The Journal of Supercomputing (2024) 80:22213–22244
https://doi.org/10.1007/s11227-024-06191-w

Enhancing self-adaptation for efficient decision-making at run-time in streaming applications on multicores

Adriano Vogel[1,2] · **Marco Danelutto**[3] · **Massimo Torquati**[3] · **Dalvan Griebler**[4] · **Luiz Gustavo Fernandes**[4]

Accepted: 3 May 2024 / Published online: 21 June 2024

Abstract

Parallel computing is very important to accelerate the performance of computing applications. Moreover, parallel applications are expected to continue executing in more dynamic environments and react to changing conditions. In this context, applying self-adaptation is a potential solution to achieve a higher level of autonomic abstractions and runtime responsiveness. In our research, we aim to explore and assess the possible abstractions attainable through the transparent management of parallel executions by self-adaptation. Our primary objectives are to expand the adaptation space to better reflect real-world applications and assess the potential for self-adaptation to enhance efficiency. We provide the following scientific contributions: (I) A conceptual framework to improve the designing of self-adaptation; (II) A new decision-making strategy for applications with multiple parallel stages; (III) A comprehensive evaluation of the proposed decision-making strategy compared to the state-of-the-art. The results demonstrate that the proposed conceptual framework can help design and implement self-adaptive strategies that are more modular and reusable. The proposed decision-making strategy provides significant gains in accuracy compared to the state-of-the-art, increasing the parallel applications' performance and efficiency.

Keywords Parallel computing · Parallelism abstractions · Self-adaptation

✉ Adriano Vogel
adriano.vogel@jku.at

[1] JKU/Dynatrace Co-Innovation Lab, LIT CPS Lab, Johannes Kepler University Linz, Linz, Austria

[2] Laboratory of Advanced Research on Cloud Computing (LARCC), Sociedade Educacional Três de Maio (SETREM), Três de Maio, Brazil

[3] Computer Science Department, University of Pisa, Pisa, Italy

[4] School of Technology, Pontifical Catholic University of Rio Grande do Sul, Porto Alegre, Brazil

1 Introduction

Currently, parallelism is everywhere as computer systems support many architectural paradigms (e.g., multicores, co-processors) [25]. Despite the great potential of parallelism to provide high performance, parallelism tends to increase programming complexity by requiring programmers to provide parallel code. Moreover, parallel executions require managing low-level aspects (e.g., communication, synchronization). Therefore, providing parallel programming abstractions to non-expert programmers is a potential alternative to mitigate the existing complexities.

In addition to programming abstractions, many configurations and optimizations are applicable to parallel executions, e.g., concerning where the tasks are executed and how many threads/processes execute a given task. Moreover, a significant part of parallel applications run for very long periods, which makes them subject to changing conditions and fluctuations at the application level (e.g., workload, input rates, and environment) or the execution environment (e.g., availability of resources, network changes). Therefore, a given parallelism configuration can provide quality of service (QoS) during an instant of the program's execution and hurt the performance in a different moment of the execution [37]. In such a scenario, applying online optimizations that autonomously adapt the parallelism configurations at run-time can improve QoS and provide new abstractions.

Consequently, many techniques and approaches can be applied to scenarios to achieve autonomous executions. Self-adaptation [13] is a relevant example that can be viewed as the capability of the systems to be autonomous, changing their behavior under conditions that can occur at run-time.

On the one hand, self-adaptation can make the executions of parallel applications more intelligent, reducing human efforts and assisting programmers in error-prone activities [32, 41]. On the other hand, self-adaptation at run-time is still challenging in terms of flexibility and efficiency: the adaptation space available at run-time is still limited. For instance, several parallel applications have complex graph topologies or composition structures, which demands flexible mechanisms to enact adaptation actions and efficient decision-making strategies to decide which configurations should be conveniently enforced.[1] Moreover, the design of the self-adaptive strategies is expected to be improved to support the decoupling of modules and enable more reusable/generic approaches.

Previously [38], we contributed with mechanisms for enabling self-adaptation on the number of replicas in complex-structured parallel applications. The solution was integrated with the FastFlow [1] C++ programming framework, and we replicated the state-of-the-art DS2 decision-making strategy [19] to decide what parallelism configurations should be applied. In previous work [38], it was notable that DS2 has limited accuracy in determining optimal parallelism configurations when applied to resource-constrained scenarios such as multicores, which reduced the overall system

[1] We consider complex structures the ones comprising more than one parallel stage.

efficiency and the performance of parallel applications. We believe that such limitations demand further research and new approaches.

Our perspective is that the decision-making should first be efficient and optimized locally and then distributed to more machines when higher performance is needed. Hence, we first consider the scenario of efficient executions on multicore machines. Therefore, in this paper, we extend our previous work and provide the following novel contributions[2]:

- A conceptual framework to design decision-making within self-adaptive strategies;
- A new decision-making strategy for a self-adaptive number of replicas in complex compositions. In Sect. 4, we provide an optimal decision-making strategy that is compared to the state-of-the-art solution called DS2 [19];
- A comprehensive evaluation methodology for benchmarking the decision-making strategies. The methodology provides relevant variations in terms of the type of applications with different numbers of parallel stages, the number of bottleneck stages and their location, stages' intensiveness, and running in different architectures of shared memory multicore machines.

This article is structured as follows. Section 2 presents this work's context. In Sect. 3, we describe the proposed conceptual framework. Then, Sect. 4 demonstrates how this conceptual framework can be applied to a concrete scenario, providing a new decision-making strategy to support advanced self-adaptation in parallel applications. Moreover, Sect. 5 discusses the comprehensive evaluation methodology proposed here. Then, Sect. 6 discusses the experimental results. Section 7 overviews the state of the art. Finally, Sect. 8 concludes this article.

2 Background

2.1 Parallel computing

Parallelism exploitation is one of the best alternatives to improve the performance of real-world applications. However, parallelism exploitation to achieve performance gains demands coding to introduce parallel routines. Moreover, it is usually necessary to configure parallelism parameters and manage the executions to achieve safe and correct executions. Thus, many applications are still executed with limited parallelism levels, providing a limited performance to end-users [8, 37].

Introducing parallelism techniques tends to be particularly challenging for application programmers who are not performance experts. Refactoring code to introduce parallelism usually results in application programmers facing a trade-off between coding productivity and performance. This occurs because parallelism increases

[2] This article is a revised and extended version of our paper "Revisiting self-adaptation for efficient decision-making at run-time in parallel executions" [38].

performance, but using efficient parallelism routines is a time-consuming task that usually reduces productivity and requires knowledge of low-level mechanisms.

Using high-level parallel programming methodologies can provide coding abstractions for application programmers, reducing the application programmers' burden. The main goal of high-level parallel programming can be defined as reducing programming efforts while ensuring reasonable performance and code portability. High-level abstractions tend to be provided by approaches that hide the complexities of parallelism from programmers.

2.2 Stream processing

Stream processing is a relevant paradigm that needs parallelism to provide QoS [2]. Moreover, this paradigm benefited from high-level parallelism and its abstractions. Stream processing applications can be defined as programs that continuously compute data items, where a *stream* is a given input that arrives from sources in the form of an infinite sequence of items [15]. Examples of stream sources include equipment (radars, telescopes, cameras, among others) and file bases (text, image). Moreover, processing stages (a.k.a. operators) consume the incoming streams by applying computations. The stages tend to be organized as a graph where each stage performs a specific computation and the stream item flows through the graph. Nowadays, many different application domains rely on stream processing architectures to handle large volumes of data in near real-time.

The characteristics of stream processing applications vary depending on the data source and computational performance. One of the most highlighted aspects is the continuous and unbounded arrival of data items [2]. Lately, we have seen a significant increase in the number of devices producing data to be processed in real-time. As stream processing systems usually have to process streams with low latency and high throughput, parallelism emerged as an opportunity to process faster those data items. Consequently, parallelism can be seen as an opportunity to increase the overall performance of a stream processing system. In the context of this study, we refer to parallelism in stream processing as the possibility to concurrently perform different operations over independent stream items. The next section provides further parallelism details related to stream processing.[3] This study uses stream processing applications as a use case to provide parallelism abstractions and as a workload to evaluate the proposed solution (see Sect. 5.3).

2.3 Self-adaptation overview

The software engineering area has been evincing that modern software systems should operate in dynamic conditions without downtime [41]. Software systems use self-adaptation concepts to collect data and adjust the system's behavior. Self-adaptation focuses on applying actions at run-time due to the need to adapt to specific

[3] Further technical material on Stream Processing can be found in references [2, 10].

unpredicted execution scenarios, which is a possible way to handle uncertainties [41]. This is usually done by collecting and extracting knowledge from the data. Then, it becomes possible for the system to decide which assertive actions can be taken. For instance, try to increase the QoS with optimal configurations. In short, this is an interpretation from an outside angle (external) where a self-adaptive system is viewed as a black box that abstractly performs optimal decisions.

QoS is a relevant concern for self-adaptation. The **SASO** (stability, accuracy, settling time, and overshoot) properties [13] are relevant ones to be considered when designing self-adaptive approaches. *Stability* refers to the capacity to produce the same output under a given condition. *Accuracy* is related to achieving the control goal with sufficiently good decision-making, and a *short settling time* is desired to reach an optimal state quickly. Moreover, *overshooting* is generally expected to be avoided in such a way that unnecessary computing resources are not used.

These properties are relevant to be included when designing decision-making strategies for self-adaptive systems, which inspire the conceptual framework (see Sect. 3) and new decision-making strategy proposed (see Sect. 4). Self-adaptation is intended to be applied to manage parallelism aspects and provide high-level abstractions, where we intend to tackle two main parallelism challenges: performance and productivity. We expect that the performance of parallel applications can be increased with optimal decision-making strategies that accurately find performatic configurations. Moreover, we expect that productivity can be improved by providing ready-to-use and more usable self-adaptive abstractions. Programmers can be more productive without worrying about complex parallelism configurations. Section 2.4 dives deeper into this works' research problem.

2.4 Research problem

One of the main research challenges is making self-adaptation more generic. This occurs because usually a given adaptation space must be managed manually or automatically by self-adaptive approaches. The appropriate actions provided by self-adaptive decision-making are the ones that provide efficient alternatives that also enable abstractions to users/programmers. However, self-adaptation is still complex to design, implement, and validate. Mainly because it is currently challenging to reuse elements/modules of an adaptive solution when implementing another one [41]. **Therefore, the first problem addressed in this study is how to make more modular and generalizable the decision-making strategies that decide which adaptation actions to apply at run-time**. We believe that better design of the decision-making strategies is one way toward more generic solutions. Consequently, we argue that decision-making should be designed and modeled with the help of conceptual frameworks. Section 3 shows our proposed conceptual framework.

