ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

ADRIANO VOGEL

SELF-ADAPTIVE ABSTRACTIONS FOR EFFICIENT HIGH-LEVEL PARALLEL COMPUTING IN MULTI-CORES

Porto Alegre
2022

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

SELF-ADAPTIVE ABSTRACTIONS FOR EFFICIENT HIGH-LEVEL PARALLEL COMPUTING IN MULTI-CORES

ADRIANO VOGEL

Doctoral Dissertation submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Ph.D. in Computer Science. This Doctoral Dissertation is in joint-supervision with the University of Pisa - Italy.

Advisor: Prof. Ph.D. Luiz Gustavo Fernandes
Advisor: Prof. Ph.D. Marco Danelutto (UNIPI)
Co-Advisor: Prof. Ph.D. Dalvan Griebler

Porto Alegre
2022

Ficha Catalográfica

V879s Vogel, Adriano José

Self-adaptive abstractions for efficient high-level parallel computing in multi-cores / Adriano José Vogel. – Porto Alegre, 2022.

176 f.

Tese (Doutorado) – Programa de Pós-Graduação em Ciência da Computação, Escola Politécnica, PUCRS.

Orientador: Prof. Luiz Gustavo Fernandes
Orientador: Prof. Marco Danelutto
Coorientador: Prof. Dalvan Griebler

1. Stream processing. 2. Self-adaptive software. 3. Parallelism abstractions. 4. Parallel programming. 5. Autonomic systems. I. Fernandes, Luiz Gustavo. II. Danelutto, Marco. III. Griebler, Dalvan. IV. Título.

Bibliotecária responsável: Loiva Duarte Novak – CRB10/2079

ADRIANO VOGEL

SELF-ADAPTIVE ABSTRACTIONS FOR EFFICIENT HIGH-LEVEL PARALLEL COMPUTING IN MULTI-CORES

This Doctoral Dissertation has been submitted in partial fulfillment of the requirements for the degree of Ph.D. in Computer Science, of the Computer Science Graduate Program, School of Technology of the Pontifical Catholic University of Rio Grande do Sul.

Sanctioned on March 21st, 2022.

COMMITTEE MEMBERS:

“And it’s whispered that soon, if we all call the tune
Then the piper will lead us to reason
And a new day will dawn for those who stand long
And the forests will echo with laughter”
(Led Zeppelin)

ACKNOWLEDGEMENTS

I would like to to express my gratitude to everyone who helped me during this doctorate in one way or another. A special thanks to my advisors, who supported and guided me throughout this project.

I would like to thank to my peers and PUCRS/GMAP research group colleagues for their assistance. I am also grateful to UNIPI’s Parallel Computing Group for the fruitful discussions. I would like to extend my sincere thanks to the evaluation committees members for the constructive advices.

I would also like to extend my deepest gratitude to my family and friends, your encouragement was essential.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nivel Superior - Brasil (CAPES) - Finance Code 001 and by SAP Labs Latin America.

Acknowledgements to the High-Performance Computing Laboratory of the Pontifical Catholic University of Rio Grande do Sul (LAD-IDEIA/PUCRS, Brazil) for providing computing resources.

RESUMO

Atualmente, uma parte significativa dos sistemas computacionais e aplicações do mundo real demandam paralelismo para acelerar suas execuções. Embora a programação paralela estruturada e de alto nível tenha como objetivo facilitar a exploração do paralelismo, ainda há questões a serem abordadas para melhorar as abstrações existentes na programação paralela, onde os desenvolvedores de aplicações usualmente precisam definir configurações de paralelismo não intuitivas ou complexas. Nesse contexto, a autoadaptação é uma alternativa potencial para fornecer um nível mais alto de abstrações autonômicas e capacidade de resposta em tempo de execução em aplicações paralelas. No entanto, um problema recorrente é que a autoadaptação ainda é limitada em termos de flexibilidade, eficiência e abstrações. Por exemplo, faltam mecanismos para aplicar ações de adaptação e estratégias eficientes de decisão sobre quais configurações devem ser aplicadas em tempo de execução. Este trabalho é focado em abstrações alcançáveis com autoadaptação gerenciando de forma transparente as execuções enquanto os programas paralelos estão sendo executados. Os principais objetivos são: aumentar o espaço de adaptação para ser mais representativo para aplicações e tornar a autoadaptação mais eficiente com metodologias de avaliação abrangentes, que podem fornecer casos de uso que demonstrem os verdadeiros potenciais da autoadaptação. Portanto, esta tese de doutorado traz as seguintes contribuições científicas: I) Uma revisão sistemática da literatura fornecendo uma taxonomia do estado da arte. II) Um framework conceitual para apoiar a concepção e abstração do processo de tomada de decisão dentro de soluções autoadaptativas, o que é utilizado nas contribuições técnicas para ajudar a tornar as soluções mais modulares e potencialmente generalizáveis. III) Mecanismos e estratégias para réplicas autoadaptáveis em aplicações com estágios paralelos simples e múltiplos, suportando múltiplos requisitos não-funcionais. IV) Mecanismo, estratégia e otimizações para autoadaptação dos Padrões Paralelos/topologias de grafos de aplicações. Aplicamos as soluções propostas ao contexto de aplicações de processamento de streams, um paradigma representativo presente em várias aplicações do mundo real que computam dados em tempo real (por exemplo, feeds de vídeo, imagem e análise de dados). Uma parte das soluções propostas é avaliada com a SPar e outra parte com o framework de programação FastFlow. Os resultados demonstram que a autoadaptação pode fornecer abstrações de paralelismo eficientes e responsividade autonômica em tempo de execução e alcançando um desempenho competitivo em comparação com as melhores execuções estáticas. Além disso, quando apropriado, a solução proposta é comparada com soluções relacionadas, demonstrando que as estratégias de decisão propostas neste trabalho são altamente otimizadas e alcançam ganhos significativos de desempenho e eficiência.

Palavras-Chave: Processamento de *streams*, Software autoadaptativo, Abstrações de paralelismo, Programação paralela, Sistemas autônomos.

SOMMARIO

Una parte significativa dei sistemi informatici e delle applicazioni del mondo reale richiede l'utilizzo di parallelismo. Sebbene la programmazione parallela strutturata miri a facilitare lo sfruttamento del parallelismo, ci sono ancora diversi problemi da affrontare per migliorare le astrazioni della programmazione parallela messe a disposizione dei programmatori. Gli sviluppatori di applicazioni al momento devono ancora utilizzare tecniche di esplicitazione del parallelismo che risultano poco intuitive e in generale piuttosto complesse. In questo contesto, l'auto-adattamento costituisce una potenziale alternativa che può essere sfruttata per fornire un miglior livello di astrazioni autonome e una migliore reattività dei runtime nelle esecuzioni parallele. Tuttavia, un problema ricorrente è che l'autoadattamento è ancora limitato in termini di flessibilità, efficienza e astrazioni. Ad esempio, mancano meccanismi per applicare azioni di adattamento e strategie decisionali efficienti per decidere quali configurazioni vadano applicate in fase di esecuzione. In questo lavoro prendiamo in considerazione le astrazioni realizzabili con l'autoadattamento che siano in gradi di gestire in modo trasparente le esecuzioni durante l'esecuzione stessa dei programmi. I nostri obiettivi principali sono quelli di espandere lo spazio delle soluzioni di adattamento per essere in grado di trattare applicazioni del mondo reale e renderne più efficiente l'autoadattamento con metodologie di valutazione complete, che possano fornire casi d'uso che dimostrano le reali potenzialità dell'autoadattamento. Questa tesi di dottorato fornisce i seguenti contributi scientifici: i) una SLR che fornisce una tassonomia dello stato dell'arte. ii) un framework concettuale per supportare la progettazione e l'astrazione del processo decisionale all'interno di soluzioni autoadattative; tale quadro concettuale viene quindi impiegato nei contributi tecnici per aiutare a rendere le soluzioni più modulari e potenzialmente generalizzabili. iii) meccanismi e strategie per repliche autoadattative in applicazioni con stadi paralleli singoli e multipli, in grado di fornire supporto per molteplici requisiti non funzionali. iv) meccanismi, strategia e ottimizzazioni per l'auto-adattamento delle topologie dei grafi di Parallel Patterns utiizzati nelle applicazioni. Applichiamo le soluzioni proposte al contesto delle applicazioni di elaborazione stream parallel, un paradigma rappresentativo presente in diverse applicazioni del mondo reale che calcolano il flusso di dati sotto forma di flussi (ad esempio feed video, immagini e analisi dei dati). Parte delle soluzioni proposte viene valutata utilizzando SPar e parte utilizzando il framework di programmazione FastFlow. I risultati dimostrano che l'auto-adattamento può fornire astrazioni di parallelismo efficienti e una reattività autonomica a tempo di esecuzione, ottenendo prestazioni competitive rispetto a quelle ottenute dalle migliori esecuzioni "statiche". Inoltre, quando appropriato, confrontiamo soluzioni all'avanguardia e dimostriamo che le nostre strategie decisionali altamente ottimizzate ottengono guadagni significativi in termini di prestazioni ed efficienza. **Parole Chiave**: Stream parallel, Software adattativo, Astrazioni di parallelismo, Programmazione parallela, Sistemi autonomi.

ABSTRACT

Nowadays, a significant part of computing systems and real-world applications demand parallelism to accelerate their executions. Although high-level and structured parallel programming aims to facilitate parallelism exploitation, there are still issues to be addressed to improve existing parallel programming abstractions. Usually, application developers still have to set non-intuitive or complex parallelism configurations. In this context, self-adaptation is a potential alternative to provide a higher-level of autonomic abstractions and runtime responsiveness in parallel executions. However, a recurrent problem is that self-adaptation is still limited in terms of flexibility, efficiency, and abstractions. For instance, there is a lack of mechanisms to apply adaptation actions and efficient decision-making strategies to decide which configurations to be enforced at run-time. In this work, we are interested in abstractions achievable with self-adaptation transparently managing the executions while the parallel programs are running (at run-time). Our main goals are to increase the adaptation space to be more representative of real-world applications and make self-adaptation more efficient with comprehensive evaluation methodologies, which can provide use-cases demonstrating the true potentials of self-adaptation. Therefore, this doctoral dissertation provides the following scientific contributions: I) An Systematic Literature Review (SLR) providing a taxonomy of the state-of-the-art. II) A conceptual framework to support designing and abstracting the decision-making process within self-adaptive solutions, such a conceptual framework is then employed in the technical contributions to assist in making the solutions more modular and potentially generalizable. III) Mechanisms and strategies for self-adaptive replicas in applications with single and multiple parallel stages, supporting multiple customizable non-functional requirements. IV) Mechanism, strategy, and optimizations for self-adaptation of Parallel Patterns/applications' graphs topologies. We apply the proposed solutions to the context of stream processing applications, a representative paradigm present in several real-world applications that compute data flowing in the form of streams (e.g., video feeds, image, and data analytics). A part of the proposed solutions is evaluated with SPar and another part with the FastFlow programming framework. The results demonstrate that self-adaptation can provide efficient parallelism abstractions and autonomous responsiveness at run-time, yet achieve a competitive performance w.r.t. the best static executions. Moreover, when appropriate, we compare state-of-the-art solutions and demonstrate that our highly optimized decision-making strategies achieve significant performance and efficiency gains.

Keywords: Stream processing, Self-adaptive software, Parallelism abstractions, Parallel programming, Autonomic systems.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

AKA	Also Known As
API	Application Programming Interface
AST	Abstract Syntax Tree
CEP	Complex Event Processing
CINCLE	Compiler Infrastructure for New C/C++ Language Extensions
CPUs	Central Processing Units
DAG	directed acyclic graph
DDR3	Double Data Rate three
DSL	Domain-Specific Language
DVFS	Dynamic Voltage and Frequency Scaling
EE	Experimental environment
ES	Experimental scenario
FPGA	Field-Programmable Gate Array
FORMS	FORmal Models for Self-adaptation
FPS	Frames Per Second
GB	Gigabyte
GHz	Gigahertz
GPU	Graphics Processing Unit
HPC	High Performance Computing
IoT	Internet of Things
IR	Input Rate
IT	Information Technology
I/O	Input/Output
I/s	Items per second

JVM	Java Virtual Machine
MAPE-K	Monitor-Analyze-Plan-Execute-Knowledge
MB	Megabyte
MBPS	Megabytes Per Second
MDP	Markov Decision Process
MHz	Megahertz
MPC	Model Predictive Control
MPI	Message Passing Interface
OS	Operating System
PARSEC	Princeton Application Repository for Shared-Memory Computers
PS	Parallel Stage
QoS	Quality of Service
RL	Reinforcement Learning
RQ	Research Questions
SASO	Stability, Accuracy, Settling time and Overshoot
SDF	Synchronous Data Flow
SF	Scaling Factor
SLO	Service-Level Objective
SLR	Systematic Literature Review
SMT	Simultaneous Multithreading
SPar	Stream Parallelism
SPE	Stream Processing Engine
TBB	Threading Building Blocks
VM	Virtual Machine

CONTENTS

1. INTRODUCTION

Currently, parallelism is everywhere as computer systems support many architectural paradigms (e.g., multi-cores, co-processors) [88]. However, automatic parallelization of sequential codes is still not fully achievable. Hence, parallel programming is essential to exploit the machines' resources properly.

Although the great potential of parallelism exploitation to provide high performance for real-world applications, it increases complexity in terms of programmability. Parallelism requires managing low-level aspects (e.g., communication, synchronization, load balancing, scheduling). Introducing details of parallelism exploitation remains too complex for application programmers, who are focused on developing applications and may not necessarily have expertise in parallel architectures.

Considering the mentioned above inherent parallelism complexities, different parallel programming frameworks and libraries are being proposed to facilitate parallelism exploitation in multi-core machines, such as OpenMP [27], Intel Threading Building Blocks (TBB) [134], FastFlow [3] and Stream Parallelism (SPar) [47]. These solutions are applicable to support parallelism in various applications domains.

A relevant example of a domain of applications is stream processing applications, which are applications that compute streams of data and provide insightful results in a timely manner [6, 122]. This application domain emerged due to the increasing use of techniques to collect data from different sources (*e.g.,* sensors, cameras, and radar). Stream processing applications have unique aspects, such as continuous data processing and varied workload trends. There is a considerable number of applications that must gather and analyze data in real (or near-real) time [58, 6, 16, 133], which is a complex demand.

Complementing the programming frameworks mentioned above for multi-core machines, there are also distributed Stream Processing Engine (SPE) in stream processing such as Apache Storm [120], Apache Spark [142], and Apache Flink [11]. These SPE are designed for large-scale clusters using Java Virtual Machine (JVM) to provide hardware abstractions. However, as shown in [143], Java-based SPE achieve a limited performance and efficiency due to the suboptimal data serialization, memory accesses, and garbage collection. Therefore, we have seen C++ based solutions running on a single multi-core machine outperforming cluster solutions, where representative case studies are provided by PiCo [94] and WindFlow [90]. Thus, we expect that the efficient use of a multi-core machine provides a level of performance suitable for a significant part of (soft) real-time stream processing applications. Therefore, in this work, we use multi-cores as execution environments.

1.1 Research problem context

Many domains of parallel applications are subject to changing conditions and fluctuations (e.g., workload, input rates, and environment) while they are running (at run-time) [6], where a configuration that sustained some quality of services can become instantly suboptimal [131]. Additionally, the unbounded data arrival requires many applications (e.g., stream processing) to execute for long or infinite periods. Therefore, applying adaptation actions online (at run-time) can improve responsiveness to changes [101].

Consequently, new techniques are being developed to cope with scenarios that suffer changes at run-time, where self-adaptation is a relevant example [111, 56, 71]. Self-adaptation can be broadly defined as the capability of the systems and environments to be autonomous, deciding and changing their behavior in response to some behaviors or uncertainties [137]. Many entities can be changed to achieve self-adaptiveness, e.g., in stream processing, self-adapting entities such as the batches size, and the parallelism degree to pursue a given performance or efficiency. Importantly, self-adaptation can be viewed as a broader context encompassing many optimizations and entities. For instance, auto-scaling can be a facet within self-adaptation as the management of computing resources can be autonomous using mechanisms for providing elasticity.

On the one hand, self-adaptation can make the executions of parallel applications more intelligent, reducing human efforts and assisting in error-prone configurations [127, 137]. On the other hand, self-adaptation at run-time is still challenging to increase flexibility and efficiency. First, the adaptation space possible at run-time is still limited. There is a lack of mechanisms to apply adaptation actions and efficient decision-making strategies to decide which configurations to be enforced. For instance, the applications' compositions structures (parallel patterns, graph topology) are non-intuitive configurations that are still up to the programmers to designate [131]. Second, a representative part of parallel applications has complex composition structures [1], making the parallelism exploitation and the enactment of adaptation actions more difficult [118]. Third, the design of the self-adaptive solutions is expected to be improved to support the decoupling of modules and enable more reusable/generic approaches. Fourth, the evaluation of self-adaptation is also a rising concern not receiving the necessary attention, where we argue that this part demands new comprehensive methodologies and guidelines [127].

Moreover, self-adaptation can also be used for providing additional parallelism abstractions to application programmers, which is a potential opportunity to simplify the process of running parallel applications [132, 127, 51]. However, considering the state-of-the-art, it is an open question to what extent self-adaptation can be applied and how

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

efficient it is to provide abstractions of the parallelism exploitation and the applications' executions [127].

1.2 Research goals

In this Ph.D. dissertation, we address parallelism abstractions for parallel applications. We aim at providing research perspectives for supporting additional parallelism abstractions and efficiency. Hence, a potential adaptation space is to apply new self-adaptive strategies to manage and optimize applications at run-time. We apply the proposed solutions to the representative context of stream processing applications. Therefore, the research goals can be summarized as follows:

- Improve the existing knowledge about the state-of-the-art of self-adaptation on stream processing;
- Increase the adaptation space to enable self-adaptation of applications' compositions structures, which can provide more flexibility and efficiency for high-level abstractions;
- Within the self-adaptable applications' compositions structures, many inner compositions are complex formed by many parallel stages. There is a need to provide self-adaptation of the number of replicas in these complex compositions;
- Improve the evaluation of self-adaptive solutions. We argue that the impact of self-adaptation can be better measured to estimate its implications in terms of resources consumption, performance, and efficiency.

1.3 Contributions

Considering the research challenges and goals explained above, we provide the following main scientific contributions in this Ph.D. dissertation:

- Categorizations, a taxonomy, a catalog of self-adaptation optimization, and a discussion of research challenges and perspectives of self-adaptive executions in parallel stream processing. More details can be seen in Chapter 3 where we present an SLR;
- A conceptual framework for decision-making in self-adaptive parallel executions, which is described in Chapter 4;

- Strategies for a self-adaptive number of replicas for applications with a single parallel stage. The strategies support non-functional requirements such as latency and throughput, and Service-Level Objective (SLO) for managing resources utilization, minimizing self-adaptation overhead, and providing seamless parallelism management. The context and these strategies are addressed in Chapter 5;
- Mechanism, models, and strategies for self-adaptive applications' compositions structures. In Chapters 6 and 7 we describe and discuss this contribution;
- Mechanism and strategy for a self-adaptive number of replicas in complex compositions within the applications' compositions structures. In Chapter 8, we provide a mechanism implementation and an optimal decision-making strategy that is compared to the state-of-the-art solution called DS2 [64].

The majority of these contributions have already been presented and published in international workshops, conferences, and journals. New ideas were published in specific workshops, such as the Workshop on Autonomic Solutions for Parallel and Distributed Data Stream Processing. Other research articles were presented in relevant conference venues, particularly in Euromicro International Conference on Parallel, Distributed and Network-Based Processing and The Parallel Computing Conference. Research articles were published in Journals of the area, such as in The Journal of Supercomputing, Communications in Computer and Information Science, Springer Computing, and Concurrency and Computation: Practice and Experience.

1.4 Document organization

The content of this Ph.D. dissertation is organized into different chapters. Chapter 2 presents this work's background. Chapter 3 provides a revision of the related literature. Then, Chapter 4 introduces the proposed conceptual framework.

Moreover, after providing the fundamentals, state-of-the-art, and conceptual framework, then in this Ph.D. dissertation, there are four technical chapters with specific scientific contributions. Chapter 5 provides strategies to self-adapt the parallelism degree in applications with one parallel stage, where we introduce relevant non-functional metrics (throughput, latency), and discuss the limits of self-adaptive abstractions, and optimize the self-adaptation overheads.

Then, considering the demand for additional mechanisms to increase the flexibility to self-adapt parallel applications, Chapter 6 introduces a new mechanism to self-adapt the Parallel Patterns and online change the applications' graphs topologies. Additionally, Chapter 7 provides an optimized decision-making strategy for the mechanism proposed in

Chapter 6. Chapter 8 introduces a proposed mechanism and decision-making strategy for supporting self-adaptation of replicas in applications with complex composition structures composed of many parallel stages. Finally, Chapter 9 contains the conclusion discussing this work’s closing remarks, implications, and potential future works.

2. BACKGROUND

In the past decades we have seen the rise of a new type of applications. The stream class of applications was needed to process data in a continuous fashion without having a predefined end to the execution (sometimes never ending) [16]. Section 2.1 introduces parallelism aspects. Then, Section 2.2 provides an overview of the stream processing paradigm. Additionally, Section 2.3 presents an overview of concepts related to self-adaptation.

2.1 Parallel computing

Potential performance optimization for real-world applications concerns parallelism. However, performance gains are usually conditioned to introducing parallel routines to applications. As emphasized in [46], a high percentage of applications are still sequential and thus cannot run in parallel. This is due to the fact that introduce parallelism techniques tends to be challenging for application programmers that are not experts in performance. Refactoring code to introduce parallelism usually results in application programmers facing a trade-off between coding productivity and performance. This occurs because parallelism increases the performance, but use efficiently parallelism routines is a time-consuming task that usually decreases the programmers' productivity.

2.1.1 Parallelism properties

In recent efforts for improving coding productivity, [118] argued that parallel programmers and frameworks should focus on three parallelism aspects: Programmability, Portability, and Performance.

Programmability concerns the development costs for coding effective and efficient parallel solutions [118]. We argue that a relevant aspect of programmability is to offer programming interfaces/languages that allow application programmers to introduce parallelism maintaining their code similar to the sequential one. This aspect has a synergy with code reuse that was a priority mentioned in [118], code reuse enables the usage of optimization available in the original sequential code without rewriting it. Moreover, abstracting from the application programmer the need to control parallelism aspects such as synchronization, communication, and task scheduling is a potential opportunity for enhancing the programmability. Another facet of programmability is code productivity, improving the programmability certainly helps application programmers to be more productive. Aiming at improving programmability and productivity, in Section 2.1.2 we present high-level parallel programming concepts.

Portability code portability is basically the idea to enable a given program to be run in different platforms and architectures. A further optimization that is far more complex is performance portability, which encompasses the idea that the same program would achieve an expected (high) performance in different environments. [118] argued that code rewriting and tuning is required for achieving the expected performance when a parallel program is ported to new hardware. We believe that performance portability can be improved by employing self-adaptivity to autonomously managing parallelism aspects, Section 2.3 will introduce foundations for this topic.

Performance is another very relevant aspect related to parallelism, which usually is the primary goal for parallel programming. However, achieve the performance needed can be a challenging task. Firstly, it is not possible to efficiently parallelize every computation, there are also code parts that cause serialization (*e.g.,* strict tasks order or dependency). A theoretical limit is known by Amdahl's Law [4]. Moreover, performance gains are even harder lately with the end of Moore's Law [95, 116]. On the other hand, the performance of running applications is expected to continue increasing, where properly use available resources and scale the performance is a common objective. We believe that scalability can be achieved while maintaining high-level abstraction for application programmers, the capacity to let an application to self-adjust its configurations is a long-term goal that is discussed in Chapters 5, 6 , 7, 8.

2.1.2 High-level parallelism

The use of high-level parallel programming methodologies is a potential alternative to provide coding abstractions for application programmers [28]. Abstractions can reduce the application programmers' burden. The main goal of high-level parallel programming can be defined as reducing programming efforts while ensuring a reasonable performance and portability. High-level abstractions tend to be provided by approaches that hide from programmers the complexities related to parallelism [85].

There are two related concepts attempting to raise the abstractions for parallel programming in a more structured mode: parallel patterns [88] and algorithmic skeletons [23, 2, 21]. The basic idea is to provide for programmers recurrent constructors (skeletons) to be used for modeling parallel applications, where the common goal is to increase the coding productivity. Such constructors are called *patterns*, where several patterns may exist with different communication models, synchronization techniques, and task execution [88].

Structured Parallel Programming may be viewed as a methodology that uses libraries or languages to facilitate coding. A usual approach towards structured parallel programming consists in using parallel patterns, which are well-accepted concepts that emerged from best coding practices in software engineering for optimizing and reusing specific code parts. The term parallel pattern refers to how the task distribution and data access are used recurrently in the design of a parallel program [88].

Patterns are expected to be generic and universal, which can theoretically be implemented in any programming interface [88]. Consequently, flexibility is targeted by providing a vocabulary with several different patterns. In this study, important examples of patterns are Map, Pipeline, and Farm [21, 88]. A Map can be simplistically defined as a function replication of elements that are processed in collections separated by indexes, where the replication is mostly suitable for independent elements. A Pipeline is a pattern handling tasks in a producer-consumer fashion (like an assembly line) that is composed of connected stages. In a Pipeline pattern, the data items flow through an acyclic graph, where each stage performs different computations/tasks.

Farm is a pattern composed of a Pipeline. While a Pipeline is composed of sequential stages, a Farm has one or more parallel (replicated) stages. A Farm has also at least one stage called Emitter (*A*), which gets the input tasks and sends them to the next stage, according to a scheduling policy. In a Farm, the stage following an emitter is usually fissioned (Also Known As (AKA) replicated, parallel) with a number N of parallel agents (called workers or parallelism degree), where N is the parallelism degree. A Collector (*C*) gathers the tasks from the workers and places them into the output stream. In the context of stream processing, the stream items flow through the graph, continuously gathering input items as well as producing output results. The Farm pattern used in a parallel execution characterizes a composition, which can be seen as the applications' graphs topologies. In Figure 2.1 we show a representation of parallel patterns, where data items can be seen as elements to be processed and tasks are a generalization of computations that are executed (*e.g.,* by a thread). Moreover, parallel patterns can be semi and arbitrary nested to compose new parallel patterns [88].

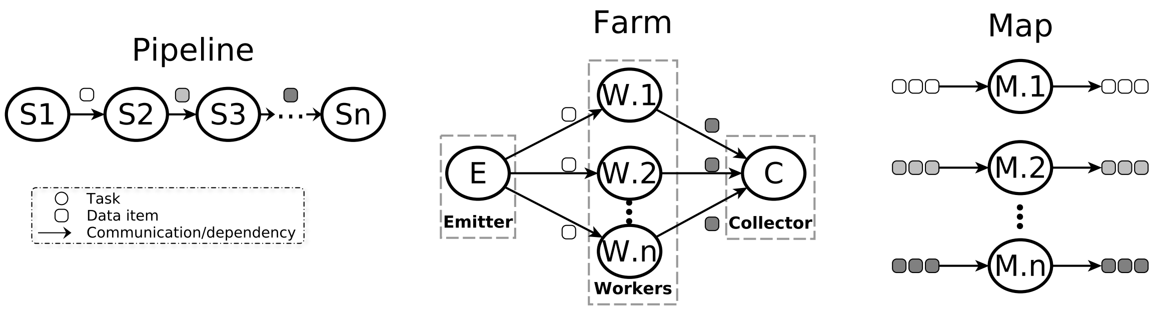


Figure 2.1: Representation of parallel pattern examples.

Source: [127].

These parallel patterns have been implemented with programming abstractions as languages and Application Programming Interface (API) for parallel stream processing [28]. Some of them are known as distributed SPE, for instance, Apache Storm [120], Apache Spark [142], and Apache Flink [11]. These SPEs are designed for large scale clusters using JVM to provide hardware and communication abstractions. There are also languages/frameworks for multi-core parallelism exploitation, such as Intel TBB [134], FastFlow [3]. Additionally, there are domain-specific languages for exploiting stream parallelism, like StreamIt [117] and SPar [47].

2.1.3 SPar

Considering the inherent challenges of high-level parallelism abstraction in stream processing applications, SPar[1] was specifically designed to simplify the stream parallelism exploitation in C++ programs for multi-core systems [46, 47]. It offers a standard C++11 annotation language to avoid sequential source code rewriting. SPar also has a compiler that generates parallel code using source-to-source transformation technique[2]. SPar uses FastFlow [3] as the main runtime library, where all low-level parallel programming advanced concepts and implementations (scheduling, load balancing or parallelism strategies) are resolved by SPar's compiler. Moreover, recent efforts have extended SPar for supporting other runtime frameworks as backends for the code generated, such as supporting TBB [59], OpenMP [60]. Other works have supported SPar to execute in additional environments: distributed [98] and Graphics Processing Unit (GPU) accelerators [102]. Additional works increased SPar abstractions for data parallelism [75] and self-adaptation [124, 129].

It is also important to note that SPar's primary goals are to increase application programmers' productivity and provide scalable higher-level abstractions. In our understanding, self-adaptation can assist in enhancing productivity and abstractions for executions in several environments and support different user goals.

Attributes can be used in C++ to express parallelism in code annotations [28]. SPar's language is composed of five attributes to express the key properties of the stream parallelism. Listing 2.1 is showing the use of the attributes in the source code annotation. The `ToStream` attribute represents the beginning of a stream region, which is the code block between the `ToStream` and the first `Stage` (lines 1 and 2). The `ToStream` attribute therefore labels where a stream parallel region starts in a given program. The `Stage`s are defined inside the `ToStream` region to label the computing phases where stream items will be processed, like an assembly line. Usually, stream processing applications will never end

[1]SPar's home page: https://gmap.pucrs.br/spar

[2]The content of this section utilizes some material from SPar's background section shown in reference [129].

like in Listing 2.1, but the users may want to finish the application at some point during the execution. They can do so by introducing a stop condition in the `while` loop as well as introducing an `if` condition before the first stage that breaks the current loop, which can be any kind of loop from the host language.

```
[[spar::ToStream]] while(1){
 i = read_item();
 [[spar::Stage,spar::Input(i),spar::Output(i),spar::Replicate(n)]]
 {
   i = filtering(i);
 }
 [[spar::Stage,spar::Input(i)]]{
  write_item(i);
 }
}
```

Listing 2.1: Demonstrative example of the SPar language.

Once the `Stage` and `ToStream` are annotated, the `Input` or `Output` attributes are inserted to define the input and output data dependencies. The attribute arguments can be one or more variables from different data types, which label the stream items that will be consumed or produced by a given region. Finally, the `Replicate` attribute may be inserted in the attribute list of `Stage` to define the degree of parallelism of that region. The argument of this attribute is the degree of parallelism (the number of stage's replicas), which can be a constant integer number or variable. SPar is not able to automatically manage stateful operations. Thus, only `Stage`s with stateless operations can actually be replicated without any user intervention.

The SPar compiler performs source-to-source transformations by generating calls to the FastFlow library. The compiler interprets the annotations in the source code, performs semantic analysis, and applies transformation rules based on stream parallel patterns [47]. The outcome is that from the high-level SPar annotations, parallel patterns such Farm, Pipeline or a combination of Pipeline with Farm stages are generated using the FastFlow programming framework. The parallel code generated is what we call the SPar runtime.

Figure 2.2 depicts the activity graph and communication between stages of the SPar runtime system generated from the example of Listing 2.1. This provides an overall idea how it works. Note that the `Replicate` attribute applies the replication role over the `Stage`. Each replicated stage has its own input and output lock-free queue, where this term refers to an operation that eventually completes after a given number of steps [118]. The first stage is actually the code left inside a `ToStream` region, which generates stream items for the subsequent stages. It is also responsible for scheduling items, which by default is round-robin. However, users may need an on-demand scheduler that is made possible through a compiler flag in the SPar compiler (`-spar_ondemand`). When this flag is present, the queues size is one (stream item). Thus, as soon as one stream item is popped by the current stage, another will be pushed from the previous stage.

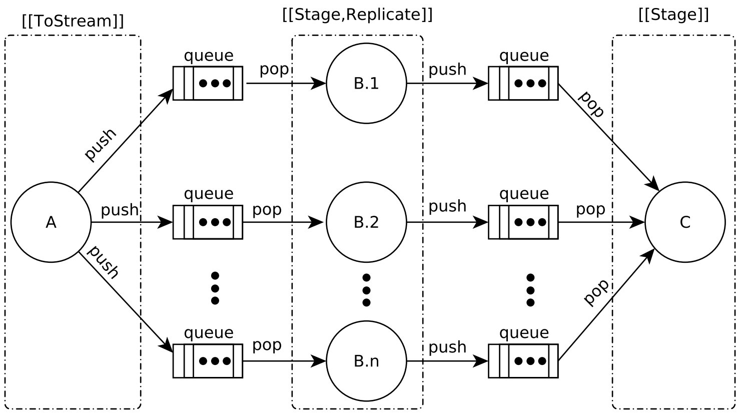


Figure 2.2: SPar runtime: activity graph and communication queues.

Source: [129].

In the SPar runtime, the default configuration is the stages actively trying to push or pop stream items from the queues. If the queue is full or empty, the stage thread remains in a loop, trying to perform push or pop until it finally succeeds. Every time that a given stage fails in perform push or pop, the stage generates a push or pop lost event. This may generate an extra overhead for coarse-grain computations. Therefore, users may set the SPar runtime to behave in a blocking mode through the *spar_blocking* compiler flag. In this case, the stage thread will not stay in a loop, it will wait until it can perform push or pop in the shared queue.

Another important concern is to preserve the order of stream items, the ordering is implemented in stage after the replicated one. SPar is able to automatically handle out-of-order stream items in the last stage when `-spar_ordered` compiler flag is defined.

A relevant part of SPar is its code generation, which is provided by a compiler tool. The SPar's compiler is powered by Compiler Infrastructure for New C/C++ Language Extensions (CINCLE)[3]. In short, CINCLE applies a parser step following the standard C++ grammar with an interface that creates the Abstract Syntax Tree (AST). As an illustrative example, Figure 2.3 shows the generic compilation flow, where blue boxes correspond to SPar implementations and the orange boxes relate to the CINCLE modules provided to generate SPar compiler. Note that semantic analysis and the AST transformations are those implemented to support SPar language and parallel code generation.

[3]Technical implementation details can be found in [46].

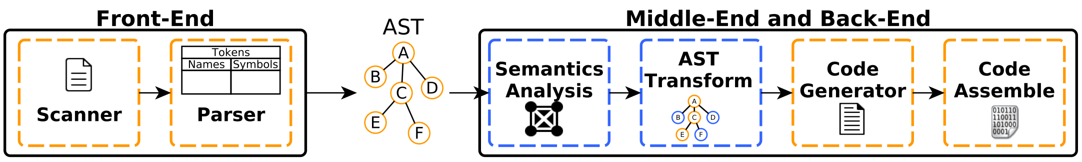


Figure 2.3: The SPar's compiler representation.

Source: [47].

2.1.4 FastFlow

FastFlow is a programming framework developed guided by the structured parallel programming methodology to support efficient parallel computing on heterogeneous multi-cores [2, 118]. FastFlow has a specific C++ API to interact with application programmers, which is included in the form of a header-only library and provides several ready-to-use parallel patterns to be instantiated. A given parallel application is usually developed using FastFlow by instantiating patterns in the application's business logic code and connecting these patterns creating a data-flow graph (e.g., composition structure). FastFlow also has its streaming runtime system that executes the data-flow graphs. Considering that the FastFlow is the primary runtime system of SPar's code generated, the description about SPar runtime provided in Section 2.1.3 is actually representative of the FastFlow runtime system.

Moreover, the reader interested in a more detailed description about FastFlow can refer to reference [118] that is a recent Ph.D. dissertation of the leading developer of FastFlow, where a new version of FastFlow is thoroughly explained that provides flexible optimizations in terms of performance and programmability.

We use FastFlow's flexibility to validate our proposed solutions in this work. As an organization matter, further FastFlow's details are presented when necessary to explain the implementations in the following chapters. Moreover, in some cases, we can improve our solutions' usability by employing SPar to generate a self-adaptive parallel code that uses FastFlow as the runtime system.

2.2 Stream Processing

Stream processing applications can be defined as programs that continuously compute data items. A *stream* is a given input that arrives from sources in the form of an infinite sequence of items [58]. Examples of stream sources can be equipment (radars, telescopes, cameras, among others) and file bases (text, image). Moreover, processing stages (a.k.a. operators) are entities that consume the incoming streams by applying computations. Usually, a stream processing system is composed of stages that communicate between them and provide results in a timely mode. The stages tend to be organized as a directed acyclic graph (DAG) where each stage performs specific computation and the stream item flows through the graph. Currently, there is available a large number of applications that characterize stream processing, these applications represent a significant workload in our computing systems.

The characteristics of stream processing applications vary depending on the data source and computations performance. One of the most highlighted aspects is the continuous and unbounded arrival of data items [6]. Lately, we have witnessed 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 as the possibility to concurrently perform different operations over independent stream items. The next section provides further parallelism details related to stream processing.

The generic concept of stream processing systems was splitted by [103] into two parts: General stream processing systems and Complex Event Processing (CEP) systems. General stream processing represents the broad concept, encompassing systems and applications that may receive different data types and produce a result in real-time. CEP systems, on the other hand, concerns systems dedicated to detecting information from streams coming in the form of events. CEP is highly associated with the processing of data coming from Internet of Things (IoT) devices, where each stream sent from a given sensor is handled as an independent event. In this work, we focus on the general stream processing system, but our solution can be also suitable for CEP systems.

2.2.1 Parallelism in stream processing

Previously, in Section 2.1, we described parallelism aspects that can be generic to a wide range of applications. Parallelism is also very important for improving the performance of stream processing applications. In stream processing, several types of parallelism can be exploited. [58] classifies the three main types of parallelism suitable for processing data streams, which are shown in Figure 2.4. Pipeline-parallel (a) is related to the concurrent running of different stages (A,B) in a producer and consumer model. Task-parallel (b) on the other hand, concerns the concurrent execution of different computations (D,E), where it is usually a function that performs operations over duplicated data. Moreover, in Data-parallel (Figure 2.4(a)) the same computation G is replicated in such a way that each instance processes different data or different partitions of the same data.

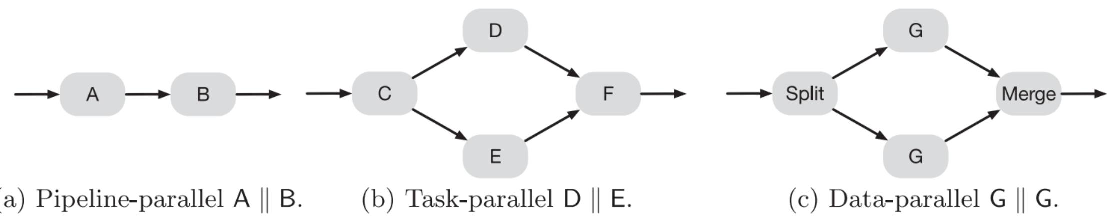


Figure 2.4: Types of parallelism in the context of data stream processing.

Source: Extracted from [58].

The classification of [58] can be viewed as a categorization for the processing of data streams. In a broader view from a stream processing perspective, we adopted and derived three parallelism types from [118, 46].

Data parallelism is similar to the data-parallel from [58] that can be generalized to data parallelism divided into sub-collection or partitions. As stated by [118], *data parallelism can be characterized by the replication of functions and partitioning of data*. Data parallelism usually provides profitability with the parallel processing of loops over independent read-only data. This technique can decrease execution times in case the data partitions are mostly independent. In the case of dependencies, an internal state is usually maintained which then characterizes *stateful* computations that are very usual in data stream processing [100, 103, 89].

Stream parallelism can be seen as a combination of pipeline and data parallelism. In this type, sequential and parallel stages run simultaneously over independent items. The logical view is usually a graph where stages communicate via dedicated channels. In stream parallelism, heavy stages can be replicated in such a way that the same computation is performed over different data items.

Task parallelism characterizes the task-parallel definition from [58] where functions are replicated attempting to improve the overall performance.

2.3 Self-adaptation overview

The software engineering field has been evincing that modern software systems should operate in dynamic conditions without downtime [18, 136, 111]. There, concepts of self-adaptation are being used in software systems to collect data and to adjust themselves. In the context of this work, foundations of self-adaptivity are aimed to be used for managing parallelism aspects as well as for providing high-level abstractions. Consequently, in this section, we provide an introduction to self-adaptive theories and concepts.

2.3.1 Definition

The concepts of self-adaptation are not strictly defined in the related literature [18, 136, 111]. In Cheng *et al.* [18], self-adaptation was described as a characteristic of a system that "is able to adjust its behavior in response to their perception of the environment and the system itself". Although other similar definitions are available, such a short and precise definition allows us to reason about the theoretical foundations of self-adaptive systems.

Moreover, Weyns [137] defines:

> "The basic idea of self-adaptation is to let the system collect new data (that was missing before deployment) during operation when it becomes available. The system uses the additional data to resolve uncertainties, to reason about itself, and based on its goals to reconfigure or adjust itself to maintain its quality requirements or, if necessary, to degrade gracefully."

From Weyns [137] quote, one can note that self-adaptation focus on applying actions at run-time due to the need to adapt to specific unknown execution scenario, which is a possible way to handle the uncertainties. This is usually done by collecting data and extracting knowledge from such data (not so easy in practice). Then, it becomes possible to decide which actions to take with knowledge. For instance, try to increase the Quality of Service (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.

Our understanding is that this view as a black box is suitable for non-expert users that determine their goals, and the self-adaptive system ideally behaves to meet them. Thus, providing user abstractions.

A complementary interpretation is the "internal" one, which Weyns [137] determines that a self-adaptive system is composed of two parts:

> "The first part interacts with the environment and is responsible for the domain concerns – i.e. the concerns of users for which the system is built; the second part consists of a feedback loop that interacts with the first part (and monitors its environment) and is responsible for the adaptation concerns – i.e. concerns about the domain concerns."

Our perspective is that these two basic principles are complementary, and using one does not implicate ignoring the other. In fact, from a practical angle, we believe that both can be employed in the same solution, where what would determine the principle is the angle and abstraction level of the one who views the self-adaptive system. The strategy designers and system developers tend to have an internal view that usually comprises two parts. On the other hand, the user tends to have an external perspective from a higher abstracted angle.

