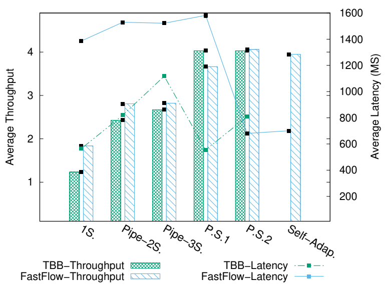
(b) Throughput with IR 20.

Fig. 7: Performance comparison with Ferret. Latency on logarithmic scale.

Figure 8 provides results from Person Recognition, where a notable outcome is that the self-adaptive executions have a good performance competitive with the best static scenarios. This is due to the accuracy of the self-adaptive strategy, especially the profiling, trial, and transitioning steps.



(a) Throughput with IR 2.



(b) Throughput with IR 4.

Fig. 8: Performance comparison: Person Recognition Application

4.5 Results summary

The evaluation provided here shows that our solution for online self-adapting the parallel patterns:

– has effective mechanisms for reconfiguring and maintaining program's executions correctness;
– accurately characterizes the applications for finding bottleneck stages;
– transparently reacts to unpredictable fluctuations (IR, workload) that occur at run-time;
– locates in a few steps, the best configuration according to different SLOs (throughput, latency) and that demands fewer resources.
– the model for transitioning is sufficiently accurate as no application downtime neither latency glitches occurred due to reconfigurations (Section 4.3);
– the new abstraction and responsiveness through self-adaptation is effective achieving a competitive performance (Section 4.4), which indicates that the overhead of instrumentation and self-adaptation can be negligible.

5 Related work

A number of entities can be adapted at run-time [15]. Although being complex to manage with applications' metrics like throughput and latency, batching can be used as an optimization in some application scenarios [4,23]. The number of cores and their frequency can be changed at run-time for reducing energy consumption [6] as well as dynamic tuning the communication queues' concurrency modes [25].

There are also works dynamically changing the degree of parallelism of parallel stages [5,18,26]. However, these optimizations are not flexible enough for the adaptations that real-world stream processing applications demand. In this vein, changing the application structure was proposed as a more powerful and flexible entity to be dynamically adapted [24].

Concurrent recompilation has been proposed for reducing application downtime [24]. However, the techniques needed for controlling downtime are intrusive, which can affect the computing (ordering, throughput) and consume additional resources. In practice, we have seen that this approach is hard to generalize to other applications and programming frameworks.

There are approaches used profiling for guiding the deployment of stream applications [20]. By contrast, we focus on more critical requirements and scenarios with online profiling, where reconfigurations are online without pausing the applications and restarting their executions from scratch.

The study of [13] proposed *Grizzly*, a solution that encompasses adaptive compilation to change the executions at run-time, which is a reaction to changes in applications' data characteristics. Grizzly's decision-making performs only speculative optimizations where in practice the accuracy can vary.

Contrasting with the related approaches, we provide a strategy that uses online profiling and tries only the suitable candidate configurations. This design is an attempt to be generic enough to a wide range of applications and frameworks. In this vein, it is possible to avoid the demand to rerun the applications to apply changes and not demanding intrusive techniques (e.g., input duplication, resource throttling, output smoothing). Recompilation is unnecessary because multiple configurations are created, and the best one for SLO and efficiency is found at run-time. Considering that our solution increases the self-adaptation space, we can combine it with other less flexible entities, for instance, self-adapting the degree of parallelism that is representative for configurations with parallel stages and where high throughput is desirable. To the best of our knowledge, this is the first approach that provides an accurate decision-making strategy for choosing the best parallel pattern configuration to be used and that online self-adapts when it is necessary.

6 Conclusion

In this paper, we presented a solution for supporting self-adaptive pattern compositions for stream processing applications. A relevant implication of these results is that self-adaptiveness can provide new efficient abstractions and autonomous responsiveness for applications that compute data in real-time.

The components of our solution can be generalized to be used in other scenarios. For instance, the online profiler has the potential to be used for other application classes and workloads. Moreover, we expect that the decision-making strategy is generic enough to be customized with other programming frameworks and execution environments, self-adapting other entities, and applicable to regular parallel applications [19]. Hence, we are in the process of documenting and open-sourcing the components of the self-adaptive strategy.

This study is limited in some aspects. We designed the solution to be generic, but mechanisms in the programming frameworks are necessary for achieving self-adaptive pattern compositions. Currently, the FastFlow framework supports such mechanisms. In future works, we intend to support additional applications and workloads in our solution. Moreover, we intend to evaluate our solution for self-adapting other parallel patterns.

References

1. M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. Fastflow: High-Level and Efficient Streaming on Multicore. *Programming multi-core and many-core computing systems, parallel and distributed computing*, pages 261–280, 2017.
2. A. Collins, C. Fensch, H. Leather, and M. Cole. Masif: Machine learning guided autotuning of parallel skeletons. In *Int Conf on H Perf Comp*, pages 186–195. IEEE, 2013.
3. M. Danelutto, G. Mencagli, M. Torquati, H. González-Vélez, and P. Kilpatrick. Algorithmic Skeletons and Parallel Design Patterns in Mainstream Parallel Programming. *Int J Parallel Prog*, pages 1–22, 2020.
4. T. Das, Y. Zhong, I. Stoica, and S. Shenker. Adaptive Stream Processing using Dynamic Batch Sizing. In *ACM Symp. on Cloud Computing*, pages 1–13. ACM, 2014.
5. T. De Matteis and G. Mencagli. Keep Calm and React with Foresight: Strategies for Low-latency and Energy-efficient Elastic Data Stream Processing. *SIGPLAN Not.*, 51(8):13:1–13:12, Feb. 2016.