Moreover, before applying adaptation actions, there is a relevant need for mechanisms that make it possible to apply adaptation actions at run-time. In parallel computing, it is necessary to achieve communication/synchronization of all the entities. Usually, a mechanism is provided through implementations on specific systems/frameworks. Considering the limited adaptation space available to apply action at

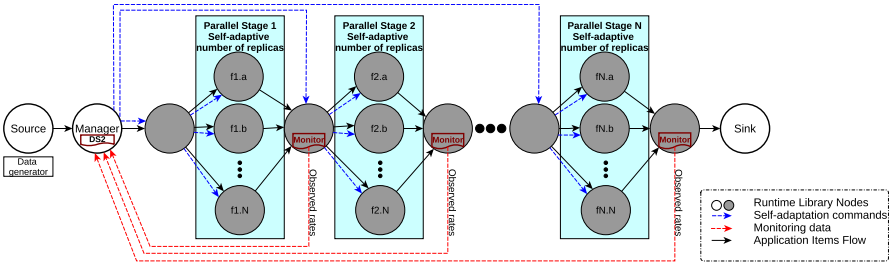


Fig. 1 Mechanism implementation and decision-making strategy integration

run-time, we provided mechanism and decision-making strategies for self-adapting the replicas in applications with a single parallel stage. Considering the demand for additional mechanisms to achieve the necessary flexibility for parallel applications, in [39], we proposed mechanism and decision-making strategies to self-adapt the parallel patterns and online change the applications' graph topologies.

Figure 1 illustrates a regular parallel application with several parallel stages and a source and sink stage. The number of parallel stages varies from one application to another, according to the number of functions and their computational weight in terms of load intensiveness. Moreover, inside the parallel stages, a relevant configuration is the number of replicas to be used on each parallel stage, which is usually defined considering their computational weight. For instance, a parallel stage that is computationally heavy and performs intense operations usually uses a configuration with more replicas, which tends to increase its performance. On the other hand, a lighter stage usually runs with fewer replicas.

In addition to defining if a stage should run in parallel or not, the problem of defining correctly the number of replicas that the parallel stages should use is still complex due to the large number of stages, the limited availability of data that requires fast processing to decide, and the many fluctuations that can occur at run-time. In terms of mechanisms to apply the adaptation actions at run-time, the FastFlow programming framework supports mechanisms to self-adapt the number of replicas in a single independent parallel stage [37]. Therefore, in previous work [38], we implemented a mechanism to coordinate between multiple parallel stages into the runtime system, which uses a manager node to send adaptation commands (increase or decrease the number of replicas).[4] The blue arrows in Fig. 1 illustrate the mechanism's communication channels implemented.

Using the validated mechanism, in [38] the decision-making of DS2 [19] was reproduced and evaluated in our scenario of C++ parallel applications running in multicore machines. Currently, DS2 is considered the most prominent approach from state of the art to determine the number of replicas in parallel stages, which was proposed as a general controller for making scaling decisions in distributed stream processing. DS2 was compared to related approaches when it was proposed

[4] The reader interested in further details about the mechanisms can refer to references [38].

(see reference [19]), and DS2's executions achieved the best performance.[5] However, it was notable that DS2's decision-making was limited in terms of accuracy under unbalanced parallel stages [38], such unbalances are usually caused by the stages' functions executed and their computational weight in terms of load intensiveness. Moreover, DS2's inaccuracies were more prominent in scenarios where lighter stages come after intensive stages and where the resources are not broadly available (e.g., multicore machines, edge computing nodes). Notably, our findings were aligned with recent related works applying DS2 to different scenarios [33]. In short, **the second research problem covered in this paper can be summarized as: How to have a decision-making system that accurately and efficiently defines a suitable number of replicas in applications with many parallel stages**. Section 4 describes the proposed solution for such a problem, and Sect. 6 shows the experimental results of evaluating the proposed approach.

3 Conceptual framework

In Sect. 7.1, we overview the existing frameworks for self-adaptation. The existing frameworks do not provide a fully decoupled decision-making strategy, which results in low flexibility and generalizability. In this work, we focus primarily on a modular design of the decision-making strategies. Then, optimal decision-making could be applied to specific contexts that provide the means (mechanisms) to apply adaptation actions.

We adopt the presentation of the proposed framework according to FORMS (FORmal Models for Self-adaptation) [17, 42] to define how our framework interacts with the components from MAPE-K (Monitor, Analyze, Plan, and Execute) feedback loop. We consider the formal model for specifying self-adaptive systems as a reference model [17, 42]. In FORMS, self-adaptive systems comprise two parts/subsystems: a meta-level that makes decisions and controls the base-level. The base level is specific to domain functionalities, such as an execution environment of a given computing application that ranges from a multicore machine to a highly distributed and flexible cloud environment [21]. Hence, we comprehend the FORMS relation to our proposed framework in the following way: the meta-level corresponds to the potentially generic decision-making strategy, while the base-level relates to the mechanisms needed for applying adaptation actions in a given scenario.

Figure 2 shows the reference framework architecture.[6] Although Fig. 2 relates to the conceptual view of self-adaptation in parallel computing, here we focus on decision-making and how the conceptual view can be practically applied. For instance, the abstract self-adaptive managing system is enacted here by an

[5] DS2 was also replicated in reference [33], DS2 is currently being also applied to other systems such as Apache Flink, which is a very popular distributed stream processing engine (see more https://nightlies.apache.org/flink/flink-kubernetes-operator-docs-main/docs/custom-resource/autoscaler/).

[6] This representation had some inspirations in how FORMS was mapped to the context of self-protecting systems in reference [43].

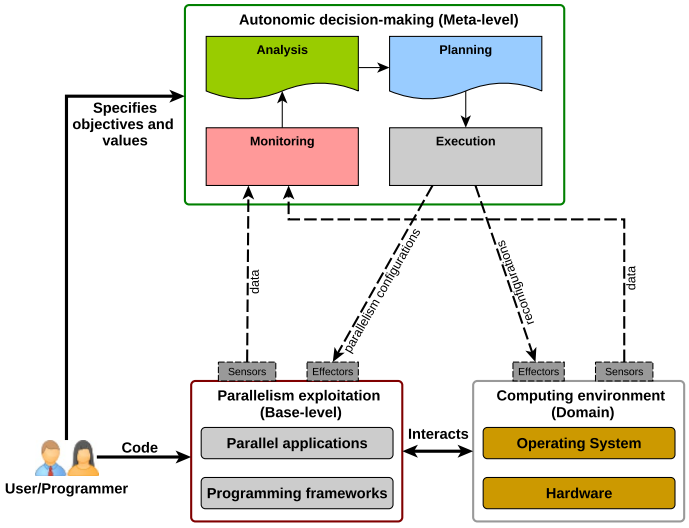


Fig. 2 Reference framework architecture

autonomic decision-making loop, a closed-loop between the decision-making and the base-level/domain.

Usability is a relevant aspect of the proposed framework. It is believed that efficient abstractions can be provided to improve the productivity of users/programmers. We understand that users should only interact with the decision-making framework via machine-readable descriptive languages or parameters to define their objectives. The users are expected to set high-level objectives (e.g., expected throughput of 10 tasks per second) and then rely on autonomous executions using the proposed framework to achieve their objectives [37]. In Sect. 4.4, we provide a concrete example of how usability can be improved in practice.

Moreover, our proposed framework is intended to be flexible and executed interactively. Hence, there is no strict order for the interactions shown in Fig. 2 to occur, e.g., one can design a given decision-making strategy with fixed steps to apply optimizations to increase efficiency without the users/programmers changing their objectives. We expect that it begins with the users/programmers defining their goals. Then, the decision-making is autonomic [13, 22] and interacts with controlling the base-level (e.g., runtime/programming frameworks, mechanisms) to enforce the user objectives. The decision-making at the meta-level collects monitoring data using sensors to verify if the user objectives are being achieved.

Since parallel computing and adaptations at run-time are complex, the main focus on generalizability is at the meta-level which must be highly customizable to different scenarios. For instance, heuristics, threshold algorithms, or auto profiling can be effective decision-making in some scenarios [39]. Balancing configurations in scenarios even more complex than the one considered in this work may require accurate decision-making strategies encompassing modern artificial intelligence approaches [12, 41]. One can expect that finding a suitable decision-making

approach to a specific scenario is complex. We believe that generalizability is needed in the sense of a more software engineering approach that decouples the mechanisms' implementations from the runtime system of a given scenario. This could allow for easy prototyping of new decision-making strategies, which is where a conceptual framework plays a crucial role.

Moreover, with a decision-making framework, it can be possible to encapsulate everything related to the best configuration to be employed, such that the framework completely and conveniently abstracts the decision-making process. For instance, one can provide a workflow in which decision-making can find the best configuration in a given adaptation space with many configurations possible. Using the implementation principles of a framework, such decision-making could be easily applied and tested in similar adaptation spaces. Additionally, in case one decision-making strategy shows to be unsuitable in a given scenario, new strategies could be easily integrated and evaluated. In this work, we show a concrete example of such a scenario. In Sect. 2.4, we show the limited decision-making implemented. Then, in Sect. 4, we provide a new strategy that is evaluated in Sect. 6.

4 A new decision-making strategy

This section describes how the conceptual framework can be applied to decision-making strategies for self-adaptation in parallel computing. Considering the limitations of previous approaches highlighted in Sect. 2.4, we proposed a new decision-making strategy to autonomously and efficiently determine optimal configurations of the number of replicas to be used in parallel stages. Although abstracting lower-level implementation details, our approach is described with enough details to enable its replication. Considering that we focused on our solutions' generalizability, modules proposed in previous works were decomposed into components and applied to the new decision-making strategy. This is made possible by adopting the design principles of the decision-making strategy following the conceptual framework (Sect. 3).

4.1 Components

The architecture of the new decision-making strategy follows the conventional architecture depicted in Fig. 1. However, the manager node runs another decision-making strategy (see Sect. 4.2) and the Monitor is extended with stages profiler. The decision-making utilizes the following updated modules:

1. *Manager* This module implements the self-adaptive strategy to decide the configurations to be enforced. In Sect. 4.2, we describe how the decision-making is performed. Importantly, w.r.t. the reference framework architecture (see Sect. 3), the *manager* module performs the Analysis and Planning steps evinced in Fig. 2.
2. *Application metrics monitor* This entity was proposed in previous work [37]. In this work, the monitor was implemented as a module and integrated into the architecture considered here. The monitor collects performance traces of the

applications, which data is used as input by the stages profiler. Considering the reference framework architecture (see Sect. 3), the *monitor* module belongs to the meta-level shown in Fig. 2.