Understanding the views from different perspectives of self-adaptive is relevant for this work because we are interested in providing abstractions by employing self-adaptation. For instance, in Chapter 4 we use the internal view and its two parts to propose a conceptual framework intended to help the strategy designers and system programmers. By using this framework, we expect that it enables us to improve the self-adaptive solutions applied to parallel computing. Then, the technical chapters 5, 6, 7, 8 use the framework and its parts in practical solutions. However, the external principle is the level of abstraction that we intend to provide to users of parallel systems and application programmers that intend to have self-adaptation in the executions of their parallel applications.

Yet regarding definitions of self-adaptive systems, a non-intuitive part is how self-adaptation relates to other concepts that are also used to apply actions and make systems more dynamic. Weyns [137] explains that other concepts like autonomous systems, multi-agent systems, self-organizing systems, and context-aware systems are not well compliant with the basic principles above-explained of self-adaptation. These additional concepts are not compliant with the second principle. They do not distinguish between the different parts. Although these concepts can be extended to comply with all self-adaptive principles, we understand that not separating the system into parts/modules implies limited modularity and flexibility. Hence, other concepts are represented in the literature but usually refer to narrow solutions where the adaptation actions are entangled within the systems, which reduces their generalizability.

2.3.2 Applying self-adaptation

Considering relevant aspect of the self-adaptation is *how* to adapt, [136] proposed a conceptual model of self-adaptive systems. This model is depicted in Figure 2.5 that encompasses the following parts:

- **Environment:** Refers to the “world” that the system runs inside. The environment tends to be a part that the system does not have full control, which is subject to uncertainties and variations. For instance, the environment of a parallel application in the operating system and machine that it runs.
- **Managed System:** Is the entity that usually receives additional components (sensors, actuators) in order to be controlled and adapted. The adaptation actions are expected to be applied while ensuring safety to the managed system. For instance,

in the case of a self-adaptive parallel application the managed system the business logic code of the application.

- **Adaptation goals:** Are usually concerns handled by the self-adaptivity. [136] provides four main types of goals: self-configuration (a system that arranges itself), self-optimization (autonomously optimizing the execution), self-healing (seamlessly detecting and repairing issues), and self-protection (a system that transparently defends itself from problems). For instance, a parallel application can have an adaptation goal to increase its performance or the efficiency of the resources usage.
- **Managing System:** Its policy, strategy or system that controls the managed system. The adaptation goals are used for deciding whats actions to take and the low-level constraints of the managed system are usually defining what to adapt.

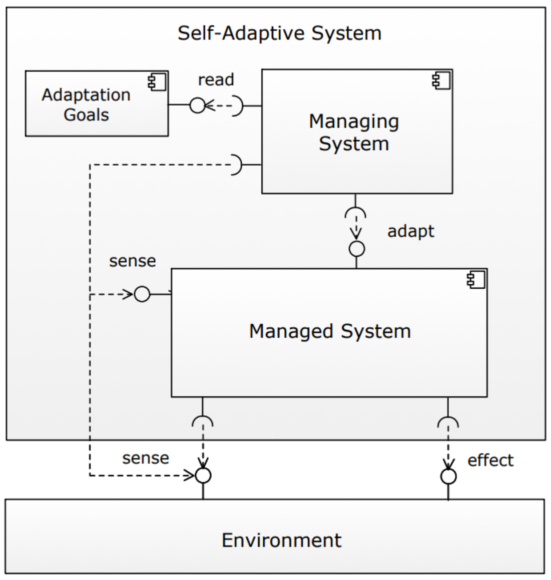


Figure 2.5: Conceptual model for a self-adaptive system.

Source: Extracted from [136].

The workflow of sensing and applying adaptations usually characterizes a feedback loop. Control engineering proposed the use of feedback loops as entities for providing adaptation capabilities. Moreover, feedback loops are conceptually essential entities for enabling self-adaptation to computing systems. The most relevant concepts of feedback loops are described in the next paragraphs.

Control engineering and theory is a concept aimed to automate systems. The concepts used in this work are based on [56], which uses the term feedback control for monitoring a system's execution to optimize it. This is performed by monitoring aspects such as throughput, latency or system utilization and, by triggering optimization in the next execution. Monitoring the outputs of the system and performing actions according to its behavior results in an architecture called feedback or closed loop [56].

Control theory can be used in the context of stream applications and it is applied in order to implement service levels regulating the application processes. The implementation of control theory into stream applications requires a control framework to manage execution [56]. An example of is a feedback control in computers is using Queueing Systems mainly for performance modeling. A simple example is shown in Figure 2.6, where the items are placed in a queue and processed by the servers (circle).

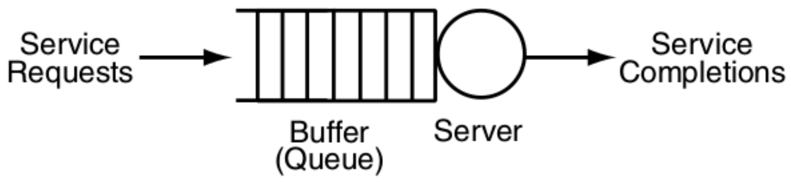


Figure 2.6: Simple example of a Queueing System.

Source: Extracted from [56].

Figure 2.7 shows an example of a feedback control loop to manage response times in a queuing system. It is relevant to point out that such a system has an SLO to maintain a given response time (constraint).

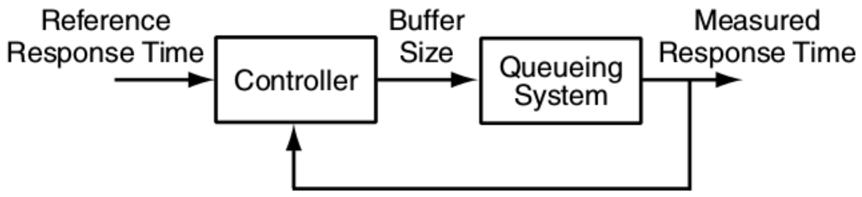


Figure 2.7: Example of a feedback control.

Source: Extracted from [56].

A closed loop using a feedback control is considered a simple system model that can adapt to disturbances [56]. Thus, it fits the requirements of a many real-world applications. Existing approaches applying self-adaptation to the context of parallel stream processing are described in Chapter 3.

3. LITERATURE REVIEW

We conducted an SLR to map and comprehend the current state-of-the-art regarding self-adaptation applied to parallel stream processing. Our research scope considers the known characteristics of dynamic, irregular, and unbounded stream processing applications, where self-adaptation can potentially improve for different goals and scenarios applications with such characteristics. The content of this chapter was published in [127] as an article presenting a SLR survey, where the material provided here is a slightly modified version reproduced here in accordance with the signed copyright agreement. The interested reader can access the complete original version in an article format in reference [127]. In summary, this chapter provides the following main scientific contributions:

- Categorizations of self-adaptation characteristics, parallelism properties, and validation categories in existing tools /frameworks for self-adaptive executions in parallel stream processing.
- A unified taxonomy for categorization and validation of self-adaptation in parallel stream processing.
- A catalog defining self-adaptation goals and entities managed.
- A discussion of open research challenges and perspectives for future enhancements on stream processing with self-adaptiveness.

3.1 Research method

This section first overviews the research questions. Then, we contextualize our search strategy and protocol.

3.1.1 Research questions

Considering the research scope and research problem, we distilled the following Research Questions (RQ):

- RQ1: Which are the publication's goals when applying self-adaptation to stream processing?
- RQ2: Which entities that enact adaptation are being dynamically adapted in existing solutions?

- RQ3: Which information is considered for performing adaptation?
- RQ4: How is the adaptation decision-making performed?
- RQ5: What parallelism aspects are being exploited in self-adaptive solutions for stream processing?
- RQ6: Are the approaches focusing on providing parallelism abstraction to application programmers when applying self-adaptation?
- RQ7: Which experimental procedures are used for validating the proposed solutions?
- RQ8: Which variations (*e.g.,* workload, application) are considered for evaluating the proposed solutions in the experiments?
- RQ9: Are the approaches considering the overhead that adaptation may cause?

RQ1 aims to show up the main goals for applying software adaptation to stream processing. RQ1 also aims to provide a better understanding of the motivations and potential advantages of self-adaptation in the context of stream processing.

RQ2, RQ3, and RQ4 concern internal aspects of the self-adaptive solutions available. As parameters or system settings may be changed, it is very relevant to list what each solution is changing at run-time. Moreover, for performing adaptation, it is required that an entity decides for it. Thus, an entity has to consider some information for deciding what change to make, RQ3 aims at answering which information or statistical data is used in the decision-making step. Additionally, RQ4 attempts to unveil which mechanisms are used on specific scenarios for actually performing adaptation actions.

RQ5 relates to how parallelism is being exploited on self-adaptive solutions, specifically the framework/language used and which architecture/environment is the solution targeting. Moreover, a relevant aspect for parallel stream processing is how many replicated stages the self-adaptive solution is able to manage, where in this broad SLR different scenarios of stream-like applications are being considered (*e.g.,* data stream, complex event processing, streaming systems). Considering the importance of providing high-level parallelism abstraction described in Section 2.1.2, RQ6 focuses particularly on evaluating if the self-adaptation properties can be used/adapted for providing parallelism abstractions for application programmers.

Another relevant aspect of self-adaptation is how to validate a proposed solution. RQ7 seeks representative aspects used for evaluating self-adaptive solutions. Additionally, a representative and the broad experimental setup is expected to properly evaluate the quality of self-adaptive solutions. Thus, RQ8 intends to identify whether the solutions are being comprehensively evaluated regarding variation that can be application characteristics, workloads, environments, etc. Yet concerning the validation of the proposed solutions, the overhead caused by the self-adaptation is very relevant. RQ9 has this concern, looking at whether the validation of the solution is considering the potential overhead of adaptation.

3.1.2 Search strategy

The search strategy is composed of incremental steps. The first step was to elaborate on the search string that was used for automating the search. As recommended by Kitchenham's guidelines [68], the research questions of this literature review were separated into facets related to three main aspects: stream processing, self-adaptiveness, and parallelism. We defined 5 control studies [44, 31, 113, 110, 107] that were previously known as relevant primary studies from our previous works [132, 129, 131, 125, 51]. Pilot searches were conducted on Scopus with the different terms and synonyms for testing if the string found the control studies. Thus, we converged to a search string evinced in Table 3.1 with the terms of the three main aspects separated by boolean ANDs, where these terms searched studies' titles, abstracts, and keywords.

In the area of stream processing, we included relevant terms known to represent stream processing characteristics and paradigms. In the self-adaptation properties, we included also terms related to "autonomic" that can be seen as a way to achieve self-adaptation. Regarding the parallelism scenario, we included also terms like the ones related to concurrency because it is used interchangeably with parallelism in some works. In short, we included similar known terms in order to find a high number of relevant studies. But at the same time, we avoid broader terms that bring irrelevant papers. In Section 3.5, we discuss a compromise between these two objectives.

Area: Stream Processing	Scenario: Parallelism	Property: self-adaptation
"stream processing" OR "data stream" OR "complex event processing" OR "streaming application" OR "streaming system"	"parallel" OR "concurren*" OR "scal*"	"adapt*" OR "autonomic" OR "autonomous" OR "elastic" OR "on-the-fly reconfiguration" OR "online reconfiguration" OR "self-*" OR automatic scaling

Table 3.1: Search terms.

The next step was running the search string in the most relevant research databases and digital libraries for finding primary studies. The research databases used are Scopus[1] and Web of Science[2]. Moreover, the two digital libraries, namely ACM Digital Library[3] and IEEE Explore[4]. The search string was defined according to the systematic review

[1]https://www.scopus.com
[2]https://webofknowledge.com
[3]https://dl.acm.org
[4]https://ieeexplore.ieee.org

procedure adapted to the syntax of each repository. The search string was executed in November of 2020.

In the third step, the titles and abstracts of the papers were read and filtered accordingly to the inclusion and exclusion criteria presented in Section 3.1.3. Then, we performed a skimming step, similar to what was performed in related surveys [100], covering a full-text view (figures, tables, flowcharts, graph results) of the papers. In the fifth step, we read the full papers for performing the final decision. This was the most intensive step, where we critically double-checked the paper considering the inclusion and exclusion criteria.

3.1.3 Study selection criteria

The following criteria were used for filtering the papers:

- **Inclusion criteria 1**: The study applies self-adaptation properties to stream processing. Rationale: we include technical studies that do not explicitly mention self-adaptive systems, but that encompass self-* properties.
- **Exclusion criteria 1**: Not a scientific paper that is not written in English or that is a short version (e.g., editorial, abstract, poster). Rationale: This type of study lacks in space for proposing and validating relevant solutions.
- **Exclusion criteria 2**: A publication that has no self-adaptation aspects, concerning only stream processing. Rationale: Our focus is on adaptive properties applied to stream processing.
- **Exclusion criteria 3**: a publication that has no aspects related to parallelism, i.e., only concerning self-adaptation. Rationale: Considering that parallelism is a pervasive topic that is relevant for several optimizations, we focus on studies addressing self-adaptation in parallel systems.
- **Exclusion criteria 4**: publications that are not considering stream processing. Rationale: we focus on solutions for the stream processing context where applications have unique characteristics, i.e., long-running and dynamic.

3.2 Self-adaptation: categorization and taxonomy

Here we propose categorizations and taxonomy to organize and extract relevant results from the literature. There are some classifications of studies addressing self-adaptive properties in the context of parallel stream processing. This study intends to unify these existing classifications into a taxonomy. Moreover, we propose new categories to improve the categorization of proposed solutions and answer our research questions. Figure 3.1 depicts the structure of the proposed taxonomy that has three main classification groups described in the next sections: self-adaptiveness (Section 3.2.1), parallelism properties (Section 3.2.2), and validation of the proposed self-adaptive solutions (Section 3.2.3).

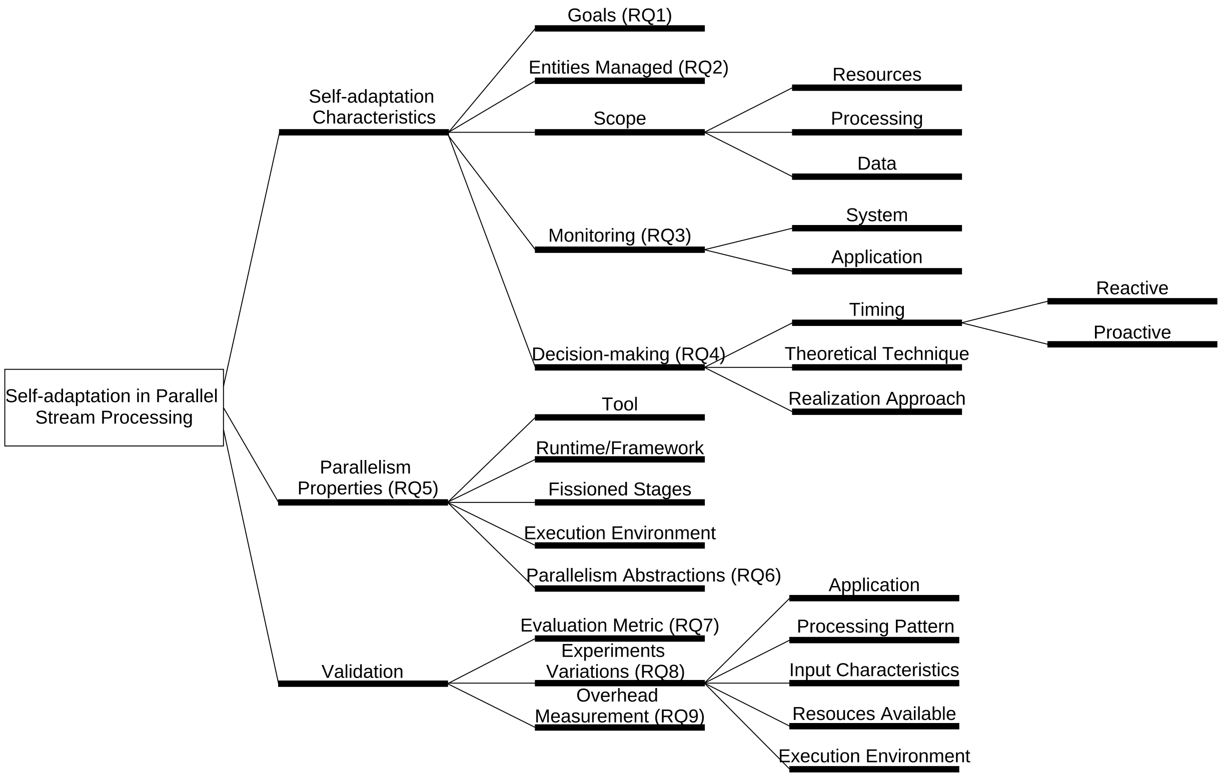


Figure 3.1: Proposed taxonomy for self-adaptiveness in stream processing.

3.2.1 Self-adaptation categories

The categorization presented here concerns self-adaptiveness foundations and we divided the self-adaptation characteristics into three major properties described below.

Category of adaptation

One category is the adaptation goals (RQ1) and the entities managed in adaptation actions (RQ2) for pursuing a given goal. Regarding where the adaptation is performed, [100] provides a taxonomy that helps in our scenario. Here, the scope of adaptation considers three adaptation classes: resources, data, and system processing. *Resources adaptation* concerns the modifications applied only to the physical execution environment that is agnostic to the application/system. For instance, adapting the amount of Central Processing Units (CPUs) computing resources available while a given application is running. *Data adaptation* is related to modifying the data stream/items that are the application is processing, such as adapting the size of data batches. The *processing adaptation* category covers adaptations performed in the application processing entities. Examples of this category are changes applied in the application processes/threads (parallelism degree) and task scheduling.

Monitoring (abbreviated: *Mon.*)

Considering that the managing system needs updated information and statistics to self-adapt the executions, monitoring is a potential way of feeding the system to measure/evaluate the execution. In this work, we divide monitoring into system and application level, which is a categorization related to RQ3. System-level relates to monitoring the operating system and hardware (the environment). Application monitoring relates to the collection of information from the application and its runtime systems, such as application performance metrics or indicators.

Decision-making

The managing systems decide to perform adaptation actions, which enables a managed system to achieve self-adaptiveness. The alternative of applying an action to change a given entity considering a goal can make a system and the executions more intelligent. RQ4 considers how is the decision-making performed. AS explained in [103], the decision may have different timing, reactive or proactive. A reactive decision responds to a specific scenario, while the proactive one attempts to anticipate a given occurrence. Additional categories that we created for evaluating the decision-making regards the *theoretical technique* and *realization approach* used. Considering that designing and implementing support for adaptation to real-world stream processing applications tends to be complex, there are several theoretical foundations available [56, 111]. Thus, the category *theoretical technique* attempts to survey which theories are being used for designing the self-adaptive strategy. Moreover, the *realization approach* category is the core of the decision-making that is represented by a decision algorithm and control mechanisms.

3.2.2 Categorization of parallelism properties

Considering that parallelism is relevant for stream processing applications and the RQ5, we also propose a categorization of the properties concerning the parallelism exploitation in self-adaptive approaches.

Tool

This category concerns which existing tools are supporting self-adaptive parallelism in stream processing. It is also important to note that the existing runtime or frameworks can be extended by approaches proposing new tools that enable self-adaptiveness.

Runtime library or framework

A relevant aspect to survey the literature relates to which existing parallel runtime libraries or frameworks have features supporting self-adaptation. Our SLR attempts to list all parallel tools/frameworks that support self-adaptation in the context of stream processing.

Parallel stages

Figure 2.1 shows a Farm with one parallel (AKA fissioned, replicated) stage, which represents a graph composition or application topology [132]. Such a graph composition is suitable for embarrassingly parallel computation that easily executes in parallel. However, there is a trend in software applications to be more complex with several functions that can be decomposed using parallel patterns, resulting in complex (AKA robust) graph compositions. We consider complex compositions the ones comprising more than one fissioned stage or a combination of patterns (e.g., a Pipeline with Farms). The property of fissioned parallel stages aims at surveying how many replicated stages are supported by the existing self-adaptive approaches/tools. Manage a single fissioned parallel stage tends to be less complex. On the other hand, complex compositions with multiple fissioned parallel stages tend to require additional control mechanisms for managing the execution at run-time.

Execution environment

The execution environment category inside the group parallelism properties considers the characteristics of the execution architecture of a given stream processing application, which is a very relevant aspect for parallelism. A single machine environment can be composed of a multi-core or heterogeneous architecture (co-processors, GPU, FPGA). The environment may also be a cluster with a distributed shared-nothing memory subsystem. The target environment used tends to be related to performance requirements. Some applications running in a multi-core machine can sustain a suitable QoS while other applications may require several machines for achieving QoS.

Parallelism abstractions

This category evaluates if the self-adaptation employed is able to optimize stream processing aspects for providing abstractions to users/programmers. Examples of abstraction can be a parallelism abstraction (Section 2.1.2), attempting to simplify the execution, or self-optimize specific concerns. Providing abstraction can be an advanced goal of self-adaptiveness because it tends to be difficult to facilitate the application programmers' tasks by transparently managing the systems. Additionally, considering stream processing and parallelism abstractions, RQ6 attempts to conceptually highlight whether parallelism abstractions are being considered for self-adaptive stream processing.

3.2.3 Self-adaptiveness validation

A relevant aspect related to RQ7 is how the literature approaches available are validated. Although self-adaptivity can provide several advantages, we expect the approaches to be properly evaluated for maintaining performance and execution safety. In [100], evaluation metrics were extracted from the papers found in their context. In this work, we intend to extract relevant information and evaluate the current validation state of the approaches. Hence, we propose a categorization to assess the characteristics and variations used in the validation. Below we describe the categories and properties covered by the proposed categorization.

Evaluation metric

Considering that the self-adaptive solutions are expected to be extensively evaluated, this category intends to survey the evaluations metrics. Examples of metrics to be considered are performance, energy, and resource consumption. For instance, the performance metric can be utilized to measure the effectiveness of the self-adaptive solutions, where examples of relevant metrics are execution time, throughput (how many stream items are computed per second), and latency (time taken to compute the stream items).

Experiments variation

Experiments variation is a relevant concern related to RQ8. Considering that a representative and broad testbed is expected to be used for evaluating the proposed solution under different scenarios, this category is divided into subcategories for better assessing the approaches:

- *Application* considers whether different applications were used for characterizing and evaluating the behavior of the proposed approaches. The self-adaptive managing systems are supported in applications that are executed for evaluation purposes.
- *Processing pattern* considers whether the tested applications present changes in behavior or processing characteristics. In our context, we consider as relevant examples of different processing patterns if the applications have a different performance trend with respect to the parallelism (degree, level, grain, scheduling, placement) used or processing characteristics (e.g., CPU bound, memory-bound, Input/Output (I/O) bound).
- *Input characteristics* evaluate if different input types or rates were considered for the applications used.
- *Resources available* considers if the number of computational resources available for the running applications changed during their executions. This is a concern of applications running in modern dynamic environments.
- *Execution environment* is a variation related to evaluating the approaches in more than one scenario/environment. An example can be executing the application and characterizing the self-adaptive decision-making in different computational architectures (multi-core, GPU, Field-Programmable Gate Array (FPGA), Clusters) or paradigms (Cloud, Fog).

Overhead measurement

This category is related to the overhead (RQ9) that can be caused by self-adaptation, where it attempts to survey what is the impact on real-world applications. For instance, self-adaptation demands additional mechanisms and processing parts, which consume more computational resources. Moreover, being adaptive can impact positively or negatively in the performance of the applications. The overhead measurement intends to evaluate whether the related literature considers the drawbacks that self-adaptiveness can cause in some cases.

3.3 Result analysis and discussion

In this section, we discuss the results found in the literature with respect to the categorizations.

3.3.1 Studies overview and their execution environments

This section presents an overview of the literature approaches that are separated according to their execution environment and ordered by publication year. The execution environment is classified according to the description provided in each study. The execution environments are classified in multi-cores, cluster, cloud, and heterogeneous environments with multi-core and accelerators (GPUs or co-processors). It is important to note that cloud environments support multi-core environments (a single instance) or a virtual cluster with multiple instances. Consequently, a parallel application running in a cloud environment can be still running in a multi-core or cluster. However, as we focus on the environment instead of the programming model, studies that are executed in clouds are only presented in the cloud category. The same organization is applied to studies showed in the accelerators section because GPUs and co-processors are added to multi-core machines that may or may not be part of a cluster.

Multi-cores

The first approach proposing self-adaptiveness for stream processing applications was found in Schneider *et al.* [106]. They proposed elasticity as an adaptation that extended SPADE [43], a language and compiler for developing stream processing applications. The solution was implemented using a dispatcher thread, which manages the stream system (load distribution, queues). The dispatcher thread is characterized as a component that manages the number of active threads. A key component of the system is the alarm thread which runs periodically for monitoring the system.

The work of Choi *et al.* [22] proposed a solution for detecting performance bottlenecks with a performance model and adaptation algorithm. The proposed solution was validated in different scenarios comparing its performance to the related solution from [106]. The validation considered a different number of bottlenecks and a scenario with changes regarding the availability of computing resources. The proposed solution outperformed the related approach in terms of performance. It is also important to note that the solution of [22] assumes a stable workload, where the application load has only one processing phase. Consequently, this approach only configures streams programs once.

Tang and Gedik [115] proposed an autopipelining solution for stream processing that transparently attempts to improve the application efficiency and throughput. It is important to note that the approach concerns a scenario where each thread executes a pipeline. Autopipelining was implemented with a runtime profiling that performs optimizations concerning the number of threads aiming to overcome application bottlenecks. The solution was evaluated on real-world applications with different tuple sizes.

Selva *et al.* [110] provided runtime adaptation for streaming languages. The StreamIt language was extended in order to allow the programmer to specify the desired throughput. The adaptation is provided by the runtime system that controls the environment. Moreover, an application and system monitor was implemented to check the throughput and system bottlenecks. The adaptation concerns the migration of actor (stage) according to the load on specific CPU cores.

Proposing the term “self-aware" to be applied in stream processing, Su *et al.* [113] introduced StreamAware. It is a programming model with adaptive parts targeting dynamic environments. The aim was to allow the applications to automatically adjust at run-time. The adaptivity implemented was based on the Monitor-Analyze-Plan-Execute-Knowledge (MAPE-K) closed-loop. The adaptive parts proposed adjusts the runtime in three aspects: integration and removing of idle nodes and adjusting data parallelism. However, the mechanisms and policies used for adapting the programs at run-time are not described. The adaptive method is evaluated and validated using the PARSEC benchmark suite.

Matteis and Mencagli [31] presented elastic properties for data stream processing to improve performance and energy efficiency (number of cores and frequency). They argue that using the maximum amount of resources is expensive and inefficient. Therefore, they proposed elasticity as a solution for efficient usage according to QoS requirements. The latency was managed using a Model Predictive Control (MPC) method while the energy consumption was reduced with Dynamic Voltage and Frequency Scaling (DVFS) techniques. The solution was validated in a high-frequency trading application as well as compared to related solutions. Matteis and Mencagli [33] extended [31] with strategies for energy-aware on data stream processing.

Additional parallelization techniques for stream processing on multi-cores were presented by Gad *et al.* [41]. There, a new Domain-Specific Language (DSL) was proposed. The self-adaptive part encompasses a mechanism providing data distribution optimizations among the CPU cores, where functions to be computed can be moved from processing elements. The solution was tested on “pleasingly" parallel tasks, which are easily parallelized and showed promising results.

Karavadara *et al.* [65] proposed a framework for stream processing on embedded systems with many-cores processors that was implemented in the library called S-Net. The adaptive part is performed by the control system that uses DVFS for reducing, at run-time, the power consumption. The DVFS proposed solution was evaluated showing significant energy savings.

Sahin and Gedik [105] proposed C-Stream, which is a stream processing engine for customizable and elastic executions. C-Stream empowers users with the option to set SLA, such as high throughput or low latency. Importantly, the solution has an adaptation module that adjusts the number of threads and the level of parallelism. The main goal of adaptation was to detect bottlenecks and increase performance. The solution was validated with real-world applications and compared to Storm, where C-Stream achieved a good performance.

Schneider and Wu [108] presented an elastic scheduler for the IBM Streams system. The work tackled the problem of determining the best number of threads in stream processing. Particularly, they proposed an elastic algorithm that finds the number of threads that yields the best performance without user inputs. The solution was tested on different machines showing a good performance. Later, Ni *et al.* [97] provided an elastic threading model for IBM Streams.

De Sensi *et al.* [35, 36] proposed Nornir, a framework applicable for stream processing. Nornir aims to predict performance and power consumption using linear regression as a learning technique. Their goal was to reduce power consumption with "acceptable" performance losses. Nornir enables the application to change different knobs at run-time. The Nornir system was aimed to satisfy power consumption or performance bounds, which have to be defined by the user. It then triggers actions when it detects changes in the input rate or application. Nornir interacts with the Operating System (OS) and with FastFlow's runtime. Importantly, Nornir also includes a flexible framework that can be used for designing custom decision-making strategies as well as non-intrusive instrumentation of executions.

The impact of parallelism on the latency of stream processing items was addressed on Vogel *et al.* [128]. There was showed that although more parallel replicas usually increase the throughput of stream processing applications, more replicas also increase the latency. The proposed solution was a compromise between latency and throughput, where the application programmer is expected to provide a latency constraint in the SPar DSL. With the latency constraint, the proposed solution controls the latency by autonomously managing the number of replicas, in such a way that the throughput is increased while the latency constraint is met. The solution was validated with a real-world application showing its effectiveness.

Griebler *et al.* [48] designed SLO attributes for the user to express the target QoS. The solution combines SPar for parallelization and Nornir for power-aware runtime. From the language side, SPar is able to generate parallel code from annotations added by the user on a sequential code. Nornir provides changes in terms of the number of cores and their frequency for enforcing a given power or performance SLO. The solution was validated with real-world stream processing applications showing its effectiveness. An extended version of this work was provided in [51] for supporting additional SLOs as well as a comparison to related solutions.

The work of Kahveci and Gedik [63] proposed optimization for solving bottlenecks on stream processing applications. They proposed a development API and a runtime library in a system called Joker, which runs in multi-core machines. The solution was validated with different applications and compared to related solutions.

The work of Vogel *et al.* [126] addressed another relevant facet of self-adaptiveness: converge faster to a suitable configuration and minimize instability for reducing the decision-making overhead. The proposed solution improved the decision-making step and the evaluation with different applications and workloads show performance gains as well as lower overhead.

Moreover, Vogel *et al.* [125] provided completely seamless parallelism management for video stream processing applications. Higher parallelism abstractions were achievable thanks to a smarter decision-making strategy that detects workload change for adapting the parallelism degree. The seamless strategy was evaluated with different applications showing its effectiveness.

Clusters

The work of Gulisano *et al.* [52] proposed StreamCloud, a solution to scale data stream processing applications running in cloud environments. StreamCloud (SC) implements parallelism to process queries and distribute the tasks among nodes where a major concern is to handle load balancing through optimized task distribution. Parallelism optimizations are triggered considering the number of active nodes and their CPU loads. The evaluation shows that elasticity and dynamic load balancing optimized resources consumption.

Balkesen *et al.* [8] proposed a framework for actively managing parallelism configurations on SPE. The framework attempts to optimize the parallelism configuration related to the cluster size where the decision considers the input events. Moreover, latency minimization was the main goal, which was optimized with load balancing. Additionally, the framework attempts to predict the future behavior of input streams. The proposed solution was integrated with the Borealis system. Although in the evaluation conducted the latency was significantly higher than static cluster size, the authors claim that the approach is effective by reducing resources consumption.

Yet another approach to stream processing is provided by Heinze *et al.* [54]. It addressed the complexity of determining the right point to increase or decrease the degree of parallelism. The authors investigated issues of elasticity in the data stream to meet requirements for auto-scaling (scaling in or out). They explored the impact of latency in distributed processing. These authors categorize the approaches for auto-scaling applications in five groups: Threshold-based, time series, reinforcement learning, queuing theory, and control theory. They argue that time series is not feasible for adaptivity on stream processing applications, because it considers historical data and the stream load is unpredictable. The queuing model was also excluded due to its limited adaptivity.

The remaining classes were then tested with stream processing applications. Threshold approaches are characterized by the need for the user to set upper and lower bounds with respect to the resource utilization and/or performance. On the other hand, reinforcement learning is based on the system state for taking optimization actions, using a feedback control that monitors the execution and chooses another configuration the next time. Control theory is based on an independent controller that responds fast to input changes based on a feedback loop.

Gedik *et al.* [44] tackled elastic auto-parallelization[107] to locate and parallelize parallel regions. They also address the adaptation of parallelism during the execution. Moreover, these authors argue that the parallelism profitability problem depends on workload changes (variation) and resource availability. They propose an elastic auto-parallelization solution, which adjusts the number of channels in their runtime to achieve high throughput without wasting resources. It is implemented by defining a threshold and a congestion index in order to control the execution regardless of if more parallel channels are required. This approach also monitors the throughput and adapts to increase performance. The experimental evaluation shows that the proposed approach performs adaptations and maintains a fair performance.

Wu and Liu [139] proposed DoDo that is a load-adaptive software layer. DoDo runs on top of cluster nodes and dynamically manages the stages distribution considering the load of the physical servers. DoDO attempts to increase the application throughput by improving the load balancing and resource utilization on the cluster nodes. The experimental evaluation conducted demonstrated performance gains with the proposed solution.

Chatzistergiou and Viglas [17] addressed job reconfiguration approaches for improving tasks distribution in stream processing. Their solution monitors the performance of running applications and reconfigures the jobs placement in case of bottlenecks. The implementation and validation of the proposed solution evinced throughput increases with synthetic and real-world workloads.

Martin *et al.* [83] proposed StreamMine3G, an Event Stream Processing engine that is scalable and elastic. A relevant part is the elasticity support that enables efficient processing under fluctuating workloads. The elasticity concerns the number of nodes used, and the decision algorithm simply monitors the utilization of the node and takes action in case of underloading or overloading.

Zacheilas *et al.* [141] proposed an approach for scaling performance in complex event processing systems. Their solution attempts to predict fluctuations in terms of input rates or latency. The technique used for prediction was Gaussian Process. However, the prediction concerns the application case study and requires previous historical data for making predictions. The challenge is that it tends to be very application specific, is hard to generalize the prediction to other applications and workloads. Moreover, it takes a significant amount of time to train the model which performs dynamic adaptations. The solution was evaluated and showed performance improvements and a fair prediction's accuracy.

Lohrmann *et al.* [76] provided a reactive strategy for guaranteeing latency constraints on stream processing. The proposed solution was validated on Nephele system, where a queuing model estimated latency responses and performed scale actions when necessary. The solution was validated with synthetic and real-world applications. The solution was also compared to the state-of-the-art and achieved performance gains.

Mayer *et al.* [86],[87] proposed a technique to timely adapt the degree of parallelism in CEP. The adaptation is performed aiming to limit the buffer size, where huge buffering is assumed to negatively affect the detection of the events. Consequently, a method was proposed to predict event rates and proactively adapt to the degree of parallelism. The validation of the proposed solution measured the queue lengths under different workloads, where the results were compared to a CPU threshold approach. The proposed solution showed to work well in the CEP scenario. However, it was not measured the impact of buffering in common performance metrics (throughput, latency, service time) of stream processing applications.

Heinze *et al.* [55] provided an elastic scaling technique focused on the trade-off between monetary cost and latency of stream processing. An online parameter optimizer was proposed for finding a scaling configuration that yields less cost. Moreover, the parameter optimizer attempts to facilitate usability by requiring the user to set only the expected service rate instead of several error-prone parameters. The evaluation of the solution evinced that it reduced costs and maintained a reasonable quality of services.

Adaptive fault tolerance for stream processing was addressed by Martin *et al.* [84]. The proposed solution dynamically changes the fault tolerance scheme (*e.g.,* active replication, active or passive standby, passive replication) aiming to facilitate for users that are not required to manually set the best replication scheme. Consequently, the users are expected only to provide high-level constraints, such as recovery time, gap, or precision. This is provided by the adaptive controller that sets and changes to the most suitable scheme considering the user constraints, workload, and the lowest resource consumption. The validation showed a low resources consumption overhead and without losses in recovery time.

Zhang *et al.* [144] aimed to minimize latency on stream processing by adapting the batch and block size. They proposed DyBBS, which uses a heuristic to learn and set the batch configuration according to the current workload. The solution was validated compared to related solutions showing that it reduces the latency.

Meet latency requirements on stream processing was also a goal of Liu *et al.* [74]. There, the latency is controlled from the perspective of resource management, scheduling, and load balancing. Presuming load fluctuations, the authors proposed scheduling solutions for tasks redistribution aiming at reducing the latency and achieving more stability by avoiding overloading specific servers. The evaluation evinced the performance gains of the proposed solution.

Gil-Costa *et al.* [24] provided an elastic strategy that balances the load according to the processing time and the load of each specific node. Using a monitor and manager, the overloaded nodes are released while new nodes process the bottleneck stages. The evaluation showed a good performance considering the throughput of applications.

Li *et al.* [72] presented an approach for elastic scaling in distributed stream processing. The proposed techniques for scheduling stream processing with batch processing. The stream processing part was implemented on Storm. A very relevant aspect of this approach is that they highlight downtimes that occur in Storm when scaling actions are taken. The downtime occurs because Storms can only scale by reconfiguring and restarting the application that results in shutting down all active operators. Consequently, the data stored in operator's memory is lost which results in downtime that lasts from 20 to 30 seconds. Previous efforts working with adaptivity in Storm did not consider this issue. Consequently, the authors proposed a solution that saves the state of the operators for minimizing downtime when scaling the application. In order to decide when to perform elasticity actions, the proposed solution includes a monitor for congestion detection attempting to avoid application bottlenecks. The solution was validated showing gains in terms of latency and throughput.

Kombi *et al.* [69] provided an approach that monitors the congestion attempting to improve the performance and resources efficiency. The continuous dynamic adaptation was proposed as a solution. The solution encompasses an estimation of the future input size with time series analysis algorithms. This estimation has the potential to forecast the load for the near future. Additionally, they proposed a technique that evaluates and adapts the degree of parallelism. The solution was validated emphasizing improvements in latency and resources consumption.

De Matteis and Mencagli [32] extended [31] and [33] for supporting execution on distributed cluster environments. They proposed a control-theoretic strategy for elastic scaling in data stream processing. The solution targets latency sensitive applications by providing a Model Predictive Control (MPC). The solution was experimentally evaluated in terms of latency and reconfigurations.

Cardellini *et al.* [12] extended [14] by proposing and evaluating optimizations regarding replication and placement of operators for stream processing. The model supports multiple QoS metrics (response time, inter-node traffic, cost, availability) and attempts to find a balance between those multiple metrics. Moreover, Cardellini *et al.* [15] proposed an elastic distributed Framework for stream processing applications. The important self-adaptive part concerns tasks migration and adaptation in the degree of parallelism. A threshold-based policy and two Reinforcement Learning (RL) policies were implemented for adapting the configurations. One RL policy is Q-learning that uses a cost function for learning from samples and estimating optimal actions. The second policy is a model-based RL algorithm that exploits different system knowledge for estimating an approximate configuration.

It is important to note that the policy proposed by Cardellini *et al.* [12] requires configurations and parameters from the user, such as cost weights, scaling thresholds, and response time. Such parameter definitions may be difficult and error-prone for application programmers. The proposed policies were implemented in Storm, which is worth mentioning the known downtime on reconfigurations [13, 72]. The validation covered a real-world application, where it was noted limitations in the resources threshold policy and that the Q-learning policy requires too much time to learn and find a suitable configuration. The model-based RL policy was the best performing solution in the evaluation conducted.

Cheng *et al.* [19] provided a new scheduler for Spark Streaming. This new scheduler is adaptive in a way that dynamically schedules and adapts the parallel jobs. Adaptivity makes it possible to reconfigure the execution with the implemented solution in such a way that the level of parallelism (number of concurrent jobs) and the sharing/availability of resources. The approach was evaluated on a security event application showing improvements in terms of performance and energy efficiency. [19] was extended in Cheng *et al.* [20] for supporting dynamic batching.

Lombardi *et al.* [77] proposed Elysium, an approach for elasticity on stream processing. The novelty of Elysium concerns evaluating, estimating, and managing elasticity in two independent dimensions: application parallelism and resources. The solution also presents a resource estimator that computed the expected resource utilization, which proactively runs elasticity actions. The solution was validated with real-world applications showing gains in terms of elasticity adaptations.

Kalavri *et al.* [64] addressed a very important aspect of online adaptiveness: find fast and accurate new configurations. They proposed the DS2 controller that has a performance model which estimates the true processing capacity of stages and converges faster, accurately, and in a stable mode to a new parallelism configuration. DS2 was implemented as a decoupled decision-making that was implemented in two frameworks, Flink and Timely. The experimental results show a fast convergence and a low monitoring overhead.

Wang *et al.* [135] proposed Elasticutor for faster elasticity on data stream processing. The solution encompasses elastic executors (operators processes) that optimizes the load balancing and a dynamic scheduler that elastically manages computational resources. Elasticutor was implemented on Storm and tested with benchmarks and real-world applications, where it demonstrated significant performance improvements.

Bartnik *et al.* [9] addressed aspects related to elasticity and fault tolerance of stateful stream processing applications in Apache Flink. Their solution was a protocol that provides the alternative to adapt the execution at run-time in three aspects: migration of operators, adding new operators, and changing the functions computed by operators. The proposed solution was evaluated with benchmarks running on a cluster, emphasizing the straightforward result that, in distributed stateful stream processing, the adaptation overhead is impacted by the job's state size that has to be migrated.

Kombi *et al.* [70] presented DABS-Storm for elastic stream processing on Storm by dynamically adapting the parallelism degree. Importantly for generalization purposes, they argued that the proposed solution could be implemented is other related solutions.

Talebi *et al.* [114] proposed elasticity for CEP running in cluster environments. The proposed solution called ACEP adapts the degree of parallelism is using a cost model for improving the load balance. ACEP was evaluated showing its effectiveness.

Clouds

Das *et al.* [29] explored the impact of batch size on the latency of stream processing applications. Based on an understanding of the relation between throughput and latency, they proposed a control algorithm that autonomously adapts the batch size. The experimental validation of the proposed solution evinced that it seamlessly optimizes the latency.