6. D. De Sensi, T. De Matteis, and M. Danelutto. Simplifying Self-Adaptive and Power-Aware Computing with Nornir. *Future Gener Comput Syst*, 87:136–151, 2018.
7. D. De Sensi, T. De Matteis, M. Torquati, G. Mencagli, and M. Danelutto. Bringing Parallel Patterns Out of the Corner: The P^{3}ARSEC Benchmark Suite. *ACM Trans Archit Code Optim*, 14(4):1–26, 2017.
8. D. del Rio Astorga, M. F. Dolz, J. Fernandez, and J. D. Garcia. A generic parallel pattern interface for stream and data processing. *Concurr Comput*, 2017.
9. A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: Self-Regulating Stream Processing in Heron. *VLDB Endowment*, 10:1825–1836, 2017.
10. D. Griebler, M. Danelutto, M. Torquati, and L. G. Fernandes. SPar: A DSL for High-Level and Productive Stream Parallelism. *P Proc Lett*, 27(01):1740005, March 2017.
11. D. Griebler, R. B. Hoffmann, M. Danelutto, and L. G. Fernandes. Higher-Level Parallelism Abstractions for Video Applications with SPar. In *Intl. Conference on Parallel Computing*, pages 698–707, Bologna, Italy, September 2017. IOS Press.
12. D. Griebler, A. Vogel, D. D. Sensi, M. Danelutto, and L. G. Fernandes. Simplifying and implementing service level objectives for stream parallelism. *J Supercompt*, 76:4603–4628, June 2019.
13. P. Grulich, B. Sebastian, S. Zeuch, and et al. Grizzly: Efficient stream processing through adaptive query compilation. In *ACM SIGMOD Intl. Conf. on Management of Data*, pages 2487–2503, 2020.
14. J. Hellerstein and et al. *Feedback Control of Computing Systems.* Wiley, 2004.
15. H. Herodotou, Y. Chen, and J. Lu. A survey on automatic parameter tuning for big data processing systems. *ACM Comput Surv*, 53(2), 2020.
16. V. Jacobson. Congestion avoidance and control. *ACM SIGCOMM Comp Com*, 18(4):314–329, 1988.
17. V. Janjic, C. Brown, K. Mackenzie, and et al. RPL: A Domain-Specific Language for Designing and Implementing Parallel C++ Applications. In *Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing*, pages 288–295. IEEE, 2016.
18. V. Kalavri, J. Liagouris, M. Hoffmann, and et al. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *USENIX Oper. Systems Design and Implemen.*, pages 783–798, 2018.
19. S. Kehrer and W. Blochinger. Equilibrium: an elasticity controller for parallel tree search in the cloud. *J Supercomp*, pages 1–35, 2020.
20. X. Liu, A. V. Dastjerdi, R. N. Calheiros, C. Qu, and R. Buyya. A Stepwise Auto-Profiling Method for Performance Optimization of Streaming Applications. *ACM Trans Auton Adap*, 12(4):1–33, 2017.
21. Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Ferret: a toolkit for content-based similarity search of feature-rich data. In *ACM SIGOPS/EuroSys European Conf. on Computer Systems 2006*, pages 317–330, 2006.
22. G. Mencagli, M. Torquati, A. Cardaci, and et al. Windflow: High-speed continuous stream processing with parallel building blocks. *IEEE T Parall Distr*, 2021.
23. P. Metzger, M. Cole, C. Fensch, M. Aldinucci, and E. Bini. Enforcing deadlines for skeleton-based parallel programming. In *IEEE RTAS*, pages 188–199, 2020.
24. S. Rajadurai, J. Bosboom, W.-F. Wong, and S. Amarasinghe. Gloss: Seamless live reconfiguration and reoptimization of stream programs. *ACM Not*, 53(2):98–112, 2018.
25. M. Torquati, D. De Sensi, G. Mencagli, M. Aldinucci, and M. Danelutto. Power-aware Pipelining with Automatic Concurrency Control. *Concurr Comput*, 31(5):e4652, 2019.
26. A. Vogel, D. Griebler, and L. G. Fernandes. Providing high-level self-adaptive abstractions for stream parallelism on multicores. *Softw Pract Exp*, 51(6):1194–1217, 2021.
27. A. Vogel, G. Mencagli, D. Griebler, M. Danelutto, and L. G. Fernandes. Towards On-the-fly Self-Adaptation of Stream Parallel Patterns. In *Euromicro Intl. Conf. on Parallel, Distributed and Network-Based Processing*, pages 89–93, Valladolid, 2021. IEEE.
28. M. Voss, R. Asenjo, and J. Reinders. *Pro TBB: C++ Parallel Programming with Threading Building Blocks.* Apress, 2019.