3. *Stages profiler* In previous work [39], we proposed an online profiler to measure the processing capacity of each stage, where the online profiler was used to find a suitable parallel pattern. Adopting the design of the conceptual framework, in this work, we reimplemented the profiler (called *auto-profiler* here) to characterize the processing capacity of the applications' parallel stages. The *auto-profiler* runs as a module along with the generic application metrics monitor.

To reduce as much as possible to the instrumentation overhead, our approach encompassed all knowledge and good practices established when applying self-adaptation to parallel computing. Using this existing knowledge of C++ solutions targeting multicores, we achieved negligible monitoring overhead by collecting and filtering data in hundreds of nanoseconds and optimizing the decision-making strategies in such a way that final performance is comparable to the best static cases [36, 37]. The final result is lightweight instrumentation for efficient execution in multicore machines [37, 39].

4.2 Decision-making strategy

The decision-making is built out of the following steps:

1. *Characterization* The parallel application runs for a given time interval (e.g., 5 s).
2. *Data collection* After the *characterization* step, the decision-making strategy collects the application's metrics data and the target execution hardware system, specifically using the metrics monitor and stages profiler modules.
3. *Finding an optimal configuration of the number of replicas* Considering the data gathered, the decision-making builds a performance model w.r.t. the relation between the processing stages and the actual performance goal. By building this relation, the decision-making estimates how many replicas are necessary for each parallel stage. It is important to note that the performance model focuses on and applies to computing capabilities. The experimental results provided in Sect. 6 comprehensively evaluate the proposed performance model, including its processing and decision-making accuracy.
4. *Resources availability analysis* It is a training step that compares the total number of necessary replicas to the actual computing capacity available in the specific machine being used to run the parallel application. This is an additional analysis step that enforces coordination between all independent parallel stages before applying the adaptation actions. The decision-making can reduce the number of replicas in a balanced way when the actual availability of resources is below the requirement to avoid resource contention.
5. *Steady state* The decision-making strategy interacts (e.g., via function calls) with the mechanism to apply the new parallelism configuration, then stabilizing

and resuming the normal application execution. In case of external changes, the decision-making can return to the first step and find a new suitable configuration.

4.3 Implementation

In previous work [38], we validated the mechanism for applying the decision-making at run-time.[7] Several parallel programming frameworks and libraries have been created to exploit parallel computing, such as StreamIt [34], Open Multi-Processing (OpenMP) [6], Intel's oneAPI Threading Building Blocks [18], and Message Passing Interface (MPI) [16]. In our work, we are using FastFlow [1] because it is the framework that we have the expertise, and flexibility support to implement and apply self-adaptation. FastFlow supports mechanisms implemented into the runtime system to coordinate between multiple parallel stages, which use a manager node to send adaptation commands (increase or decrease the number of replicas). The blue arrows in Fig. 1 illustrate the mechanism's communication channels implemented.

Abstracting specific implementation technicalities, the proposed strategy was integrated into FastFlow as a ready-to-use C++ library. The implementation of the decision-making strategy is included in regular parallel applications as header-only files. The monitoring and profiling are additional lines to be included.[8] The implementation of the decision-making strategy is abstracted from the users (see Sect. 4.4), which adopts the design principles of the conceptual framework (Sect. 3).

4.4 Usability

We expect that the solution proposed in this section can be a tool for (non-expert) application programmers to execute their applications with suitable configurations and more autonomous executions.

In the long term, we expect that no additional coding should be required from the application programmers to use our solution. They should have ready-to-use abstractions that are automatically integrated within the programming frameworks. From a usability perspective, a more relevant facet is using our solution from the applications' execution angle. The application programmers should be (ideally) only concerned with defining high-level goals, such as a service level objective (SLO). Another way to enable the application programmers to configure their high-level goal is through declarative files, e.g., XMLs used in [8]. Our solution is currently implemented to enable the application programmers to set SLO as execution parameters when they run the applications. For instance, to define the self-adaptive strategy to enforce a target throughput with a value of 100 (items/second):

[7] The reader interested in further details of the mechanism can refer to reference [38].

[8] A complete functional example of integrating self-adaptive decision-making strategies into C++ parallel applications can be found in reference [37].

```
-application-binary throughput 100
```

Executing the application binary integrated with the decision-making plus the target SLO and its value enables the self-adaptive strategy to make decisions at run-time fully transparently. Hence, this is another abstraction intended for application programmers: execution abstractions. Such an abstraction is expected to be very relevant for long-running applications, where we expect to avoid the need for human operators to apply adaptation actions under the usual changes at run-time manually.

It is important to note that the solution proposed here is intended to improve the configurations with multiple parallel stages by supporting a self-adaptive number of replicas within each parallel stage, which paves the way to increase the system's efficiency and applications QoS. Moreover, the solution proposed here was designed for this work's specific scenario. However, our solution is not limited to a specific application. In fact, the proposed decision-making is expected to be generic enough to apply to the vast majority of complex structured parallel applications that execute following the dataflow processing model, allowing the monitors to collect application metrics and profile the stages' intensiveness. We also expect conceptual and technical efforts provided here to be easily applied to classical parallel applications that usually have less dynamic executions, making it easier to utilize decision-making strategies.

Considering that our primary focus is on providing flexible and feasible self-adaptive strategies, Sect. 5 shows the methodology to evaluate the proposed decision-making strategy. Our approach is applied and validated to the execution of stream processing applications, which is a representative paradigm present in several data-intensive applications (see Sect. 2.2). The rationale behind this decision is that stream processing is a scenario with more strict requirements and dynamic executions, which demands elaborate techniques and methods.

5 Experimental plan

5.1 Experiments' objectives

The approach proposed in Sect. 4 is evaluated here. Our proposed approach is compared to the most prominent approach from state of the art called DS2 [19]. [9] Although there are aspects of DS2 that do not apply to the multicore scenario, e.g., serialization and deserialization times, the DS2's decision-making strategy was reproduced in previous work [38] following its description from reference [19]. The following are the specific objectives to evaluate the decision-making strategies.

[9] The comparison with DS2 is expected to be sufficient as DS2 outperformed related approaches [19].

- Evaluate the reliability and generalizability of the mechanism for applying self-adaptation and the conceptual framework. The evaluation covers different decision-making strategies in high throughput applications running in robust machines.
- Measure the impact of the number of replicas in the application metrics and system resources.
- Evaluate the accuracy of decision-making strategies when applied to a concrete scenario.
- Measure the impact of configurations provided by the decision-making strategies in the performance of the applications in the different metrics and the systems resource efficiency.
- Measure the impact of configurations provided by the decision-making strategies in the FastFlow runtime system communication behaviors, namely, blocking and non-blocking (See Sect. 5.2 for definitions).
- Measure the stability of the decision-making strategies when executed in a varied range of application types, different numbers of parallel stages, the number of bottleneck stages and their location, stages' intensiveness, and running in different architectures of shared memory multicore machines.

It is important to note that both decision-making strategies have a training step where monitoring and profiling data is collected. Then, each decision-making strategy infers the optimal number of replicas for each parallel stage. We present results with different training step times in the strategies to evaluate if such value impacts QoS.

5.2 Experimental setup

We first executed the experiments in the same multicore machine utilized in previous chapters and sections called M1, equipped with two Intel Xeon E5-2620 processors (a total of 12 cores - 24 threads) and 32 GB of memory for running experiments. M1 runs with Ubuntu Server 16.04 and G++ compiler (7.5.0). We also provide complementary results from additional machines, M2 and M3. M2 has two Intel Xeon Silver 4210 (a total of 20 cores - 40 threads) and 64 GBs of memory. M2 runs with Ubuntu Server 20.04 and G++ compiler version 9.3.0. M3 is a machine equipped with 2 sockets AMD EPYC 7551 processors 32-Core Processor (in total 64 cores, 128 threads) and 128 GBs of memory.

Moreover, we executed the experiments in the machines in a dedicated mode. No other workloads were running simultaneously. Evaluating with multiple applications running simultaneously is left for the future.

The FastFlow's runtime system parameters of the threads/nodes were configured without any custom pinning policy so that the OS's scheduler can allocate the threads in the cores.[10] The rationale behind this choice is that we focus on evaluating the impact of adaptations at the system/application level.

[10] Low-level policies can be designed to optimize the mapping of threads to physical cores, which can achieve performance gains by improving memory accesses and cores affinity.

The FastFlow runtime system offers two communication behaviors: non-blocking and blocking. Non-blocking, the default behavior, utilizes active waiting rather than blocking mechanisms in send/receive nodes. While this can enhance performance, it typically results in higher resource utilization. Conversely, the blocking mode is a customization that conserves resources by keeping executing nodes/threads blocked when no items or tasks are to be computed. Both communication behaviors are examined in our experiments. We include these evaluations to expand our assessment of the programming framework's performance metrics and resource consumption, aiming to provide a comprehensive understanding of decision-making impacts.[11]

It is important to note that the experiments provided in this section intend to evaluate the impact of the decision-making strategies on QoS and system efficiency. Therefore, validating our approach in one practical framework (FastFlow) is expected to be enough to provide relevant accuracy and QoS insights to the multi-core scenario. Throughput is considered the most relevant metric measured in items per second (I/s) processed. Consequently, the target throughput is defined as an SLO with the same value as the input rate. Average latency is also measured, which refers to the time to compute a stream item. The data items are set to arrive at different fixed speeds (using a data generator module). Such values are set according to representativeness to real-world applications and the feasibility of the machines used. The decision-making strategies use a single training step to determine the number of replicas, which is to evaluate the decision-making accuracy to achieve low settling time.

5.3 Applications

The first experiments are from a simple application where synthetic arithmetic computations are performed over 50,000 independent items. In this application, a stream item is a record that flows through the pipeline triggering computations. This synthetic application allows several customizations relevant to simulate representative behaviors. For instance, a parametric version enables the creation of graph topologies with different numbers of stages, customizations in the computing weight of each parallel stage, and different data input rates. Importantly, such a flexible parameterization is intended to evaluate decision-making with scenarios representative of real-world conditions.