Tudoran *et al.* [121] tackled inter network transfer overhead by providing an adaptive transfer strategy that self-optimizes the batch sizes. The strategy sets the batch size to the value that considers the instant environment condition, such as transfer rates, aiming to reduce the latency. The proposed solution was validated on a real-world cloud environment showing its effectiveness.

Heinze *et al.* [53] presented an approach addressing latency aspects on stream processing. There, it was stated that scaling decisions performed for optimizing resources utilization tend to result in latency violations. Consequently, they proposed a solution attempting to minimize latency violation by estimating latency spikes before running scaling decisions. The adaptation regarding the scaling strategy was implemented and compared to related solutions, where their solution evinced fewer latency violations.

A game-theoretic controller was proposed by Mencagli [89] as a control strategy for distributed stream processing. In this model, each fissioned parallel stage is managed with a local controller that sets the internal degree of parallelism, while there are several global controllers. In order to settle globally, two strategies were proposed, a non-cooperative and an incentive-based. In the non-cooperative, each controller only considers its local configuration. On other hand, the incentive-based proposes a controller that senses global conditions for improving the system globally, which tends to improve in terms of performance and efficiency. The theoretical solution was validated with simulations of a mobile cloud computing platform.

An application profiler for stream processing was proposed by Liu *et al.* [73]. Profiling was used to identify potential bottlenecks and applying self-adaptivity in the parallelism configuration to improve the performance. The relation between resources and application performance was handled by the profiler with resource provision at an optimization level, attempting to further improve the performance with the combination of provisioning, scheduling, and placement. The solution was evaluated with real-world applications, where it achieved higher performance in comparison to related solutions.

Adaptivity was considered by Floratou *et al.* [40] for real-time stream processing analytics, where the notion of self-regulation in Twitter's Heron framework was introduced with the proposed system called Dhalion. In this solution, the user sets a target throughput and Dhalion transparently configures the number of processes and cloud instances for achieving the user goal. Dhalion was validated with real-world applications, showing that the system can dynamically self-regulate to meet SLOs.

Venkataraman *et al.* [122] observed that stream processing systems need to constantly adapt to failures and workload fluctuations. They proposed Drizzle, a solution for reducing the overhead in case of a recovery adaptation that maintains a QoS. Compared to related solutions, Drizzle achieved lower latencies and significantly faster recovery adaptability.

Tolosana-Calasanz *et al.* [10] proposed an autonomic controller based on queueing theory for managing resources. The controller manages and controls the number of VMs allocated for the running stream processing applications. The controller also monitors the items queuing time. The approach was evaluated showing its effectiveness.

Mai *et al.* [80] proposed Chi, a control-plane that dynamically adapts stream processing applications. The goal can be seen as facilitating the usage and efficiency of users. With Chi, users can simply express SLOs (latency, throughput) while the control plane enforces the goal by using feedback for adjusting the system parameters/configurations. The adaptivity concerns mainly the number of nodes used. The solution was validated in comparison to related solutions showing significant performance gains.

The very relevant problem of downtime on reconfigurations of distributed stream processing was addressed in Rajadurai *et al.* [101]. There, Gloss was proposed as a solution for avoiding downtime on Synchronous Data Flow (SDF) applications. Such an application scenario is arguably a narrow part of stream processing where parallel executions are stable, static, synchronous, and deterministic. Consequently, SDF is a solution only for specific regular applications. However, Gloss can be viewed as a quite elaborated solution. With Gloss, live reconfiguration and optimizations are possible at run-time. In order to avoid downtime, Gloss employs input duplication and concurrent execution of new and old graph topologies during reconfiguration. The solution was validated on real-world cloud environments showing to be effective for avoiding application downtime.

Fardbastani *et al.* [39] proposed adaptive load balancing for complex event processing. The solution periodically collects the load of each node and decides if the nodes are balanced or not. Considering the decision based on load, the solution redistributes the tasks. The approach was evaluated emphasizing an increase in terms of performance.

Marangozova-Martin *et al.* [81] provided a strategy for elastic management of resources on Storm in cloud environments. This approach attempts to reduce latency while consuming minimum resources. The solution is multi-level in the sense that covers application (performance) and environment (number of Virtual Machine (VM)). The solution was validated in a real-world stream processing application.

Lombardi *et al.* [78] proposed PASCAL for automatic scaling of distributed applications. PASCAL is combined with a performance model attempting to predict incoming workloads and a system for estimating the number of computing resources to be provisioned. The performance model was evaluated with synthetic and real-world traces that generate the input load (workload). However, it is difficult to estimate how representative and generalizable are the workloads used for other stream processing applications. Importantly for stream processing, different workloads were tested using Storm. The solution was validated in a cloud environment.

Abdelhamid *et al.* [1] showed Prompt, a solution for the dynamic data partition. In this scenario, the data is partitioned in micro-batches and the dynamism concerns the size of the batches. Noteworthy, adaptive parallelism properties are also covered with an elastic part that dynamically adapts the parallelism degree in case of workload changes. The solution was validated with different applications and workloads showing its effectiveness.

Russo *et al.* [104] addressed the very relevant problem of heterogeneous computing infrastructures for running stream processing applications. Most of the works considering elasticity for distributed applications assume that all machines and clusters will be homogeneous in such a way that each machine provides the same performance/profitability. However, this assumption is not true anymore in most cases because the environments and machines are becoming more dynamic and irregular. The authors use Markov Decision Process (MDP) for controlling elasticity and Reinforcement Learning acRL algorithms for optimizing the definition of parameters. The solution was validated in cloud environments in terms of performance and cost.

Accelerators

Lars *et al.* [109] proposed AdaPNet, an approach aiming at maximizing the performance of streaming applications by adapting the degree of parallelism. In AdaPNet, parallelism is adapted for responding to changes in terms of resources availability. The performance and overhead of the proposed solution were evaluated, where the overhead considers the time taken to change the application at run-time as well as the related memory usage.

Vilches *et al.* [123] addressed aspects related to efficiently running stream processing applications on heterogeneous architectures composed of CPU and GPUs. They proposed a framework that at run-time finds the best mapping of processing stages on CPU cores, GPUs, or the combination of them. The framework collects runtime statistics for performing the decision-making, where the goal was throughput, energy, or a trade-off between both goals. The solution was validated on different architectures showing performance improvements.

Mencagli *et al.* [91] proposed Elastic-PPQ, a layered autonomic system for dynamic data stream processing. The layered architecture comprises two adaptation levels. One is a load balancing mechanism for fast variations using the control-theoretic approach. Moreover, a relevant part regarding the parallelism level for slower variations uses Fuzzy Logic. The solution was evaluated showing effectiveness in terms of adaptability and performance.

Matteis *et al.* [34] proposed Gasser, a system for running windowed stream processing applications on hybrid architectures composed by CPU and GPUs, where computations are offloaded to GPUs. Importantly, the proposed solution has an adaptive feedback part for auto-tuning, which tests and tries several configurations for finding the one that achieves the best performance. The configuration considers the different batch sizes and degrees of parallelism. Gasser was validated with data stream processing applications and compared to related solutions. An arguable shortcoming of Gasser is that it only performs adaptation once, which is only suitable for stable and regular workloads.

Stein *et al.* [112] provided techniques for dynamically adapting the batches size of the specific class of stream processing applications that perform data compression. Moreover, the work targets particularly applications suitable for running on GPUs, where a different decision-making algorithm was proposed. The solution was evaluated considering latency as the target and the suitability of algorithms varies according to the specific workloads.

3.3.2 Self-adaptation classification

Table 3.2 [5] characterizes self-adaptation and parallelism properties of the approaches described in the previous section. Table 3.2 has the following abbreviations: Experimental environment (EE), migration (mig.), propagation (propag.), theoretical technique (theoret. tech.), system (sys.), distribution (distr.), adaptation (adapt.), algorithm

[5]In reference [127], the interested reader can view the content of this table in another format.

(algo.), frequency (freq.), utilization (util.), regression (regr.), theory (th.), reinforcement learning (reinf. learn.), placement (placem.), Optimal DSP Replication and Placement (ODRP), Model Predictive Control (MPC).

It is important to note that here the approaches are separated according to their execution environment and ordered by publication year. Regarding the RQ1 and the categorization proposed in Section 3.2.1, in column “Goal" of Table 3.2 it is possible to note that there are several purposes of applying self-adaptiveness to stream processing. Thus, we propose the following catalog and definitions to organize the self-adaptation goals:

- **Throughput** (*maximize*): is the number of stream items/tasks processed in a given time interval. A high throughput tends to be a goal. Additionally, there are some studies that intend to allow the user to define throughput as a performance goal, which we refer to as *target throughput*.

- **Latency** (*minimize*): Latency in stream processing is a performance metric that refers to the time taken to process stream items/tasks. A lower latency tends to be better, which can be set as a *constraint* in some approaches.

- **Resources usage** (*maximize, limit, optimize*): In the context of this study, we refer to resources as computational power available for processing computations. Other terms like *resource utilization*, *system utilization* (abbreviated *sys. util.*) are considered synonymous and used interchangeably in this study. Some works attempt to utilize resources as much as possible. On the other hand, some approaches try to limit how much resources a given application uses in order to avoid interference in multi-tenant environments or for limiting energy consumption. *Energy* consumption is also a goal that we categorize as related to resources. There are also goals for optimizing (abbreviated *opt.*) the usage of resources in such a way that a performance goal is met with minimum resources, which is also related to limiting resource usage. This aforementioned resource optimization characterizes the goal of computing *efficiency*, which is also pursued by some approaches.

- **Buffering** (*limit*, *minimize)*: we refer to buffering as queueing stream items before processing them. The term *queue size* tends to mean a similar aspect in the stream processing applications. Limiting the buffer sizes impacts the performance of stream processing mainly in terms of latency, which is a potential optimization.

- **Cost** (*reduce*): On *pay-per-use* paradigms like cloud computing, the resources usage impacts directly the cost. Consequently, there are efforts that aim to reduce the cost of stream processing by self-adapting executions considering resource usage.

- **Fault** (*tolerance*): This aspect concerns the running stream processing applications, where avoiding faults is relevant. There are some approaches that autonomously adapt to the fault tolerance scheme for optimizing executions. Moreover, *avoid downtime* when running a stream processing application is a facet of fault tolerance considered by some approaches.

- **Load** (*balance*): Balancing the load of stream processing is an objective pursued for optimizing executions, which can achieve gains in terms of performance or resources. Some approaches explicitly mention load balance as a goal, where it is assumed that optimizing the load balance will provide gains to applications.

Throughput improvement of stream processing applications was the goal of 31 studies. Latency Reduction was mentioned as a goal in 29 studies. Noteworthy, some of these studies attempt to reach a combination of optimal throughput and latency. In fact, a significant part of approaches has more than a single objective, often targeting a trade-off between different metrics.

Considering the motivation for a taxonomy discussed in Section 3.2, in this survey, it was possible to extend the revision for covering unique approaches focusing on additional applicabilities of self-adaptiveness. For instance, studies with new goals were found, such as minimize energy consumption [35, 36, 65, 123, 51], avoid downtime [101], and tolerate faults [84]. We believe that this is relevant to enable self-adaptiveness to be evaluated in the future for supporting new applicabilities.

Approach	**Goal**	**Adaptation action/Entities**	**Theoret. tech.**	**Realization Approach**	**Tool**	**Framework**	**EE**
Schneider [106]	Throughput	Parallelism degree	/	Adapt. algo.	Algorithm	SPADE-Sys. S	
Choi [22]	Throughput	Parallelism degree	Delay propag.	Bottleneck detection	Algorithm	/	
Tang [115]	Throughput	Parallelism degree	/	Optimization algo.	Algorithm	System S	
Selva [110]	Throughput	Task mig.	/	Adapt. algo.	Algorithm	StreamIt	
Su [113]	Resource util.	Parallelism degree	Feedback loop	Adapt. algo.	StreamAware	StreamMDE	
Matteis [31]	Energy, latency	Parallelism degree, cores freq.	MPC Queuing	Runtime mechanisms	Strategy	FastFlow	
Gad [41]	Throughput	data distr.	Feedback loop	Adapt. algo.	java DSL	/	Multi-cores
Karavadara [65]	Energy	Cores freq.	/	DVFS strategy	Algorithm	S-Net	
Gedik [105]	Latency,throughput	Parallelism degree	/	Bottleneck det.	Algorithm	C-Stream	
Schneider [108]	Throughput	Parallelism degree	/	Adapt. algo.	Scheduler	SPL	
De Sensi [36]	Efficiency	Parallelism degree, cores freq.	/	Linear regression	Nornir	FastFlow	
Vogel [128]	Latency	Parallelism degree	Feedback loop	Adapt. algo.	Algorithm	SPar	
Griebler [51]	Performance, energy	Parallelism degree, cores freq.	Feedback loop	Linear Regression/Adapt. algo.	SPar and Nornir	FastFlow	
Kahveci [63]	Throughput	Parallelism degree	/	Adapt. algo.	Algorithm	Joker	
Vogel [126]	Throughput	Parallelism degree	Feedback loop	Adapt. algo.	Algorithm	SPar	
Vogel [125]	Throughput	Parallelism degree	Feedback loop	Adapt. algo.	Algorithm	SPar	
Gulisano [52]	Throughput	Tasks distr., nodes	/	Load thresholds	StreamCloud	Borealis	
Balkesen [8]	Latency	Tasks distr., nodes	/	Adapt. algo.	Algorithm	Borealis	
Heinze [54]	Resource util.,latency	Task mig.	/	Threshold, R.L.	Algorithm	FUGU	Clusters
Gedik [44]	Throughput	Parallelism degree	/	Control algo.	algorithm	SPL IBM S.	
Wu [139]	Throughput	Task distr.	/	Adapt. algo.	DoDo	S4	
Chatzistergiou [17]	Throughput	Tasks distr.	/	Group-aware	Algorithm	Storm	
Martin [83]	Throughput	Nodes, tasks mig.	/	Threshold algo.	Algorithm	StreamMine3G	

Table 3.2 continued from previous page

Approach	**Goal**	**Adaptation action/Entities**	**Theoret. tech.**	**Realization Approach**	**Tool**	**Framework**	**EE**
Zacheilas [141]	Resource util.,latency	Parallelism degree	/	Short path algo.	Esper	Storm	
Lohrmann [76]	Latency	Parallelism degree	Queuing th.	Scaling policy	Algorithm	Nephele	
Mayer [86]	Limit buffering	Parallelism degree	Queuing th.	Adapt. algo.	Algorithm	/	
Mayer [87]	Limit buffering	Parallelism degree	Queuing th.	Adapt. algo.	Algorithm	/	
Heinze [55]	Reduce cost	Nodes	/	Threshold algo.	Algorithm	FUGU	
Martin [84]	Fault tolerance	Tolerance scheme	Controller	Checkpoint algo.	Algorithm	StreamMine3G	
Zhang [144]	Latency	Batch and block size	/	Isotonic Regr.	DyBBS	Spark	
Liu[74]	Latency	Tasks distr.	/	Adapt. algo.	Algorithm	Storm	
Gil [24]	Throughput	Nodes, tasks distr.	/	Adapt. algo	Algorithm	S4	
Li [72]	Latency,throughput	Nodes	/	Adapt. algo.	Algorithm	Storm	
Kombi [69]	Latency	Parallelism degree	/	Time service	Autoscale	Storm	
Matteis [32]	Latency	Parallelism degree	Control theory	Adapt. algo.	Algorithm	FastFlow	Clusters
Cardellini [15]	Latency	Parallelism degree, tasks mig.	Feedback loop	Reinf.learn.	Algorithm	Storm	
Cardellini [12]	Response time	Task placem., parallelism degree	/	Additive Weight.	ODRP	Storm	
Cheng [19]	Throughput	Job parallelism level	/	Reinf. learning	A-scheduler	Spark Stream.	
Cheng [20]	Latency,throughput	Batch size, jobs	Feedback loop	Fuzzy, reinf.learn.	A-scheduler	Spark Stream.	
Lombardi [77]	Throughput	Parallelism degree, nodes	/	Adapt. algo.	Elysium	Storm	
Kalavri [64]	Throughput	Parallelism degree	/	Controller	DS2	Flink, Timely	
Wang [135]	Throughput,latency	CPUs, tasks mig.	/	Adapt. algo.	Elasticutor	Storm	
Bartnik [9]	Latency	Parallelism degree	/	Adapt. protocol	Protocol	Flink	
Kombi [70]	Latency	Parallelism degree, Tasks distr.	/	Adapt. algo.	DABS	Storm	
Talebi [114]	Latency	Parallelism degree	MPC Queuing	Adapt. algo.	Algorithm	/	
Das [29]	Latency	Batch size	Feedback loop	Control algo.	Algorithm	Spark Stream.	
Tudoran [121]	Latency	Batch size	/	Simple testing	Algorithm	JetStream	
Heinze [53]	Latency	Task mig.	Cost model	Bin packing	Algorithm	FUGU	
Mencagli [89]	Efficiency	Parallelism degree	Game th.	Adapt. algo.	Algorithm	/	
Liu [73]	Latency,throughput	Tasks distr., IR	Feedback loop	Trial algo.	Algorithm	Storm	
Floratou [40]	Throughput	Operator instances	/	Adapt. algo.	Dhalion	Heron	
Venkataraman [122]	Latency, throughput	Batch size, nodes	/	Adapt. algo	Drizzle	Spark	
Tolosana [10]	Minimize Queues size	Nodes	/	Adapt. algo.	Algorithm	CometCloud	Clouds
Mai [80]	Throughput, latency	Nodes, batch size	Feedback loop	Adapt. algo.	Chi	Flare, Orleans	
Rajadurai [101]	Avoid downtime	Data, grain, nodes	/	Adapt. algo.	Algorithm	StreamJIT	
Fardbastani [39]	Load balance	data and tasks	/	Adapt. algo.	Algorithm	CCEP	
Marangozova [81]	Resource util., latency	Nodes	/	Adapt. algo.	Algorithm	Storm	
Lombardi [78]	Resource util.	Nodes	/	Neural network	PASCAL	Storm	
Abdelhamid [1]	Throughput,latency	Parallelism degree, batch size	/	Adapt. algo.	Prompt	Spark	
Russo [104]	Reduce cost	Parallelism degree	/	Markov, reinf. learn.	/	/	
Schor [109]	Throughput	Parallelism degree	/	Adapt. algo.	AdaPNet	POSIX	
Vilches [123]	Throughput,energy	Stages Mapping	Queuing th.	Adapt. algo.	Algorithm	Intel TBB	
Mencagli [91]	Throughput	Parallelism degree	Feedback loop	Fuzzy logic	Elastic-PPQ	FastFlow	Accelerators
Matteis [34]	Throughput,latency	Parallelism degree, batch	/	Adapt. algo.	Gasser	FastFlow	
Stein [112]	Latency	Batch size	Queuing th.	Adapt. algo.	Algorithms	SPar	

Table 3.2: Self-adaptive properties and tools.

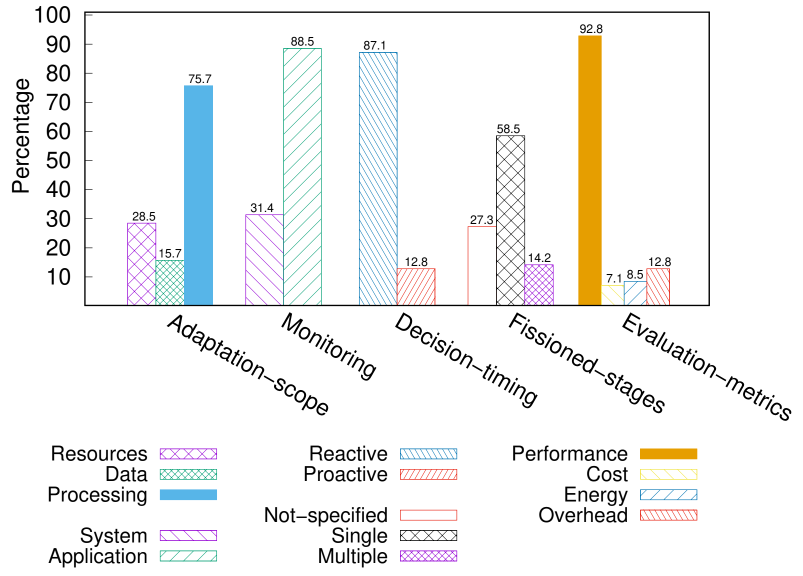


Figure 3.2: Results overview.

3.3.3 Adaptation actions and entities managed

Considering the classification of the scope of adaptation from Section 3.2.1, the 'adaptation-scope' section of Figure 3.2 shows the percentage of approaches that adapted a given scope. These results represent the sum of the percentages exceeds 100% because some approaches adapted more than one item, for instance, adapting the parallelism and the underlying resources. Noteworthy, the majority of approaches are applying adaptations at the application level, specifically the processing and data aspects. In some cases adapting the resources utilized may not be so important because it can be adapted transparently by the abstraction, like in a cloud environment where the applications are usually adapted to reduce the costs and the resources are adapted by the lower infrastructure layer [26].

Regarding the entities controlled in self-adaptive solutions (RQ2), in Table 3.2 it is possible to note a high number of different entities. In the environment, it is possible to manage nodes, CPUs, cores, frequency, and cores mapping. Moreover, in the application, it is possible to manage the placement, scheduling, parallelism degree, batches size, data distribution, etc. Thus, we propose the following catalog and description of entities self-adapted:

- **Parallelism degree** (a.k.a. Degree of parallelism): is a generalization for adaptations at the system/application level related to the number of active processing elements. The parallelism degree is also referred to as the number of threads/processes and the number of replicas. The number of threads/processes is the degree of parallelism in applications running on multi-core or cluster machines. Moreover, the number of replicas is the number of entities processing in a given fissioned parallel stage, where each replica sometimes is related to one thread at the OS level. For the sake of precision, we have a subcategory inside the parallelism degree called *job parallelism level*, which is specific to some scenarios where parallelism is achieved by running multiple simultaneous jobs. The number of concurrent jobs can be self-adapted characterizing an autonomous *job parallelism level*. Other similar terms used in specific contexts for referring to the parallelism degree are *Operator instances* and *Operator Parallelism*.

- **Data**: is a generalization for adaptation concerning the data items. Adaptation of *Batch and block size* concerns the changes performed at run-time in the data granularity. *Data speed* Attempts to control the speed that data is ingested, such as the arrival time. *Data migration* corresponds to changing the place where data items are executed. *Data distribution* relates to changing the way that data items are assigned to computing elements. In data stream processing, a data item tends to be treated as a task. Consequently, *task distribution* is a generalization that we used when the distribution of task changes but it is not defined if each task corresponds to a data item.

- **Cores**: this category encompasses changes at the computing resources level, particularly in the CPUs. The *Number of cores* corresponds to the number of active cores that can be changed in modern processors. Additionally, the *Cores frequency* (abbreviated freq.) can be modified by setting fixed clock rates. *Mapping* refers to policies for mapping software threads to CPUs.

- **Nodes**: is a resource adaptation that refers to the number of physical computing nodes where applications are executed. Usual adaptation actions are adding or releasing nodes.

- **Tolerance scheme**: concerns the technique(s) used for assuring fault tolerance mechanisms in stream processing tools. A tolerance scheme has been implemented in distributed processing scenarios.

Different entities can be managed for pursuing the same metric (throughput, latency). However, the difference lays in how effective each entity is for optimizing the executions. The entities used are also highly related to the approaches' runtime, specific solutions may or may not support adapting a given entity at run-time. For instance, several approaches adapt the parallelism degree at run-time [44, 105, 89, 128] while others have to change the graph topology to adapt to the parallelism degree [101].

In [101] the graph topology is transformed at run-time, but it remains unclear if the graph was transformed because of their runtime constraints or if their targeted advanced optimizations (e.g., efficiency). A runtime constraint can prevent adaptation at run-time. Thus, in [101] the graph topology is transformed because this could be the only way to adapt the executions at run-time. We identify graph topology transformations as a complex adaptation action that has the potential to further optimize stream processing applications in terms of performance and efficiency.

3.3.4 Monitoring on self-adaptation

Information about the executions is necessary to perform adaptation actions, where RQ3 examines which data is used according to the category described n Section 3.2.1. It is possible to note from the section ’Monitoring’ of Figure 3.2 that the majority of approaches are monitoring information from the application level, but there is also a reasonable number of studies that monitor low-level system indicators. Noteworthy, the section ’Monitoring’ of Figure 3.2 shows the percentage of approaches that collected information from the applications and/or from the system, where the sum of the percentages exceeds 100% because some approaches monitored the application and the system.

The monitoring part is also related to the tools/technologies used, as in some cases, it is only possible to monitor the system, or only monitoring the application when system indicators are unreachable. It is also important to note that monitoring actions are performed to some extension in all approaches, adapting the execution without collecting information can be unfeasible. However, in the literature, we lack in-depth discussions of the limitations and advantages of specific monitoring statistics and the potential overhead that the monitoring routines can cause.

3.3.5 Adaptation decisions

Regarding how the decision-making for adaptation is performed (RQ4) described in the proposed classification in Section 3.2.1, the section ’Decision-timing’ of Figure 3.2 shows that the timing of the majority of studies is reactive. The reason behind this it is challenging to predict the load of stream processing applications. Consequently, reactive approaches may eventually violate QoS, but they tend to respond better to fluctuating workloads. References [141, 86, 87, 31, 33, 32, 77, 70, 78] claimed to be proactive.

Another aspect from RQ5 is related to the *theoretical technique* used for decision-making, some utilized are feedback loops from control theory. However, the majority of works in the fourth column of Table 3.2 are filled with a slash “/” meaning that they do not mention the theoretical technique used when designing self-adaptiveness. The use of a slash is one to represent the lack of information in surveys [130]. In the categorization of *Realization approaches*, we catalog algorithms designed for performing the decision-making with different goals, design goals, and entities controlled. These algorithms that are designed in specific solutions are called here *Adaptation algorithm*, where there are some approaches that it is not possible to classify how the realization is performed. Yet regarding the decision-making, there are also more complex ones like heuristics and reinforcement learning, however, a low complexity is expected for a given approach to be computationally feasible on highly dynamic stream processing scenarios. For instance, in [15] the authors concluded that some learning algorithms require a long time to find an optimal decision policy. Some approaches use trial-and-error for finding a configuration with the best performance. However, it tends to be less efficient as several suboptimal or poor configurations lead to performance losses. Additionally, under eventual temporal changes, several trials are required again.

3.3.6 Self-adaptive parallelism in stream processing

Parallelism is commonly used for improving the performance of stream processing applications [30, 103]. RQ5 concerns parallelism aspects of self-adaptive approaches proposed in the classification in Section 3.2.2. In the last column of Table 3.2 we present the parallel library or framework used, 13 works used Storm for providing self-adaptive properties. It is also notable that in some studies it remains unclear in which tool they designed and implemented the self-adaptive entities, the last column of Table 3.2 is filled with a slash “/”. In such cases, it is assumed that self-adaptiveness was implemented on prototypes, which are not necessarily integrated with any existing runtime library/framework.

The number of fissioned parallel stages (described in Section 3.2.2) is another relevant aspect related to RQ5. It is possible to note in the section ’Fissioned-stages’ of Figure 3.2 that the majority of studies focus on managing applications’ graphs topologies with a single fissioned parallel stage. It is also notable that a part of the approaches does not describe well enough the characteristics of the applications used, for the sake of precision these approaches are classified as “Not specified". Importantly, there are also approaches for self-adaptiveness in applications with multiple fissioned parallel stages [106, 22, 115, 44, 76, 89, 77]. However, the application having a multiple stage topology does not mean that such a complex graph is adapted at run-time. The multiple stage topology can be static while other aspects are adapted, such as the batch size [76] or tasks distribution. In this sense, multiple fissioned parallel stage are utilized on specific scenarios, where the mechanism and the strategies’ decision-making are limited in terms of generalization.

In this survey we are interested in specific aspects of parallelism exploitation, focusing particularly on self-adaptive parallelism abstractions. Additional parallelism details can be found in references [103, 30]. RQ6 and the category described in Section 3.2.2

relate to whether the existing approaches focus on providing parallelism abstractions to application programmers. Although nowadays we have frameworks providing high-level programming abstractions for stream processing [30], a limited number of approaches [106, 84, 89, 80, 128, 51, 70, 35, 36] mentioned abstractions as relevant for using/implementing self-adaptation. Considering that it tends to be very complex and time-consuming for application programmers to achieve self-adaptation in their domain-specific applications, we believe that the tools /frameworks should come with ready-to-use abstractions. Examples are offering flags and parameters to enable users to provide hints on their objectives at a higher level. Such objectives could be met by intelligent and autonomous systems that seamlessly use entities to self-adapt executions.

Providing additional parallelism abstractions is one aspect that certainly will require more effort in the future. We expect that the specific characteristics of stream processing would need to be considered for assessing the feasibility of self-adaptive abstractions. Moreover, evaluating if the existing approaches are suitable for parallelism abstractions requires in-depth analysis, and new evaluation methodologies would need to be proposed.

3.3.7 Validation metrics and variations

An often neglected aspect in literature approaches is the comprehensive validation, which is a concern covered by RQ7 and RQ8 and organized in taxonomy in Section 3.2.3. Importantly, the section 'Evaluation-metrics' of Figure 3.2 evinces the percentage of approaches that considered a given metric in the evaluation. Hence, the sum of the percentage exceeds 100% because some approaches considered more than one metric, e.g, covering the performance and the cost. In Figure 3.2, it is notable that performance is the most evaluated aspect of self-adaptive solutions. The approaches [141, 55, 84, 114, 104] considered in the evaluation of the relevance of the cost and the approaches [65, 123, 19, 36, 51] evaluated energy consumption.

Concerning the variations (RQ8) considered for evaluating the solutions, Table 3.3 evinces the results from the proposed taxonomy. A number of works were only tested with more than one application, but it is mostly unclear how different are the processing characteristics of those applications. Only a few studies considered the processing pattern of applications in their validation. The solutions mostly neglect the variability of resources available and environments. Moreover, several studies have no variations, meaning that the solution was validated with a single application, one workload, running in one environment. Evaluate a solution with different applications is relevant, mostly because each application has specific characteristics (processing behaviors, memory access, I/O, communication, etc.). Moreover, different workloads are relevant for evaluating the self-adaptiveness of algorithms because if the testbed has only one behavior, it is hard to estimate how the decision-making algorithms will behave under other conditions.

Approach	**Application**	**Processing pattern**	**Input characteristics**	**Resources available**	**Execution environment**
Schneider [106]	X	X			
Choi [22]	X			X	
Tang [115]	X				
Selva [110]	X				
Su [113]	X		X		
Matteis [31, 33]				X	
Gad [41]					
Karavadara [65]	X				
Gedik [105]	X	X			
Schneider [108]	X				X
De Sensi [36]	X	X			
Vogel [128]					
Griebler [51]	X				
Kahveci [63]	X				
Vogel [126]	X	X	X		
Vogel [125]	X				
Gulisano [52]					
Balkesen [8]		X			
Heinze [54]			X		
Gedik [44]	X				
Wu [139]					
Chatzistergiou [17]	X		X		
Martin [83]					
Zacheilas [141]					
Lohrmann [76]	X				
Mayer [86]	X				
Mayer [87]	X				
Heinze [55]	X	X			
Martin [84]	X				
Zhang [144]			X		
Liu[74]	X				
Gil [24]					
Li [72]			X		
Kombi [69]	X		X		
Matteis [32]			X		
Cardellini [15]			X		
Cardellini [12]					
Cheng [19]					
Cheng [20]					
Lombardi [77]	X		X		
Kalavri [64]	X	X	X		
Wang [135]	X		X		
Bartnik [9]	X				
Kombi [70]	X		X		
Talebi [114]			X		
Das [29]		X		X	
Tudoran [121]			X		
Heinze [53]		X			
Mencagli [89]			X		
Liu [73]	X				
Floratou [40]			X		
Venkataraman [122]					
Tolosana [10]		X			

Table 3.3 continued from previous page

Approach	**Application**	**Processing pattern**	**Input characteristics**	**Resources available**	**Execution environment**
Mai [80]			X		
Rajadurai [101]	X		X		
Fardbastani [39]			X		
Marangozova [81]			X		
Lombardi [78]	X		X		
Abdelhamid [1]	X		X		
Russo [104]			X		
Schor [109]	X			X	
Vilches [123]	X	X			X
Mencagli [91]	X	X			
Matteis [34]	X				
Stein [112]			X		

Table 3.3: Self-adaptive validation

3.3.8 Overhead measurement

Regarding RQ9 that related to the overhead category described in Section 3.2.3, the section 'Evaluation-metrics' of Figure 3.2 provides results concerning the overhead measurement in the literature. The measurement of the overhead requires a comprehensive validation scenario, only the references [8, 109, 110, 84, 31, 33, 77, 126, 125] considered the overhead that can be caused by performing adaptation actions at run-time. Relevant aspects could be the resource utilization and performance of the application. The monitoring, self-adaptation algorithms, and reconfiguring entities for pursuing goals can also cause overhead. We argue that self-adaptive solutions should further consider the potential overhead caused, we believe that new validation methodologies for assessing the overhead should be proposed.

Overhead is also highly related to the environment and programming models used. For instance, it tends to be more complex to apply changes in distributed stream processing and when the adaptation requires state migration [44, 9]. A relevant example of overheads is an adaptation to Storm's topologies, where changing the number of replicas causes application downtime [13]. In [15] the reconfiguration applied in Storm caused performance losses, thus, they neglected such overhead by excluding the performance results up to 2 min after each reconfiguration. Consequently, for two minutes, there are no QoS guarantees. Thus, we argue that there is a need for better mechanisms and decision algorithms for improving stream processing applications for real-world scenarios.

3.3.9 Results summary

Table 3.4 provides an overview of the research questions and the results aforementioned throughout this section.

R.Q.	Context	Overview
RQ1	Goals	Achieve goals such as latency and throughput are mostly pursued.
RQ2	Entities being self-adapted	Adaptation can be in the environment (e.g., nodes, cores, frequency) and at the application level (parallelism degree, placement, scheduling, batches).
RQ3	Information used	Most approaches are collecting information from the applications to utilize for decision-making.
RQ4	Decision-making timing	Most actions are reactive. Although feedback loops are widely used as a theory for designing self-adaptation, several approaches do not explain the theory used. There are a large number of adaptation algorithms for decision-making.
RQ5	Parallelism characteristics	Several frameworks are being used. Storm is still popular in stream processing. Most studies focus on applications with a single fissioned stage.
RQ6	Parallelism abstractions	A few approaches mentioned abstractions as relevant for using/implementing self-adaptation, which is a potential aspect to be considered in future efforts.
RQ7	Experiments	Performance is mostly considered as an evaluation metric.
RQ8	Variations in experiments	Most studies were tested with different applications. The variations in the validation of solutions certainly require more attention in the future.
RQ9	Measuring the overhead	Few studies considered the overhead caused by adaptation actions, which is a concern that arguably has to be included in new evaluation methodologies.

Table 3.4: Summary of research questions and literature results.

3.4 Research challenges

In Section 3.3 we reviewed and discussed the current approaches from the literature. In this section, we introduce and discuss important aspects to be enhanced in the future, such aspects are considered as open research challenges.

3.4.1 Self-adaptive parallelism in complex compositions

Considering that in this work we are focused on self-adaptation for parallel executions, it is relevant to cover parallelism adaptation in complex compositions (defined in Section 3.2.2). The section ’Fissioned-stages’ of Figure 3.2 introduced how many fissioned parallel stages are used in the related literature. Adapting parallelism aspects at run-time in complex compositions is not a trivial problem. For instance, a simple strategy that only adapts the parallelism degree would need to take actions considering at least safety, load balancing between stages/compositions, and the amount of resources available. Consequently, such a solution would require several sensors and actuators with or without coordination among them.

Furthermore, the approaches available in the literature are still not presenting a self-adaptive solution that is generic and comprehensively validated. For instance, the IBM stream tools are tuned to their runtime library and constraints. Such a claim is supported by considering the high number of parameters that have to be set for using these tools and by the comparison between different approaches provided by [31]. In Matteis and Mencagli [31], the strategy from [44] was reproduced for comparison, where it achieved a poor performance by being unable to adapt under unbalanced workloads. The implication of this result provided in [31] is that the strategy of [44], which is a well-known solution for complex application compositions, lacks in terms of generality by performing poorly even with a simplistic composition of only one replicated stage. Kalavri *et al.* [64] found a similar outcome where Dhalion [40] was replicated and showed poor performance and slow convergence. Consequently, we argue that the scenario of complex application compositions demands a comprehensive evaluation of the literature's decision algorithms.

3.4.2 Improving resources efficiency and performance

As far as the entities found in the SLR are concerned, a potential optimization is to perform dynamic adaptations in the applications' graphs topologies (e.g., compositions), which can potentially self-optimize the application runtime in such a way that additional efficiency and flexibility is achieved. Hence, only the work of [101] was found that addresses graph adaptation at run-time, where the application is recompiled and changed without downtime using input duplication. However, this solution only performs such a complex optimization because runtime constraints make it unfeasible to adapt the parallelism degree at run-time, requiring program recompilation. However, dynamically changing the parallelism degree is possible in other runtimes without the need for recompilation.

Considering that stream processing applications execute for long periods with fluctuations, we argue that further enhancements are needed for providing the flexibility for changing the applications' graphs topologies. Optimization in these aspects can potentially improve the performance (stages separation), reduce resource consumption (fusion), or achieving a trade-off between performance and resources. Also, we believe that a mechanism for adapting graph topologies combined with self-adaptive strategies could make it possible to detect and overcome bottleneck stages at run-time. The importance of the aforementioned aspects can be also seen for optimizing resource usage when running stream processing applications in modern environments (e.g., Fog, Edge). In such environments, the availability of computational resources tends to be more restricted than in highly used multi-core machines.

3.4.3 Improving self-adaptation for dynamic environments

Our SLR also covered the important aspect related to the environment and architecture that stream processing applications are being executed. Notable, considering the evolution of the architectures, there are already efforts using hardware accelerators (GPUs, FPGAs) for stream processing [109, 123, 91]. Additionally, distributed cluster environments are highly used. Multi-cores also have the potential for providing the performance level required by a significant part of stream processing applications. This is achievable considering the increasing number of cores and sockets available in a single machine.

In the SLR it is notable that cloud environments are increasing in use for stream processing applications. Cloud computing can be seen as a flexible and dynamic execution environment, which can also be seen as a starting point for other environments like Fog and Edge. However, there are still challenges. For instance, Rajadurai *et al.* [101] demonstrated a peculiar issue of stream processing applications running in cloud environments. Live migration, which is a technique for moving a virtual machine from one physical node to another, causes downtime in stream processing applications. Downtime occurs due to the characteristics of these applications of constantly inserting new data that modifies the data saved in memory. A representation of this problem is provided by [101] where the application throughput drops to 0 for several seconds. In the narrow scenario of [101], they proposed techniques for mitigating downtime. However, a relevant aspect is that the potential downtime of stream processing applications running in cloud environments is not mentioned by other works targeting this kind of environment.

3.4.4 Self-adaptiveness validation and overhead measurement

Table 3.3 highlights the evaluation aspects of approaches from the literature. With a critical view, it is possible to note that the validation of the proposed solutions is not receiving the necessary attention. Although there are available efforts for benchmarking the adaptiveness of stream processing systems [57], we argue that we need further improvements in terms of methodologies and representative benchmarks for stream processing characteristics [82, 42]. The majority of approaches consider only the performance, neglecting the potential overhead caused. Additionally, only a few works considered a comprehensive testbed, with different application processing characteristics, inputs, and running architectures and environments. In fact, we argue that no work yet considered a wide combination of these evaluation categories in order to characterize how the proposed solution would behave in different scenarios.

Although every work tends to have specific scenarios and goals, we believe that the impact of self-adaptiveness may cause in applications must be better measured. The validation must encompass different scenarios being as broad as possible. Proposing new evaluation/validation methodologies and guidelines is a potential opportunity to improve the validation of self-adaptive solutions, where it can potentially improve the QoS of applications using adaptiveness. These new approaches are expected to be representative for measuring self-adaptation performance, energy, resource utilization metrics, and overhead.

3.4.5 Generalization and reproducibility of self-adaptive solutions

The conducted SLR has covered and extracted relevant information regarding the decision-making of existing self-adaptive solutions. Noteworthy, several approaches used as theoretical techniques the feedback loops adapted from control theory [56]. Another significant number of works also used queuing theory for modeling their solutions. The theories used have similarities, and one complements another (discussed in Section 2.3), but, notably, each approach tends to adapt suitable aspects to its specific scenario. Hence, there is a lack of considering the potential of generalizing and make the proposed solutions reproducible.

The proposed solutions could enable reproducibility by decoupling specific technical and low-level mechanisms from the potentially generalizable decision-making strategies. Hence, the design of new solutions can be improved by modeling what can be generalizable, which would enable new solutions to reuse existing parts and only implementing specific runtime mechanisms.

Additionally, comprehensive evaluation methodologies would allow one to validate the literature's strategies to a broader context or different scenarios to determine whether new strategies or decision algorithms must be proposed. Significantly, new tools and languages being designed could include mechanisms to adapt parallel mechanisms, especially for applications with multiple fissioned parallel stages. Hence, new tools could support features based on generically designed and comprehensively validated strategies for providing self-adaptation. For instance, reference [36] proposes a framework for developing self-adaptive solutions. We argue that there are further opportunities for providing frameworks for modeling self-adaptive solutions on runtime libraries/tools.

3.5 Threats to validity

There are also potential threats to the validity of the SLR to be considered. Similar to related SLR [100], the search for studies can be considered limited. Finding all relevant studies is a known challenge in literature revisions. We attempt to mitigate this aspect by searching in different databases and libraries as well as performing multiple search rounds.

The search terms used can also limit the results, as generic terms could bring much more irrelevant results. On the other hand, it is challenging to extract insightful results from terms used on very narrow scenarios. For instance, we recently found the term “malleability" being used as a synonym of adaptation, but “malleability" focuses on dynamic modifications of the granularity in Message Passing Interface (MPI) distributed applications [38]. In practice, the relevance of evaluating studies from other contexts is low. Our search terms were validated considering their completeness, with pilot searches evaluating whether previously known relevant studies were found.