The default training step set in this synthetic application was one second. Moreover, the buffer sizes of the runtime system used a maximum length of 20 items, which is a value for balancing throughput and latency. Noteworthy, having buffer space available does not mean more items will be enqueued and buffered. In practice, this configuration is only relevant from the perspective of Fastflow's runtime system and is not expected to impact the decision-making strategies significantly.

[11] For more detailed insights into these communication behaviors, readers can refer to reference [35].

This is due to the fact that enqueuing and buffering of items primarily happens when the application processing speed falls below the input rate.

In addition to the comprehensive characterization provided with the synthetic application, we also considered Ferret, which is a stream-parallel benchmark that searches for similarities in data items like audio, images, and video [24]. Ferret's original version has four thread pools that run in parallel, meaning that Ferret has four parallel stages. Ferret's unstable workload was notable in reference [39], even without using parallel stages, so we increased the training step to 5 s to attempt a more reliable training step [19]. Moreover, in Ferret the buffer sizes of the runtime system were set to a maximum length of 10. This value was set in [39] as a suitable value to achieve high throughput and stability. Moreover, Ferret was executed with the PARSEC native input, which is a representative workload from the original version.

6 Experimental results

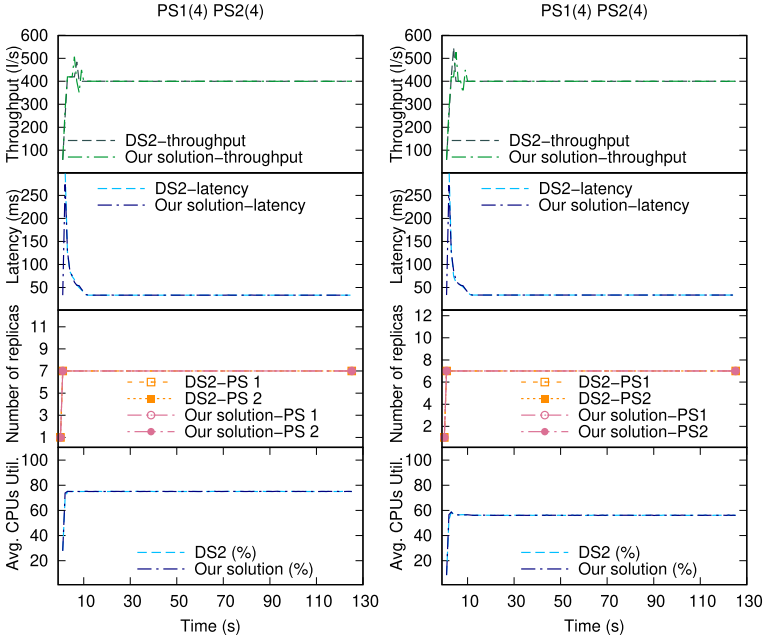
Here, we present the most relevant and insightful results. In Sect. 6.1, we present the first results from customizing the synthetic application to simulate an application's two parallel stages. There are results simulating scenarios where two parallel stages are balanced and scenarios where the applications have unbalanced stages (more representatives of real-world applications). The rationale for such scenarios is to evaluate if the strategy can detect the imbalance and optimally estimate the resources needed at each stage.

Section 6.1 also demonstrates a representative scenario where the application has four parallel stages, which is expected to be a more challenging scenario for the self-adaptive strategies due to the higher number of stages and the potential unbalance and higher resource consumption that, in theory, requires a better resources management achievable with optimal decision-making w.r.t. the configurations to be applied. Moreover, we extend the evaluation with a real-world application presented in Sect. 6.2.

To assess if the decision-making strategies work consistently in different machine architectures, we also include in Sect. 6.3 results from execution in the M2 and M3. For the sake of conciseness, we present and discuss some insightful results from scenarios replicated in M2 and M3, which are expected to be enough to extend the analysis and demonstrate the consistency of the decision-making strategies under different architectures.

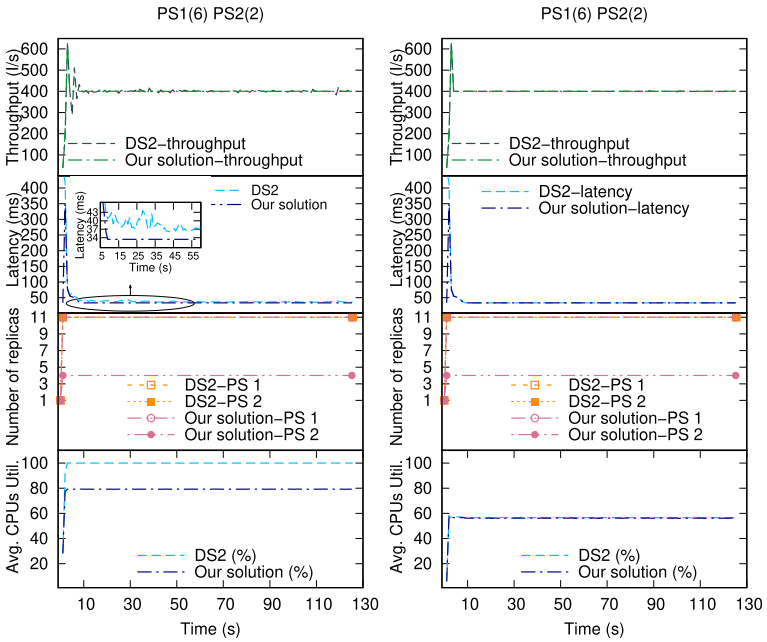
6.1 Synthetic application

The first results are from a synthetic application that is customized to simulate an application composed of two parallel stages. Figure 3 shows the first experimental results from a scenario where the different parallel stages (PS) are balanced in this



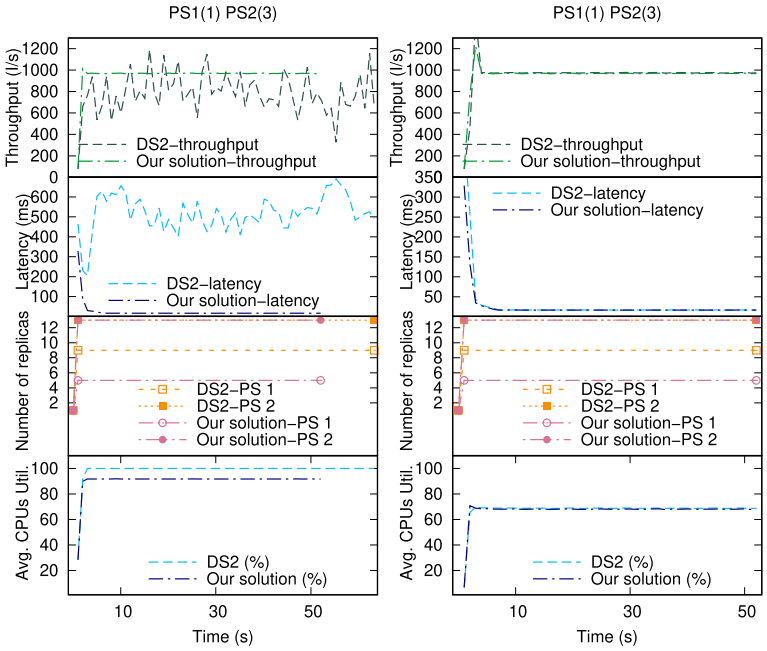
(a) Non-blocking Mode. (b) Blocking Mode.

Fig. 3 Synthetic App with input rate (IR) and Target Throughput of 400 I/s



(a) Non-blocking Mode. (b) Blocking Mode.

Fig. 4 Synthetic app with IR and target throughput of 400 I/s



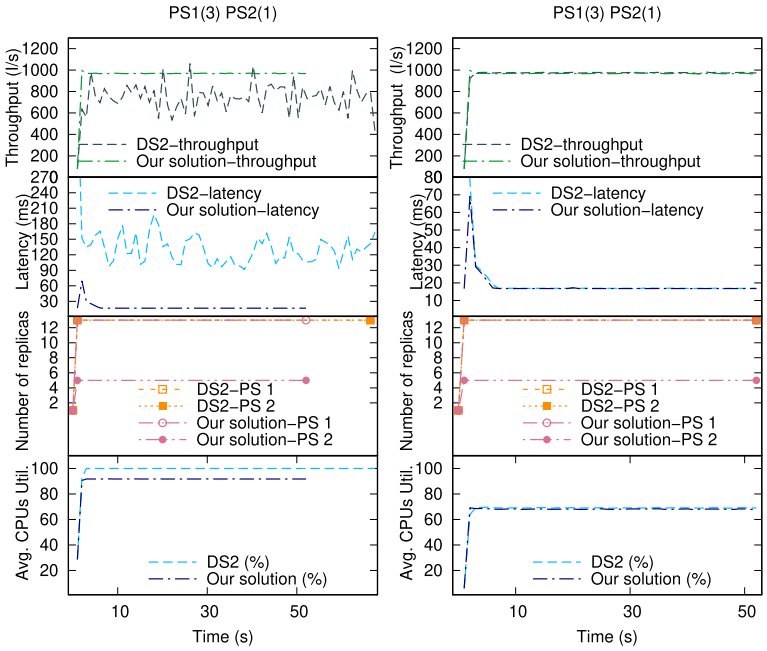
(a) Non-blocking Mode. (b) Blocking Mode.

Fig. 5 Synthetic: unbalanced stages. IR and target throughput of 1000 I/s

case with the value 4 (ms of processing time). We refer to balanced stages as a scenario where the stages have the same computational weight.

It is important to note that the input rate (IR) and the target throughput were not high for the machine used. An indicator of this is the CPUs utilization below 60% in the blocking mode (lower part of Fig. 3b). After the training step, the two strategies inferred that seven replicas were suitable for achieving QoS regarding the decision-making strategies. **Hence, both strategies achieved similar performance**. This outcome is representative of scenarios where the parallel stages are perfectly balanced, where the decision-making strategies also performed similarly under other scenarios of balanced stages. However, having perfectly balanced stages in the real world is unusual because each stage performs specific computations that cause contrasting computational weights.

Figure 4 presents a scenario with unbalanced stages where the first parallel stage has a weight of 6 ms and the second a weight of 2 ms. DS2 configured each parallel stage to execute with eleven replicas. In contrast, our solution detected that the first parallel stage was heavier and set it to run with eleven replicas. Our solution set the second (lighter) stage to execute with four replicas. **The main implication of our solution using fewer threads is in terms of resource consumption and efficiency, where it is possible to note in the lower part of Fig. 4a that our solution is more efficient**, having CPUs utilization around 20% lower.



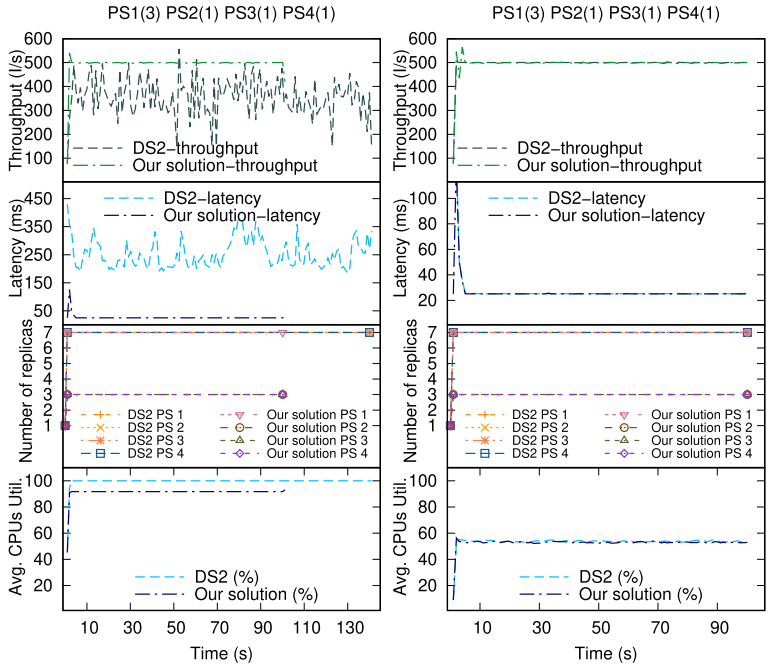
(a) Non-blocking Mode. (b) Blocking Mode.

Fig. 6 Synthetic: unbalanced stages. IR and target throughput of 1000 I/s

In some cases, **our solution achieved a throughput that is stabler than DS2's throughput. Moreover, the DS2 decision to utilize more replicas causes instability and performance losses**. For instance, when zooming the latency 55 s after the training step, our solution achieved a lower latency ranging from 7.66% to 23.27% better results. In the blocking mode, it is not possible to note the impact of DS2 utilizing more replicas due to the efficiency of FastFlow's blocking mode [35, 38].

In previous work [38], we conducted a further analysis w.r.t. the results from Fig. 4 to understand the reasons behind the DS2's lack of capacity to detect that the first stage was the bottleneck. This event is because DS2 collects the output rates of each stage. Hence, as the first stage is the bottleneck (computes and outputs fewer tasks), the subsequent stage seems to the decision-making to be a bottleneck too because it can only output as many items as it receives from the bottleneck stage. Finally, our profiling solution was better for detecting this complex scenario by making a decision based on measurements of the actual service times of each stage.

Figure 5 introduces a representative scenario where a higher throughput (1000 I/s) is required and the last stage is the bottleneck. DS2 detected that the last stage was the bottleneck, determined that thirteen replicas were suitable, and set the first stage to run with nine replicas. Our solution based on profiling estimated five replicas for the first stage. Considering that it is notable in Fig. 5b that both solutions coped with the IR in the blocking mode and achieved similar performance, our



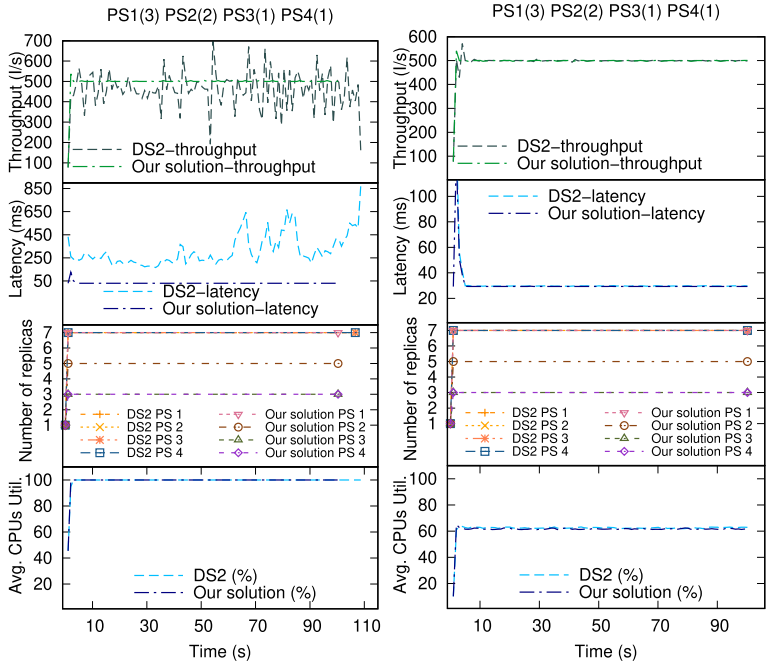
(a) Non-blocking Mode. (b) Blocking Mode.

Fig. 7 Synthetic: unbalanced stages. IR and target throughput of 500 I/s

decision-making strategy correctly inferred that five replicas were enough for the first stage. Thus, our solution was more accurate and efficient in this test scenario.

Figure 5a relates to the non-blocking mode showing a distinct performance trend compared to Fig. 5b. In this scenario, the additional unnecessary replicas added by DS2 in the first stage consumed more resources due to the nature of the non-blocking mode. First, using this extra resource reduced the efficiency by demanding 100% of the machine resources. Second, this additional consumption of resources in the first lighter stage seems to "steal" resources necessary for the actual bottleneck stage (the last one). Consequently, the **DS2's suboptimal decision-making made the application execution unstable and reduced the application throughput**. Then, a throughput lower than the IR increased the latency due to the buffering of tasks in the bottleneck stage. In conclusion, **our solution outperformed DS2 with an optimal estimation of the number of replicas for all the parallel stages**. Figure 5a shows a relevant scenario of potential performance and efficiency gains that such optimal decision-making can provide, achieving low latency and high throughput that reaches a plateau compatible with the input rate.

Figure 6 shows a scenario that shares some similarities to the one seen in Fig. 4. Still, an application scenario is being simulated here, where higher throughput is aimed, and the stages are lighter. Figure 6a shows that our solution outperforms DS2 by detecting the optimal stages' computational weight supported by the profiling step. This accurate information was again employed to estimate a suitable number of



(a) Non-blocking Mode. (b) Blocking Mode.

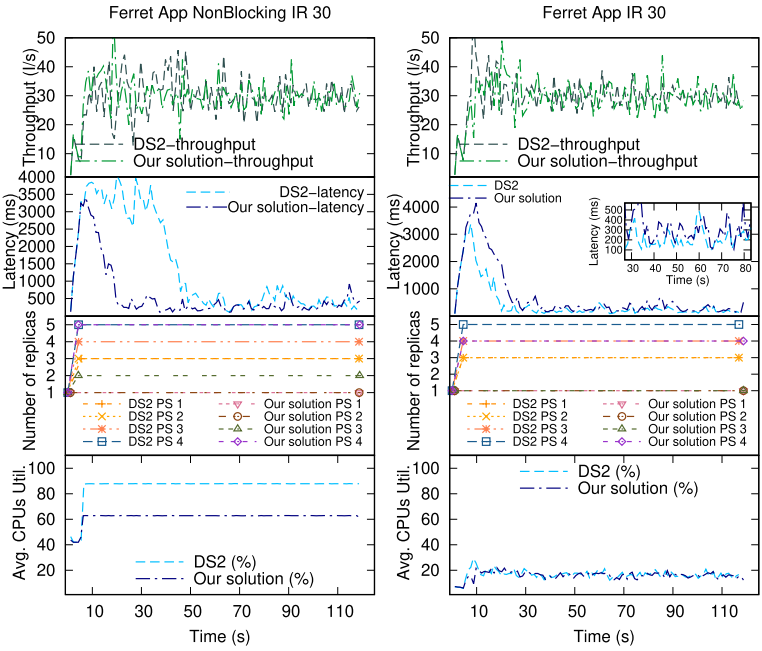
Fig. 8 Synthetic: unbalanced stages. IR and target throughput of 500 I/s

replicas that avoids resource contention by activating the actual necessary number of replicas. Moreover, the position of the bottleneck and the lighter stage is inverse in Fig. 6 compared to Fig. 5. Still, our proposed solution could optimally estimate the number of replicas for each parallel stage.

The previous results provided insightful outcomes from the synthetic application with two parallel stages. We also provide results from representative scenarios where the application has four parallel stages. Figure 7 shows the first results with an IR of 500 I/s where the first stage is the bottleneck. In general, Fig. 7 shows results aligned with the previously seen two parallel stages. DS2's decision-making does not fully detect each stage's computational weight and enforces suboptimal configurations. Hence, in the non-blocking mode, DS2's execution achieved a limited throughput and a high latency. Consequently, **the optimized decision-making of our solution results in a parallel execution that significantly outperforms DS2 in terms of performance and efficiency**.

The outcome provided in Fig. 7b is also aligned with the previous results, where the DS2's additional replicas do not compromise the performance due to the efficiency of the runtime blocking mode.

Figure 8 shows an even more complex scenario where two stages are heavier yet have different computational weights. Figure 8a shows a representative scenario of simulating an application with four parallel stages with three different computational weight levels. In this scenario, the first parallel stage is the heaviest, followed by the



(a) Non-blocking Mode. (b) Blocking Mode.

Fig. 9 Ferret with IR and target throughput 30 I/s

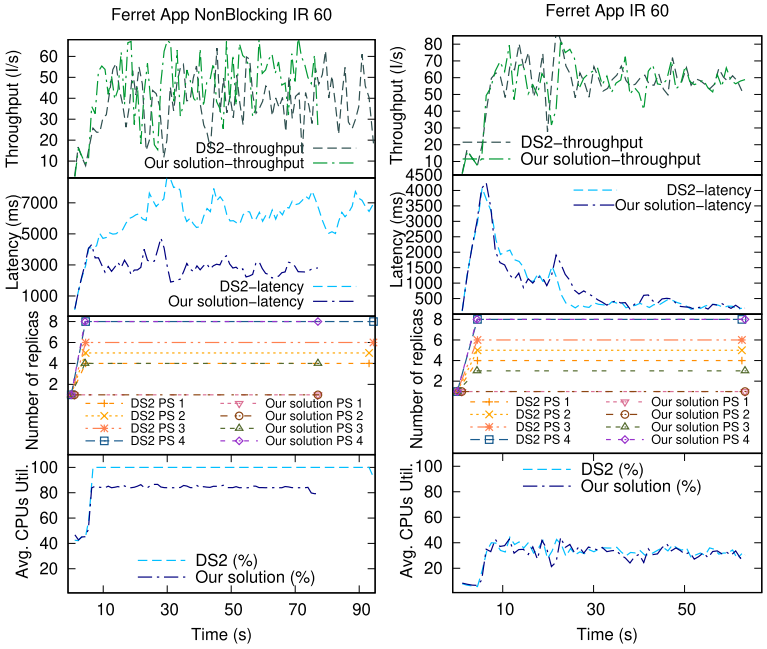
second one, which is lighter than the first stage but is still two times more intensive than the light stages (third and fourth).