Filtering studies and extracting data is another potential threat because different terms can be used for referring to a single characteristic. Consequently, we considered the general terminology of the area, where it is worth mentioning the catalog of similar terms provided by [58]. The threats to the consistency of data extraction were potentially minimized with a multi-step revision and an explicit revision protocol. Moreover, some short abstracts were excluded as well as works that fail to provide detailed information about their context and proposed solution. Other works without technical aspects and experimental evaluation were also removed as they fail to provide a suitable validation.

3.6 Summary

In this chapter, we proposed a taxonomy of relevant aspects to comprehend the related literature. We believe that self-adaptive parallelism can be exploited in several classes of applications, even where online, dynamic, and continuous adaptation is not needed. However, the strategies for monitoring and decision-making are still essential. Consequently, provide flexibility and modularity for future solutions is of paramount importance for their generalization and increase usage.

Considering the literature results, it is possible to observe that the research area is expanding, but several research challenges are still existing. Noteworthy, new self-adaptive approaches could provide flexible adaptations and parallelism abstractions to applications with realistic complex compositions. In the next chapters we propose solutions for help in addressing the above-mentioned research challenges.

4. A DECISION-MAKING FRAMEWORK FOR SELF-ADAPTATION IN PARALLEL APPLICATIONS AT RUN-TIME

Previously, in Section 3.4 from Chapter 3, we discussed many relevant open research challenges. In this chapter, we introduce a conceptual framework aiming at making the decision-making of the self-adaptive solutions more generic and flexible.

4.1 Context

A recurrent challenge in real-world parallel applications is autonomous management of the executions. This occurs because it is unfeasible for humans to monitor and manually change the long-running executions continuously, timely, and efficiently. Consequently, new techniques are being developed to cope with scenarios that could benefit from autonomicity, e.g., applications that suffer changes or fluctuations at run-time. A relevant example of a technique is self-adaptation [111, 56, 71] that can be broadly viewed as the capability of the systems/environments to be autonomous, deciding and changing their behavior under specific conditions.

Figure 4.1 shows a conceptual representation of our perspective of self-adaptation applied to parallel computing. In this representation, the main objective is to provide abstractions to the users/programmers. They are supposed to be enabled to define adaptation goals driving a self-adaptive managing system. It is important to note that the elements presented in Figure 4.1 refer to the taxonomy described previously in Section 3.2.

The self-adaptation workflow shown in Figure 4.1 comprises decision-making strategies and managed entities. The entities (AKA effectors, actuators, knobs) are controlled to apply adaptation actions (Execute), see more details in Section 3.2.1. The mechanisms for controlling specific entities tend to be more specific to each scenario. For instance, a low-level mechanism for online changing the number of replicas in a given runtime system or programming framework.

On the other hand, the decision-making strategy that decides when and how to adapt (additional details can be seen in Section 3.2.1) is usually composed of analytical steps, where the decision-making can ideally be designed to work in more significant scenarios. For instance, we expect that decision-making that is well-validated in one programming framework can also work very well in similar systems with unique mechanisms for applying changes. In short, a strategy can be viewed as a generic solution while a mechanism represents ways to target more specific context.

Decision-making strategies and mechanisms comprise a self-adaptive managing system, which is the main focus of this work. Such a system collects information (e.g.,

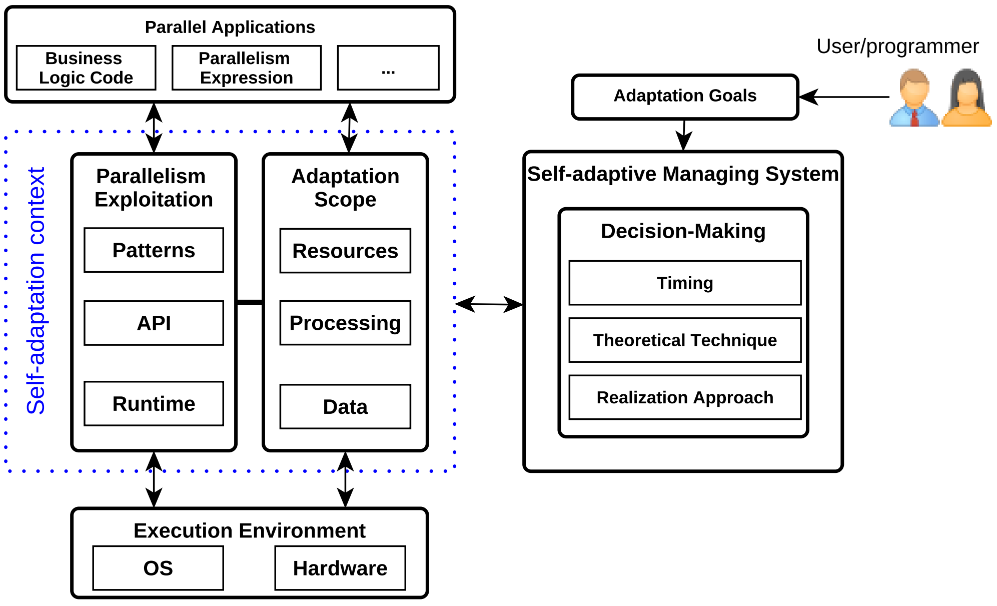


Figure 4.1: A conceptual view of self-adaptation in parallel computing.

monitoring) from the environment and software layers for sensing changes and applying necessary adaptation actions. As displayed in Section 3.2.1, the scope of adaptation has three classes (resources, data, system processing) used for applying adaptation. Importantly, in this work, we are interested in the self-adaptation context at the applications and parallelism exploitation runtime libraries/frameworks[1].

Traditional parallel applications and other specific applications classes such as stream processing applications execute on specific computational environments, where an environment is characterized by at least an operating system and a given hardware architecture. In the software layer, right above the operating system, we usually have the runtime library or framework used to support parallelism exploitation. Also, programmers can use API and Patterns (Section 2.1.2) for introducing parallelism to their applications.

The top part of Figure 4.1 refers to the applications domain, which is a sensitive layer where code and execution intrusiveness is mainly to be avoided. The applications' business logic code represents the functional code of a parallel application. Moreover, the parallelism expression is the first step for the parallel execution. When expressing parallelism, the application programmers can define code regions that are suitable and profitable for running in parallel.

In this sense, we argue that abstraction must be provided for application programmers since there is a significant gap between parallelism expression and an actual self-adaptive parallel execution. Effective parallelism abstractions can be achieved when self-adaptive systems regulate low-level mechanisms, which prevents users/programmers

[1]We acknowledge the existence and relevance of other entities in lower hardware and OS layers that can be self-adapted, but here we focus on flexible adaptations applicable at the higher-level that can provide powerful abstractions and efficiency.

from performing non-intuitive/error-prone activities. In this context, parallelism abstractions can potentially increase programmers' productivity as well as provide system optimizations (additional performance or efficiency). In summary, self-adaptation can be a potential solution to reduce costs and human efforts by reducing the need for human interventions.

In Section 3.4, we discussed that one of the main research challenges is to make the self-adaptation more generic. In practice, 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 [137]. Hence, we expect that enhanced self-adaptive systems can mitigate such challenges by ensuring the following properties:

- Modularity: The components of a system (e.g., elements, modules) needed in a given context are expected to be designed in a modular way to allow such parts to be decomposed/decoupled. Then, these parts can potentially apply to other solutions with technical compatibility. For instance, a monitor implemented in a given programming framework designed in a modular way (decoupled from the programming framework) could be easily integrated within other programming frameworks that use the same programming language.

- Abstraction: Relieve application programmers from the burden of finding the best configurations. Separation of concerns can enable programmers to set high-level objectives like SLO instead of hand-tuning configurations.

- Lightweightness: Execute without demanding a significant extra amount of resources or the self-adaptation causing intensive resources usage.

- Efficiency: An optimal configuration meets user/programmer goals and requires fewer computing resources. Consuming fewer resources increases the system efficiency, and reduces energy consumption and costs.

Therefore, we argue that the core of such a challenge is the decision-making process that determines which adaptation actions to be applied. A better design of the decision-making strategies is one way towards more generic solutions, which can be achievable by treating the decision-making within a self-adaptive solution in the form of a conceptual framework. In Section 4.2, we introduce a conceptual framework and describe how it can help address the challenges and properties mentioned above.

4.2 Conceptual framework

Providing autonomous solutions in the form of frameworks is already present in some scenarios. Noteworthy, NORNIR [36] was proposed as a framework for simplifying the management of energy consumption in parallel applications. Additionally, E2DF [96] was proposed to adapt the infrastructure resources availability and optimize the deployment of applications' stages. From an interesting decoupling perspective, [92] 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 [93], a theoretical model of a framework was proposed. The framework provides optimizations and mapping algorithms (only at compile time) targeting heterogeneous architectures, which is based on a static performance model.

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 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, an optimal decision-making could be applied to specific contexts that provide the means (mechanisms) to apply adaptation actions. Here, we propose a framework to make the design and decision-making of self-adaptive solutions more modular and generic. It is important to note that the decision-making framework proposed here is intended to help design, implement, and evaluate the proposed self-adaptive solutions. Such a conceptual decision-making framework is distinct from programming frameworks. Consequently, providing specific technicalities such as declarative API can be addressed in future efforts.

Usually, a given adaptation space must be managed manually by humans or automatically by self-adaptive approaches. In this work, we are interested in studying the new advances possible by employing self-adaptation to exploit online/at run-time the adaptation space to find appropriate configurations or parameters. The appropriate actions provided by the self-adaptive decision-making are the ones that provide efficient alternatives that enable abstractions to users/programmers. Consequently, here we argue that the decision-making should be designed and modeled with the help of conceptual frameworks. We expect that considering an adaptation space in a given parallel computing scenario, conceptual frameworks can help in designing and implementing efficient self-adaptive solutions.

From the parallelism abstraction standpoint, the decision-making system can provide new efficient abstractions to users/programmers. For instance, one can provide a workflow that in a given adaptation space with many configurations possible, the decision-making can find the best configuration.

Hence, we propose a decision-making framework to improve self-adaptation in terms of design, generalizability, and efficiency. Such improvements are going to be discussed in upcoming chapters. With a decision-making framework, it could 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.

The presentation of the proposed framework can be consistently linked with FORmal Models for Self-adaptation (FORMS) [138], a formal model for specifying self-adaptive systems that we consider as a reference model. In FORMS, self-adaptive systems are composed of 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 multi-core machine to a highly distributed and flexible cloud environment [127, 66, 26]. 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 4.2 shows the reference model of the proposed framework[2]. Although Figure 4.2 relates to the conceptual view of self-adaptation in parallel computing illustrated in Figure 4.1, here we focus on decision-making and how the conceptual view can be practically applied here. For instance, the abstract self-adaptive managing system shown in Figure 4.1 is here enacted by an autonomic decision-making loop, which is 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 [51, 129].

Moreover, our proposed framework is intended to be flexible and execute interactively. Hence, there is no strict order to the interactions shown in Figure 4.2 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 [67, 56] and interacts 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.

[2]This representation had some inspirations in how FORMS was mapped to the context of self-protecting systems in reference [140]

Since parallel computing and adaptations at run-time are complex, the main focus on generalizability is at the meta-level that must be highly customizable to different scenarios. For instance, heuristics, threshold algorithms, or auto profiling can be effective decision-making in some scenarios [126, 132]. On the other hand, balancing complex configurations in larger adaptations spaces may require advanced strategies encompassing artificial intelligence and machine learning [137, 45].

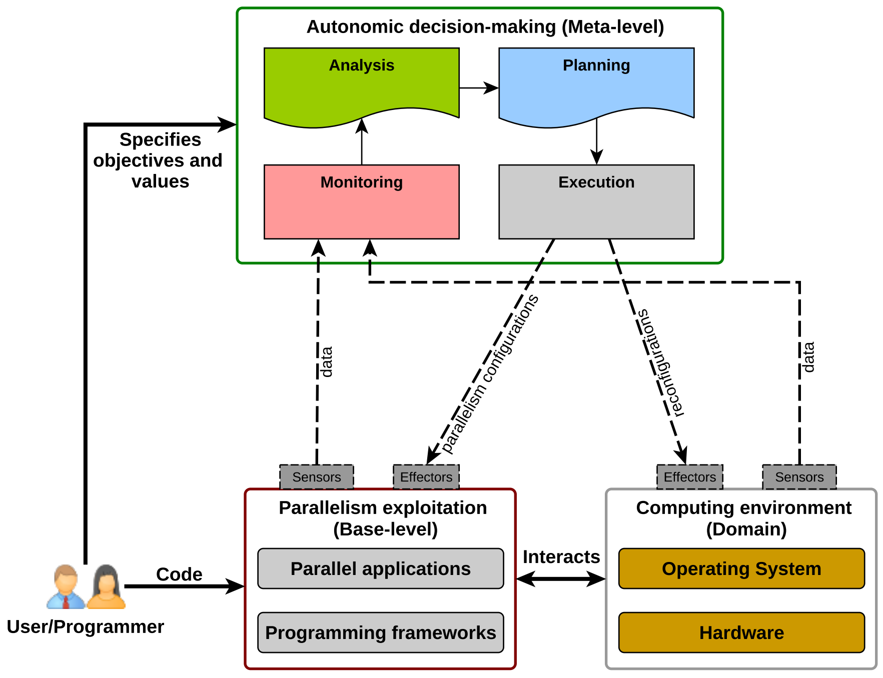


Figure 4.2: Framework's reference model.

Considering that generalization is very relevant, we expect some modules/system parts can be flexible: a manager module that makes decisions, a monitor (sensors), and an interfacing system module that integrates the meta-level with the base-level.

4.3 Applying the proposed framework

Previously, in Section 4.2 we proposed a conceptual framework for decision-making in parallel systems. In this section, we describe how this conceptual framework can be applied to provide decision-making for self-adaptive actions in a concrete scenario. Background concerning this context can be found in Sections 2.2 and 2.1.

Considering the reference model evinced in Figure 4.2, this structure was applied for implementing potentially generalizable decision-making solutions. Figure 4.3 shows how the conceptual framework was ported to a real-world scenario of the FastFlow programming framework (see Section 2.1.4), which is one of the runtime system generated by the DSL SPar [47, 60, 129].

The decision-making framework represented in Figure 4.3 comprises generic modules to be implemented for an effective strategy:

- **Application profiler**: This is a module intended to measure the actual processing capacity and computational weight of application's stages, which is very relevant for estimating the resources needed and the characteristics of an optimal configuration. For instance, the application profiler helps in characterizing the applications for improving the accuracy of the decision-making. See a concrete example in Section 7.2.
- **Configuration characteristics**: This is a module planned to percolate the possible configurations (adaptation space) and characterize them. This can be achieved in different ways. For instance, the application programmers can provide the characteristics of the configurations in a machine-readable descriptive way, or runtime systems can have modules for auto-detecting the configuration characteristics.
- **Search suitable configurations**: Considering the applications and configurations characteristics, this generic module is expected to utilize computational methods to find which configurations can be suitable for a given user objective.
- **Transitioning model**: Applying adaptation actions at run-time should be implemented as a smooth process such that they do not compromise QoS. Hence, an appropriate transition can be necessary to be applied when changing from one configuration to another. Section 7.2.1 provides a concrete example of employing a draining phase to avoid adaptation instability.
- **Data generator**: Usually the data items are real-time generated/produced from the network at a given speed, which can be simulated by a data generator module.

In Figure 4.3, it is evinced that the effectors managed in FastFlow are the replicas and the parallel patterns. The replicas effector corresponds to managing the number of replicas, e.g., the parallelism degree of parallel stages, and the Parallel Patterns is a more powerful adaptation that changes the entire applications' graphs topologies/composition structure.

In this work, our solution is applied and validated to implement stream processing applications, which is a representative paradigm present in several applications (see Section 2.2) of typically long-running applications. The rationale behind this decision is mostly because stream processing is a scenario with more strict requirements and dynamic executions, motivating to propose elaborated solutions, i.e., the reshaping of parallel pattern compositions used to exploit parallelism. We expect conceptual and technical efforts provided here to be easily applied to traditional parallel applications, which usually have less dynamic executions, making it easier to utilize decision-making strategies.

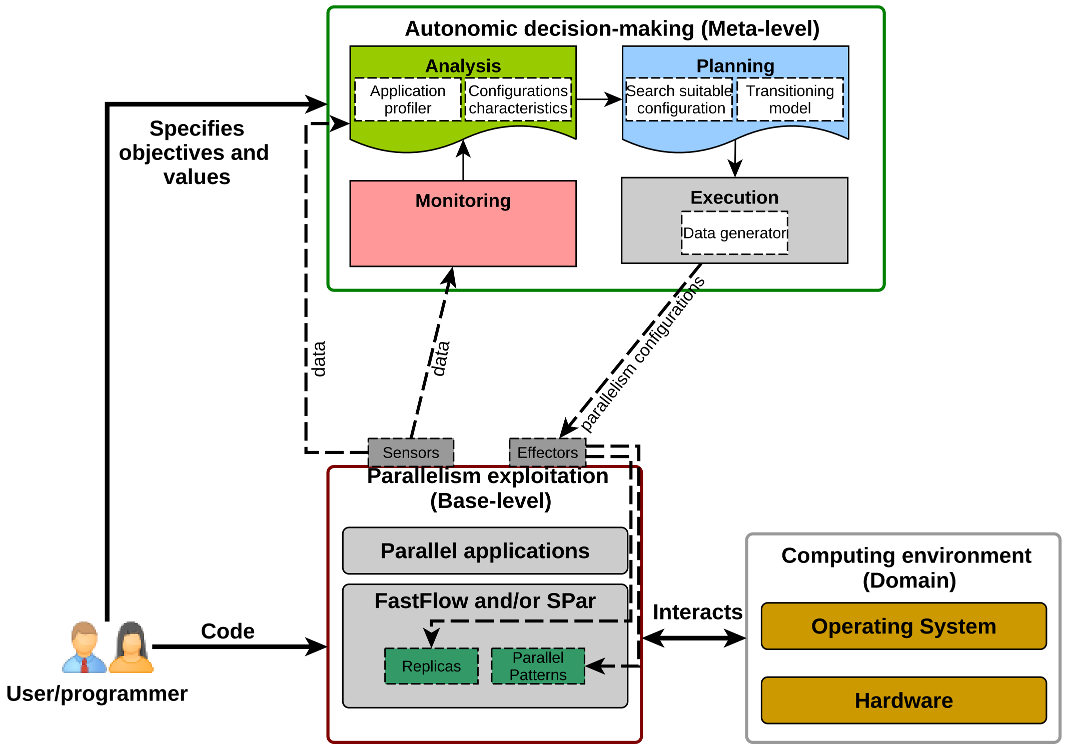


Figure 4.3: Framework's architecture implemented.

4.4 Summary

The proposed framework is a step to improve the design and engineering of self-adaptive systems to provide efficient and more generalizable parallelism abstractions. The framework's model is based on components decoupling and separation of concerns, which enables the self-adaptive solutions to be consistent with the conceptual definitions of self-adaptive systems [137].

In the next chapters, we introduce and provide further details of cases on how the decision-making framework was applied to design self-adaptive solutions and contribute to the research area. Chapter 5 describes an existing replicas effector and discusses strategies for self-adapting the parallelism degree in applications with one parallel stage, where it was possible to provide parallelism abstractions, introduce relevant non-functional metrics (throughput, latency), study the limits of self-adaptation and transparent executions, and minimize overheads that can arrive when applying self-adaptation.

Moreover, considering the demand for additional mechanisms to achieve the necessary flexibility for parallel applications, Chapter 6 introduces a new mechanism and a simple strategy to self-adapt the Parallel Patterns and online change the applications' graphs topologies. Then, Chapter 7 provides an optimized decision-making strategy for the mechanism proposed in Chapter 6. Finally, Chapter 8 introduces a proposed mechanism and decision-making strategy for supporting self-adaptation of the replicas effector in applications with complex composition structures composed of many parallel stages.

5. SELF-ADAPTIVE AND SEAMLESS DEGREE OF PARALLELISM

This chapter presents the proposed solutions for self-adapting the number of replicas in a parallel stage. Section 5.1 describes the first efforts proposed in the Master Thesis. Then, in Section 5.2 we provide strategies provided for self-adapting the number of replicas in applications with one parallel stage. Finally, Section 5.3 discusses this chapter's perspectives and closing remarks.

5.1 Previous work

In the Master Thesis [124], which was later published([129]), we introduced efforts for autonomously managing (through self-adaptation) the degree of parallelism in stream processing applications. There, it was noted that the parallel programming frameworks and libraries available for C++ programs like Intel TBB [134], StreamIt [117], FastFlow [3], and SPar [47] require users/programmers to define the degree of parallelism. The degree of parallelism was also too static for stream processing applications that have dynamic executions due to the many changes and uncertainties that occur at run-time.

A notable aspect unveiled was that using a static degree of parallelism during the entire execution was a suboptimal approach. Firstly, it was noted that it tends to be complicated and time-consuming to set parallelism parameters because a programmer would need to run the same program several times to find the optimal configuration. Secondly, the degree of parallelism depends on several aspects, which are usually related to computer architecture, the application processing characteristics, and input rates.

With the Master Thesis [124] and in [129], we validated mechanisms for applying adaptation and implemented reactive strategies for self-managing the degree of parallelism. The implemented solution works in Farm patterns (one parallel/fissioned/replicated stage).This was a first step for improving the parallelism abstractions related to the definition of the degree of parallelism in stream processing applications.

5.2 Self-adaptive strategies

The first part of the doctorate was related to improving the previously proposed strategies for self-adapting the parallelism, characterizing these strategies under different applications and workloads as well as proposing new strategies.

Section 5.2.1 describes the already implemented solution to self-adapt and abstract the degree of parallelism. Then, Section 5.2.2 shows the solution proposed in the

Master Thesis [124] for parallelism adaptations when the goal is throughput. Moreover, Section 5.2.3 introduces a strategy published in [128] that supports users/programmers to define applications' latency constraints, where a self-adaptive strategy autonomously manages the latency.

Moreover, Section 5.2.4 describes a proposed strategy to enable the users/programmers to manage better the computational resources CPUs, where users are empowered to provide a target utilization goal (SLO), and the self-adaptive strategy autonomously enforces the goal at run-time. Additionally, Section 5.2.5 presents the optimizations proposed in [126], where we minimized the self-adaptation overhead with a new decision-making strategy that reduces the settling times and increases the stability. Finally, Section 5.2.6 shows a strategy that detects performance fluctuations to manage the parallelism configurations transparently.

5.2.1 Self-adaptive degree of parallelism in multi-cores

Here we describe the proposed solution for self-adapting the degree of parallelism. Modules such as a monitor and actuator (effector) is a regulator are used for autonomously adjusting the number of replicas. Figure 5.1 shows the architecture used for performing adaptation in the number of replicas. This workflow was designed and implemented following the conceptual framework proposed in Chapter 4. Hence, in the *control steps* that applications and/or environments are monitored and decision-action are taken when it is necessary to optimize the execution. By doing so, it is possible to be reactive to select the best number of replicas even in presence of workload fluctuations, which is a common characteristic of stream processing applications.

In Section 2.1.2, we introduced some patterns suitable for stream parallelism. Considering that Farm is a very relevant pattern, in Figure 5.1 a Farm composition is evinced that is used for implementing self-adaptivity. In this Farm, the first stage is the task emitter that distributes the tasks to the next stage using a given scheduling policy. Furthermore, the second stage is the most computationally heavy one, which is replicated in several parallel replicas. Also, the last stage acts as a collector that gathers the tasks.

The strategy is implemented with different components. The collector stage runs also the *monitor* routine that periodically collects performance metrics while the application is running. Also, in the first stage runs the *regulator* component that corresponds to the Analysis and Planning modules of the conceptual framework, where *regulator* gets the information collected by the monitor, and decides which actions to take to optimize the parallelism configuration. The actuator (execution module) changes the number of replicas at run-time without restarting the application.

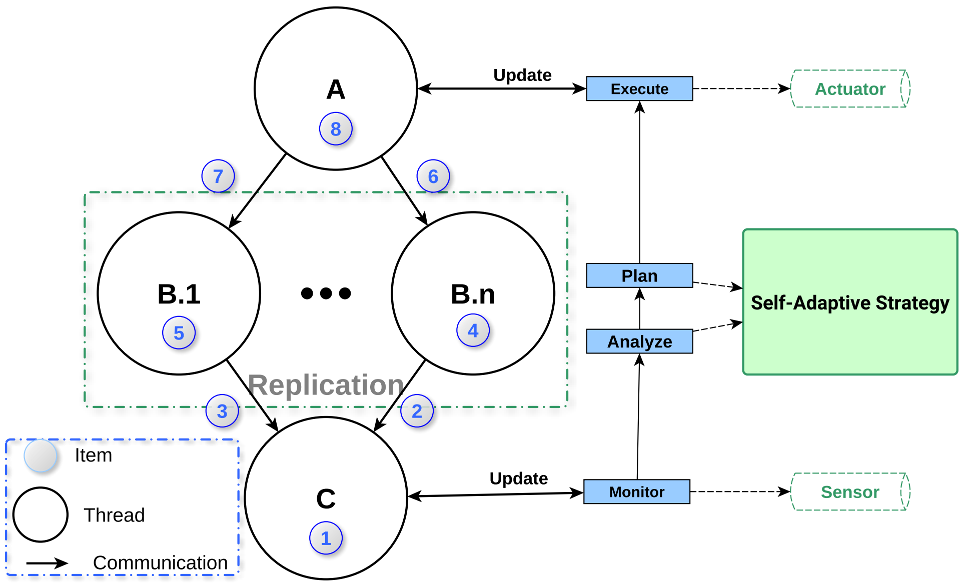


Figure 5.1: Workflow of self-adaptive parallelism management.

Source: [51].

5.2.2 Self-adaptation for throughput

This idea was introduced in [124] and the runtime system support was published in [129] as a language abstraction. The motivation was that defining a performance goal is presumably easier for application programmers than defining a low-level parallelism parameter of the runtime library. Therefore, we studied ways to handle the configuration challenges and abstract them from programmers to meet the requirement of a transparent degree of parallelism. Hence, we implemented a strategy that adapts the degree of parallelism based on the applications' throughput.

Considering the workflow shown in Figure 5.1, this strategy monitors the execution considering a performance metric to optimize the performance of the computation in the next iteration. In this case, the actuator has a maximum value for the number of replicas, which is defined according to the machine's CPUs availability. The execution is started when the emitter sends items to the active worker replicas.

We implemented this strategy through an algorithm that changes the number of replicas and continuously monitors the program's execution. The changes in the degree of parallelism are based on the target (expected) and measured (actual) throughput. On each adaptation step, several replicas can be activated or suspended, here the number of replicas changed is called a Scaling Factor (SF).

The throughput value (tasks/second) is calculated by the monitor component considering the number of tasks processed in the current iteration by subtracting the previous total number of tasks from the current total number of tasks. In each iteration, the throughput is the result of dividing the number of processed tasks by the time taken. Consequently, the last stage gathers the tasks and also measures the throughput. The throughput rates are then stored and accessed by the regulator/actuator. This solution provides information to the actuator to decide if an adaptation is required and only requires a target throughput to be defined by the programmer.

This strategy was firstly tested by adding the adaptive part to a parallel version of the Lane Detection video application. **Lane Detection** is an application used on autonomous vehicles to detect road lanes, which is used for maintaining the car on the road. This is performed by reading a video feed from a camera. The road lanes are detected through a sequence of operations where the parallel implementation is like an assembly line composed of three stages, where the second stage is stateless and therefore replicated [49]. The experiments shown here were carried out on a multi-core machine with 2 Sockets Intel(R) Xeon(R) CPUs 2.40 Gigahertz (GHz) (8 cores-16 threads), with a memory of 16 Gigabyte (GB) - Double Data Rate three (DDR3) 1066 Megahertz (MHz). The operating system used was Ubuntu Server, G++ v. 5.4.0 with the -O3 compilation flag. The parallel version used the on-demand scheduling policy that is suitable for stream processing, which improves the load balancing by distributing one item to each replica.

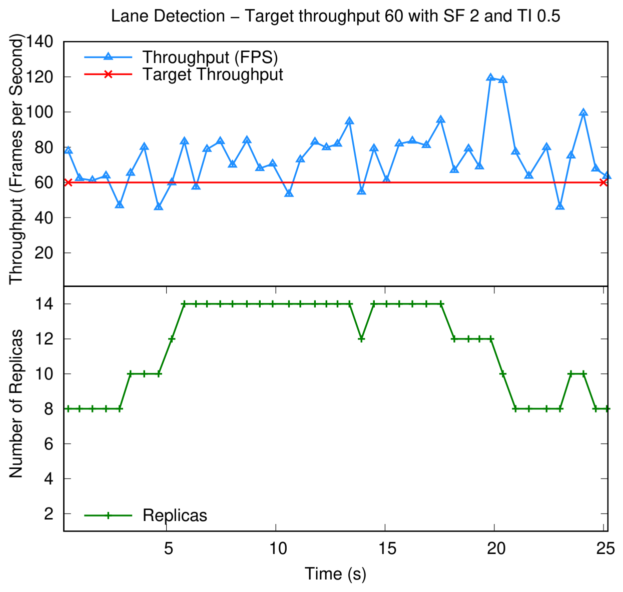


Figure 5.2: Target Throughput 60.

Source: [129].

In Figure 5.2, we present an experiment with a target throughput of 60 frames per second and a scaling factor of 2. In the tested machine, the number of replicas used varies from 8 to 14. As the number of replicas impacts the actual throughput, the execution starts using half of the total available cores (with Simultaneous Multithreading (SMT)). In the experiments, it is possible to notice a need to increase the number of replicas right after the execution starts. The load from the input file caused most of the throughput fluctuations. It is noteworthy that in some specific instances, even using the maximum number of replicas, it was not possible to achieve the target throughput. However, it was not caused by the adaptive strategy but is a consequence of the machine's limited processing capability.

This self-adaptive strategy was also tested in comparison with regular parallel executions that use a static (AKA fixed) number of replicas, ranging in the tested machine from 2 to 16 replicas. As aforementioned, the self-adaptive strategy tends to have a more elaborate execution with monitoring and adaptations, which can reduce the overall application performance. In this evaluation, we present the final throughput, which was derived by taking the total number of processed items in a given execution divided by the final execution time. Each execution was repeated 10 times and the results presented are the arithmetic means. The results also present the respective standard deviation.

Figure 5.3 presents the results from the lane detection application. As expected, in the static executions the throughput increased as more parallel replicas were added until it reached the scalability limit of the application. The static parallelism achieved the highest throughput with 14 replicas. The performance of the self-adaptive strategy with a target throughput was almost as good as the best static parallelism configuration (14 replicas). This result demonstrates that even with the additional parts implemented, a self-adaptive strategy can achieve performance similar to the best static cases. Additional performance results of this strategy can be found in references [129, 51].

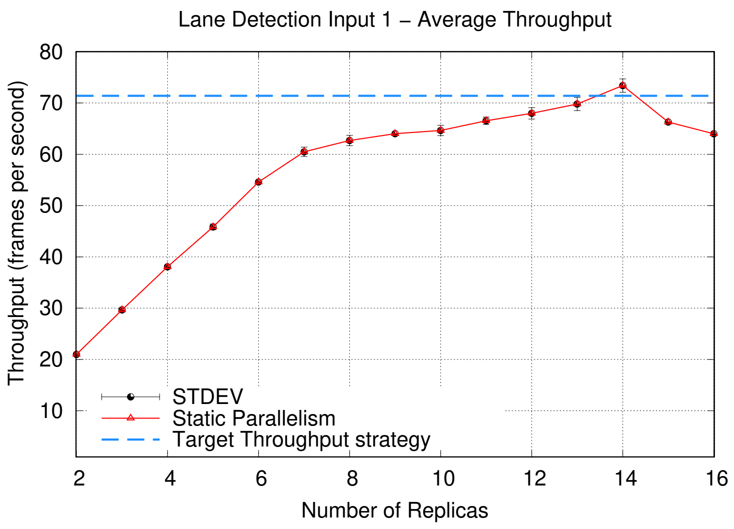
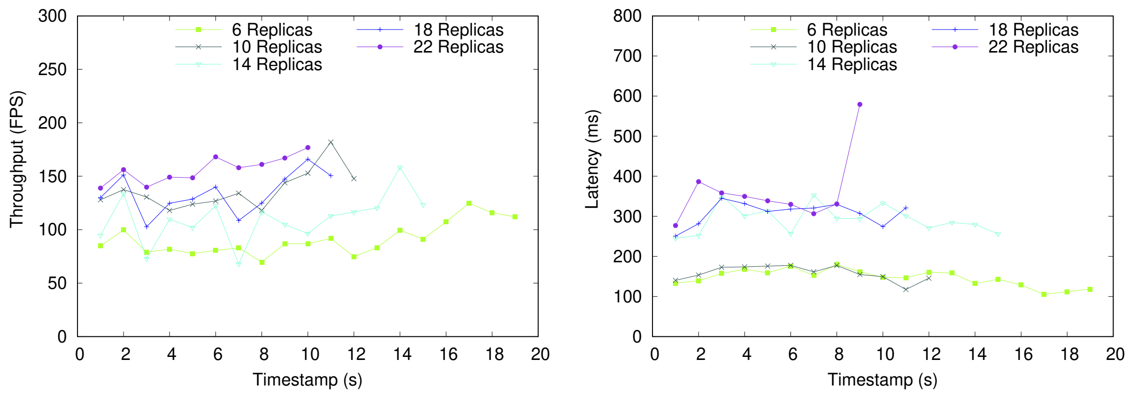


Figure 5.3: Average Throughput of Lane Detection.

Source: [129].

5.2.3 Self-adaptive parallelism with latency constraints

In previous work [128], we demonstrated how the degree of parallelism impacts the latency of stream items. For instance, there we executed again the Lane Detection application on a multi-core machine composed of 12 cores with 2-way SMT for a total of 24 hardware threads. Extracted from [128], Figure 5.4(a) shows the throughput of the application (i.e. how many stream elements per second are processed) for a different number of replicas and time timestamps of one second[1]. In this case, the number of replicas is statically defined and is not modified during the execution. These results prove that the use of SMT is beneficial for the throughput of this kind of application since the best throughput is obtained by using 22 replicas.



(a) Throughput. (b) Latency.

Figure 5.4: Lane Detection Characterization.

Source: [128].

Figure 5.4(b) extracted from our previous work [128] shows that increasing the number of replicas may have detrimental effects on the latency of the application. It is worth noting that a significant [2] increase in the latency (as well as a decrease in the throughput) can be observed when more than 10 replicas are used. Moreover, it is possible to note a significant increase in the oscillation of the latency when using more replicas. These effects are caused by the contention between stages running on two SMT cores corresponding to the same physical core.

There can be seen a correlation between throughput and latency. Achieving a high throughput using many replicas tends to increase the latency. On the other hand, using too few replicas decreases the throughput and latency. Consequently, it can be relevant to achieve a balance between these two performance goals. The challenge is

[1]Some textual parts were extracted from[128]

[2]The term "significant" refers to a performance difference that can achieve a statistical significance [129].

that high throughput is commonly pursued, and at the same time low latency may also be necessary. Consequently, in [128], we proposed a strategy that attempts to manage the latency of stream items by continuously adapting the degree of parallelism. This strategy uses a workflow similar to the one shown in Figure 5.1, the implementation and decision strategy are abstracted here, as such aspects are described in [128]. Importantly, here we present a representative result of the validation of this solution.

In [128] we demonstrated that pursue only the maximum throughput is not suitable for latency sensitive applications that need to rapidly produce results. Also, there we showed that using a minimal number of replicas for reducing the latency tends to result in a low throughput as well as inefficient usage of computational resources. Thus, the decision-making strategy proposed in [128] attempts to find a balance between the applications' throughput and latency by increasing the number of replicas when the latency is below the constraint defined by the user. In relevant experiments, we tested SF of the parallelism regulator was 1 or 2, meaning that on each reconfiguration one or two replicas could be activated or suspended. Also, 1 second is a time interval between adaptation actions that are sensitive enough to react without causing performance instability. Latency thresholds were used, where the threshold is a toleration value used in order to avoid oscillation in the number of replicas. The tolerated threshold makes the number of replicas not be increased when the latency is lower but close to the constraint.

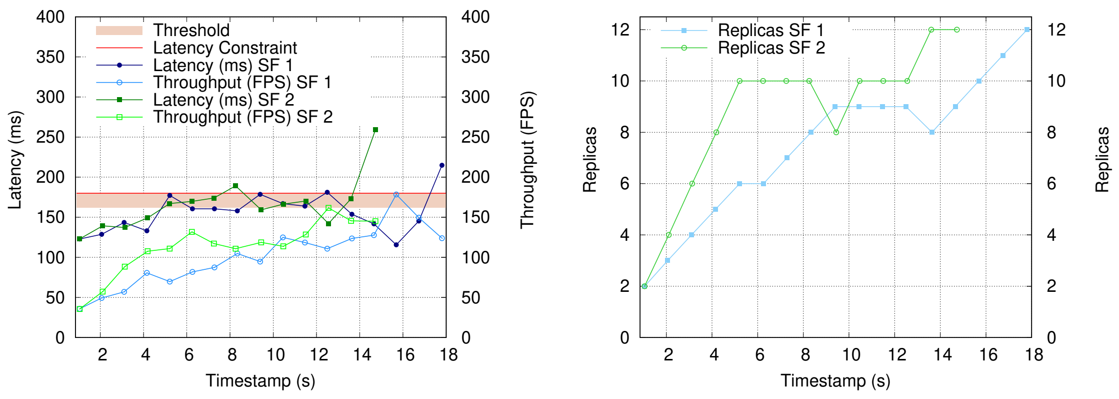


Figure 5.5: Latency Constraint of 180 ms (Left) and Replicas Used (Right).

Source: [128].

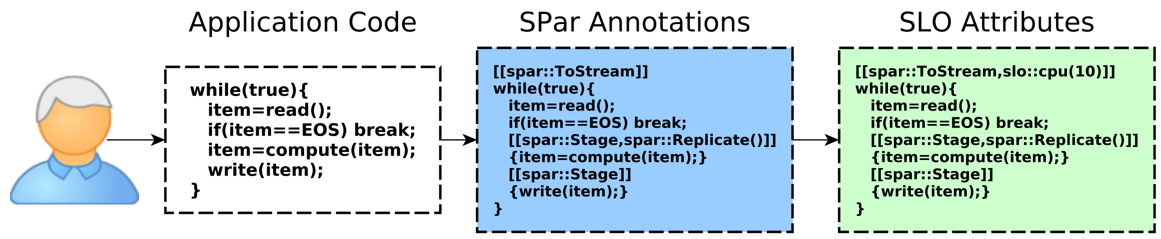
In Figure 5.5, the left side shows the throughput and latency of the application, while on the right side we plot the number of replicas used during the execution. In this experiment, the latency constraint was set to 180 milliseconds with a 10% threshold. As we can see from Figure 5.5, the number of replicas is reduced when the latency increases, and the number of replicas are changed several times due to oscillations in the input video. Comparing the configurations, we observed that SF of 2 reacts faster to changes and increases the throughput at the price of more latency violations.

Finally, as emphasized in [128], there we provided a new parallelism abstraction for SPar for latency sensitive applications. This was accomplished by implementing a strategy that adapts, without any programmer intervention, the number of replicas in order to have a latency lower than the constraints specified by the user. This is particularly useful for stream processing applications, which are characterized by fluctuations in the input rates.

5.2.4 Managing resources utilization through self-adaptation

We also implemented abstractions and strategies [48, 51] that enable users/programmers to express SLO, such as energy bounds, system utilization, and throughput. There, we evinced the need to impose limits in terms of resource usage while improving system utilization. We have shown that it is also relevant to allow programmers to define objectives regarding the consumption of resources.

Therefore, in [51] we provided several SLO options that allow users/programmers to set specific performance goals in source code. Relevant SLO provided were latency, throughput, CPUs utilization, and power consumption. Figure 5.6 depicts the methodology proposed in [51] to express SLO in the application source code. The first step in the developing process is to code the stream processing application (not needed for legacy applications). After, the programmer inserts the SPar annotations to express the stream parallelism. Lastly, the programmer can insert SLO attributes along with SPar's annotations in the source code. Therefore, the only requirement is to choose the SLO metric and its initial target value. No extra details have to be provided by the application programmers, which can spend most of their time coding the sequential application. For instance, in Figure 5.6, an slo::cpu is set to use no more than 10% of the machine processing resource.



```
while(true){
   item=read();
   if(item==EOS) break;
   item=compute(item);
   write(item);
}
```

```
[[spar::ToStream]]
while(true){
   item=read();
   if(item==EOS) break;
   [[spar::Stage,spar::Replicate()]]
   {item=compute(item);}
   [[spar::Stage]]
   {write(item);}
}
```

```
[[spar::ToStream,slo::cpu(10)]]
while(true){
   item=read();
   if(item==EOS) break;
   [[spar::Stage,spar::Replicate()]]
   {item=compute(item);}
   [[spar::Stage]]
   {write(item);}
}
```

Figure 5.6: A methodology to define SLOs for stream parallelism.

Source: [51].

A very relevant SLO that is pursued by adapting the number of replicas is the CPUs utilization SLO. Although there are available OS-level tools for controlling the CPUs usage (*e.g.,* CPUlimit [25]), such tools are arguably not flexible. Considering the dynamic nature of stream processing applications, we expect to adapt the degree of parallelism of the application at run-time for optimizing the CPUs utilization and meeting the target SLO. The workflow scheme used by the self-adaptive strategy is also similar, which here we again abstract the implementation aspects that are available in [51].

The `slo::CPU` SLO was validated in [51]. Here we present some representative results that are relevant to be discussed in the current research context. The results presented here are extracted from [51] and some text descriptions were also extracted from there. These results were collected from the same machine described in the previous section and the application used was Bzip2. Bzip2 is a data compression application that uses Burrows-Wheeler algorithm for sorting and Huffman coding. This application is built on top of libbzip2. Its parallel version using SPar is described in [50].

Figure 5.7 shows the execution of the Pbzip2 application with the attribute defining the maximum utilization to 60%, which is an empirically defined scenario, simulating an execution that could have a CPUs load slightly higher than half of the machine's resources. Such a scenario is representative of applications running on shared environments. We tested this SLO strategy with two representative threshold values: 10 and 20%. These were the most suitable thresholds for stream parallelism, as seen in [128]. We also executed a variant using the blocking mode (`-spar_blocking` compilation option in SPar) that tends to consume fewer CPUs resources by only distributing new tasks upon requests from the active threads. The results are compared to the CPUlimit utility tool, which also was set to limit the CPUs usage is 60%. For the tests using CPUlimit, we set a number of application threads equal to the number of hardware threads, which is what is done by default in several runtimes. The self-adaptive strategy, on the other hand, uses a custom number of active threads by changing the status of the replicas at run-time according to the decision-making strategy implemented.