Figure 8a shows how **our solution could correctly estimate the computational weight of the stages to set an appropriate number of replicas on each stage. Hence, our solution performs better than DS2** with higher throughput, lower execution time, and significantly lower latency.

6.2 Ferret application

In reference [39], Ferret was tested with IR of 10 and 20 I/s. Considering that the executions were without optimizations in the number of replicas, here we first tried Ferret with an IR of 30 I/s, as shown in Fig. 9. In the non-blocking mode shown in Fig. 9a, it is notable that both strategies enforced more replicas to the last stage Rank stage, which is the heavier one. Moreover, DS2's decision-making enforced more replicas in lighter stages compared to our solution. Considering a QoS perspective, both solutions present a fluctuating throughput due to Ferret's unstable behavior. Our solution achieved a more consistent, mostly lower latency and consumed fewer resources.

Figure 9b evinces a scenario where **both decision-making strategies achieved a similar performance and resource consumption in the Blocking mode**. Noteworthy, there are instances where DS2 has lower latency and others where our solution



(a) Non-blocking Mode. (b) Blocking Mode.

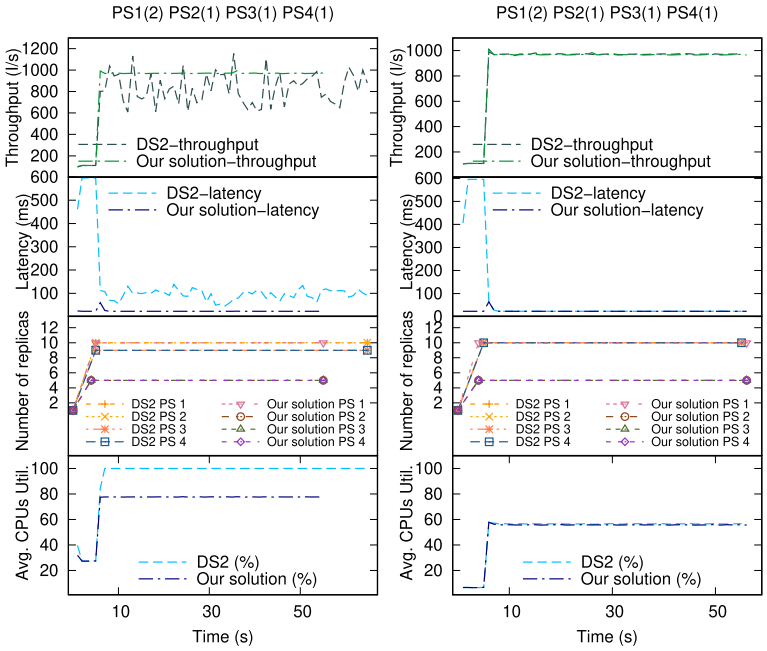
Fig. 10 Ferret with IR and target throughput 60 I/s

has a lower one. Still, it seems inconclusive where the contrasts are potentially due to specific workload peaks.

In Fig. 10, the input rate is doubled to simulate a scenario that demands a higher throughput and utilizes more resources. Considering the number of replicas enforced by the strategies, the outcome is similar to the one seen with IR 30 in Fig. 9b, where DS2 estimates the need to use more replicas in lighter stages. However, under a higher workload like IR 60, unnecessary resource consumption caused more contention, reducing DS2's execution throughput and increasing the latency. Hence, **in Ferret, it is also notable that our solution can be more accurate in determining the appropriate number of replicas for the parallel stages and can significantly outperform DS2's performance.**

6.3 Complementary results

This section presents insightful results from running the evaluation scenarios in additional machines. Here, we present only the most relevant results to assess whether decision-making strategies work consistently in different machine architectures. M2 and M3 are more recent and powerful machines compared to M1. Hence, we expect some contention in previous sections to be avoided in M2 and M3. Moreover, in M2 and M3, all the experiments utilized a training step time of five seconds.



(a) Non-blocking Mode. (b) Blocking Mode.

Fig. 11 M2—synthetic: unbalanced stages. IR and target throughput of 1000 I/s

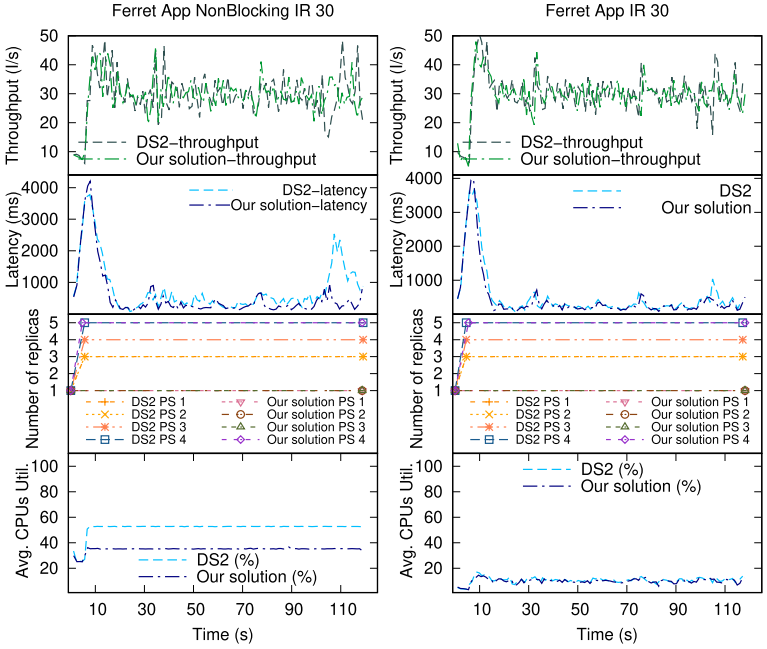
Here, we increased (from one to five seconds) in the synthetic application to test if the training step time impacted the DS2's limited decision-making accuracy.

Figure 11a evinces **more accurate decision-making of our solution compared to DS2, which resulted in our solution achieving a higher throughput** compatible with the IR. This results in lower latency and more efficient resource usage. Moreover, another relevant outcome from Fig. 11 is that **a higher training step time did not improve DS2's limited decision-making accuracy**, which indicates that the low DS2's accuracy is caused by its limited generalizability to multicore machines [38].

Considering that for the sake of conciseness, the results from Fig. 11 addressed only the more complex outcomes from M1, the rest of this section focuses on additional results from the real-world Ferret application executed in M2.

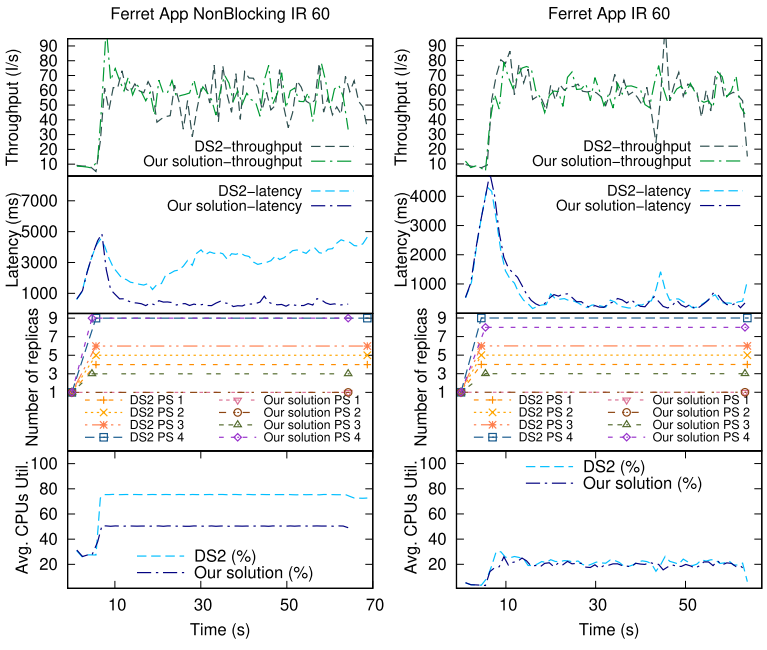
The results provided in Fig. 12 extend the evaluation from Fig. 9 with experiments from M2. In general, the results from both machines are very consistent. In both cases, our solution optimally detected the stages' characteristics and enforced an appropriate number of replicas on each one. Consequently, in Fig. 12a it is notable that our solution consumed significantly fewer resources and achieved stabler throughput and lower latency, where the rationale is still that our approach is more accurate and avoids resource contention.

Figure 13 highlights Ferret with IR 60 items/s executed in M2, which complements the same experiment shown in Fig. 10 executed in M1. Here, the optimal



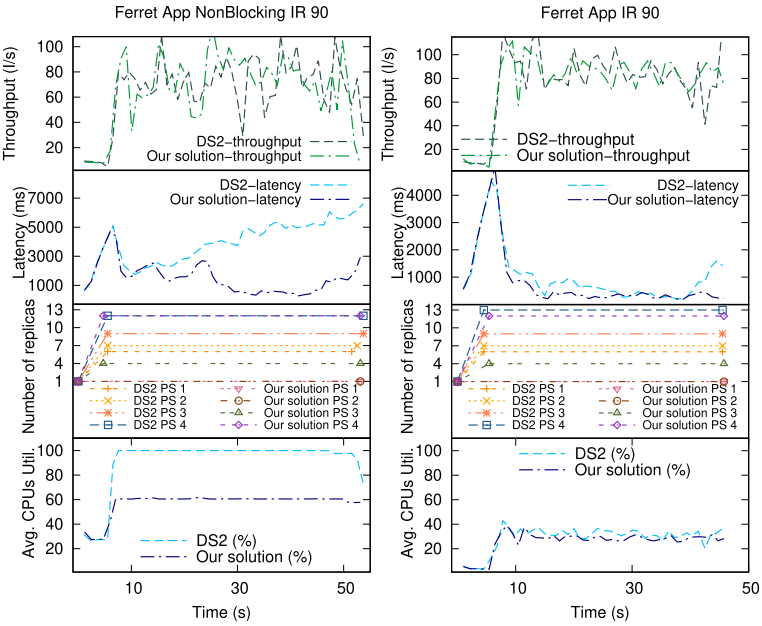
(a) Non-blocking Mode. (b) Blocking Mode.