In the results from Figure 5.7, we can observe that CPUlimit was unable to enforce the required SLO. It is relevant to highlight that all executions presented a high CPUs utilization in the first second. This event is caused by the application startup routines, such as threads and queues creation. The threshold of 10% introduced instability by triggering too frequent changes in the number of replicas, which also induced variation in CPUs utilization. On the other hand, the threshold of 20% was the most accurate and stable one. By using the `-spar_blocking` compilation flag, it reduced the CPUs utilization. Consequently, this resulted in an opportunity to use more replicas in the parallel region.

We now show the results obtained by running with the `slo::CPU` SLO with all the considered applications. The results presented are an average of 10 executions. In Figure 5.8, is shown the throughput in Megabytes Per Second (MBPS) of the execution considering the three representative applications, and two representatives `slo::CPU` SLO configurations: 60 and 90%. It is important to note that the SLO strategies are compared to a static degree of parallelism version using the CPUlimit for SPar and Intel TBB.

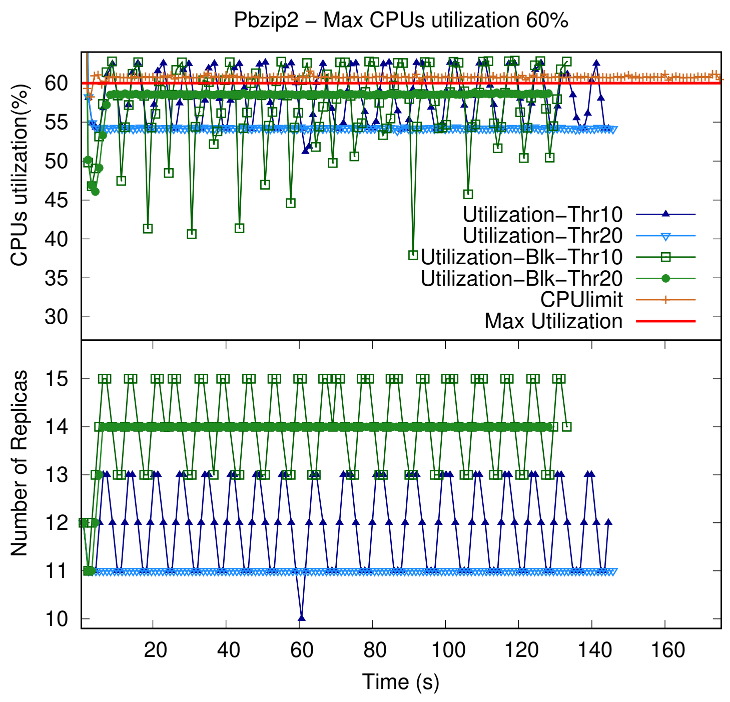


Figure 5.7: Characterization of *Pbzip2* Application with `slo::CPU(60)`.

Source: [51].

Considering the SLO of 60%, it is possible to identify a similar outcome regarding the different applications. In the self-adaptive executions, when using the `spar_blocking` compilation flag, it achieved higher throughput rates than the default nonBlocking execution. The self-adaptive strategy dynamically tunes the number of replicas resulting in the highest throughput rates. This result indicates that the way in which CPUlimit works (*i.e.*, continuously pausing and resuming the target process) causes performance overhead. CPUlimit in SPar had a lower throughput, while TBB and SPar Blocking achieved better performance.

The result of running with `slo::CPU` in 90% showed similar results with respect to 60%. Although the contrasts between our generated self-adaptive strategy and CPUlimit were smaller, our strategy again was significantly better in most cases. In Lane Detection with 90% CPUs utilization SLO, both TBB and SPar blocking achieved the highest throughput. CPUlimit blocked significantly less the threads with the low CPUs restriction of 90% CPUs utilization SLO, which increased the application performance. In Lane Detection, the TBB version outperformed SPar because TBB improves the load balancing, while in Person Recognition and Pbzip2 both versions achieved similar performance. Considering the different applications and their execution characteristics, it is possible to note that CPUlimit performed better in those applications with a more balanced load, while performing worst in the irregular processing applications (Person Recognition). This indicates that CPUlimit is not a suitable alternative for limiting CPUs utilization in stream processing applications, which are usually unbalanced because of their intrinsic dynamic nature.

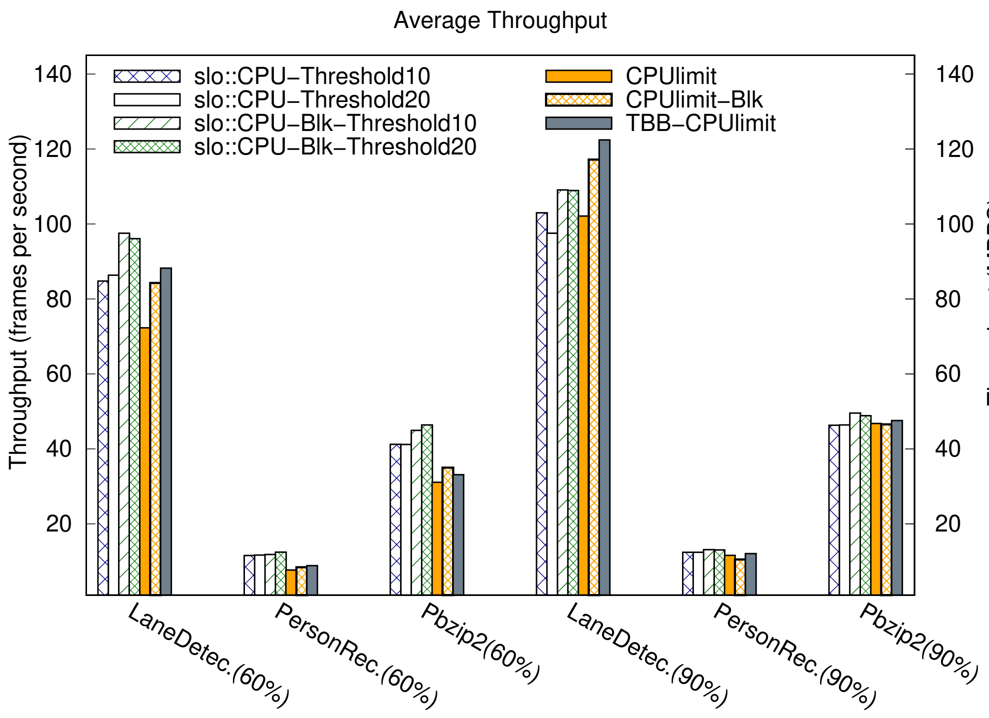


Figure 5.8: Applications Throughput.

Source: [51].

In order to further characterize CPUlimit, we also evaluated the impact of the number of replicas. Figure 5.9 presents the results on Pbzip with a representative `slo::CPU` SLO of 60%. In this test, the results from our self-adaptive strategy are compared to a static number of replicas in SPar and TBB managed by CPUlimit. The throughput of our strategies is presented in all numbers of replicas because any of those numbers could be used during the execution, depending on the decisions made by the regulator algorithm. It is possible to note that the configuration using 12 replicas was the best CPUlimit configuration in SPar and TBB, although the self-adaptive strategy in blocking mode still achieved the highest throughput. Regarding CPUlimit, the blocking mode only achieved a better performance in specific cases compared to the default nonBlocking mode. Comparing the results where TBB outperformed SPar running with one application thread per hardware thread in Figure 5.8, the several numbers of replicas in TBB only won with 14 and 16 replicas. On the other hand, SPar with the blocking mode outperformed TBB in most cases.

The outcome from Figure 5.9 highlights the correlation between the number of replicas and the application throughput, showing that using a tool like CPUlimit for limiting the CPUs utilization SLO is inefficient in the stream processing context. The results indicate that even if CPUlimit is used, a suitable number of replicas has still to be found. However, finding a suitable number of replicas tends to be a complex task in stream processing applications. Additionally, the number of replicas often has to be adapted during

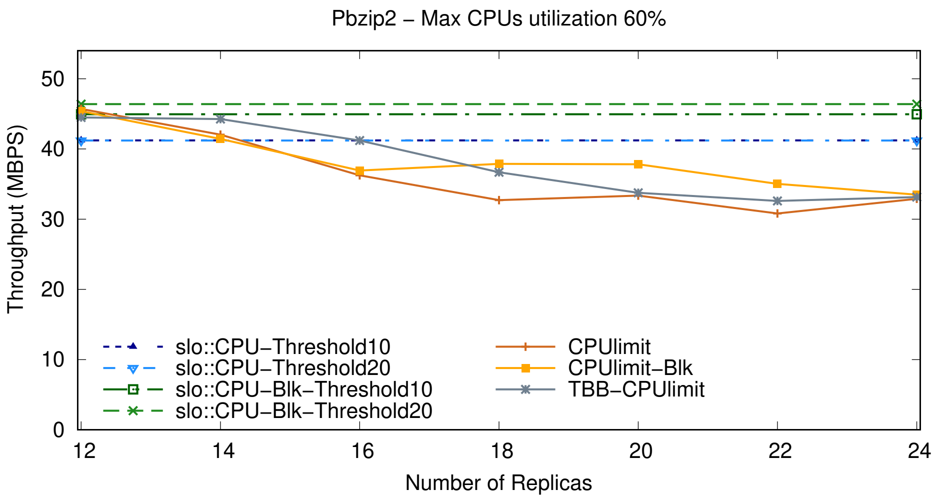


Figure 5.9: Characterization with Different Number of Replicas.

Source: [51].

execution according to performance or efficiency goals, because this class of applications runs without a defined end of the computation. Therefore, rerun the application multiple times until a suitable number of replicas is found, it becomes unfeasible for stream processing applications. Consequently, our strategy that dynamically adapts the number of replicas in SPar at run-time is a feasible and effective approach, which showed promising performance outcomes.

5.2.5 Minimizing self-adaptation overhead in stream processing

Employing self-adaptation to stream processing applications can provide higher-level programming abstractions and autonomic resource management. However, in [126] we have shown that there are cases where the performance is suboptimal. There, we optimized parallelism adaptations in terms of stability and accuracy, which improved the performance of parallel stream processing applications. The previous implementation [128, 51] was extended to better encompass the Stability, Accuracy, Settling time and Overshoot (SASO) properties [56]. *Stability* refers to the capacity of producing the same output under a given condition. *Accuracy* is related to achieving the control goal with sufficiently good decision-making, and *Short settling times* are desired for reaching fast enough an optimal state. Moreover, *overshooting* should be avoided by using only the amount of resources needed.

The validation of these self-adaptive strategies presented in [51, 129] showed that comparing only the target throughput (expected) and measured (actual) throughput sometimes resulted in too many and frequent reconfigurations, which in some events caused performance instability. For efficiency purposes, the number of replicas was reduced only when the actual throughput was significantly higher than the target one. The throughput oscillations and peaks induced the regulator to reduce the number of replicas. But, it was notable that when the unstable workload trend passed, using fewer replicas sometimes caused a lower performance than the target one. Also, the previous strategies increased 1 or 2 replicas while the throughput was smaller than the target, which resulted in settling times higher than the ideal one.

Consequently, in [126], we extensively analyzed the root causes and elaborated mechanisms to improve it, resulting in a new optimized strategy for handling the stability and performance violations. Here we present relevant pieces of information of this new decision-making strategy [3]. Figure 5.10 shows a high-level representation of the steps performed by the new strategy's decision-making. In order to respond to fluctuations, the decision-making follows the conceptual framework design (described in Chapter 4) to periodically (e.g., every second) iterate the steps: execute, decide, and when necessary apply changes.

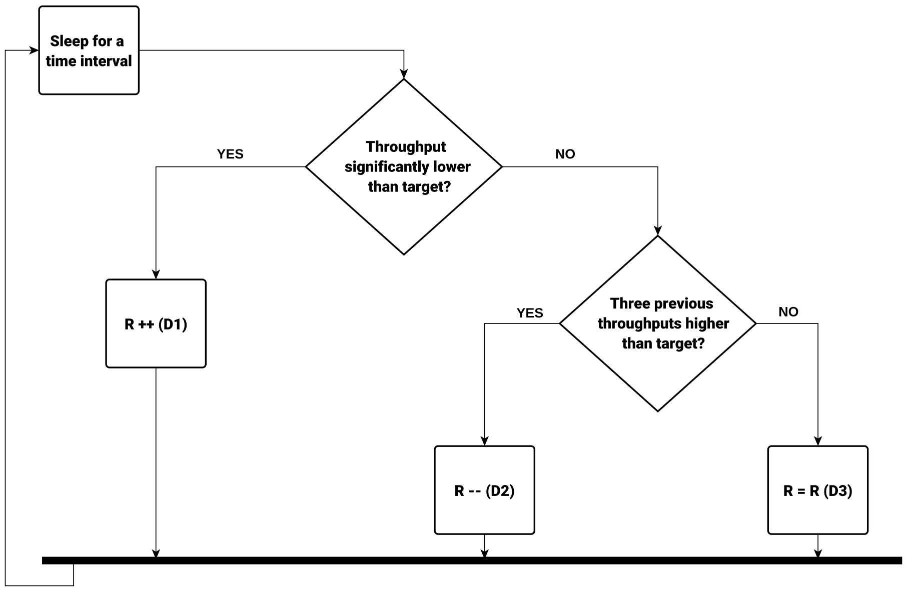

Sleep for a time interval
Throughput significantly lower than target?
YES
NO
R ++ (D1)
Three previous throughputs higher than target?
YES
NO
R -- (D2)
R = R (D3)

Figure 5.10: High-level representation of the decision-making.

Source: [126].

The new decision-making strategy checks if the current throughput is significantly [4] lower than the target one. If true, it enters the **Decision 1 (D1)** for increasing

[3]Some textual parts and results were extracted from [126]

[4]The term "significantly" refers to a performance difference that can achieve a statistical significance [129].

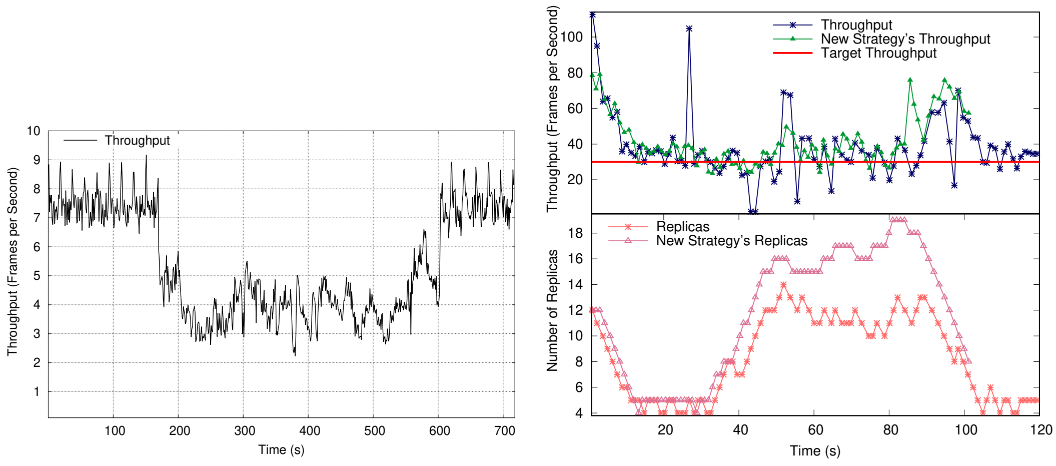
the number of replicas (**R**) with the following steps: 1) detects the machine processing capabilities; 2) calculates the percentage that each processor has from the total processing capability; 3) calculates the percentage of the difference between actual and target throughput; 4) according to the percentage of the difference and the processing power that each processor holds (from step 2), the regulator estimates how many replicas should be added. Consequently, if the actual throughput is extremely lower than the target, this new strategy attempts to increase the throughput by adding several replicas in one step (the previous strategy added 1 or 2 per step), such a decision has the potential to reduce the setting time. The regulator also checks if the current and previous throughputs are higher than the target ones. If false, it applies **Decision 3 (D3)** maintaining the same number of replicas. But, if this condition is true, the regulator applies **Decision 2 (D2)** that decreases the number of replicas. Pursuing stability and avoiding frequent/unprofitable changes, three previous throughputs are compared to the target throughput [5] (the previous strategy considered only one).

The new strategy was compared to the existing one. The adaptive part was included in Lane Detection application, which was described previously and used the same machine from the previous section. Figure 5.11(a) shows the throughput of a serial execution of Lane Detection with the tested video file Input-1 (260 Megabyte (MB)), which characterizes the load and shows the usual throughput fluctuations in stream processing, between 2 and 9 Frames Per Second (FPS). Some frames require more (or less) time to be processed resulting in load fluctuations, significant fluctuation can be viewed around the second 180 with throughput falling and increasing after 600 seconds in another workload phase.

Figure 5.11(b) characterizes the previous strategy and the new one with a target throughput of 30 FPS. The top part shows the measured throughput, where the new strategy had a stabler throughput that resulted in fewer throughput violations. Moreover, the lower graph referring to the number of replicas highlights the stability of the new strategy. The previous strategy reconfigured the number of replicas too many times causing additional throughput instability. Regarding the settling time, it is possible to note that between the seconds 40 and 60 of Figure 5.11(b) a new workload phase required parallelism reconfiguration. The new strategy reacted faster by adding 9 replicas that increased the throughput. The execution of the new strategy ended before due to its higher throughput.

We also tested this solution in terms of performance compared to static executions. Figure 5.12 shows the results of Lane Detection using input, presented in Figure 5.11(a). In the static executions, the throughput increased as more replicas were added until it reached the maximum performance of the application. It is notable some performance oscillations in the static executions between 10 and 21 replicas. These

[5]The three previous samples' values can be customized to different scenarios. Here, it was determined for this strategy considering empirical experiments.



(a) Input Workload Characterization. (b) Parallel Strategies.

Figure 5.11: Lane Detection - Sequential (Left) and Parallel Strategies (Right). Source: [126].

events were caused by the combination of these input load oscillations and the ordering performed in the last stage. When the load is too unbalanced (items have significant computing time differences), there will be more unordered items in the last stage, where a single thread has to reorder the items along with its operations (*e.g.*, write). Therefore, it becomes a bottleneck when there is such a combination of load oscillations and ordering requirements.

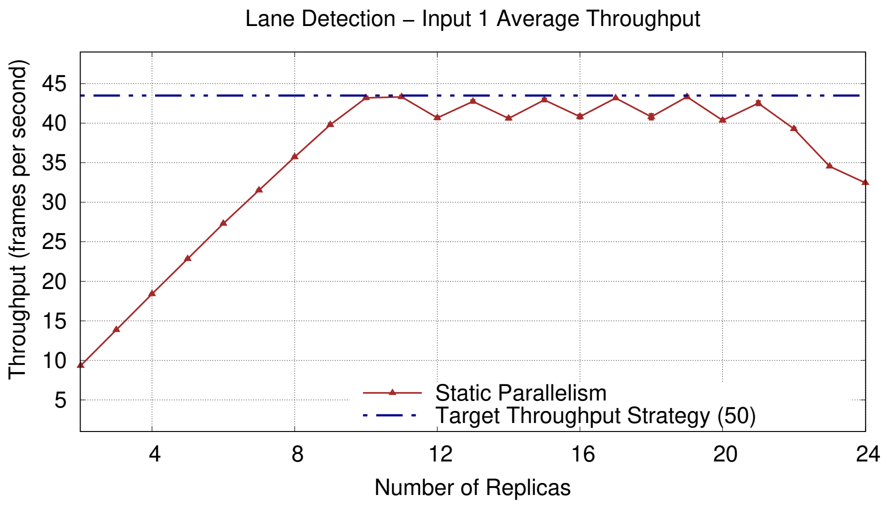


Figure 5.12: Average Throughput of Lane Detection. Source: [126].

In the used machine, the self-adaptive execution started using 12 replicas, since it is the number of available physical cores collected by the parallelism regulator. The throughput from self-adaptive executions is the same for all replicas since any number of replicas could be used during the execution. The performance of the self-adaptive strategy with a target performance was as good as the best static parallelism configurations. This demonstrates that even with the additional parts implemented, the self-adaptive strategy can achieve a performance similar to the best static executions that have no control or adaptability with respect to the number of replicas.

Several aspects of the execution are relevant to evaluate the strategies' executions. Noteworthy, memory usage is relevant for evaluating the amount of resources that a given program demands in order to run. The total memory usage was collected showing average execution results.

Figure 5.13 illustrates how the number of replicas impacts memory usage. Although the self-adaptive strategy has additional processing parts that could use additional memory, it consumed less memory than the static execution with more than 12 replicas. It is also worth noting a variation in memory consumption on the static execution with more than 12 replicas, this aspect is caused by a combination of the load unbalance of threads and by the ordering constraint. The results from memory usage of the self-adaptive strategies demonstrated that decision-making does not cause a significant additional resources consumption, which is relevant for running under a low overhead.

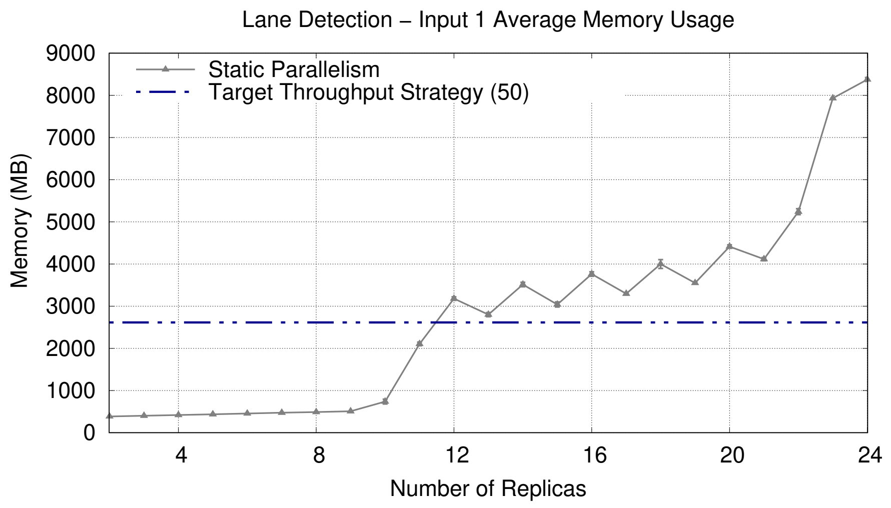


Figure 5.13: Average Memory Usage of Lane Detection.

Source: [126].

In short, our solution proposed in [126] provided high-level programming abstractions reduced the adaptation overhead and achieved a competitive performance with the best static executions.

5.2.6 Seamless parallelism management for stream processing

The previous proposed strategies [128, 126, 124, 51, 129] require from users the input of performance hints for adapting the number of replicas. However, performance aspects tend to be complex for application programmers. Additionally, stream processing applications are usually long running and with significant load fluctuations, where temporal changes could require different performance objectives. Consequently, in [125] we proposed a new strategy to manage the execution in an autonomous and seamless way. The new strategy abstracts from users the parameters set. This solution enabled a fully seamless execution, which was achieved by a new decision strategy that monitors the application, detects changes in the workload, and performs optimizations in the number of replicas used.

The workflow is similar to previous strategies, further details are available in [125]. Noteworthy, here we present relevant aspects fully described and discussed in [125]. The relevant decision-making strategy's parts decides whether the number of replicas should be adapted using the Analyze and Plan modules from the cocneptual framework with the following steps: 1) stores data regarding the application performance collected by the monitor; 2) After the execution starts, when it has a minimum of three (a number defined from empirical tests for having a balance between fast and accurate decisions) performance results from monitor iterations, compares this previously collected data with the current performance; 3) If the current performance is significantly lower than the previous one, a new replica (R) is activated (D1); 4) If the current performance is significantly higher than previous results, an active replica is suspended (D2); 5) After the monitor executed at least 10 iterations with performance results [6], the regulator enters a new phase where it has more performance data for deciding, which tends to improve the decisions accuracy. Then, for the sake of stability, the average of the previous three throughputs collected is compared to the average throughput from all historical data.

Figure 5.14 shows a high-level representation of decision phases and iterations performed. It is important to note that in addition to decisions 1 and 2 (D1 and D2), there is also the D3 that is performed when the decision is for maintaining the same number of replicas. Moreover, the self-adaptive strategy runs continuously and decides if the number of replicas should be adapted. Although the strategy runs several times and changes the configuration, the adaptations do not affect the regular computations of the application. In fact, while the application is running, the strategy periodically runs and then sleeps for a time interval. In this study, we consider 1 second as the default sampling time interval, which allows the strategy to achieve a suitable level of sensitivity to workload fluctuations. Too frequent adaptations can cause instability, while too high sampling times

[6]This number of iterations can be customized to different scenarios. Here, it was determined for this strategy considering empirical experiments.

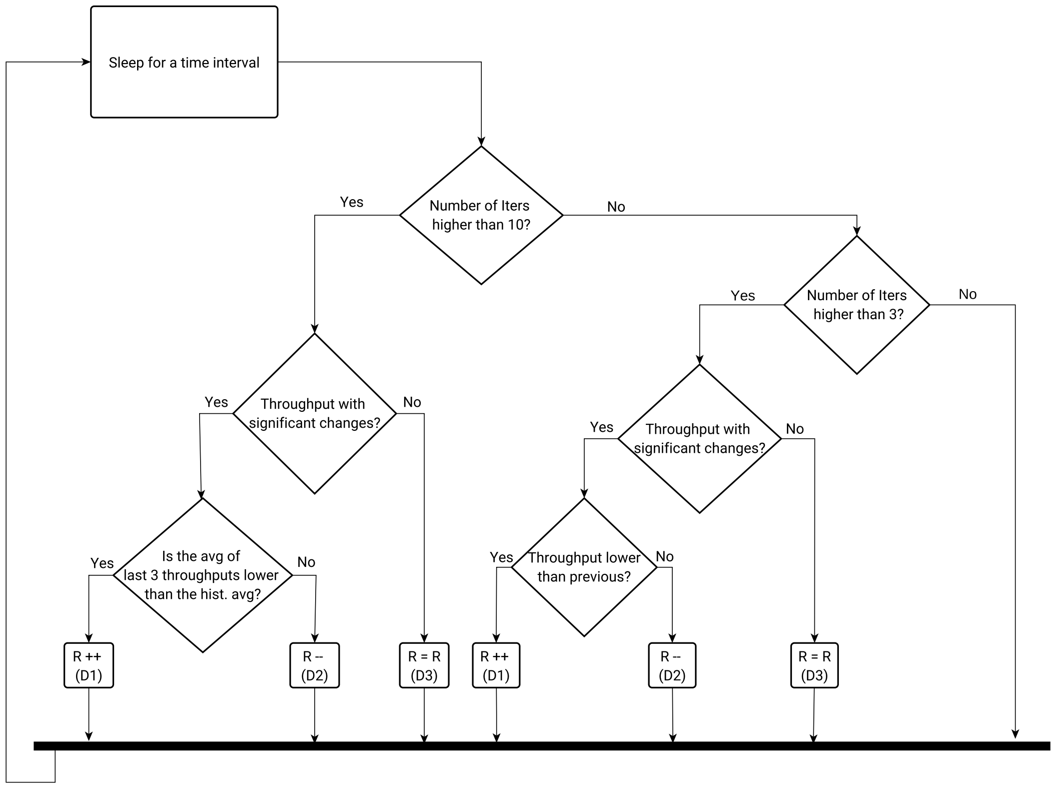

Sleep for a time interval
Number of Iters higher than 10?
Yes
No
Number of Iters higher than 3?
Yes
No
Throughput with significant changes?
Yes
No
Throughput with significant changes?
Yes
No
Is the avg of last 3 throughputs lower than the hist. avg?
Yes
No
Throughput lower than previous?
Yes
No
R ++ (D1)
R -- (D2)
R = R (D3)
R ++ (D1)
R -- (D2)
R = R (D3)

Figure 5.14: Overview of the Analyze and Plan Phases.

Source: [126].

can result in unresponsiveness to changes. Also, two is the minimum number of replicas in a parallel stage, which is a value for minimum parallelism. The maximum number of replicas is defined by the self-adaptive strategy by detecting the machine configuration. The maximum number of replicas is set to at most one application thread per hardware thread, also counting threads from other sequential stages (*e.g.,* Read, write).

The new seamless strategy is characterized and compared to an existing one [126] that requires a manual definition of a target performance, which was defined to a throughput of 50. The applications and machines used were the same of previous experiments. Moreover, in order to avoid additional variations, the emitter and collector stages were placed on dedicated physical cores.

The seamless strategy behavior is characterized in Figure 5.15 using the Lane Detection application and the input workload was a file of 260 MB [126]. The experiment demonstrates the throughput and the number of replicas used by each strategy in parallel executions. Moreover, the self-adaptive strategies are compared to static executions running with a fixed number of replicas. For the sake of visual clarity, we only show representative results of static executions with 10 and 20 replicas.

In Figure 5.15, we can observe throughput fluctuations caused by the input workload [126]. The executions with a static number of replicas also presented throughput fluc-

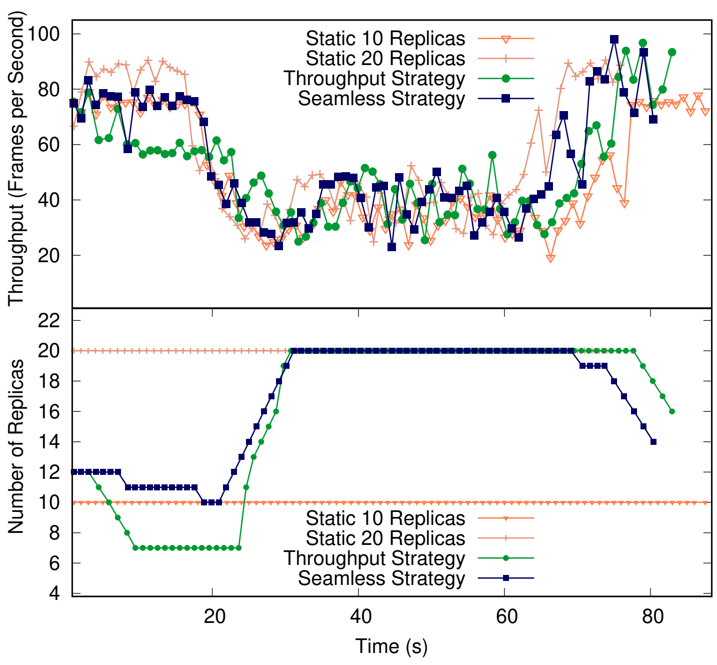


Figure 5.15: Characterization - Parallel Executions.

Source: [126].

tuations, which emphasizes that the oscillations were caused by input workload instead of the self-adaptive strategies. Regarding the proposed seamless performance strategy, it is important to note that after the first iterations, the throughput increased because of the workload fluctuation. As a consequence, the parallelism actuator changed the number of replicas from 12 to 11. Noteworthy, considering the workload fluctuations around the middle of the execution, the actuator responded to this fluctuation by increasing the number of replicas between seconds 21 and 36. Another event that highlights the correct sensitivity of this strategy is that the number of replicas was reduced when the execution entered a new phase that increased the throughput (near the second 70).

Comparing the strategies, it is possible to note a similar performance trend caused by the input workload. The strategy based on a manual target performance presented a short settling time, which is notable in the adaptation of the number of replicas after the second 20. The seamless performance strategy required more time to respond to workload fluctuations, which can impact negatively on those applications that demand very fast adaptations. Moreover, it is possible to note in Figure 5.15 that the seamless performance strategy had a slightly lower execution time, which occurred because this execution had a higher throughput in the first seconds by using more parallel replicas.

Moreover, in [125] we have seen aspects related to the complexities of abstracting parallelism and autonomously managing parallelism configurations at run-time. The new proposed strategy that abstracts the need to set the parallelism and performance configuration shown to be effective. The alternative that required the definition of a target performance increases the flexibility at the price of additional complexities. On the other hand, running an application transparently increases the abstraction level, but tends to provide less flexibility and lower performance. Some users/programmers may have performance expertise, in which case they may customize their execution by setting system parameters and target performance. However, the provided strategy for seamless execution is designed for users/programmers with no performance and system expertise. Additionally, in [125] the performance of the proposed strategy was validated compared to static executions where the self-adaptive Seamless strategy achieved a competitive performance. Consequently, an implication from the experimental results from [125] is that self-adaptivity is suitable for seamlessly managing parallelism configurations.

5.3 Discussion

5.3.1 Applying the proposed strategies

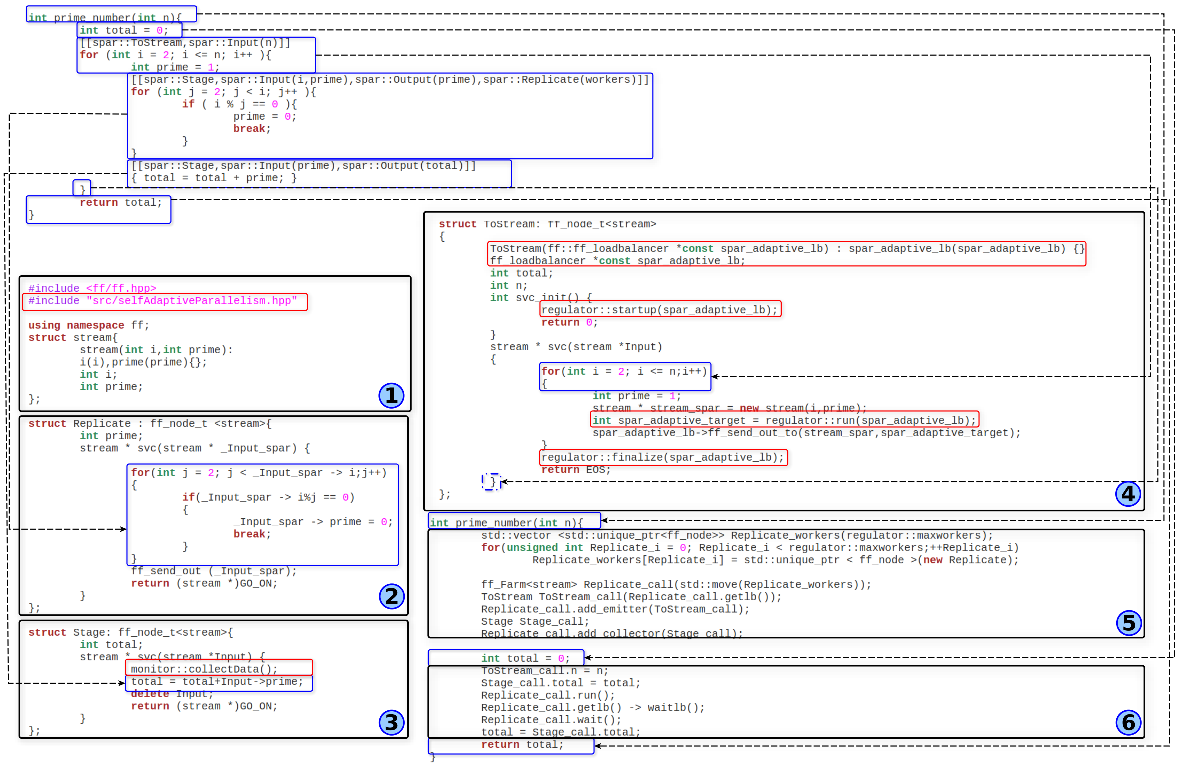
In [129], a case study demonstrated how one could use the previously described and evaluated self-adaptive strategies for providing efficient parallelism abstractions. A notable part is the self-adaptive code generation in SPar, where a compiler is used to generate additional code blocks so that the productivity of users/programmers is not impacted when using self-adaptation.

In [129], a trivial application that calculates the prime numbers as a code example was used to illustrate how the adaptive code generation works with SPar's source-to-source transformations[7]. In Figure 5.16, we show six steps of the code generated by the SPar compiler, where at the top it showed the code with SPar's annotation scheme. SPar's code generation follows some specific steps [47]. First, the compiler detects input and output dependencies for generating the step block ①. Then, the stages are built within blocks ② and ③ we must properly manage data, taking care of the associated input and output dependencies. The next step is ④ that creates the emitter (task scheduler) for the stream region. Finally, an example of a creation of a Farm pattern is shown in block ⑤ and in ⑥ the variable values are updated as well as the Farm starts running.

In this example, SPar generates a parallel code supported by the FastFlow runtime[8]. Importantly for supporting the generation of self-adaptive code, the code parts highlighted in the color red in Figure 5.16 are changes or additional parts for enabling self-adaptive executions using the proposed strategies. The self-adaptive code is generated using the already described Rule 2. Firstly, it is necessary to include the header files,

[7]Here we provide only a high-level description, the reader interested in more details can refer to reference [129].

[8]Additional details concerning the SPar's default code generation can be found in the reference [47].



```
int prime_number(int n){
        int total = 0;
        [[spar::ToStream,spar::Input(n)]]
        for (int i = 2; i <= n; i++ ){
                int prime = 1;
                [[spar::Stage,spar::Input(i,prime),spar::Output(prime),spar::Replicate(workers)]]
                for (int j = 2; j < i; j++ ){
                        if ( i % j == 0 ){
                                prime = 0;
                                break;
                        }
                }
                [[spar::Stage,spar::Input(prime),spar::Output(total)]]
                { total = total + prime; }
        }
        return total;
}
```

```
#include <ff/ff.hpp>
#include "src/selfAdaptiveParallelism.hpp"

using namespace ff;
struct stream{
        stream(int i,int prime):
        i(i),prime(prime){};
        int i;
        int prime;
};
```

1

```
struct Replicate : ff_node_t <stream>{
        int prime;
        stream * svc(stream * _Input_spar) {

                for(int j = 2; j < _Input_spar -> i;j++)
                {
                        if(_Input_spar -> i%j == 0)
                        {
                                _Input_spar -> prime = 0;
                                break;
                        }
                }
                ff_send_out (_Input_spar);
                return (stream *)GO_ON;
        }
};
```

2

```
struct Stage: ff_node_t<stream>{
        int total;
        stream * svc(stream *Input) {
                monitor::collectData();
                total = total+Input->prime;
                delete Input;
                return (stream *)GO_ON;
        }
};
```

3

```
struct ToStream: ff_node_t<stream>
{
        ToStream(ff::ff_loadbalancer *const spar_adaptive_lb) : spar_adaptive_lb(spar_adaptive_lb) {}
        ff_loadbalancer *const spar_adaptive_lb;
        int total;
        int n;
        int svc_init() {
                regulator::startup(spar_adaptive_lb);
                return 0;
        }
        stream * svc(stream *Input)
        {
                for(int i = 2; i <= n;i++)
                {
                        int prime = 1;
                        stream * stream_spar = new stream(i,prime);
                        int spar_adaptive_target = regulator::run(spar_adaptive_lb);
                        spar_adaptive_lb->ff_send_out_to(stream_spar,spar_adaptive_target);
                }
                regulator::finalize(spar_adaptive_lb);
                return EOS;
        }
};
```

4

```
int prime_number(int n){
        std::vector <std::unique_ptr<ff_node>> Replicate_workers(regulator::maxworkers);
        for(unsigned int Replicate_i = 0; Replicate_i < regulator::maxworkers;++Replicate_i)
                Replicate_workers[Replicate_i] = std::unique_ptr < ff_node >(new Replicate);

        ff_Farm<stream> Replicate_call(std::move(Replicate_workers));
        ToStream ToStream_call(Replicate_call.getlb());
        Replicate_call.add_emitter(ToStream_call);
        Stage Stage_call;
        Replicate_call.add_collector(Stage_call);
```

5

```
        int total = 0;
        ToStream_call.n = n;
        Stage_call.total = total;
        Replicate_call.run();
        Replicate_call.getlb() -> waitlb();
        Replicate_call.wait();
        total = Stage_call.total;
        return total;
}
```

6

Figure 5.16: Example of SPar's self-adaptive code generation.

Source: [129].

which is where the monitor and regulator components are actually implemented. Second, in the code generated corresponding to the emitter (A), the first step is to declare the *ff_loadbalancer* that is the runtime mechanism that controls the status of replicas. Third, it is necessary to call the regulator's function that prepares the system for starting the execution of the applications. Then, the regulator's *run*() function returns an ID of the replica to send the task and periodically verifies if the number of replicas has to be changed or not. In case a change is needed, the *run*() function transparently interacts with the runtime library and updates the configuration. Finally, when there are no more tasks to be processed, the regulator's *finalize*() function is called for cleaning up step that suspends the replicas to enable the runtime library to end the execution.

Moreover, in the stage represented in step 3 of Figure 5.16 that corresponds to C, a single extra code line is necessary for calling the monitor component, which collects relevant statistics and feeds the regulator. Although here we are describing how the solution for a self-adaptive number of replicas is integrated into SPar and its runtime library, these additional lines can be also manually instrumented/added to the other parallel codes. Consequently, we expect that similar programming frameworks/DSLs that have mechanisms for controlling the status of the threads could also use this self-adaptive library and its decision-making strategies.

5.3.2 Closing remarks

The proposed strategies extended previous work [124, 129]. Here, it was possible to design and validate new strategies assisted by the conceptual framework (described in Chapter 4) to provide additional SLO to users/programmers, increase the abstraction level, and minimize the self-adaptation overheads. However, a potential limitation of these strategies is that they support adaptation in applications with only one parallel stage, where supporting adaptation in the representative scenario of applications with more complex compositions is an open research challenge.

Therefore, we argue that more mechanisms and new strategies are needed to increase the adaptation space and existing abstractions. We understand that it is necessary to expand the area beyond optimization problems such as execution parameters and support efficient, flexible adaptations (reconfigurations) at run-time. Therefore, the subsequent chapters provide new conceived mechanisms and efficient self-adaptive strategies.

6. A MECHANISM FOR SELF-ADAPTATION OF STREAM PARALLEL PATTERNS AT RUN-TIME

In structured parallel programming, application programmers can easily create different application structures by instantiating high-level pattern constructors and combining them in compositions structures (AKA pattern compositions, stream graph, graph topology). However, it can be complex to find a pattern composition that provides QoS under dynamic executions (e.g., stream processing). Supporting dynamic changes in the pattern compositions used is expected to be a potential solution for abstracting complexities from users/application programmers and at the same time providing QoS or efficiency. For instance, alternative pattern compositions could be automatically discovered and instantiated, then the best one can be found and activated transparently.

Dynamically reshaping the pattern compositions can be relevant for several domains. In this work, we focus on stream processing applications where it tends to be more challenging. First, effective and safe mechanisms for applying changes are complex. Second, there is a need for more generic strategies for self-adaptive decision-making. Third, applying changes can have detrimental effects on the QoS like application downtime[1]. In this chapter, we tackle these main challenges

This chapter is a slightly modified version of the paper [131][2] and has been reproduced here in accordance with the signed copyright agreement and the copyright holder. In reference [131] the interested reader can access the content in an article format. This chapter adopts the following structure. Section 6.1 shows a motivational scenario. Then, Section 6.2 presents the proposed solution and Section 6.3 provides an experimental evaluation. Finally, in Section 6.4 we highlight the conclusions and discuss the perspectives of the chapter in the context of this dissertation.

6.1 Context

Attempting to simplify the parallelism exploitation, parallel patterns are usually composed and combined by users/programmers creating composition structures. Therefore, pattern-based parallel programming provides composable recurrent structures instantiated by programmers, combining the patterns creating different configurations. Stream processing applications running in multi-core machines can use high-level parallel programming frameworks, where Intel TBB [134] is an example from the industry and FastFlow [3], and SPar [47] are academic frameworks.

[1]A time period where a given application does not produce output
[2]Towards On-the-fly Self-Adaptation of Stream Parallel Patterns, 2021 Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP) - © 20XX IEEE

In TBB, the application programmers can create a parallel Pipeline by declaring each function as a filter and defining if a stage is parallel or sequential. In FastFlow, the user can also create Pipelines and replicate (run in parallel) specific stages using the Farm pattern.

TBB creates tasks that are scheduled to a pool of threads in a runtime system's perspective, where dynamic scheduling controls thread oversubscription by avoiding context switching and time-sliced execution. However, TBB can incur scheduling overheads with fine-grained tasks and I/O blocking operations. 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 suit stream processing with 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 in existing programming frameworks.

In Figure 6.1 we show pattern compositions[3], where a number of functions (f1, f2, f3) are decomposed in stages. A representative for stream processing applications, there is a data source and at least a Sink stage that will collect the results for producing an output. In the middle part, different compositions can be used according to specific application characteristics and user goals.

Configuration 1 represents a sequential stage (1S.) running the three functions. *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 (PS) running all functions. Considering that functions can be decomposed into multiple parallel stages, *Configuration 5* provides a variation of *Configuration 4* where 2 parallel stages (PS2) are employed, which can be useful for applications that are not embarrassingly parallel. For instance, there can be an internal state that prevents the easy replication of independent computations, i.e., first performing (in parallel) a filtering step, then computing the filtered data.

A stage can be represented as a node in the programming framework, where an important characteristic is the mapping of nodes to threads. In some cases, the nodes are mapped and executed by software threads. There is also the concept of tasks that are logical entities executed as independent operations, where software threads compute the tasks, i.e., a given computation in a stage can be processed as a task. Importantly, the mapping of nodes to threads and the pattern compositions shape the application structure, which can have a high impact on the application's performance and resource consumption. For instance, in a mapping of each node to a thread (one to one), there is only

[3]Here the terms composition and configuration are used interchangeably.

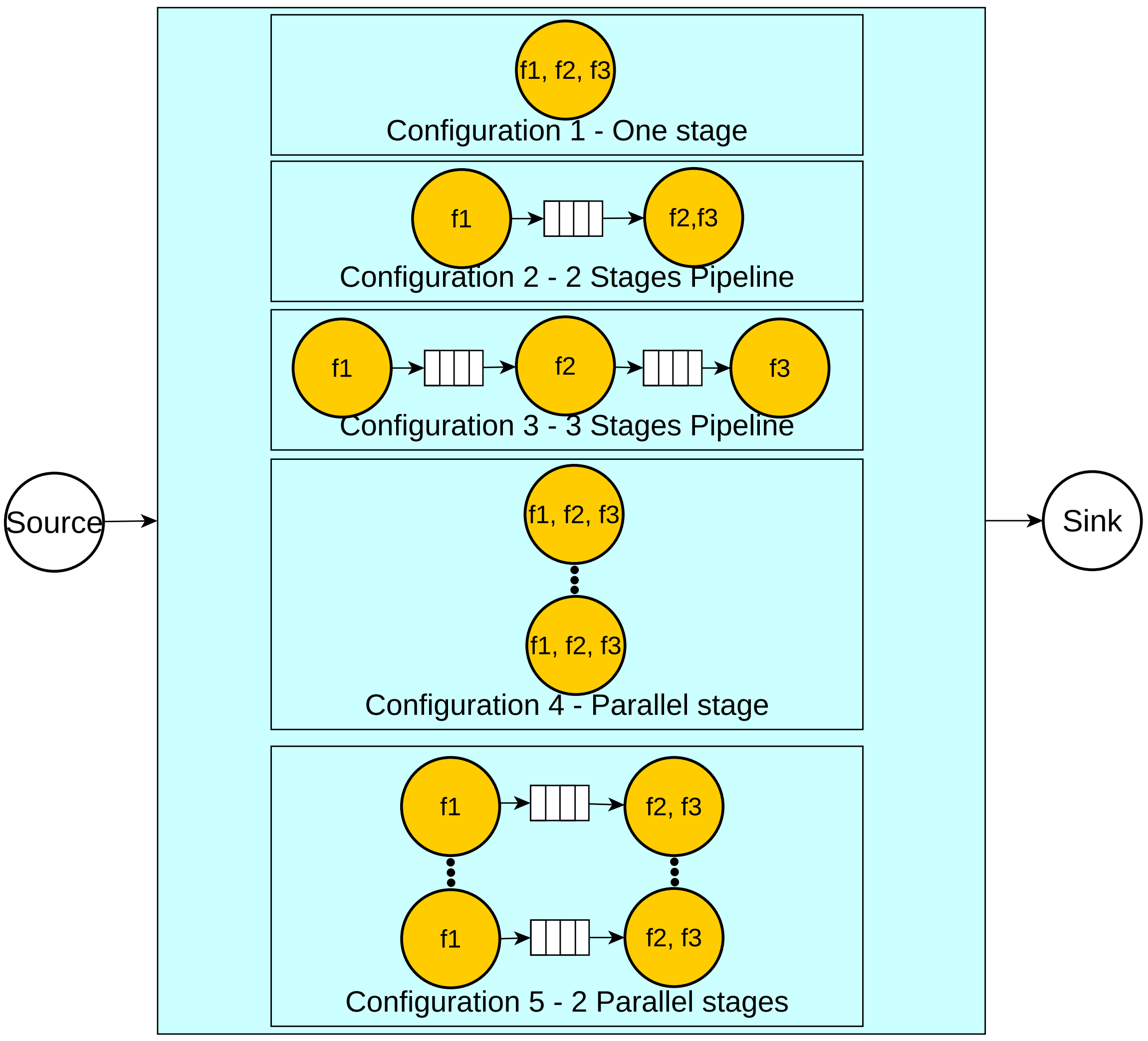

f1, f2, f3
Configuration 1 - One stage
f1
f2,f3
Configuration 2 - 2 Stages Pipeline
f1
f2
f3
Configuration 3 - 3 Stages Pipeline
Source
Sink
f1, f2, f3
f1, f2, f3
Configuration 4 - Parallel stage
f1
f2, f3
f1
f2, f3
Configuration 5 - 2 Parallel stages

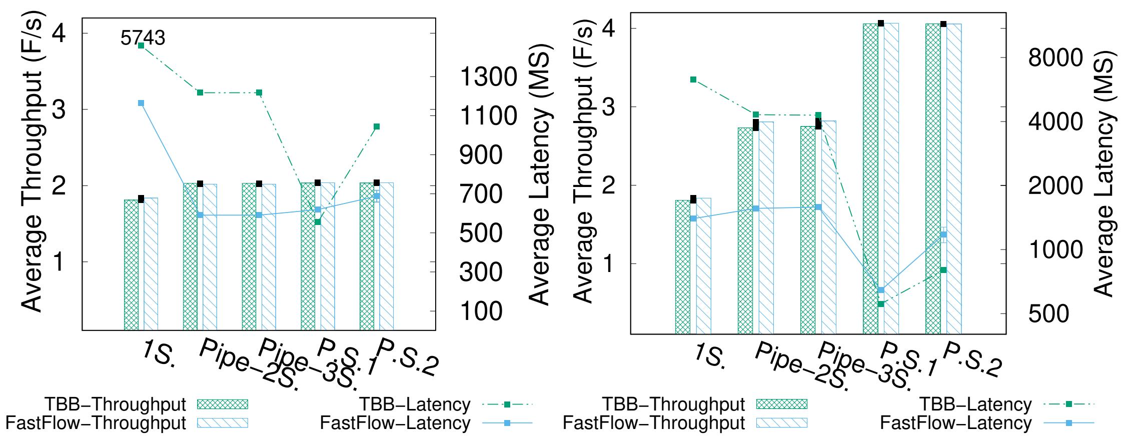
Figure 6.1: Example of compositions for stream processing.

Source: [131] © 20XX IEEE.

one software thread in the middle part of *Configuration 1*, such a configuration is suitable only for applications with a low-performance demand.

Figure 6.2 shows the performance of a stream processing application (setup described in Section 6.3) with the configurations from Figure 6.1 using two programming frameworks: Intel TBB [134] and FastFlow [3]. The data arrives at a fixed Input Rate (IR) of 2 FPS, where 2 is a suitable throughput for sustaining the IR. Latency is another relevant metric that corresponds to the time taken to compute a given item, low latency is a constraint for many applications.

Figure 6.2(a) evinces that in FastFlow *Configuration 2* was the best one by sustaining the input rate, providing lower latency, and using fewer nodes that consume fewer resources. In case other entities were being adapted this best Pipeline configuration with a given number of stages would not be achievable. For instance, adapting the number of workers (parallelism degree) would only be suitable for configurations using parallel stages. In TBB only the configuration with one parallel stage achieved a competitive latency. Under a higher input rate, Figure 6.2(b) shows that only the configurations with one parallel stage sustained the input rate with low latency.



(a) Input Rate 2 FPS.

(b) Input Rate 4 FPS.

Figure 6.2: Example on a Video Processing App.

Source: [131] © 20XX IEEE.

The results from Figure 6.2 emphasize that different configurations can be necessary to be used at run-time because the input rate can change due to network fluctuations or variations in the number of devices producing data [126]. Resource availability can also fluctuate in shared/dynamic environments like Clouds. Consequently, stream processing applications are expected to support dynamic adaptations at run-time. Considering that there are several aspects that correlate in a nonlinear manner, the user/programmer should not be expected to hand-tune the configurations at run-time. A solution is to support users/programmers to set only high-level goals like throughput or latency and rely on expert strategies that enforce a suitable QoS by finding and enforcing optimal parallel pattern configurations at run-time.

6.2 Proposed solution

6.2.1 Design goals and requirements

An effective approach of dynamic compositions for stream processing have different goals and requirements. 1) control loop with periodic monitoring applying changes only when necessary. 2) the adaptation should not cause application downtime. 3) smooth transition between configurations. 4) a suitable approach is lightweight without demanding a significant extra amount of resources and optimizes resource consumption while achieving the user goals.

6.2.2 Decision-making strategy

Proposing a flexible and generalizable decision-making strategy demands several assumptions to abstract specific implementation technicalities. Runtime mechanisms should be available for applying changes in programming frameworks. Moreover, the strategy receives a number of configurations to be tested i.e., by a user or system. Finally, external entities fed the strategy with information and alerts for making decisions.

The designed self-adaptive decision-making for the pattern compositions is described at a high-level abstracting the formalism. Such a description is expected to be sufficient for the reproducibility of the proposed solution that has three steps:

1. Training step: it activates each configuration for a time interval (e.g., 1 second) and collects execution statistics.
2. Optimal configuration: is the one that sustains QoS and needs fewer nodes. If the user goal is not achievable, enforces the configuration with the closest value.
3. Steady-state: returns to step one if the monitoring detects changes or if the user-defined goal is violated.

The proposed strategy is expected to enable non-functional requirements for stream processing. The users/programmers set an objective to be pursued by the self-adaptive strategy. Detecting fluctuations is an important part where values are considered significantly different when they have a contrast equal to or higher than a threshold of 20%, the suitability of such a value was ascertained in [128]. Moreover, pursuing stability and avoiding response to minor fluctuations, adaptation is triggered when three successive values indicate a change.

6.2.3 Implementation

Frameworks and libraries available were considered for implementing the proposed solution. There are available industry and academic solutions such as Intel TBB [134], FastFlow [3], and SPar [47]. Considering the support for performing adaptations at runtime, TBB supports only dynamic task distribution and load balancing for stream processing applications, where other adaptations have to be implemented at the lower level. 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 [119]. Thus, FastFlow was used for implementing the proposed solution.

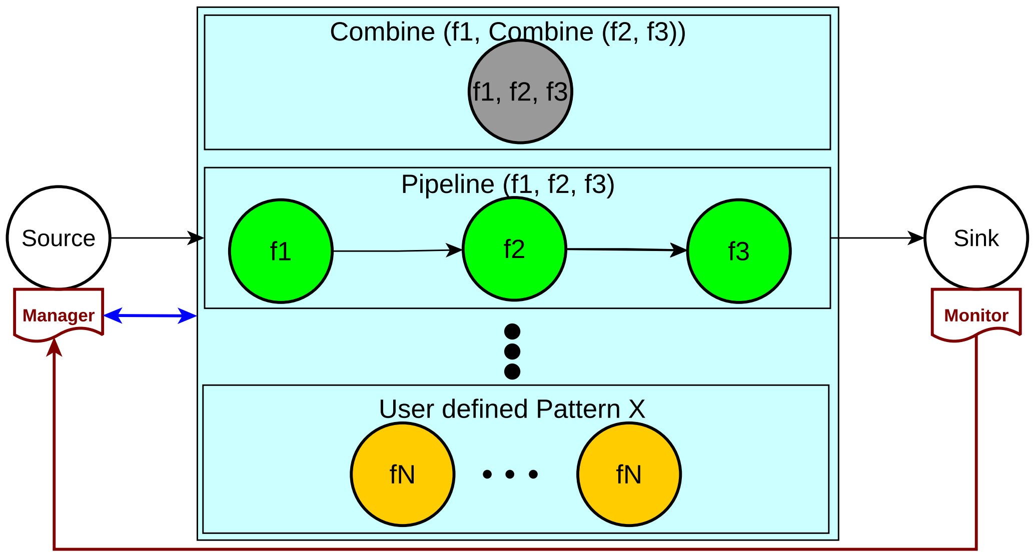


Figure 6.3: Proposed solution implemented in FastFlow.

Source: [131] © 20XX IEEE.

Abstracting specific 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 6.3 provides a representation of the implementation, where one entity is the *Manager* is embedded in the data source and uses runtime mechanisms for applying changes in an autonomous mode. Another entity is *Monitor* implemented as another embedded entity within the Sink stage that periodically collects data and feds the *Manager*. Figure 6.3 shows some possible configurations from Figure 6.1, where a Pipeline with 3 stages is active and the other is inactive. Moreover, the lower part of Figure 6.3 demonstrates the achievable flexibility because several other configurations can be composed by the user/programmer.

6.2.4 Solution's usability

Considering the usability of our solution from the programming perspective, it is not necessary to propose a new programming API. In fact, we intend to use the existing/known languages and enhance them in terms of execution abstractions with self-adaptation at run-time. Therefore, integrating our self-adaptive strategy within the FastFlow programming interface allows the business logic code inside the applications' functions to be reused instead of duplicated. One can provide parallel executions with FastFlow by declaring and instating parallel patterns using its skeleton library. For instance, an expert system programmer can declare and add with two C++ code lines the three-staged Pipeline used in Figure 6.3, other compositions/configurations can be expressed and included with similar coding productivity.

In addition to including headers and patterns instantiation, the self-adaptive strategy only requires two extra code lines in the applications' functions (stages) for calling the *Manager* and *Monitor*. We expect that (non-expert) application programmers can be assisted with tools for designing additional configurations and coding, such as RPL [62] and SPar's compiler [47]. Future research could make it even easier by exploiting potential ways to automatically derive and generate alternative configurations in a completely abstract way.

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 the usage of our solution from the applications' executions angle. The application programmers should be (ideally) only concerned with defining high-level goals, such as the SLOs defined in the code in Section 5.2.4. Another way to enable the application programmers to configure their high-level goal is through declarative files, e.g., XMLs used in [36]. Our solution is currently implemented to enable the application programmers to set their goals as execution parameters when they run the applications. For instance, to define the self-adaptive strategy to enforce a goal throughput with a value of 10 (items/second):

```
-application-binary throughput 10
```

By executing the application binary compiled with the programming concerns above explained, the goal and value are enough for the self-adaptive to parse such information and use to make decisions at run-time in a fully transparent way. 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. Importantly, considering that our primary focus is on providing flexible and feasible self-adaptive strategies, Section 6.3 evaluates the mechanism and the decision-making strategy proposed here.

6.3 Evaluation

6.3.1 Experimental setup

A multi-core 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 runtime buffer sizes were set to 1. The configurations illustrated in Figure 6.1 were used for evaluating if the proposed strategy is able to find the best configuration. Testing five configurations in different applications can be considered a pessimistic scenario that increases the training step. However, each application can have specific configurations and our solution provides flexibility for programmers to instantiate the configurations to be tested.

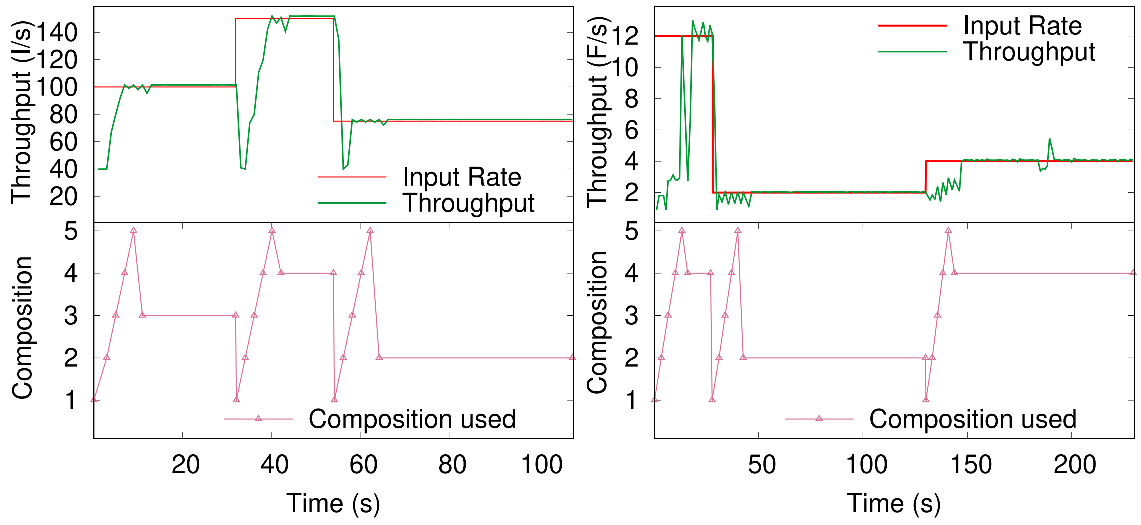
The strategy is characterized in a scenario simulating unexpected input rate changes. The performance is also evaluated with static executions using the same configurations as a baseline, where parallel stages configurations used a parallelism degree equal to the number of cores. Moreover, the *Adaptive* (abbrev. *Adapt.*) executions are the ones relative to the proposed strategy and $Adaptive - A.T.$ (abbrev. $Adapt. - A.T.$) refers to the performance collected after the training step. Executions correctness was ascertained by hashing the outputs.

6.3.2 Experimental results

The first application is a synthetic where 10,000 stream items are processed and each one has a service time of 24 milliseconds (ms). Figure 6.4(a) shows results from the execution of the self-adaptive pattern composition configurations, where the user-defined goal is throughput in items per second (I/s) equal to the input rate. The execution starts with a training step of the self-adaptive strategy that tests each configuration. After the training, the decision-making enters a stable phase with configuration 3, which is a 3 staged Pipeline configuration that sustained the input rate demanding the fewest amount of resources. Then, around the second 30, another training step was necessary because the input rate changed, where the strategy stabilizes with configuration 4. Another training step was performed when the input rate changed again to 75 I/s, where the execution stabilized with configuration 2. From a QoS perspective, it is possible to note that the self-adaptive strategy was able to effectively change and find the best configuration for achieving a suitable throughput. This simulated a pessimistic scenario where the user defines suboptimal configurations (i.e., a single stage under a high input rate).

Results from real-world applications are also provided with a customized version of the Person Recognition application [49], which has multiple functions to recognize people in video streams. Firstly, it receives video input and applies a denoising step for improving quality. Then, it detects and marks the faces with a red circle. These faces are then compared with the training set of faces. The face is marked when the comparison matches.

The input used was a 30 seconds long video with frames resolution of 260 pixels. Figure 6.4(b) shows the results where the goal of the self-adaptive strategy was to achieve an application throughput that would sustain the input rate. Importantly, the input rate



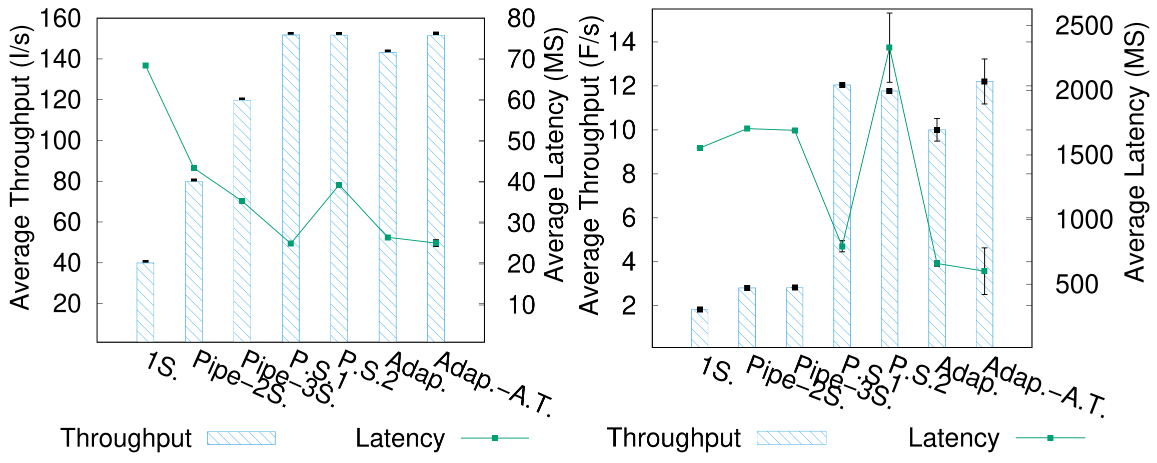
(a) Synthetic Application.

(b) Person Recognition.

Figure 6.4: Self-adaptation characterization.

Source: [131] © 20XX IEEE.

varies from 2 to 12 tasks per second, wherein in this application, each task is a video frame. Although the throughput fluctuated when testing suboptimal configurations, the proposed solution effectively reconfigured to find the best configuration.



(a) Synthetic Application.

(b) Person Recognition.

Figure 6.5: Performance evaluation.

Source: [131] © 20XX IEEE.

Figures 6.5(a) and 6.5(b) provides performance results with the metrics of throughput and latency. The collected metrics are an average of all processed tasks in a given execution. Moreover, each execution was repeated ten times (except in $Adaptive - A.T.$). $Adaptive - A.T.$ is a relevant outcome of the performance without the overhead of the training step. Figure 6.5(a) shows a representative execution of the synthetic application with an input rate of 150 I/s, where the best static configuration was with a parallel stage. Noteworthy, this was the configuration used by the self-adaptive strategy showing its effectiveness.

Figure 6.5(b) shows a representative result of the Person Recognition application with an IR of 12 FPS. The self-adaptive strategy was effective by choosing a parallel stage after the training step, which is the best static configuration under the high input rate of 12. However, the average throughput is lower than the static one, because the input used was not large with a short execution time of around 75 seconds. Consequently, the adaptive execution spent a significant amount of time in training (around 15 seconds) that reduced the throughput average. In long-running executions, the training impact could be lower with a performance similar to the $Adaptive - A.T.$.

6.4 Summary

In this chapter, we presented a solution for supporting self-adaptive pattern compositions, which was validated in stream processing applications. There are implications of the achieved results as well as some limitations that are relevant to emphasize. The results show that the proposed solution is technically feasible for adapting pattern compositions at run-time. Importantly, the adaptation is possible with a reasonable performance. A relevant implication of these results is that new abstractions can be provided for users/programmers. The dynamic adaptations and self-adaptive executions can provide additional flexibility for improving QoS (throughput, latency) and/or system efficiency.

This solution is limited in some aspects, i.e., mechanisms are necessary for the programming framework for achieving self-adaptive pattern compositions on each scenario. Moreover, the adaptation space can be limited by the alternative configurations provided, but this potential limitation can be mitigated. One way is to increase the adaptation space by combining the dynamic compositions with other less flexible optimizations such as batching and dynamic number of replicas.

Having mechanisms for changing the applications' graphs topologies is very relevant to achieve the flexibility needed for optimizing the performance and efficiency. Moreover, the dynamic adaptation using the mechanism is a step towards additional parallelism abstractions by enabling executions to be self-managed. In the next chapter we advance in this topic by extending the decision-making and the overall solution's validation.

7. AN OPTIMIZED DECISION-MAKING STRATEGY FOR SELF-ADAPTATION OF STREAM PARALLEL PATTERNS

In paper [131], we contributed with mechanisms for online self-adaptiveness of parallel patterns. The solution was integrated with a C++ programming framework (FastFlow [3]) and experimentally evaluated. In this chapter, we present the extended version [132], where online self-adaptation is achieved with a profiler that characterizes the applications. The profiler is combined with a new self-adaptive strategy and a model for smooth transitions on reconfigurations. Hence, we contribute with additional efficiency and flexibility for self-adaptation at run-time.

This chapter is a slightly modified version of the paper published in reference [132], in reference [132] the interested reader can access the version in an article format.

7.1 Motivation

In previous work [131], we evinced that streaming applications computing data in real-time require the programmers to create a configuration of sequential or parallel replicated stages. However, maintaining such a configuration for the entire execution can be limited due to the dynamic nature of applications. For instance, it was demonstrated with a video stream processing application executing under fluctuations in the IR that the best configuration combining parallel replicated, sequential, and merged stages varies with different scenarios that occur at run-time and from one programming framework to another.

A more challenging scenario is applications with complex composition structures composed of several sequential or parallel replicated stages. For instance, Figure 7.1 shows the structure of the Ferret [79] application from the PARSEC suite, where the four middle stages are thread pools that can run in parallel. However, the profitability of running them in parallel can vary from how balanced the stages are and according to the expected performance and QoS. Ferret has a complex structure modeled with different shapes by composing and nesting parallel patterns [37]. Figures 7.2 and 7.3 provide performance results with different Ferret's configurations, where the setup is described in Sections 7.3.1 and 7.3.2. This new streaming version of Ferret computes data at a given IR and provides stream processing performance metrics like throughput and latency. In Figure 7.2 the IR is 10 items per second, and 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 [31].

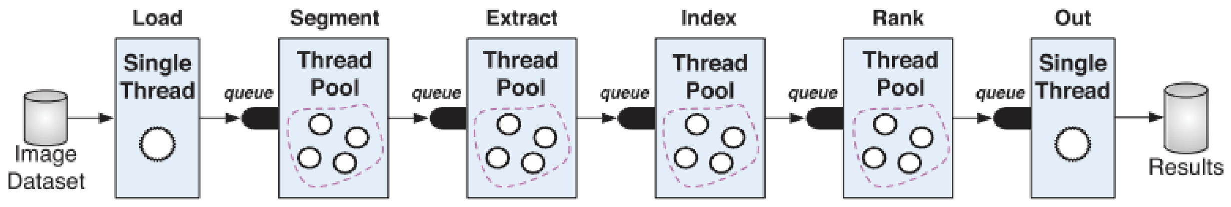


Figure 7.1: PARSEC's Ferret Pipeline structure.

Source: Extracted from [37].

Here we show the performance from 15 configurations described in Section 7.3.2. There are no results from configurations 13, 14, and 15 with TBB because merging functions would require refactoring the business logic code. Configuration 12 corresponds to the native implementation using Pthreads where all stages are parallel[1], demanding more resources and not performing so well in terms of latency. In Figures 7.2 and 7.3 it is possible to note that the performance varies according to the configurations, which have a significant impact on the throughput and latency. The throughput increases, and the latency decreases in configurations with a suitable level of parallelism in computationally intensive stages. The best performance is achieved when the last stage (the most intensive one) is parallel. Although declaring such a configuration is trivial, it is not intuitive for application programmers and significantly impacts QoS.

Moreover, the previous solution [131] described in Chapter 6 was still limited in some aspects such as settling times and accuracy. It needed to test all configurations available to find the best one. Hence, here we provide the following contributions that extend previous work:

- 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 SLO.
- 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 7.2.1).
- Extended validation of the proposed solution, including new scenarios and applications. Noteworthy, we provide a custom version of the Ferret application to regulate the IR and support user-defined SLOs (throughput, latency). Ferret (see Section 7.3.2) is a realistic application from the Princeton Application Repository for Shared-Memory Computers (PARSEC) benchmark suite.

[1]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 such a comparison can refer to reference [37].

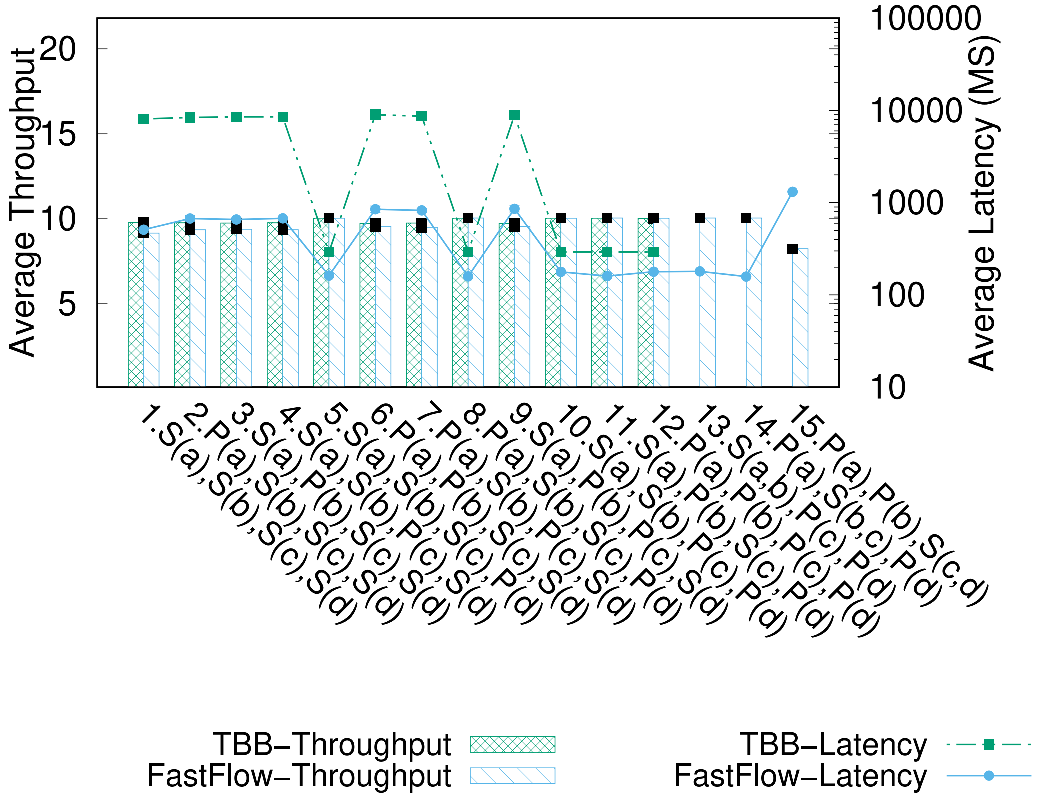


Figure 7.2: Input Rate 10 items/s.Latency is on logarithmic scale

Source: [132].

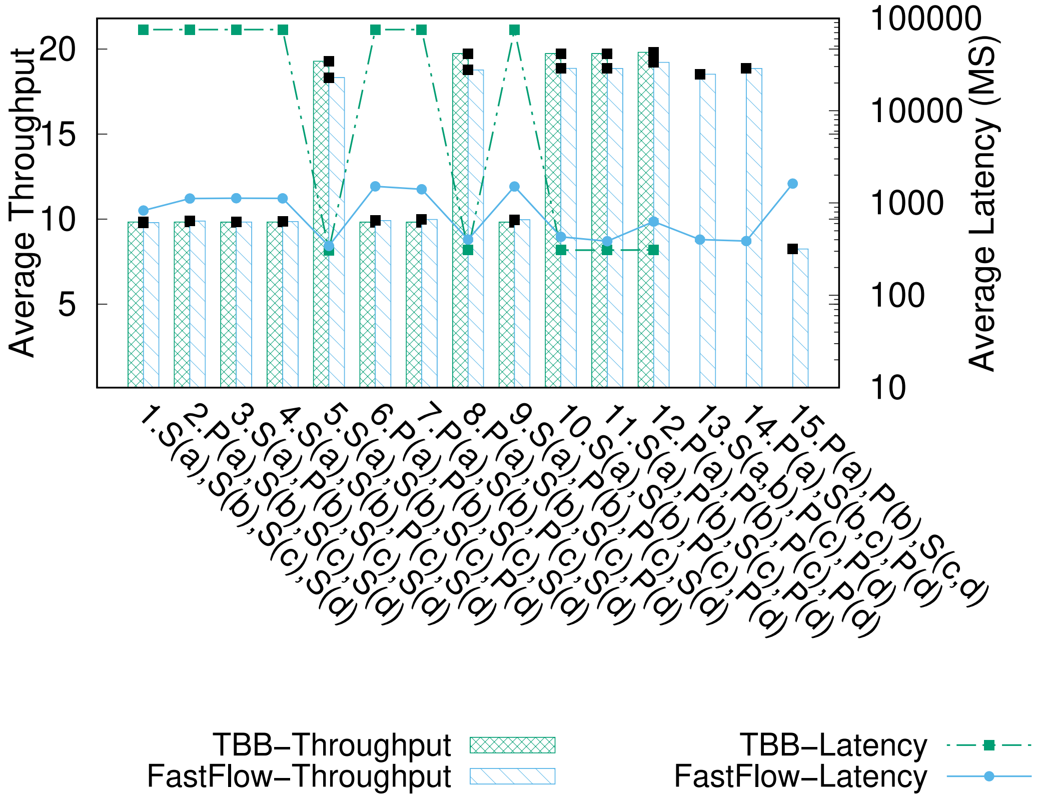


Figure 7.3: Input Rate 20 items/s.Latency is on logarithmic scale

Source: [132].

7.2 An optimized 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 6.2.3 we show an example of implementation in a C++ programming framework. The main necessary assumptions are:

- Runtime system's mechanisms are available supporting dynamic changes to configurations from one configuration to another.
- The strategy receives alternative configurations to be considered at run-time. 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.

Considering that self-adaptation is expected to encompass relevant properties, **SASO** properties, described in Section 5.2.5. The designed decision-making for the pattern composition configurations is described in a manner that abstracts lower implementation details. However, the description provided is expected to be sufficient for replicating the proposed solution. The decision-making is built out of the following steps:

1. **Online profiling step**: Lightweight instrumentation gathers execution statistics from each stage, which helps in characterizing how computationally heavy and balanced they are. The profiling step measures the actual processing capacity of each stage and ranks them by their computational weight. 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), which is a configurable parameter that the used value of 20% was ascertained in previous work [129].

3. **Shortlisting configurations**: Previous work [131] applied experimental runs with all configurations. In practice, this can affect QoS because bad configurations could be used. Considering that the new proposed strategy aims to reduce the settling time without limiting the configuration space, the profiling step's information is used to 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 parallel 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 execution statistics. The rationale for executing each shortlisted configuration is to increase the accuracy by finding the configurations that perform better for the specific application, workloads, and environments. Considering that the time interval in which each configuration is tested is a relevant parameter that expert users can customize, 5 seconds is the default value ascertained from empirical results. The previous implementation from [131] tested all configurations for only 1 second because it did not profile and shortlisted the best ones. 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 parallel replicated stages. This decision is to enforce the most optimal configuration that maintains QoS and at the same time consumes fewer resources for avoiding overshooting.

6. **Steady state**: Stabilizes in a configuration and periodically evaluates if the SLO is being satisfied. In practice, every 10 seconds, the current status is verified to maintain responsiveness to potential fluctuations/changes. 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 7.2.1 addresses the relevant aspect of how to achieve safety and stability when transitioning from one configuration to another.

7.2.1 Transitioning between configurations

Reconfigurations at run-time should be smooth such that they do not compromise the QoS. One possible 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 [101]).

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 [61] 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 weight, 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 7.1 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 system's 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 7.3.3, we characterize the transitioning between configurations using this model.

7.2.2 Solution's usability

The new optimized decision-making proposed does not differ from the aspects discussed w.r.t. the mechanism (see Section 6.2.4) from the programming usability perspective. The self-adaptive strategy proposed here is also easily integrated within the FastFlow programming framework.

From the applications' executions usability perspective, the application programmers can set their goals as execution parameters. The current decision-making strategy also supports a throughput goal (likewise previously demonstrated in Section 6.2.4):

```
-application-binary throughput 10
```

Moreover, the new decision-making strategy supports application programmers to define also latency constraints, where we discussed latency's relevance and peculiarities in Section 5.2.3. Importantly, from the applications programmers' usability perspective, it is just a matter of defining the latency as the goal and the constraint value (in milliseconds or seconds), e.g., a latency constraint of 500 milliseconds:

```
-application-binary latency 500
```

Hence, the self-adaptive strategy parses the arguments/parameters for the decision-making to pursue the user goals at run-time fully transparently. Therefore, in Section 7.3 we evaluate quantitatively the decision-making of the strategy proposed here.

7.3 Evaluation

7.3.1 Experimental setup

We used a multi-core machine equipped with two Intel Xeon E5-2620 processors (a total of 12 cores-24 threads), 32 GB of memory for running experiments, Ubuntu Server 16.04 as operating system, and G++ compiler (7.5.0) with -O3 compilation 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 checked by hashing the outputs. In Section 7.3.2 we describe the applications and configurations tested. Then, in Section 7.3.3 the decision-making is characterized with the different SLOs supported and application characteristics, Section 7.3.4 evaluates the performance of self-adaptive executions compared to baseline static executions.

7.3.2 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.

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.

Ferret

Ferret is a stream-parallel benchmark that searches for similarities on data items like audio, images, and video [79, 37]. 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 four parallel stages from Figure 7.1 are represented as user functions a,b,c,d. Moreover, the configurations from 1 to 12 explore possible combinations of sequential (S) and parallel (P) stages, whereas configurations 13, 14 and 15 cover the merging of sequential stages. Merging can be relevant when the stages are unbalanced and the lighter ones can be merged. Importantly, the self-adaptive strategy has a profiling step for characterizing the stages and their workload.

Person Recognition

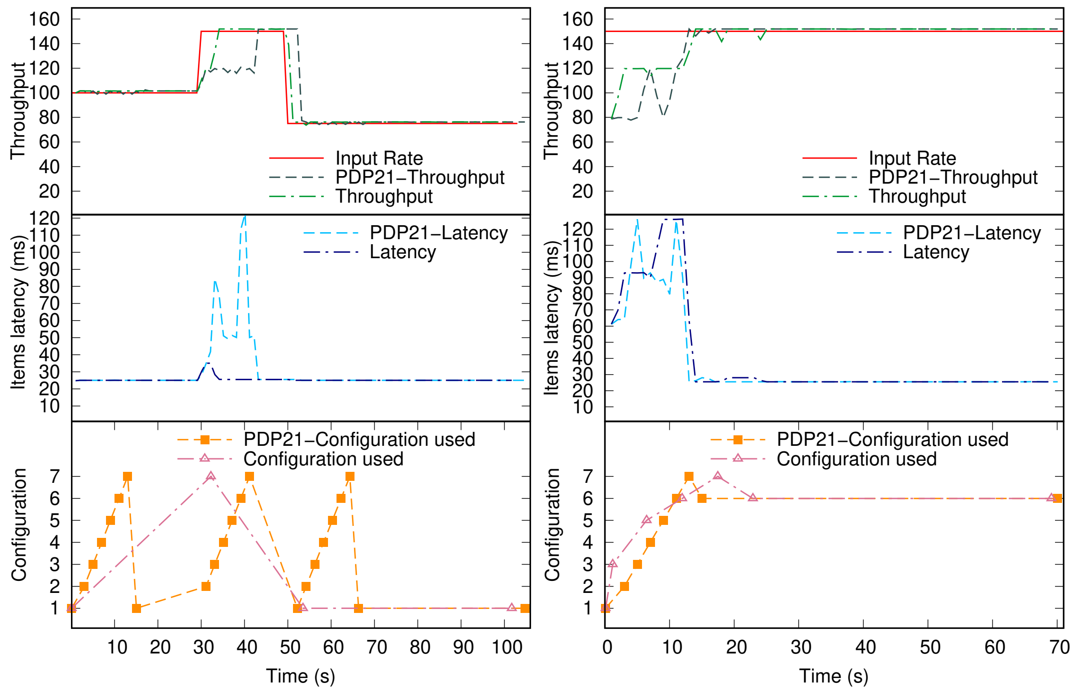
The Person Recognition is a stream processing application [49] where we used a customized version that has three functions to detect and verify people in video streams. It receives a video input and applies a denoising step for improving the quality. Then, it detects and marks the faces with a red circle. 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 [131] 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.

7.3.3 Self-adaptive strategy characterization

This section characterizes the decision-making process of the self-adaptive strategy. The first results to characterize the solution are from the synthetic application. The proposed solution is compared to the previous one called PDP21 [131]. Figure 7.4(a) shows the results of balanced application stages where the defined SLO is to have a throughput (items/s) outcome equal to the IR, where there are two changes in the IR representing fluctuations that can occur at run-time in stream processing.

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 7.2). By Reacting to the IR change around the second 30, the new strategy accurately went on one step 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 without demanding additional resources.