Fig. 12 M2—Ferret with IR and target throughput 30 I/s



(a) Non-blocking Mode. (b) Blocking Mode.

Fig. 13 M2—Ferret with IR and target throughput 60 I/s



(a) Non-blocking Mode. (b) Blocking Mode.

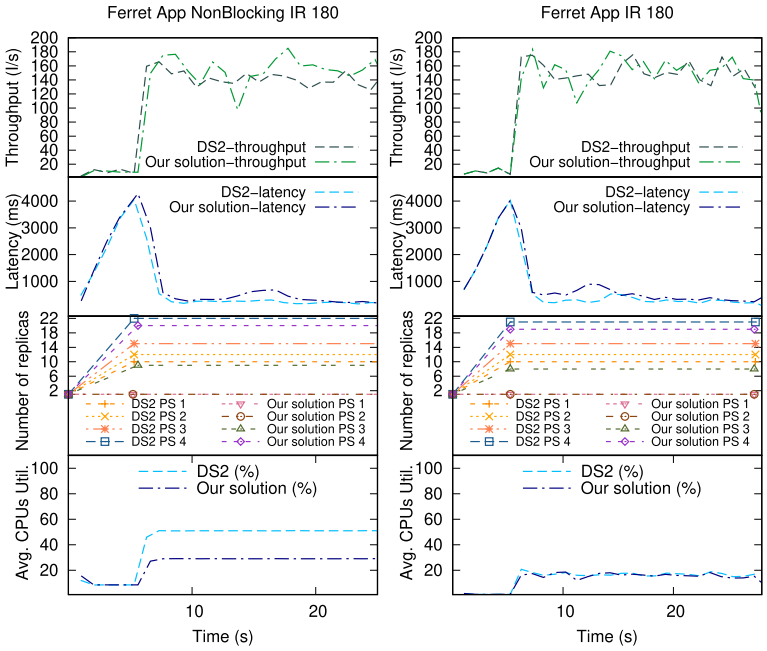
Fig. 14 M2—Ferret with IR and target throughput 90 I/s

decision-making of our solution again outperformed DS2 in terms of resource efficiency and QoS. Contrasting with the outcome shown in Fig. 10a, in M2, the non-blocking mode shown in Fig. 13a did not consume 100% of the machine's resources in DS2. However, the higher (and unnecessary) resource consumption due to DS2's suboptimal decision-making still caused contention and performance degradation, e.g., the high latency notable in Fig. 13a.

Figure 13b shows the blocking mode evaluation that is notable for similar resource consumption. Our solution's throughput and latency are stabler than DS2 due to the use of fewer replicas. Moreover, the latency achieved with our solution is slightly lower than DS2's latency.

Considering that in Fig. 13 the IR of 60 I/s running in M2 did not demand all the machine's processing capacity, in Fig. 14 we provide an additional result scenario with a higher IR of 90 I/s. It is important to note that this experiment can be executed with stability in M2 due to its additional processing capacity compared to M1. With empirical characterization tests, M1 did not cope with the high IR of 90, which compromised the stability of the decision-making strategies.

Figure 14 provides the results from the executions with IR 90 I/s, which compared to Fig. 13 (IR 60 I/s) we can note a similar performance and resources efficiency trend. Such an outcome again indicates the consistency of the decision-making of our strategy outperforming DS2. Importantly, although the final throughputs are similar, our solution has notable latency gains both in the Blocking and non-blocking modes.



(a) Non-blocking Mode. (b) Blocking Mode.

Fig. 15 M3—Ferret with IR and target throughput 180 I/s

Moreover, with the IR 90 (I/s), the DS2 execution in the non-blocking mode (Fig. 14a) fully utilized the machine's resources, which further increased the latency. **In summary, the additional results from Ferret running in M2 further confirmed that our solution can significantly outperform DS2 in multicore machines.**

Figure 15 shows an additional scenario of Ferret executed in Machine 3, which is a machine with more processing resources. Hence, in M3 the highest IR of 180 items/second was used in order to benchmark the decision-making strategies.

Considering the results evinced in Fig. 15 from a perspective of QoS, **it is notable that in M3 both solutions achieved similar performance trends in terms of throughput and latency**. On the one hand, the application's execution time was significantly reduced in a faster machine. On the one hand, significant fluctuations can be seen due to the combination of Ferret's unstable nature and the IR for high throughput.

Moreover, it is notable that **our solution's decision-making is more accurate by inferring that fewer replicas were necessary and still achieved performance comparable to DS2**. Such an aspect corroborates that our solution outperforms DS2 from an accuracy perspective. In Fig. 15a, the additional replicas that DS2 enforced caused a significantly higher consumption of resources. For instance, after the training step in the non-blocking execution, our solution utilized, on average, 29% of CPUs. DS2's execution consumed around 50%. Beyond the resource consumption, this means that DS2 used 64 CPUs while our solution used 37 CPUs. The high

difference in the CPUs resource utilization shows that our solution provided more efficient executions, consumed less energy, was more sustainable, and the executions then could cost less.

6.4 Evaluation summary

The following are the key insights from the extensive experimental evaluation:

- The conceptual decision-making framework (Sect. 3) was effectively applied here. This facilitated generalization and flexibility, enabling the easy use of different decision-making strategies by only changing the implementation of monitoring modules and decisions (e.g., in practice, simply changing C++ header files).
- The decision-making strategies were able to quickly (in order of milliseconds) determine the configurations to be applied. However, in this work, we are more interested in evaluating how accurate are the configurations found in the decision-making step.
- The usage of an optimal number of replicas on multicore machines is still essential for efficiency and QoS.
- Our solution is stabler and more efficient compared to DS2. Our solution's decision-making outperforms DS2 by better estimating the optimal number of replicas for each parallel stage. Moreover, our solution considers the resources available, reducing contention and improving performance in several scenarios.
- Although the impact of the decision-making in the QoS achieved in different scenarios is not trivial to predict, the experimental results demonstrated that the decision-making strategies are consistent across different applications. The new decision-making strategy proposed here is equally or more accurate and efficient than the existing approach. Such an outcome comes from a comprehensive evaluation that covers different numbers of parallel stages, the number of bottleneck stages and their location, stages' intensiveness, and running in different architectures of shared memory multicore machines.

7 Related work

This section overviews the relevant related works separated into frameworks for self-adaptation and the proposed technical mechanisms or strategies.

7.1 Frameworks for self-adaptation

Providing autonomous solutions in the form of frameworks is already present in some scenarios [26]. Noteworthy, *NORNIR* [8] was proposed as a framework for simplifying the management of energy consumption in parallel applications. Additionally, *E2DF* [29] was proposed to adapt the infrastructure resources availability and optimize the deployment of applications' stages. From an interesting decoupling

perspective, [27] proposed an analytical framework for deciding the size of batches in parallel processing, where the framework is decoupled from the runtime system utilized. Moreover, in reference [28], a theoretical model of a framework was proposed. The framework provides optimizations and mapping algorithms (only at compile time) targeting heterogeneous architectures based on a static performance model. From the functional perspective of autonomous and self-adaptive systems, [5] proposed an evaluation framework for self-adaptation, mainly focusing on learning to handle unexpected conditions and evaluate if changes provide improvements. *AgileCtrl* [40] was also proposed for adaptation mechanisms in software systems.

However, the frameworks mentioned above are applied to specific contexts, which results in low flexibility and generalizability. For instance, it is unclear what and how those frameworks' given components could be applied to self-adapt other entities or design new decision-making strategies. In our understanding, a potential approach is to focus primarily on a modular design of the decision-making strategies. Then, optimal decision-making could be applied to specific contexts that provide the means (mechanisms) to apply adaptation actions.

7.2 Mechanisms and decision-making strategies for self-adaptation

Many entities can be adapted at run-time to provide self-adaptive optimizations and/or abstractions, spanning from hardware and operating system level optimizations [3, 4, 8, 20] to higher-level of optimization applied at application's level [7, 9, 14, 30, 33, 37]. While different viewpoints may diverge and can be integrated to tackle analogous challenges, our methodology is underpinned by high-level parallel computing principles. We focus on optimizations at the application level applying self-adaptation parallel streaming applications. From a practical perspective, this scenario alone is already very challenging because the complex structured applications usually are composed of deep pipelines, creating data items' dependency and ordering constraints. This requires coordination between the parallel stages to avoid a scenario where one stage becomes a bottleneck slowing down the performance of all the subsequent ones [38].

Our vision is that programmers should be empowered by efficient execution abstractions, i.e., released from the need to define the number of replicas in complex-structured applications. Several studies have assessed how to determine best the optimal number of replicas in parallel applications [7, 9, 14, 30, 33, 37]. However, the aforementioned approaches support self-adaptation in a single parallel stage. Therefore, the adaptation space is limited for complex-structured parallel applications composed of two or more parallel stages.

Other approaches support self-adaptation in many parallel stages, such as [9, 11, 19, 33]. Such solutions tackled different execution environments, including multicores and distributed systems. A prominent solution is DS2 [19]. In previous work [38], we evaluated DS2's feasibility for efficient decision-making, where it became notable that DS2's decision-making is limited in accuracy when applied to multicores. In this work,

we proposed and extensively evaluated a new decision-making strategy that focuses on accurate decision-making for parallelism configuration, which can provide better performance, efficiency, and QoS to the applications.

8 Conclusions

This paper presented an extended approach for supporting self-adaptation in complex structured parallel applications, focusing on providing parallelism abstractions. The new abstractions are provided by self-adaptive strategies autonomously managing and optimizing at run-time the parallel applications.

8.1 Implications

The proposed decision-making strategy for self-adaptive number of replicas is achieving encouraging results in terms of accuracy, performance, and efficiency. This demonstrates that our approach is ready to be integrated with larger adaptation spaces such as the entire applications' composition structures [35, 39]. Adapting the entire structure is a more powerful but intrusive adaptation that requires creating several configurations at compile time. Notably, the solution proposed here is complementary and can self-adapt only the number of replicas in the case of parallel stages, empowered with more flexible mechanisms that do not require changes at compilation time. In short, this paves the way to potential additional flexibility by combining the self-adaptation of different entities.