(a) Balanced stages.

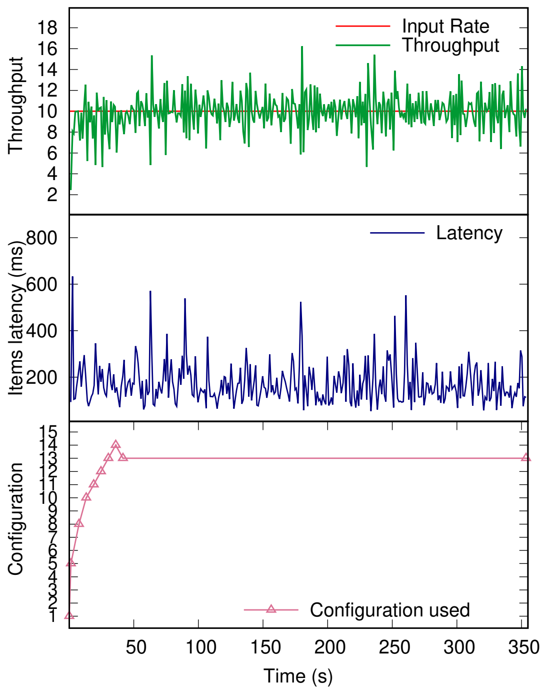
(b) Unbalanced stages.

Figure 7.4: Characterization with the synthetic application.

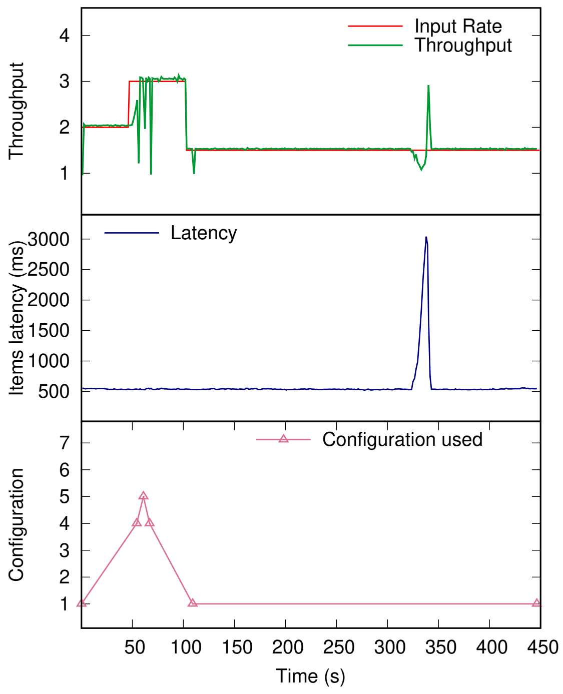
Source: [132].

Figure 7.4(b) shows a distinct outcome in a scenario with unbalanced stages. The execution starts with a throughput lower than the IR. Therefore, 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 7.3.2) 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 provides 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 real-world applications [64]. 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. Although the time interval can be customized for specific application's characteristics, five seconds is expected to be a suitable value for a wide range of applications. Another relevant aspect evinced Figures 7.4(a) and 7.4(b) concerns the transitioning model. Notably, the transitions between configurations are smooth without throughput drops or latency glitches.



(a) Ferret with IR 10.



(b) Person Recognition with a changing IR.

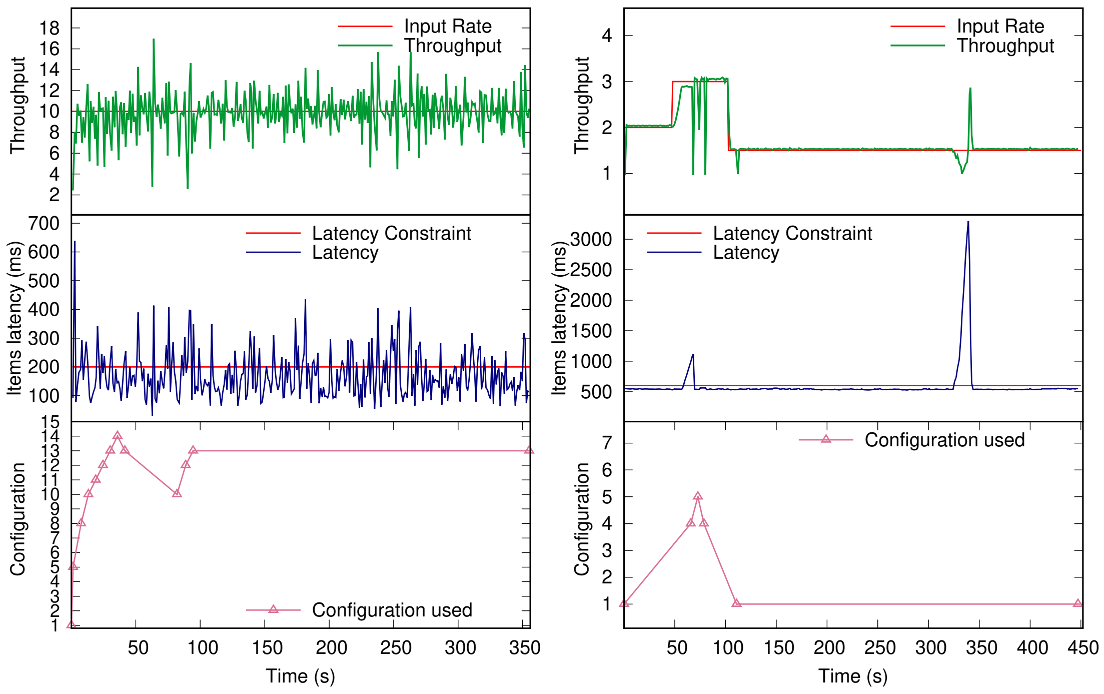
Figure 7.5: Throughput (items/s) Characterization.

Source: [132].

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. Figure 7.5(a) 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 was reduced during some reconfigurations, the transitioning model showed accuracy because there was no application downtime.

Figure 7.6(a) evinces Ferret with an SLO of 200 ms with fluctuations due to Ferret's characteristics. The strategy stabilizes with configuration 13, overcoming the bottle-



(a) Ferret SLO 200 ms. (b) Person Recognition SLO 600 ms.

Figure 7.6: Latency Characterization.

Source: [132].

neck on stage *Rank*. Near 100 seconds time, 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.

Figure 7.6(b) 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 7.5(b)) that caused the throughput to decrease and the latency to increase. This fluctuation was 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 shows that the model's estimation is accurate.

7.3.4 Performance evaluation

In this section, we compare the final performance of the self-adaptive executions to static ones using real-world applications. The results represent 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.

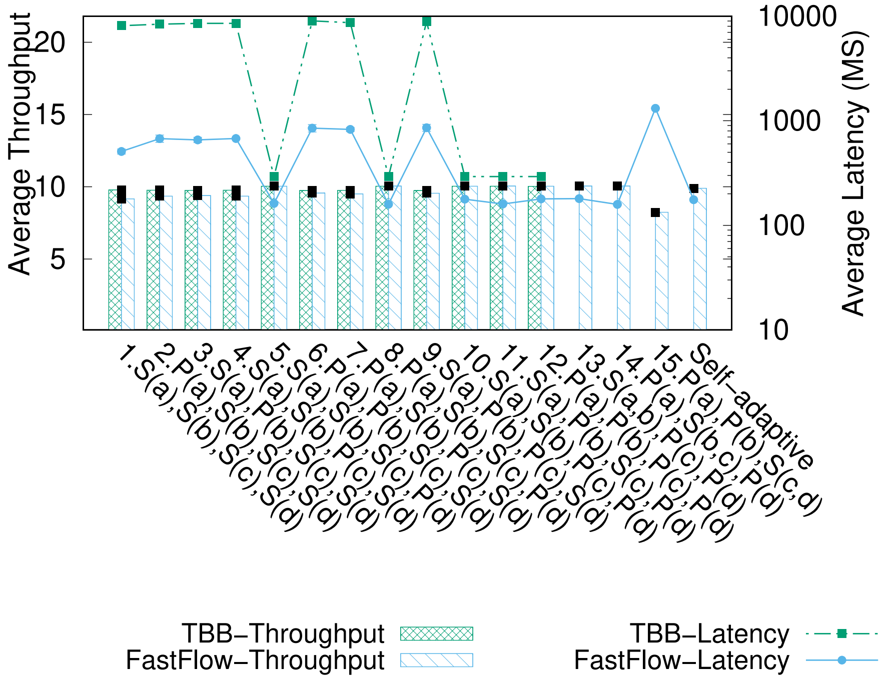


Figure 7.7: Ferret with IR 10. Latency on logarithmic scale.

Source: [132].

Figures 7.7 and 7.8 show 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.

Figure 7.9(a) and 7.9(b) 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.

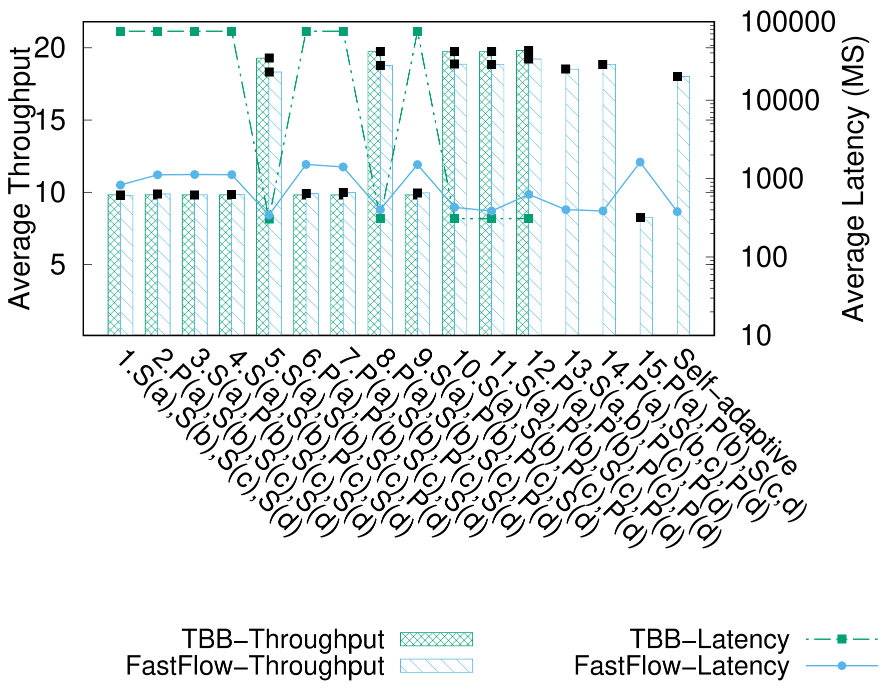
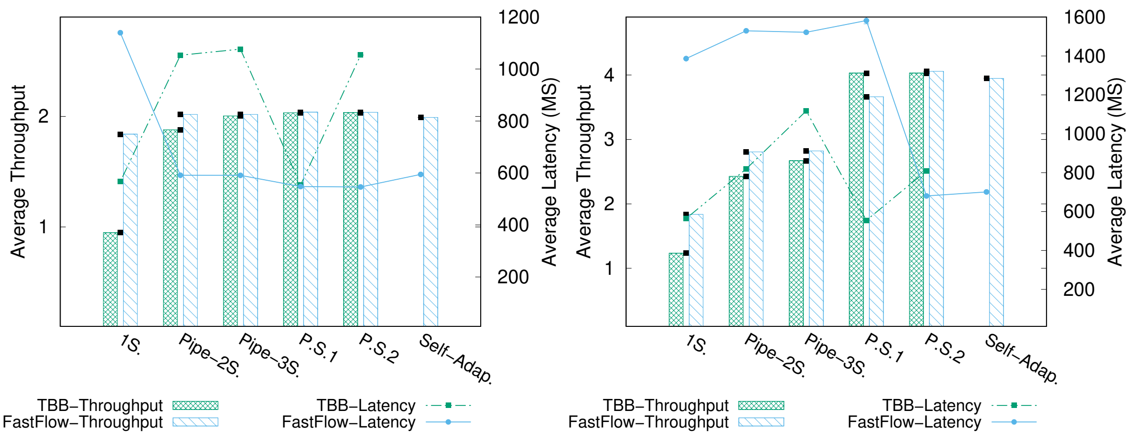


Figure 7.8: Ferret with IR 20. Latency on logarithmic scale.

Source: [132].



(a) Throughput with IR 2. (b) Throughput with IR 4.

Figure 7.9: Performance comparison with the Person Recognition Application.

Source: [132].

7.4 Remarks

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;
- can transparently react to unpredictable fluctuations (e.g., 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.
- provides a transitioning model that is sufficiently accurate as no application downtime neither latency glitches occurred due to reconfigurations (Section 7.3.3);
- has a negligible overhead of instrumentation by the fact of achieving a competitive performance (Section 7.3.4).

A relevant implication emphasized is the importance of having well-defined building blocks components as composable and nestable objects. The building blocks enable the creation of complex, well-defined structures (e.g., patterns) that we have shown to be possible to self-adapt at run-time. Moreover, the results demonstrated that self-adaptiveness could 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, the self-adaptive strategy can be generic enough to be customized with other programming frameworks and execution environments. We expect that one could apply the strategy to provide self-adaptations and abstractions for regular parallel applications.

The solution described in this chapter was designed supported by the decision-making framework from chapter 4. Hence, in addition to the parallel abstraction provided, we envision that a certain level of generalizability is achievable. The decision-making is decoupled from the mechanisms necessary to apply the changes in the specific programming framework utilized. In the next chapter we provide a novel solution for self-adapting the number of replicas in compositions with many parallel stages (e.g., Ferret), where some modules of the solution proposed here were reused in the decision-making of a different entity.

8. SUPPORTING SELF-ADAPTIVE DEGREE OF PARALLELISM IN COMPLEX COMPOSITION STRUCTURES

Previously, in Chapter 5, we presented the efforts aimed at self-adapting the number of replicas (one facet of the parallelism degree) in application with one parallel stage (AKA Farm). Moreover, in Chapter 6, we viewed that it is possible to adapt the entire application topology by online self-adapting the parallel patterns used. The previous Chapter 7 also evidenced that online self-adapting the parallel patterns is a powerful adaptation action that enables high flexibility for reconfiguring the executions to a suitable configuration.

In practice, we have seen that many real-world applications have a complex (AKA robust) composition structure (see Section 3.2.2) with multiple parallel stages (PS). In addition, Section 3.4.1 of Chapter 3 evinced that it is still a challenging scenario to prove self-adaptiveness for complex structured applications. A powerful entity that can be self-adapted is the parallel patterns. Such an entity creates an adaptation space that can be leveraged to tackle the challenge of finding the most efficient configurations, i.e., identifying user functions/stages to be merged or separated and executed sequentially or in parallel.

However, in many cases, when utilizing parallel stages, they still demand a dynamic tuning of the number of replicas. Hence, we argue that it is relevant to support the inner part of the powerful adaptation of the parallel patterns to self-adapt the number of replicas utilized within each parallel stage. In Section 8.1, we discuss the context and the related literature. Then, in Section 8.2 we describe our proposed solution that contributes with a new mechanism and decision-making strategy to support advanced self-adaptation in parallel applications. Section 8.3 discusses a comprehensive evaluation methodology. Then, in Sections 8.4, 8.5, 8.6, and 8.7 we show and discuss the experimental results. Finally, Section 8.8 provides closing remarks of this chapter.

8.1 Context

The literature revision (Chapter 3) found many solutions that seemingly adapt the number of replicas in application with complex structures. Such solutions tackled different execution environments such as multi-cores and distributed systems. A prominent solution is DS2 [64] which was proposed as a general controller for making scaling decisions in distributed stream processing. Noteworthy, DS2 was compared and significantly outperformed Dhalion [40] in the distributed stream processing context.

8.1.1 DS2's Decision-making

Abstracting the mechanism implemented within DS2, a relevant part is its decision-making system that collects performance traces and estimates in at most three steps which configuration should be employed on each parallel stage.

The estimation of the optimal number of replicas on each parallel stage considers a performance model built by monitoring the processing capacity of each stage. Then, DS2 builds a relationship between each stage's processing speed and the target performance. DS2 estimates how many replicas to add if the actual processing capacity of a given stage is below the target performance.

A complementary view of DS2's decision-making in distributed environments can be seen in Figure 8.1, showing that the stage O_1 is the bottleneck by computing only ten tasks/items per second where the target performance of to process 40 Items per second (I/s). Consequently, DS2 detects this bottleneck and increases four times the number of replicas of stage O_1.

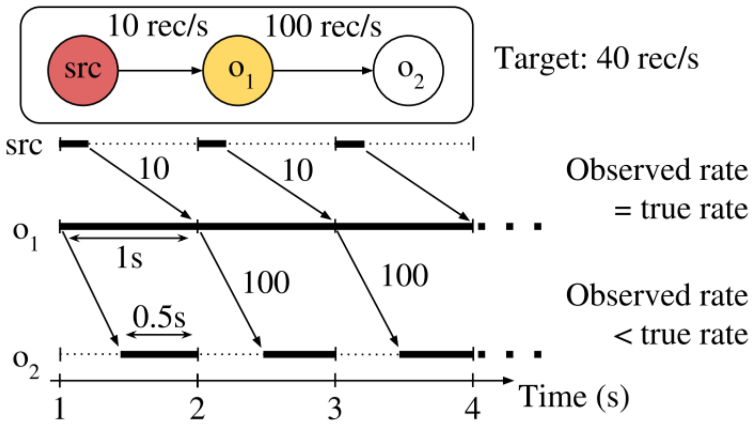


Figure 8.1: DS2 representation.

Source: Extracted from [64].

8.1.2 Potential limitations of DS2's decision-making

Although DS2 is an approach that works well on distributed environments, several concerning aspects emerge when looking from a conceptual perspective of generalizing and widely applying DS2's decision-making:

- it focuses only on distributed scaling, the useful time metrics considered like serialization and deserialization are not generalizable nor representative for other environments like multi-cores and manycores.

- the decision about new configurations performed on each operator without coordination is arguably limited. When considering the context of adapting distinct parts (e.g., stages, entities) of a given software system, our understanding is that a compromise between an inflexible centralized decision-making systems and the fully independent decision-making concerning correlated parts. For instance, in multi-cores where the availability of the resources is a constraint, there could be need to implement coordination between decentralized decision-making entities. We believe that further improvements are possible with certain level of coordination when deciding configurations for different parallel stages.
- although the DS2 is described as running under a low instrumentation overhead ranging from 13% to 20% in distributed environments, it is arguable too high for achieving efficiency and high-performance on multi-cores. In practice, optimizing the decision-making can help in reducing even instrumentation overheads.

Section 8.2 describes a new solution to contribute with a self-adaptive number of replicas in complex structured applications running in multi-cores. The proposed solution comprises a new decision-making strategy and a mechanism for applying adaptation in a shared-memory programming framework. In Section 8.3, we introduce an evaluation plan of our proposed solution using this mechanism compared to a version of DS2's decision-making replicated to our context.

8.2 Proposed Solution

The decision-making strategy is one of the main parts of the proposed solution and also encompasses the SASO properties [56]. Here we describe the solution abstracting lower implementation details. However, this description is intended to be sufficient for replicating the proposed solution. Moreover, considering that we focus on the generalizability of our solutions, modules that were proposed and validated in other contexts were reused and applied to solve the specific challenges tackled here. Noteworthy, the decision-making utilizes the following existing modules:

1. **Online profiling step**: was proposed in Section 7.2 and published in reference [132]. By reusing such a generalizable module, our solution applies a profiling step for characterizing the computational weight of the applications' stages.
2. **Monitor**: It is a module utilized in all solutions proposed and described in Chapters 5, 6, and 7. The monitor module was used here for the same purpose of collecting performance traces of the applications.

Moreover, the decision-making executes within a manager entity, similar to the previous solutions. However, here the decision-making strategy utilizes a different decision algorithm. Considering the data gathered by the profiling step and monitoring modules, decision-making builds a relation between the heavy stages and the actual performance goal (ex: SLO). By applying this relation, the decision-making estimates if and how many replicas are ideal for each parallel stage. However, before applying it, an additional monitoring and analysis step enforces coordination between all theoretically independent parallel stages. It is a training step that compares the total number of replicas to the actual computing capacity of the running machine. The decision-making reduces in a balanced way (w.r.t. the optimal value for each parallel stage) the number of replicas to avoid resources contention [7, 99, 128].

The overall training step runs for a given period (e.g., one second) and, in one step, determines the optimal number of replicas for each parallel stage. Then, the strategy enters a steady-state (similar to the one proposed in Section 7.2 and published in reference [132]) where adaptation actions are only performed in case of changes.

Considering that the instrumentation overhead is a potential limitation of DS2, our solution encompasses all knowledge and good practices made when applying self-adaptation to one parallel stage and parallel patterns. Using this existing knowledge of C++ solutions targeting multi-cores, we achieved a negligible monitoring overhead by collecting and filtering data in hundreds of nanoseconds [125] and optimizing the decision-making strategies in such a way that final performance in comparison to the best static cases [126]. The final result is lightweight instrumentation for efficient execution in multi-core machines [132, 129].

In addition to a potentially highly optimized decision-making strategy, there is also a relevant need for mechanisms for applying the adaptation actions and for communication and synchronizing all the entities' execution concurrently in complex structured parallel applications. We utilized once more the flexibility of the FastFlow framework for implementing and validating our solution. The mechanisms were integrated into the runtime system, and communication channels were implemented to enable the manager to send the adaptation commands to the parallel stages.

A representative example of such a solution is provided in Figure 8.2, where the decision-making manager is one additional runtime node: the “Manager" node. Figure 8.2 illustrates a regular parallel application with several parallel stages and with 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. Importantly, it is expected to work as long as the manager node is able to send adaptation commands to the parallel stages. The manager sends commands to the parallel stages to adapt (i.e., increase or decrease) their number of replicas.

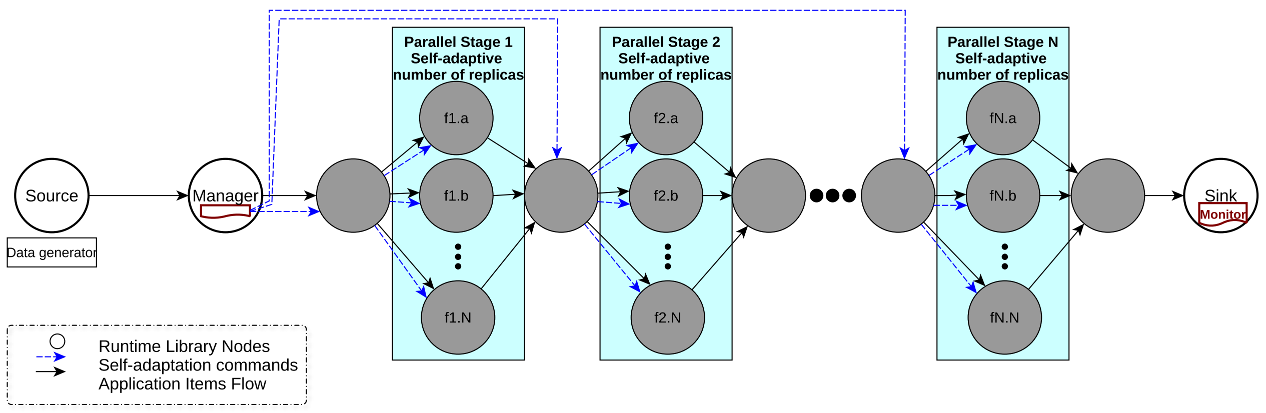


Figure 8.2: Integrated of the proposed solution within the FastFlow framework.

When focusing on the proposed solution from a broader perspective, the composition represented in Figure 8.2 can be viewed as a Pipeline with multiple parallel stages (Farms). Although the representation evinced in Figure 8.2 supports several parallel stages, it can be seen as one of those configurations described in Chapters 6 and 7. In fact, the composition shown in Figure 8.2 is represented in Figure 6.1 from Chapter 6 by the *Configuration 5*, which can be created at compiling time and used at run-time. 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.

The solution proposed here was designed to this work's specific scenario. However, the design of the proposed solution was inspired in previous works and related approaches, such as [128, 126, 132, 7, 89, 36]. The following section provides a comprehensive evaluation of the proposed solution described here.

8.3 Experimental Plan

The solution proposed in Section 8.2 is evaluated here. This proposed solution is compared to a relevant solution from the state-of-the-art called DS2 that was described in Section 8.1.1.

Although there are aspects of DS2 that do not apply to the multi-core scenario, e.g., serialization and deserialization times, we reproduced the DS2's decision-making strategy following its description from reference [64]. Considering that DS2 decision-making relies on the actual processing capacity of each stage, we implemented monitor modules similar to the ones of our solution on each parallel stage. Importantly, we also incorporated our solution's good practices and low overhead mechanisms into the reproduced DS2 version. Thus, the main differences in the DS2 version are: I) the need for additional monitor modules, II) the decision-making strategy that runs in a manager node as our proposed solution, but we will see in practice how the DS2's decision-making principles work in our context.

Another relevant aspect is that reproducing existing solutions comes with several inherent limitations. In addition to the elements mentioned above that are generics and do not apply to our context, every description and possible interpretation of a given solution is not always complete and precise. For instance, although DS2 promises to converge to an optimal solution in one step (as desired for limiting the settling times), in reference [64] it is stated that DS2 may need more steps to find an optimal configuration. However, here, we are unable to reproduce such additional steps because in reference [64] the decision-making descriptions of these steps is omitted. Thus, the reproduced version of DS2 follows the available, but there are aspects about DS2's reproduced version that can be further considered in the future.

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.

8.3.1 Experimental Setup

We executed the experiments firstly in the same multi-core machine utilized in previous chapters and sections called here M1, equipped with two Intel Xeon E5-2620 processors (a total of 12 cores - 24 threads), 32 GB of memory for running experiments. M1 runs with Ubuntu Server 16.04 and G++ compiler (7.5.0). We also provide some complementary results from another machine, called M2, equipped with two Intel Xeon Silver 4210 (a total of 20 cores - 40 threads) and 64 of memory. M2 runs with Ubuntu Server 20.04 and G++ compiler version 9.3.0.

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

Regarding the FastFlow runtime system parameters, the threads/nodes were configured without any custom pinning policy[1] so that the OS's scheduler can allocate the threads in the cores. The rationale behind this choice is that we focus on evaluating the impact of adaptations at the system/application level. Hence, less flexible and loosely generalizable low-level optimizations are not customized in our experiments. The main concern is to assure similar conditions for fairness in quantitative comparisons.

[1]Generally, custom low-level policies can be designed to optimize the mapping of processes or threads to physical cores. Such optimization can offer performance gains in some scenarios by improving local memory accesses and cores affinity.

Yet regarding the runtime system, FastFlow provides two runtime communication behaviors: non-blocking and Blocking. The non-blocking is the default one of the FastFlow framework, and it works in such a way that nodes/threads continuously perform push or pop operations in the communication queues. This communication behavior can improve performance but usually increases resources utilization. On the other hand, the Blocking mode is promising customization that tends to consume fewer resources, where the executing nodes/threads remain blocked while they have no items/tasks to compute. Both communication behaviors are covered in our experiments. The rationale for considering these two communications is to extend the evaluation of the programming framework utilized and further assess the impact of the decision-making strategies in the performance metrics and resource consumption.

It is important to note that the experiments provided in this section intend to evaluate the impact of the decision-making strategies in QoS and system efficiency. Throughput is considered the most relevant metric. In these experiments (as were previously seen in Chapters 6 and 7), the data items to be processed arrive at a given speed supported by a data generator module. Consequently, the target throughput is defined as an SLO with the same value as the input rate (IR). Notably, several input rates are tested, and such values were set to be representative of the applications and machines used.

8.3.2 Applications

The first characterization and performance results concern executing a simple synthetic application where 50,000 tasks are processed. The Synthetic application implements the same application used previously in Sections 6.3 and 7.3.2. However, the version used here relates to *Configuration 7* described in Section 7.3.2 that has all computing stages running in parallel. However, in contrast to the previous one, the application version utilized here was modified to support self-adaptation in the number of replicas within each parallel stage accordingly to the decision-making strategy and the mechanism proposed in Section 8.2.

This synthetic application allows several customizations relevant to simulate representative behaviors. For instance, a parametric version of this application allowed us to set a different number of stages, customize the computing weight to each parallel stage, different data input rates, and target performance in terms of throughput. Importantly, this flexible parameterization space makes it possible to evaluate the decision-making strategies under representative real-world scenarios. The training step set for this application was one second. Moreover, in the synthetic application the buffer sizes of the runtime system used a maximum length of 20, which is a value for balancing between throughput and latency. Noteworthy, having queue space available does not mean that more items/tasks will be enqueued and buffered. In practice, the items only remain in the queues when the application output processing (measured throughput) is lower than the input rate.

In addition to the comprehensive characterization provided with the synthetic application, we also considered Ferret (described in Section 7.3.2) that is a more realistic application. Ferret's original version has four threads pools that run in parallel, meaning that Ferret has four parallel stages. Ferret's unstable workload was notable in Section 7.3.3, even without using parallel stages, so we increased the training step to 5 seconds to attempt a more reliable training step [64]. Moreover, the buffer sizes of the runtime system were set to a maximum length of 10, trying to limit buffering and instability. Moreover, Ferret used as input the PARSEC native that is a representative workload.

8.3.3 Experiments roadmap

In this chapter, we present the most relevant and insightful results. Figure 8.3 illustrates the organization of the experimental results displayed here. The majority of results are from the characterization executed in M1.

In Section 8.4 we present the first results from customization of the synthetic application to simulate an application's two parallel stages. Experimental scenario (ES) 1 shows the results of a case where the two parallel stages are balanced. Considering that most real-world applications have unbalanced stages, the ES 2 and 3 present scenarios that will test the self-adaptation strategies under applications that have stages unbalance between their stages. The rationale for such scenarios is to evaluate if the strategy can detect the unbalance and optimally estimate the resources needed on each stage.

Additionally, Section 8.5 presents a representative scenario where the application has four parallel stages, which is expected to be a more challenging scenario for the self-adaptive strategies. Because of the higher number of stages and the potential unbalance and higher resources consumption that in theory requires a better resources management achievable with optimal decision-making w.r.t. the configurations to be applied. ES 4 and 5 cover scenarios where the first and last stages are the bottlenecks, and the middle ones are lighter. Then, ES 6 assesses a more challenging scenario where two stages are the bottleneck, and there is still unbalance between the two bottlenecks compared to the lighter ones.

Many other customizations could be implemented with the flexible Synthetic application. However, we believe that these six scenarios are enough for a starting point to evaluate the self-adaptive strategies' decision-making. Moreover, we extend the evaluation with a real-world application presented in Section 8.6 that corresponds to the ES 7.

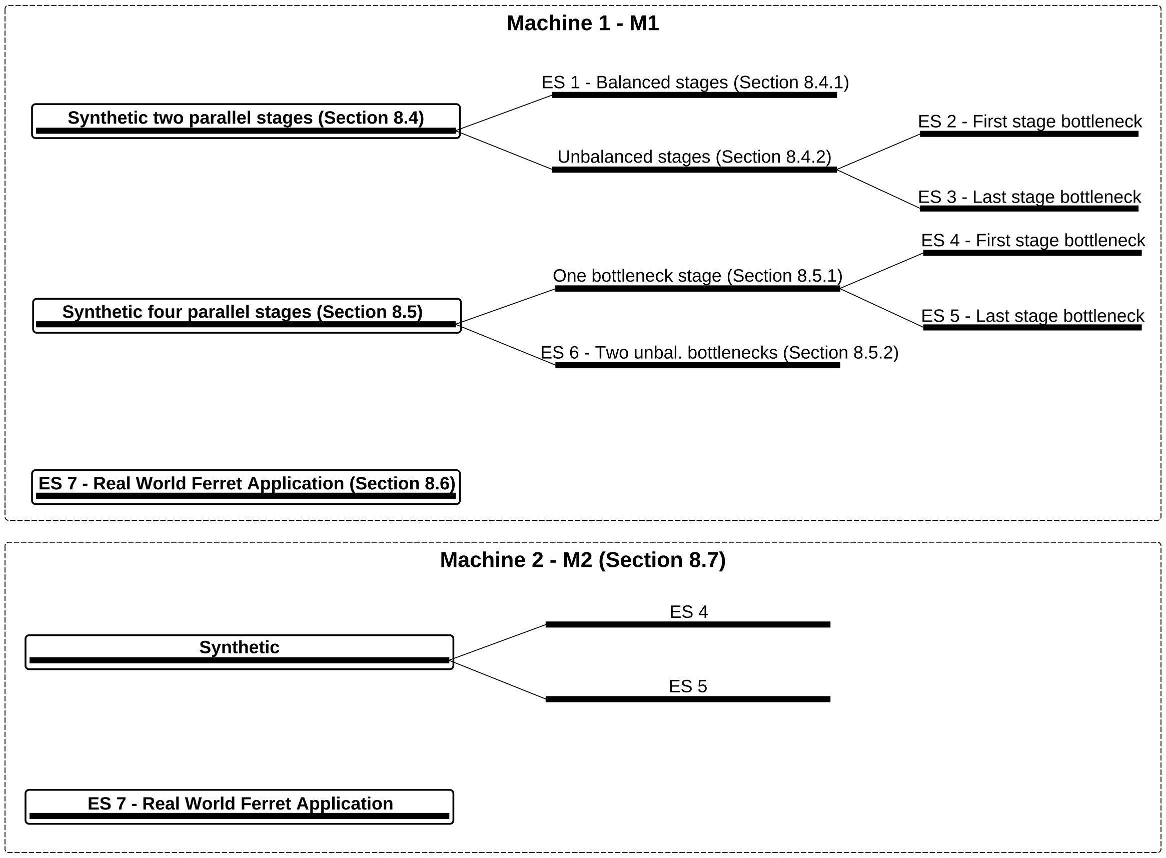

Machine 1 - M1
Synthetic two parallel stages (Section 8.4)
ES 1 - Balanced stages (Section 8.4.1)
Unbalanced stages (Section 8.4.2)
ES 2 - First stage bottleneck
ES 3 - Last stage bottleneck
Synthetic four parallel stages (Section 8.5)
One bottleneck stage (Section 8.5.1)
ES 4 - First stage bottleneck
ES 5 - Last stage bottleneck
ES 6 - Two unbal. bottlenecks (Section 8.5.2)
ES 7 - Real World Ferret Application (Section 8.6)
Machine 2 - M2 (Section 8.7)
Synthetic
ES 4
ES 5
ES 7 - Real World Ferret Application

Figure 8.3: Experiments' roadmap. ES means experimental scenario.

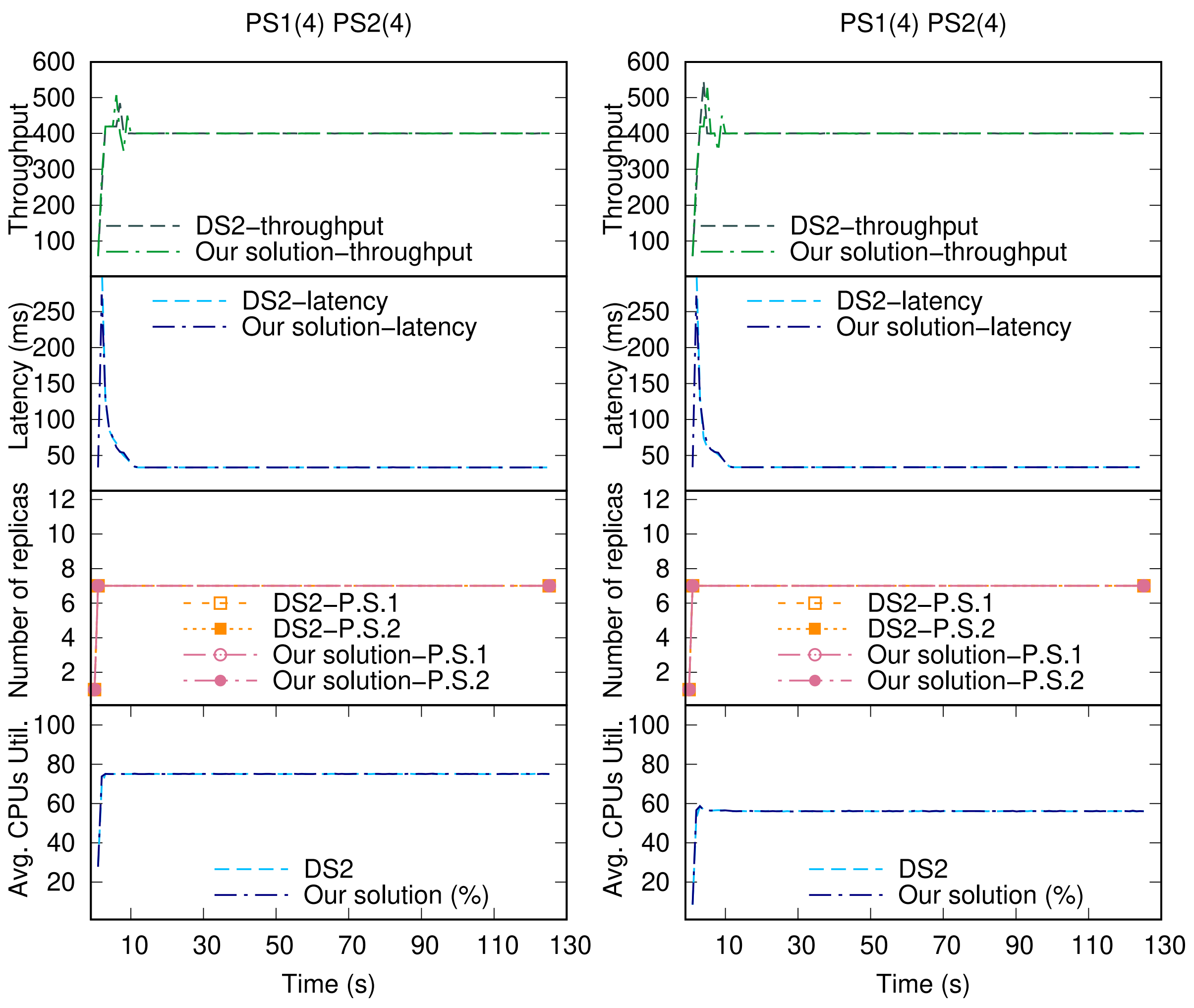
To assess if the decision-making strategies work consistently in different machines architectures, we also include in Section 8.7 results from execution in the M2. For the sake of conciseness, we present and discuss some insightful results from scenarios replicated in M2, which are expected to be enough to extend the analysis and demonstrate the consistency of the decision-making strategies under different architectures.

8.4 Evaluation with two parallel stages

8.4.1 ES 1 - Balanced stages

Figure 8.4 shows the first experimental results in a synthetic application scenario with two parallel stages. Each stage has a weight of four milliseconds (ms), meaning that it is a scenario where the stages are perfectly balanced.

It is important to note that the 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



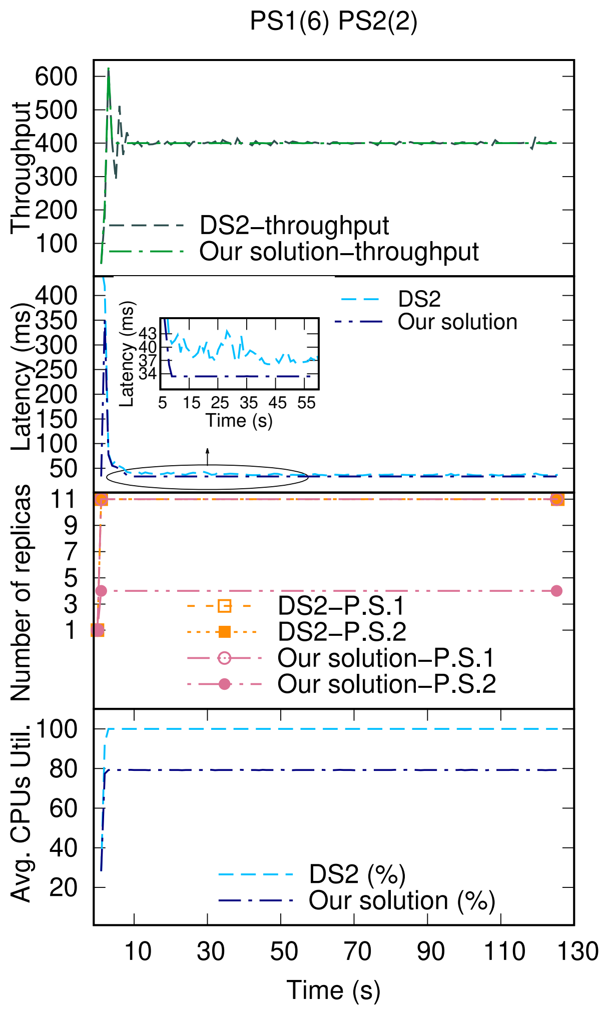
(a) Non-blocking Mode. (b) Blocking Mode.

Figure 8.4: Synthetic App With IR and Target Throughput of 400 I/s.

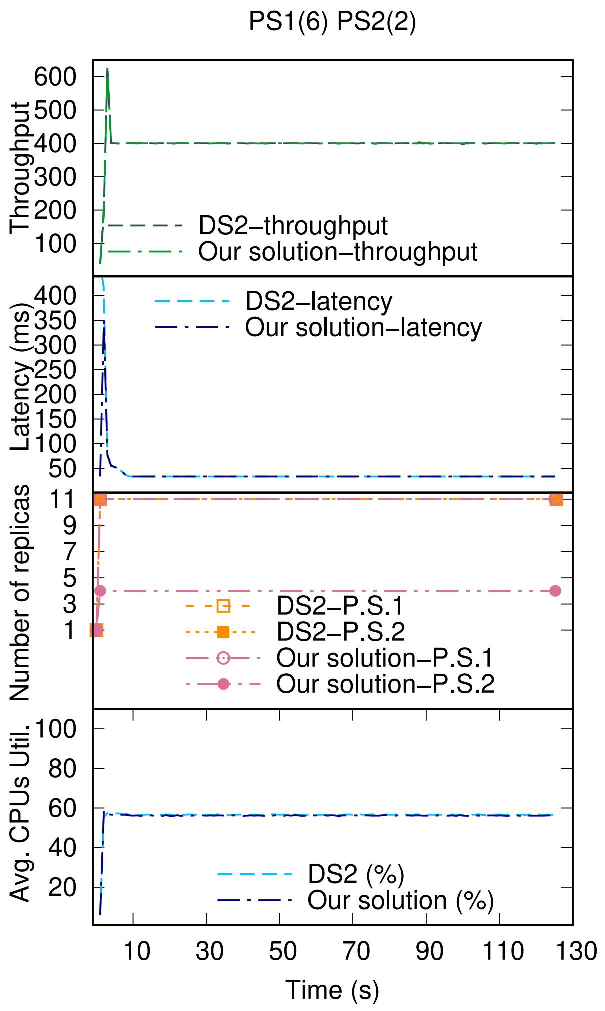
(lower part of Figure 8.4(b)). The two strategies after the training step inferred that seven replicas were suitable for achieving QoS regarding the decision-making strategies. Hence, both strategies achieved similar performance. This outcome is representative of scenarios where the parallel stages are 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 not usual because each stage performs specific computations that cause contrasting computational weights.