In practice, we expect that the approach proposed here is also applicable to stateful applications. The management of stateful applications' state executing in shared-memory multicore machines is easy to manage. However, the applicability of the decision-making strategies in stateful applications executed in distributed environments would require further steps for managing (saving and migrating) the applications' state during entities' adaptation. One concrete example of complexities with stateful applications is downtimes that can occur when the state is lost and has to be restored to apply adaptation actions [23]. There are techniques for such a challenge, such as saving the state and migrating before applying actions, concurrent recompilation, and input duplication [31]. Moreover, one could implement such techniques within the planning step of the conceptual framework presented in Sect. 3.

Beyond the technicalities involved, one can extract relevant scientific and technical implications. A significant implication is that in concrete implementations and use-cases, self-adaptation has been demonstrated to be effective in making the applications' executions more abstract and yet achieve high performance. Moreover, our approach was designed to be ready-to-use in the sense of working without the need to install external systems or libraries. We expect that this capacity will facilitate new use cases and modular integrations with other systems.

8.2 Limitations and future work

In this study, we provide a self-adaptation approach that was validated with a real-world application (Ferret) and a synthetic application, allowing multiple representative scenarios to be created. Although the applications are representative, slightly different results may be achieved under other application characteristics. However, in the experiments of our previous works, we found similar behaviors with the inclusion of other applications such as video processing and compression [37, 39].

It is important to note that one of the goals of this work is to improve the design of the self-adaptive approaches to make them more generalizable. The practical solution covered in this work is only evaluated in the context of multicore machines. Theoretically, the decision-making strategy could be applied to distributed executions. However, with the current state of our research, we do not cover distributed scenarios nor many-core platforms [4]. The necessary implementations and validations are left for the future.

From a technical perspective, we expect it would not be easy to implement our self-adaptive solution in runtime libraries based on task processing. However, we understand that one can apply this work's solutions to other software systems with the popular threading model based on nodes, for instance, to the majority of distributed stream processing frameworks where each node or stage is translated into a thread or process. In this scenario, expert programmers could implement mechanisms that could be managed by self-adaptive strategies using the generic decision-making modules.

The demand for mechanisms to apply adaptation actions in software systems is a known limitation in the field. Hence, considering that generalization is a relevant aspect of this work, the availability of mechanisms can be viewed as a partial limitation to our self-adaptive solutions' broad applicability. Considering that computing applications and software systems are becoming (or need to become) more dynamic and modular, we believe that providing mechanisms for enabling self-adaptation in a given software system is a step in the right direction. Consequently, this work's framework and optimized decision-making strategy can help inspire future solutions.

In this work, we focus mostly on how accurate the self-adaptive strategies make the decisions at the start of the applications' executions. In the future, we intend to evaluate whether the results found in the decision-making strategies also occur in more dynamic workloads requiring several reconfigurations during execution. Moreover, considering that we focus on self-adaptation at the application level, a future step is to open-source the code artifacts of the decision-making strategies.

Author contributions All authors contributed to the study conception and design. Adriano Vogel wrote the main manuscript text. Adriano Vogel implemented the approach, executed the tests, and provided the visualizations. Massimo Torquati assisted in the implemented mechanism integration into FastFlow. Adriano Vogel, Marco Danelutto, and Dalvan Griebler discussed the performance analysis. Luiz Fernandes ensures that questions about the accuracy or integrity of any part of the work are appropriately investigated and resolved.

Funding Open access funding provided by Johannes Kepler University Linz.

Declarations

Conflict of interest The authors declare no competing interests.

References

1. Aldinucci M, Danelutto M, Kilpatrick P et al (2017) Fastflow: high-level and efficient streaming on multicore, vol 13. Wiley, Chap, pp 261–280
2. Andrade H, Gedik B, Turaga D (2014) Fundamentals of stream processing: application design, systems, and analytics. Cambridge University Press
3. Barati S, Bartha FA, Biswas S et al (2019) Proteus: language and runtime support for self-adaptive software development. IEEE Softw 36(2):73–82
4. Bellasi P, Massari G, Fornaciari W (2012) A RTRM proposal for multi/many-core platforms and reconfigurable applications. In: International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip. IEEE, pp 1–8
5. Bombarda A, Bonfanti S, De Sanctis M, et al (2022) Towards an evaluation framework for autonomous systems. In: Proceedings of IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion. IEEE, pp 43–48
6. Dagum L, Menon R (1998) OpenMP: an industry standard API for shared-memory programming. IEEE Comput Sci Eng 5(1):46–55
7. De Matteis T, Mencagli G (2016) Keep calm and react with foresight: strategies for low-latency and energy-efficient elastic data stream processing. ACM SIGPLAN Not 51(8):13:1-13:12
8. De Sensi D, De Matteis T, Danelutto M (2018) Simplifying self-adaptive and power-aware computing with nornir. FGCS 87:136–151
9. Floratou A, Agrawal A, Graham B et al (2017) Dhalion: self-regulating stream processing in heron. VLDB 10:1825–1836
10. Fragkoulis M, Carbone P, Kalavri V, et al (2023) A survey on the evolution of stream processing systems. VLDB J:1–35
11. Gedik B, Schneider S, Hirzel M et al (2014) Elastic scaling for data stream processing. IEEE TPDS 25(6):1447–1463
12. Gheibi O, Weyns D, Quin F (2021) Applying machine learning in self-adaptive systems: a systematic literature review. ACM Trans Auton Adapt Syst 15(3):1–37
13. Hellerstein J, Diao Y, Parekh S et al (2004) Feedback control of computing systems. Wiley, IEEE
14. Hidalgo N, Rosas E, Vasquez C et al (2018) Measuring stream processing systems adaptability under dynamic workloads. FGCS 88:413–423
15. Hirzel M, Soulé R, Schneider S et al (2014) A catalog of stream processing optimizations. ACM CSUR 46(4):46
16. Huss-Lederman S, Gropp B, Skjellum A, et al (1997) MPI-2: extensions to the message passing interface. University of Tennessee, available online at http://www.mpiforum.org/docs/docs.html
17. Iftikhar MU, Weyns D (2014) ActivFORMS: Active formal models for self-adaptation. In: International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp 125–134
18. Intel (2024) oneAPI Threading Building Blocks . https://github.com/oneapi-src/oneTBB
19. Kalavri V, Liagouris J, Hoffmann M, et al (2018) Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In: USENIX OSDI, pp 783–798

20. Kanduri A, Miele A, Rahmani AM, et al (2018) Approximation-aware coordinated power/performance management for heterogeneous multi-cores. In: Annual Design Automation Conference, pp 1–6
21. Kehrer S, Blochinger W (2019) Elastic parallel systems for high performance cloud computing: state-of-the-art and future directions. Parallel Process Lett 29(02):1–20
22. Kephart J, Chess D (2003) The vision of autonomic computing. Computer 36(1):41–50. https://doi.org/10.1109/MC.2003.1160055
23. Li J, Pu C, Chen Y, et al (2016) Enabling elastic stream processing in shared clusters. In: Proceedings of the IEEE International Conference on Cloud Computing. IEEE, pp 108–115
24. Lv Q, Josephson W, Wang Z, et al (2006) Ferret: a toolkit for content-based similarity search of feature-rich data. In: ACM SIGOPS/EuroSys, pp 317–330
25. McCool M, Reinders J, Robison A (2012) Structured parallel programming: patterns for efficient computation. Elsevier Science
26. Metzger A, Quinton C, Mann ZÁ, et al (2022) Realizing self-adaptive systems via online reinforcement learning and feature-model-guided exploration. Computing:1–22
27. Metzger P, Cole M, Fensch C, et al (2020) Enforcing deadlines for skeleton-based parallel programming. In: IEEE Real-Time and Embedded Technology and Applications, pp 188–199
28. Miller J, Trümper L, Terboven C et al (2021) A theoretical model for global optimization of parallel algorithms. Mathematics 9(14):1685
29. Nardelli M, Russo GR, Cardellini V, et al (2018) A multi-level elasticity framework for distributed data stream processing. In: Proceedings of the European Conference on Parallel Processing. Springer, pp 53–64
30. Omoregbee P, Forshaw M (2023) Performability requirements in making a rescaling decision for streaming applications. In: Gilly K, Thomas N (eds) EPEW. Springer, pp 133–147
31. Rajadurai S, Bosboom J, Wong WF et al (2018) Gloss: seamless live reconfiguration and reoptimization of stream programs. ACM Not 53(2):98–112
32. Rashid TA, Hassan BA, Alsadoon A, et al (2023) Awareness requirement and performance management for adaptive systems: a survey. J Supercomput:1–23
33. Siachamis G, Kanis J, Koper W, et al (2023) Towards evaluating stream processing autoscalers. In: International Conference on Data Engineering Workshops. IEEE, pp 95–99
34. Thies W, Karczmarek M, Amarasinghe S (2002) StreamIt: a language for streaming applications. In: Proceedings of the International Conference on Compiler Construction, pp 179–196
35. Torquati M, Sensi DD, Mencagli G et al (2019) Power-aware pipelining with automatic concurrency control. CCPE 31(5):e4652
36. Vogel A, Griebler D, Danelutto M et al (2020) Minimizing self-adaptation overhead in parallel stream processing for multi-cores. LNCS 11997:30–41
37. Vogel A, Griebler D, Fernandes LG (2021) Providing high-level self-adaptive abstractions for stream parallelism on multicores. SPE 51(6):1194–1217
38. Vogel A, Danelutto M, Griebler D, et al (2023a) Revisiting self-adaptation for efficient decision-making at run-time in parallel executions. In: PDP. IEEE, pp 43–50
39. Vogel A, Mencagli G, Griebler D et al (2023) Online and transparent self-adaptation of stream parallel patterns. Computing 105(5):1039–1057
40. Wang S, Hoffmann H, Lu S (2022) AgileCtrl: a self-adaptive framework for configuration tuning. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp 459–471
41. Weyns D (2020) An introduction to self-adaptive systems: a contemporary software engineering perspective. Wiley
42. Weyns D, Usman Iftikhar M, De La Iglesia DG, et al (2012) A survey of formal methods in self-adaptive systems. In: Proceedings of International Conference on Computer Science and Software Engineering. ACM, pp 67–79
43. Yuan E, Esfahani N, Malek S (2014) A systematic survey of self-protecting software systems. ACM Trans Auton Adapt Syst 8(4):1–41