8.4.2 ES 2 and 3 - Unbalanced stages

Figure 8.5 presents a scenario where the first parallel stage has a weight of 6 ms and the second a weight of 2 ms. On the one hand, DS2 configured each parallel stage to execute with eleven replicas. On the other hand, 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 execute with four replicas. The main implication of our solution using fewer threads is in terms of resources consumption and efficiency, where it is possible to note in the lower part of Figure 8.5(a) that our solution is more efficient, having a CPU utilization more than 20% lower.



(a) Non-blocking Mode.



(b) Blocking Mode.

Figure 8.5: Synthetic App With IR and Target Throughput of 400 I/s.

Our solution achieved a throughput that is even stabler than DS2's throughput in some cases. Moreover, the DS2 decision to utilize more replicas causes instability and performance losses. For instance, when zooming the latency of the 55 seconds 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, which corroborates the known efficiency of the blocking mode [119].

We conducted a further analysis w.r.t. the results from Figure 8.5 to understand the reasons behind the DS2's lack of capacity to detect that the first stage was the bottleneck. Our understanding is that 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. In short, if the first stage is the bottleneck, the subsequent one will have a similar output capacity because they can not process tasks before they pass the bottleneck. Finally, our solution using profiling showed to be better to detect this complex scenario by making a decision based on measurements of the actual service times of each stage.

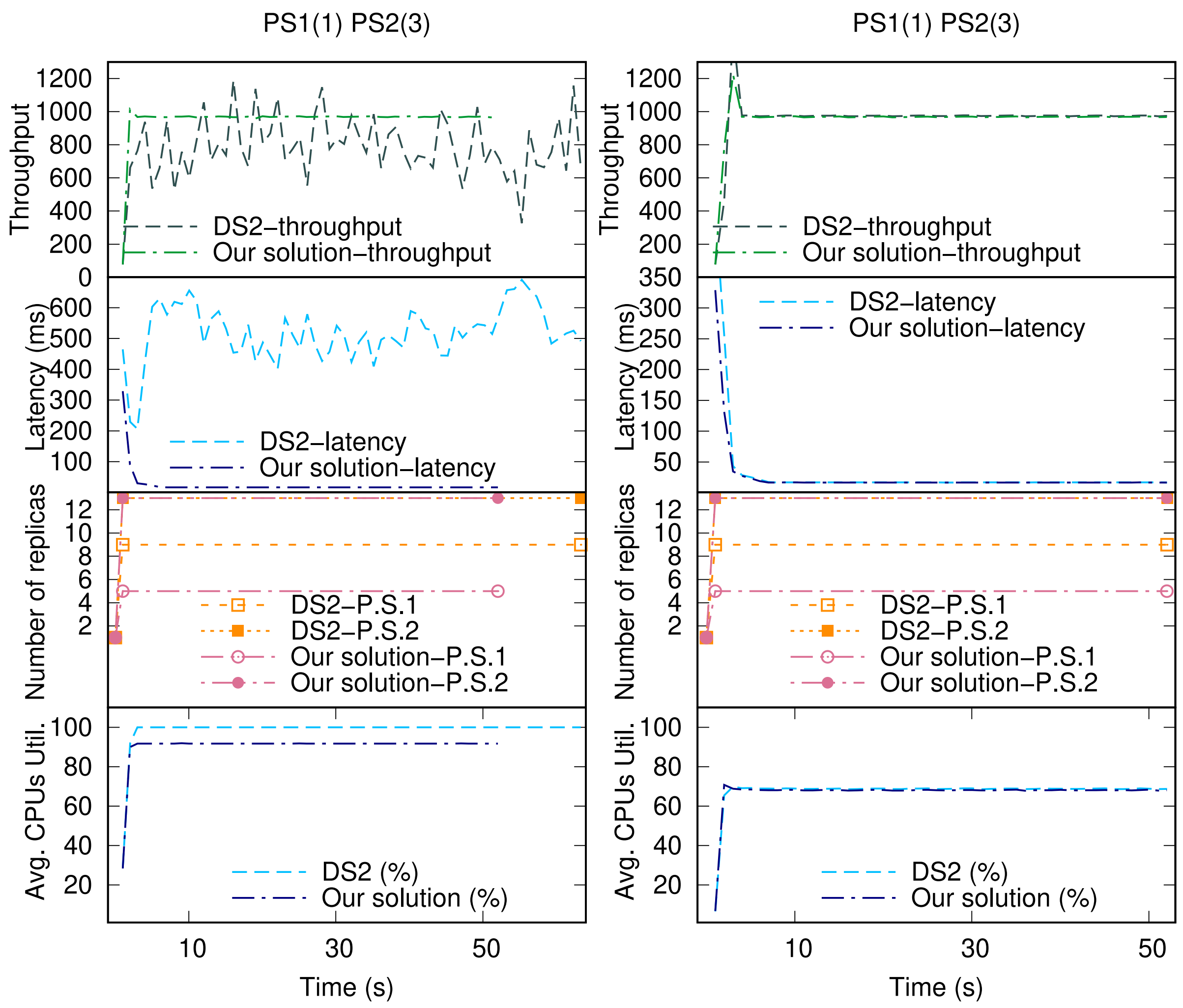
Figure 8.6 introduces another representative scenario where a high throughput (1000 I/s) is a required throughput SLO with lighter stages[2]. However, contrasting with the results from Figure 8.5, in Figure 8.6 the last stage is the bottleneck one. In Figure 8.6, DS2 detected that the las stage was the bottleneck and determines that thirteen replicas were a suitable value. DS2 sets the first stage to run with nine replicas, while our solution based on profiling estimates five replicas as appropriate value for the first stage.

Figure 8.6(b) evinces that both solutions coped with the IR in the Blocking mode and achieved similar performance. This outcome implies that our decision-making strategy correctly inferred that five replicas were enough for the first stage as this value did not make the first stage a bottleneck.

Figure 8.6(a) relates to the non-blocking mode showing a distinct performance trend compared to Figure 8.6(b). In this scenario, the additional (seemingly 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 to 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 the parallel stages. Figure 8.6(a) 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 8.7 shows a scenario that shares some similarities to the one seen in Figure 8.5. Still, here it is being simulated an application scenario where higher throughput is aimed, and the stages are lighter. Figure 8.7(a) 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 replicas that avoid resource contention by activating the actual necessary number of replicas. Moreover, the position of the bottleneck and the lighter stage is inverse in Figure 8.7

[2]The main focus is to compare the strategies’ decision-making. In some figures, the scale is different between the non-blocking and blocking modes, which was necessary to improve the visual clarity.



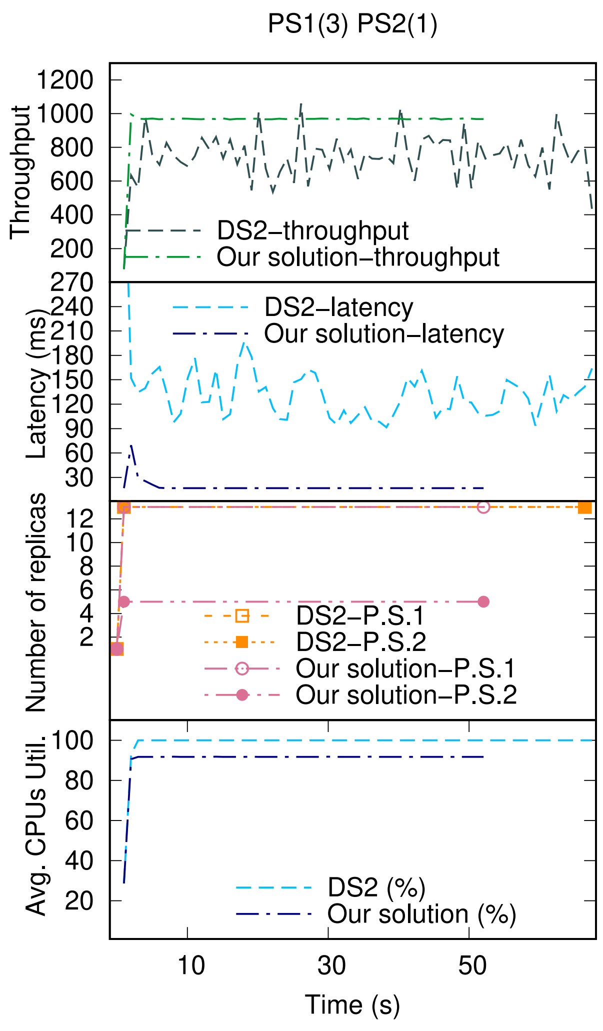
(a) Non-blocking Mode.
(b) Blocking Mode.

Figure 8.6: Synthetic: Unbalanced Stages. IR and Target Throughput of 1000 I/s.

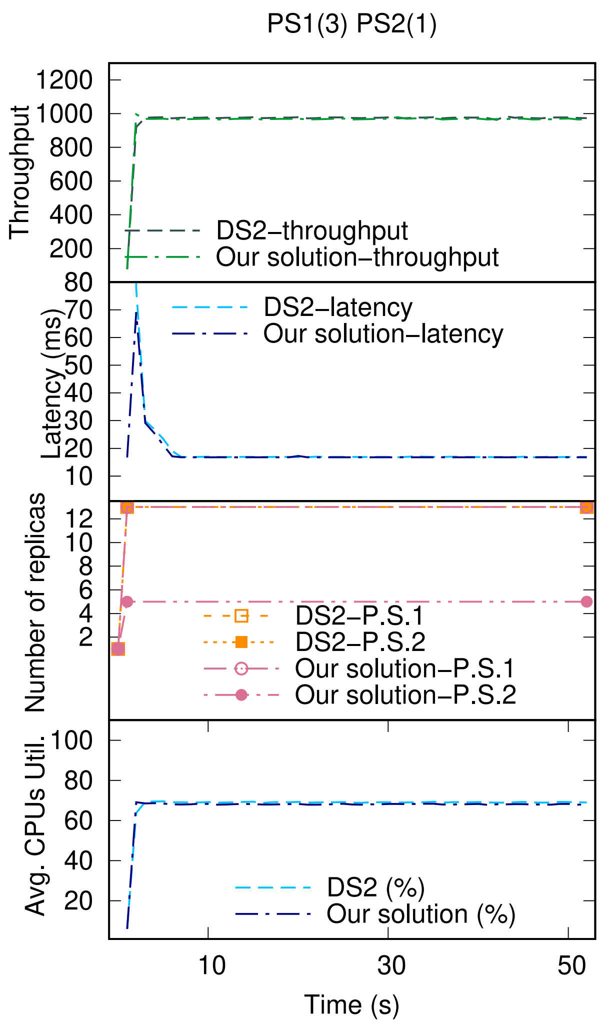
compared to Figure 8.6. Still, our proposed solution was able to estimate the number of replicas for each parallel stage optimally.

In Figures 8.8 and 8.9 an even higher throughput is aimed to generate a high load and extend the strategies' evaluation. Figure 8.8 shows the scenario where the last stage is the bottleneck. The non-blocking mode (Figure 8.8(a)) shows a similar trend seen in previous experiments and corroborates that our solution's decision-making outperforms DS2. A notable new aspect is a higher throughput instability caused by the full utilization of physical cores and Hyperthreads that results in fluctuations [128]. Although consuming 100% of resources, our solution stills outperform DS2 in terms of throughput and latency.

Concerning the blocking mode, Figure 8.8(b) shows a higher throughput and stabler executions than the non-blocking mode. Both decision-making strategies achieved a similar throughput plateauing when reaching the input rate speed. Still, our solution can achieve some minor latency gains exemplified in the zoomed part of the latency.



(a) Non-blocking Mode.

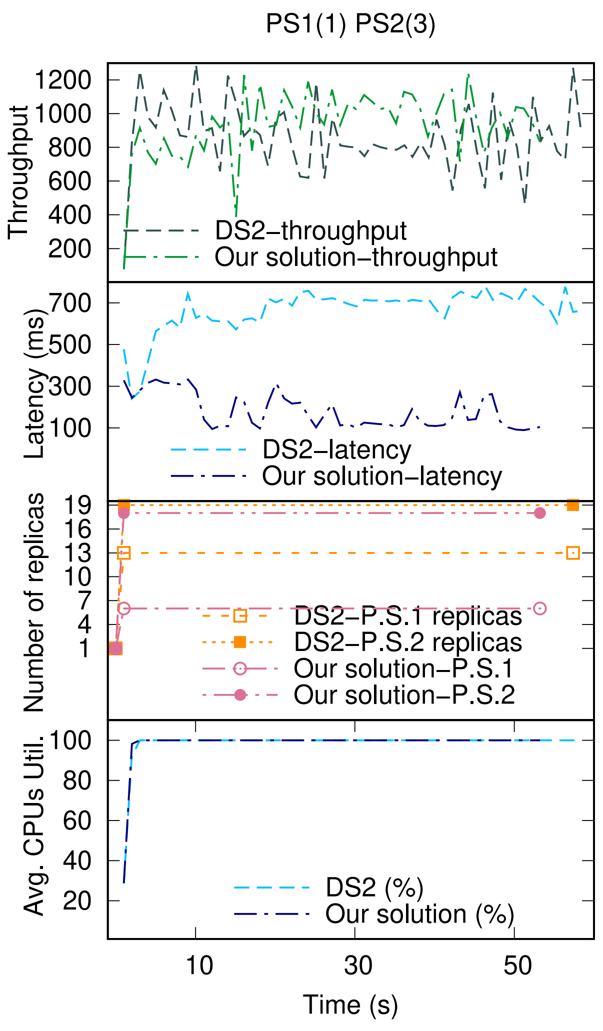


(b) Blocking Mode.

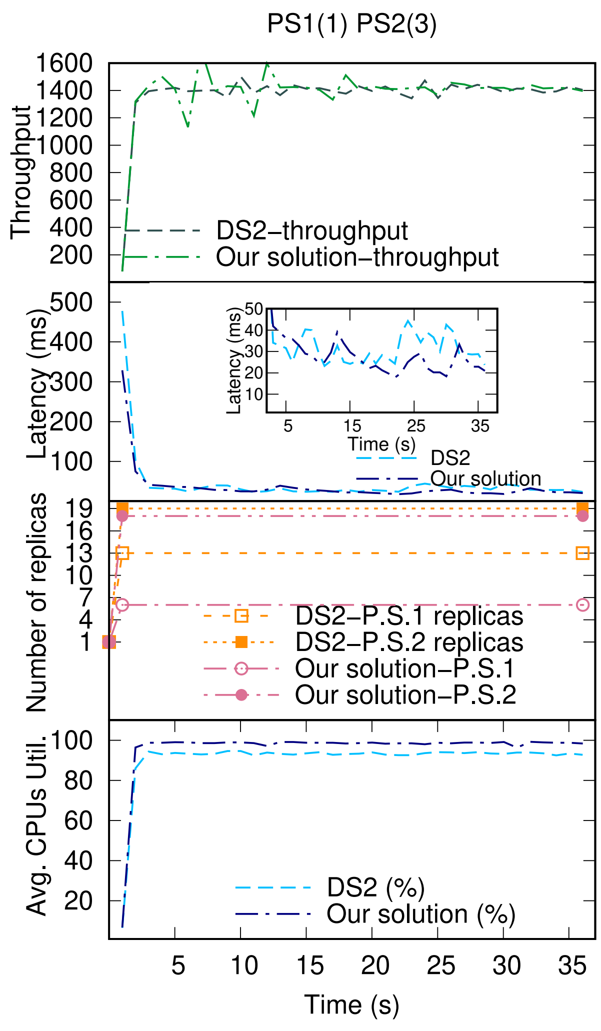
Figure 8.7: Synthetic: Unbalanced Stages. IR and Target Throughput of 1000 I/s.

Figure 8.9 presents a case where the target throughput is high (at least for the running machine), and the first stage is the bottleneck. The non-blocking mode from Figure 8.9(a) evinced an overall fluctuating throughput and latency due to fully utilizing the machine's processing capacity. Regarding the self-adaptive strategies, our solution enforces the utilization of fewer replicas, eighteen in the first stage (bottleneck) and six in the second stage, resulting in twenty-four replicas that is the exact value of the total cores count of the machine. This decision was enforced by the training step (see Section 8.2) that attempts to avoid resources contention by fairly reducing the number of replicas when they exceed the machine's resource availability.

A practical implication of this optimized decision-making of our solution is the performance gains notable in Figure 8.9. In the non-blocking mode, although our solution also utilized 100% of the CPUs, it achieved an overall higher throughput and a lower latency. Moreover, in the blocking mode, our solution achieved a higher throughput. In terms of latency, it is notable that the results present fluctuations. The results indicate that our



(a) Non-blocking Mode.

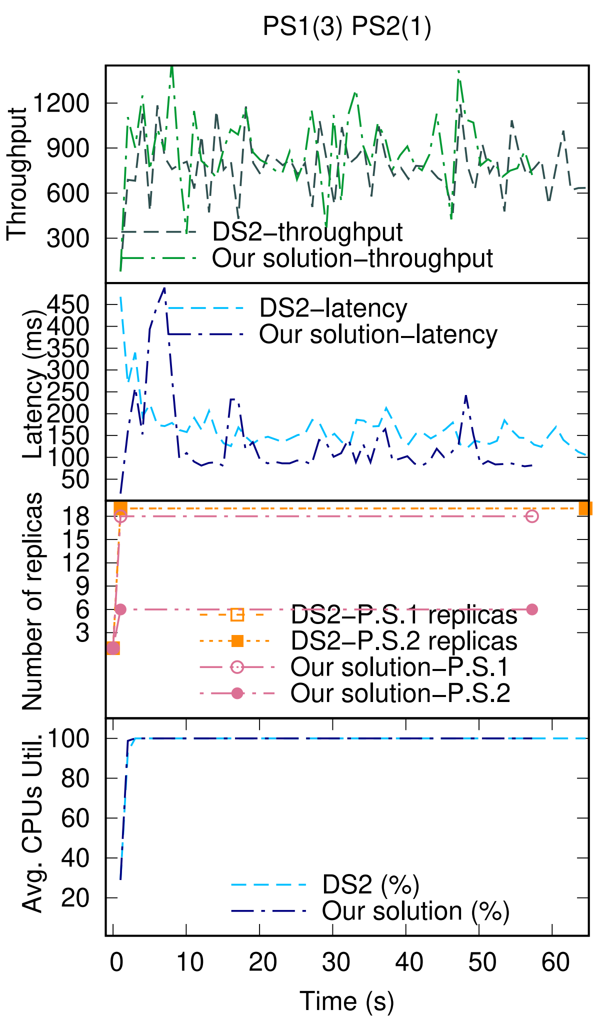


(b) Blocking Mode.

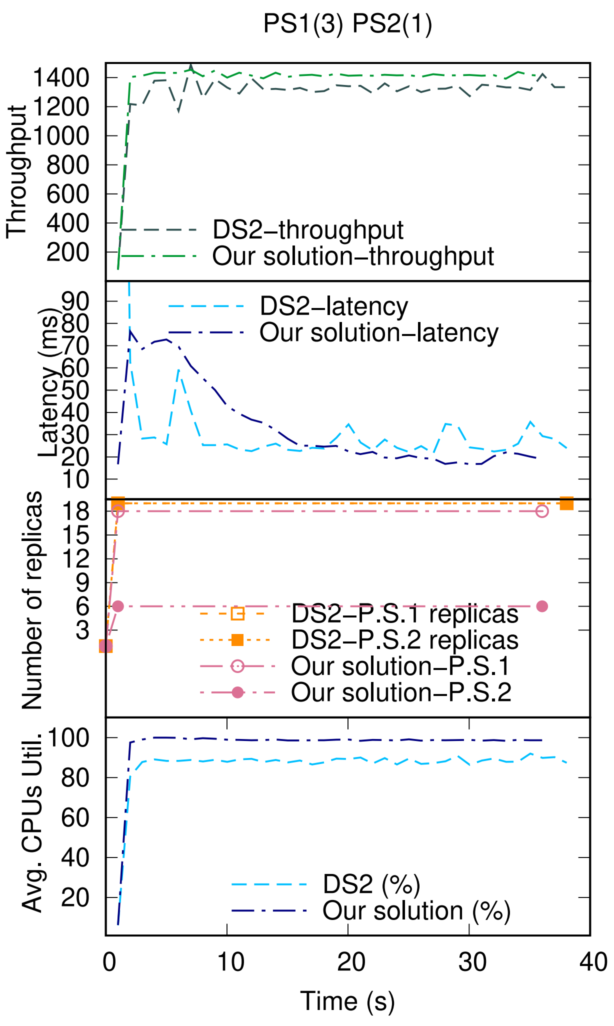
Figure 8.8: Synthetic: Unbalanced Stages. IR and Target Throughput of 1500 I/s.

solution had a higher buffering of items in the first part of the execution, which increased the latency to achieve high throughput. Then, our solution's execution could cope with the IR and reduce the latency to achieve the lowest values.

It is also notable that Figure 8.9(b) shows our solution consuming more CPU resources than DS2. One may argue that this is not intuitive as our solution utilized fewer replicas. However, a given replica running in the blocking mode tends only to consume resources when there are items/tasks to be processed. Consequently, the fact that DS2's execution did not consume the total resources available indicates that it is due to the imbalance of stages. Our understanding is that in DS2, the second (lighter) stage was mostly idle while the excessive number of replicas spawned competed for the processing resources needed in the bottleneck stage.



(a) Non-blocking Mode.



(b) Blocking Mode.

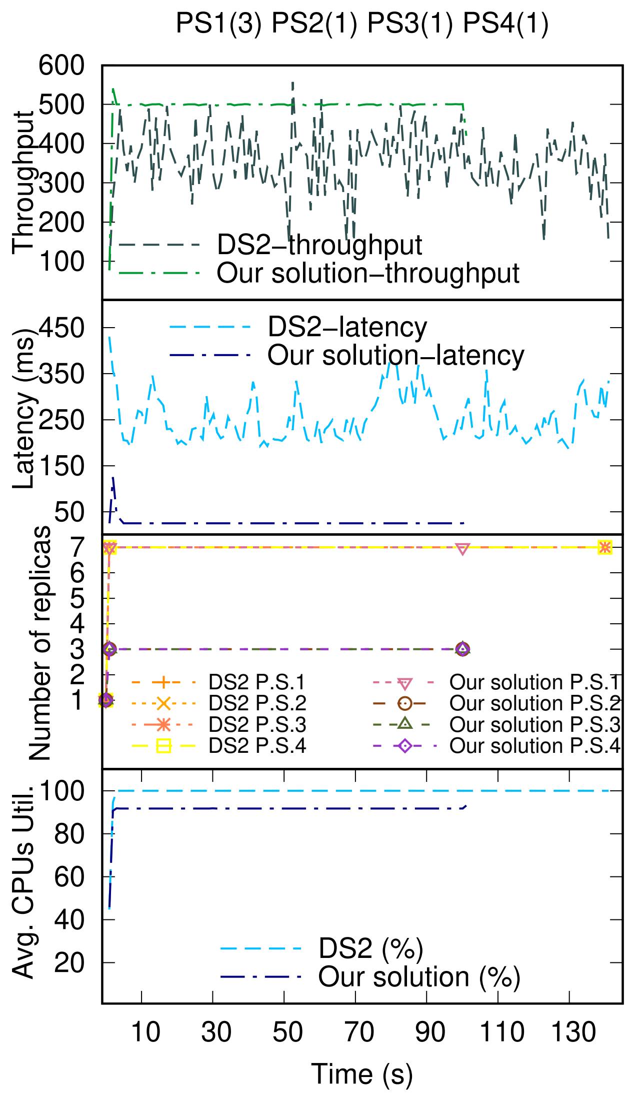
Figure 8.9: Synthetic: Unbalanced Stages. IR and Target Throughput of 1500 I/s.

8.5 Evaluation with four parallel stages

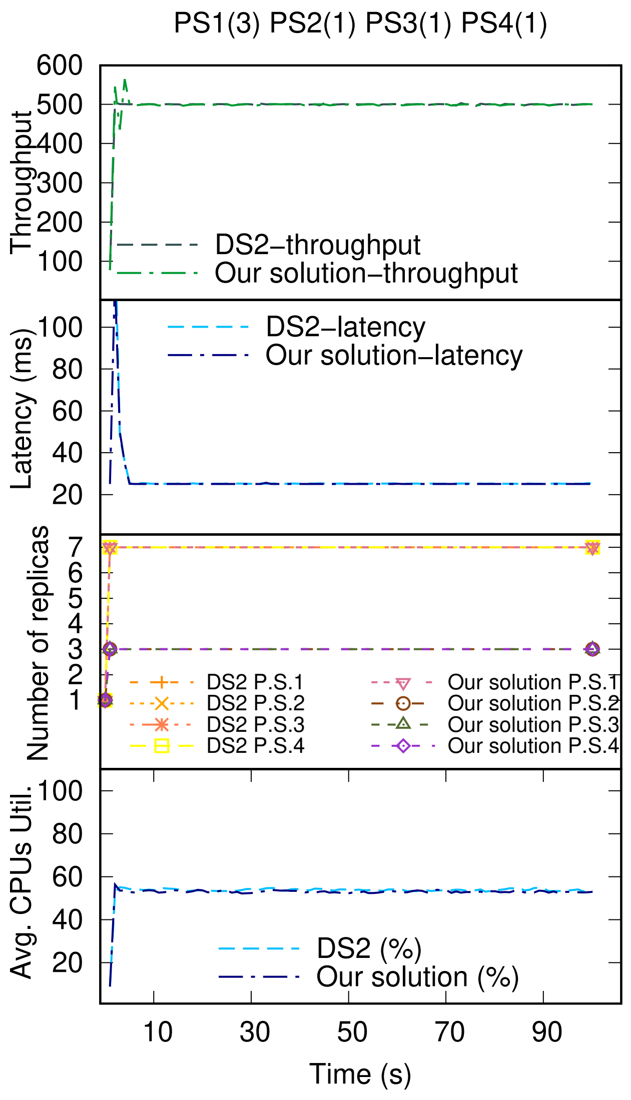
The previous Section 8.4 provided insightful outcomes from the synthetic application with two parallel stages. This section presents a representative scenario where the application has four parallel stages.

8.5.1 ES 4 and 5 - One bottleneck stage

Figure 8.10 shows the first results with an IR of 500 I/s and where the first stage is the bottleneck. In general, Figure 8.10 shows results aligned with the ones seen in Section 8.4, DS2's decision-making cannot 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 the DS2's one in terms of performance and efficiency.



(a) Non-blocking Mode.

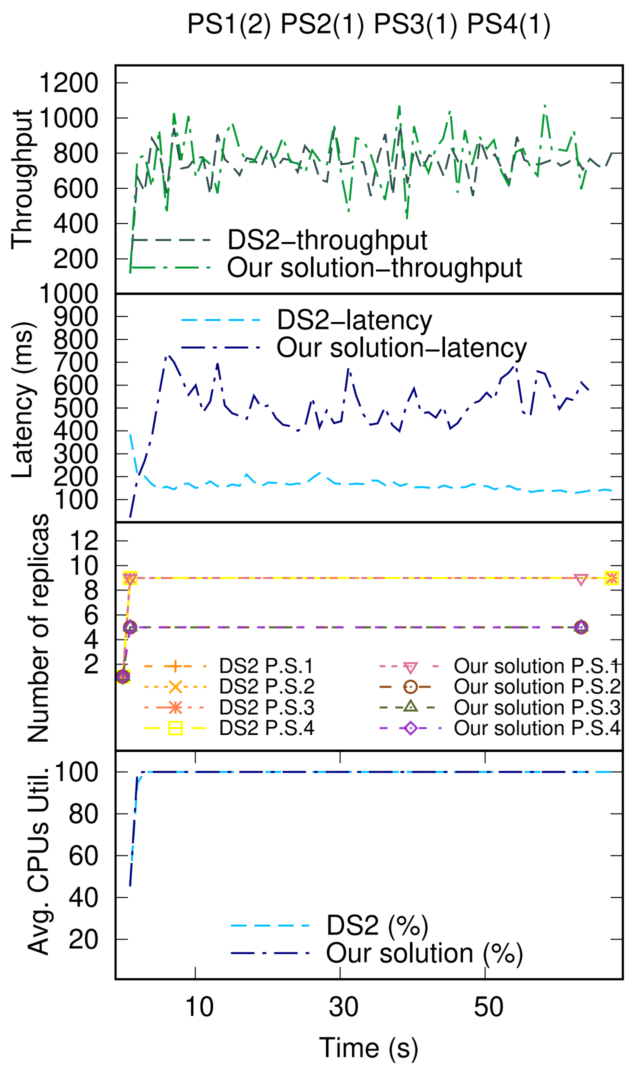


(b) Blocking Mode.

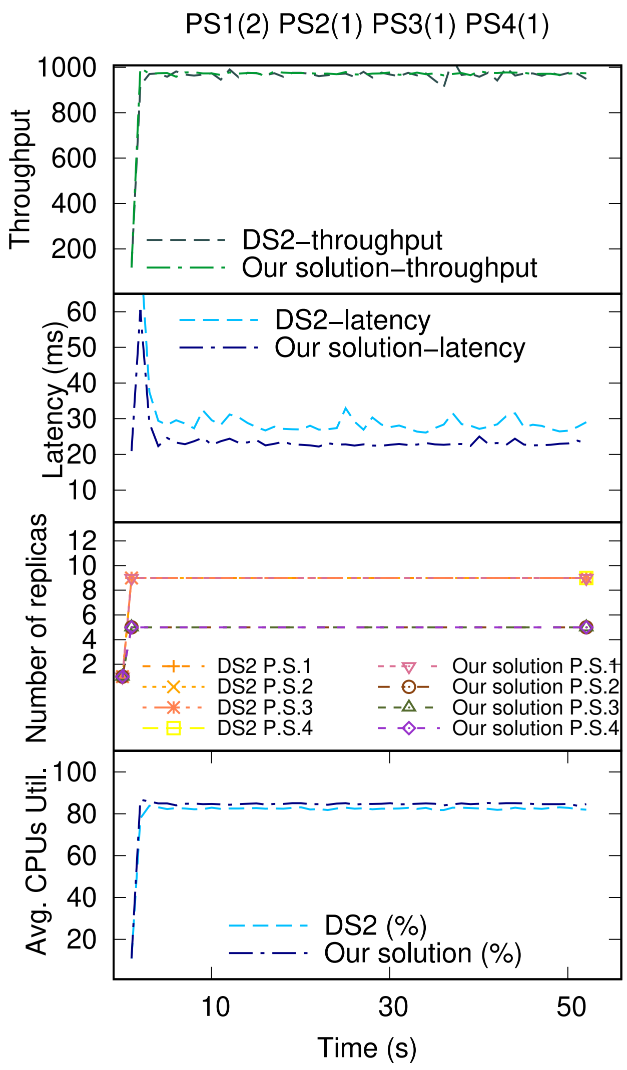
Figure 8.10: Synthetic: Unbalanced Stages. IR and Target Throughput of 500 I/s.

The outcome provided in Figure 8.10(b) 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 8.11 evinces a rather complex scenario with an IR of 1000 I/s. Although we reduced to two the weight of the bottleneck stage in this experiment, the high IR seemingly was still too high for the running machine, i.e., even the efficient runtime's blocking mode achieved only a throughput close to the IR.

Figure 8.11(a) shows the outcome of the execution in the non-blocking mode. In terms of throughput, although under fluctuations, our solution outperformed DS2 achieving a performance closer to the IR target. On the other hand, although DS2 enforced more



(a) Non-blocking Mode.

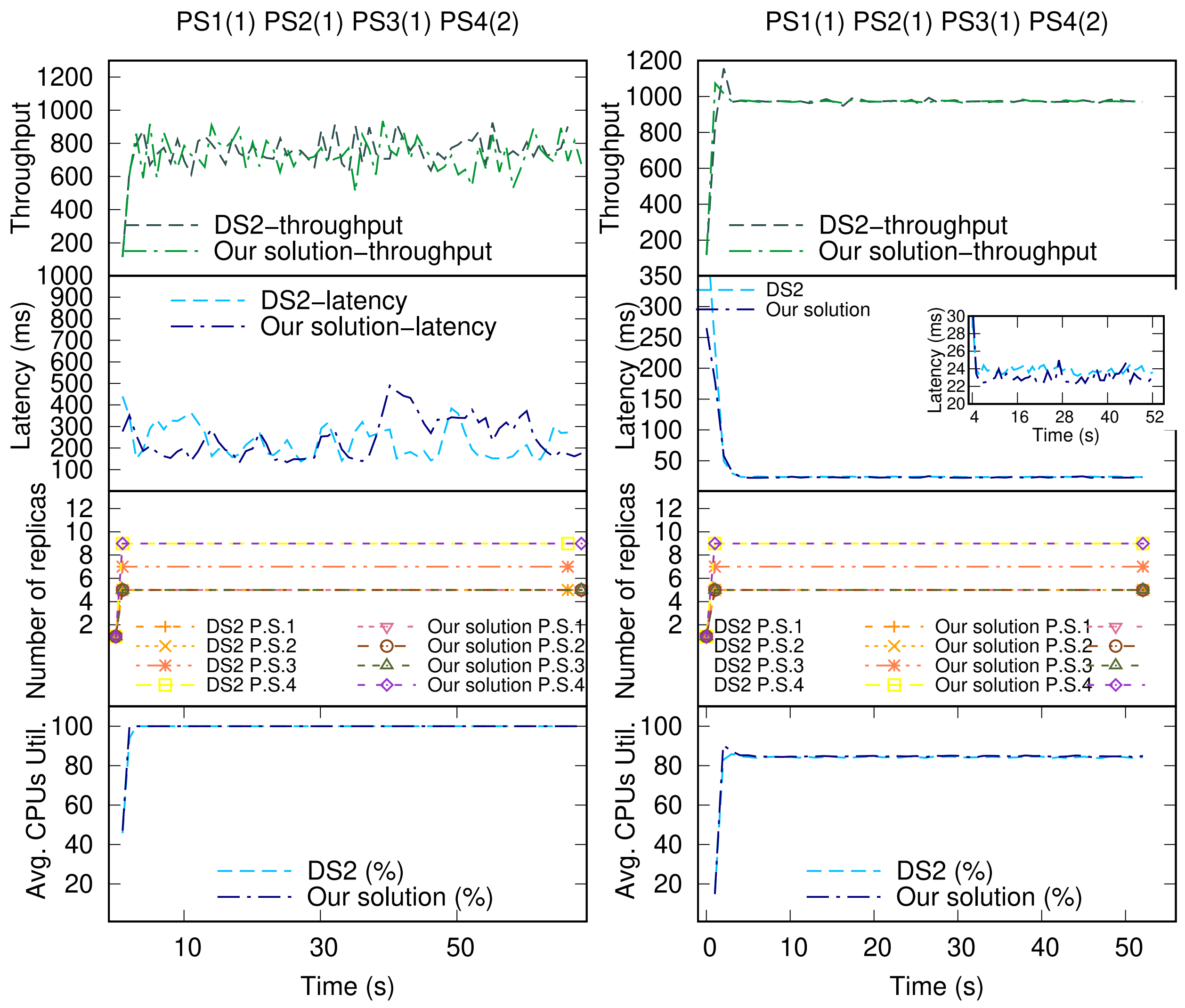


(b) Blocking Mode.

Figure 8.11: Synthetic: Unbalanced Stages. IR and Target Throughput of 1000 I/s.

replicas, it reached a lower latency. A rationale for such an outcome is the combination of high throughput and high CPUs utilization (causing contention) tends to increase the latency [128], which increased the buffering in our solution towards the end of the Pipeline. In the DS2's execution, the bottleneck stage limited the throughput and consequently the buffering in subsequent stages. Hence, less buffering prevented the latency from increasing significantly. Moreover, it is essential to note that one can extend in the future the decision-making strategies to improve latency in complex scenarios like this one with resource contention.

Figure 8.11(b) provides results in the Blocking mode, showing that both solutions achieved a higher throughput compared to the non-blocking mode. Moreover, this interesting outcome showed significant latency contrasts in the blocking mode, where our solution achieved a lower latency. This result corroborates that the resources contention in the non-blocking mode from Figure 8.11(a) was more detrimental to the latency in our solution.



(a) Non-blocking Mode.

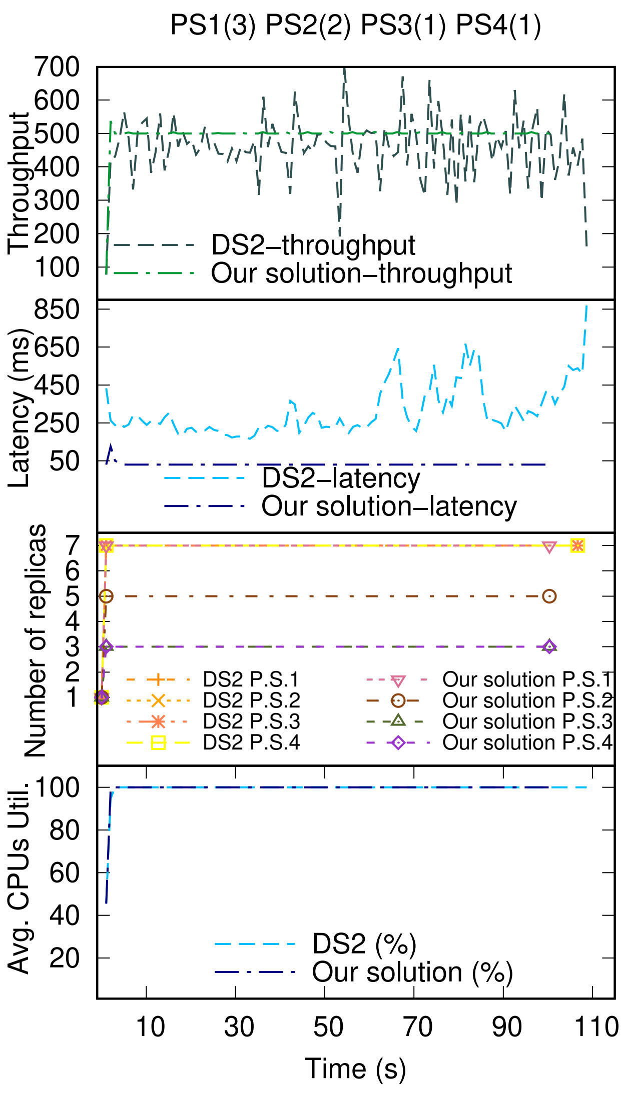
(b) Blocking Mode.

Figure 8.12: Synthetic: Unbalanced Stages. IR and Target Throughput of 1000 I/s.

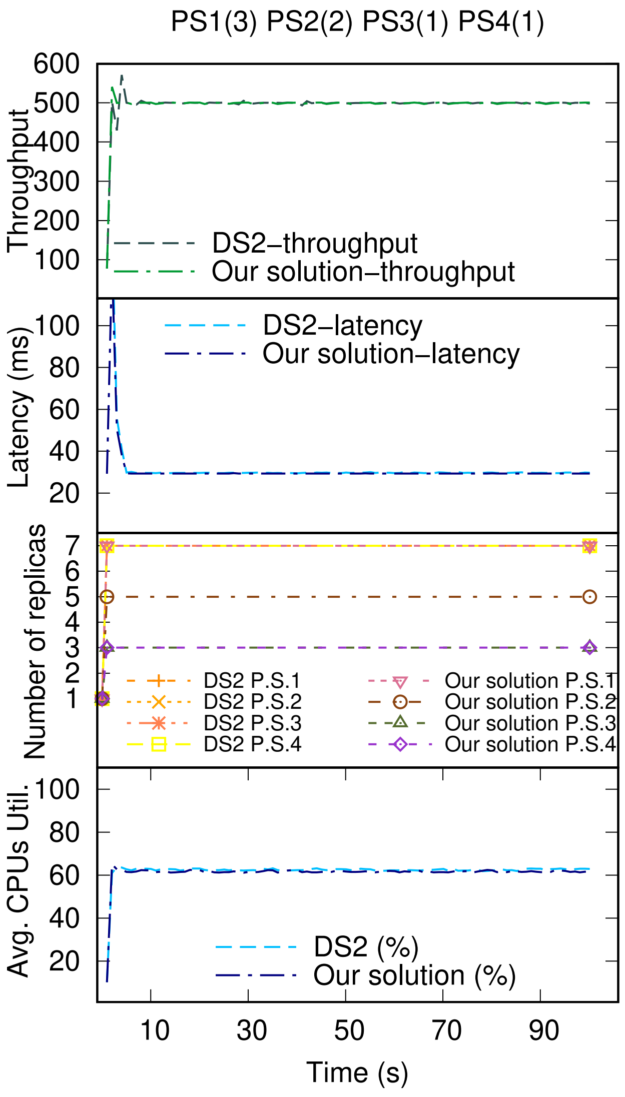
Figure 8.12 provides an additional scenario of four parallel stages and a high input rate. However, contrasting with Figure 8.11, in Figure 8.12 the last stage is the bottleneck. In short, in Figure 8.12 it is difficult to note significant differences from a QoS perspective, where both solutions consumed a similar amount of resources and achieved similar fluctuating performance. A notable insight from Figures 8.11 and 8.12 is that a future optimization for the strategies could encompass (in a training step) the actual resource consumption. Under high utilization, a potential alternative could be to self-configure to more efficient modes like the blocking one in the FastFlow programming framework.

8.5.2 ES 6 - Two unbalanced bottleneck stages

Figures 8.13 and 8.14 show an even more complex scenario where two stages are heavier and yet have different computational weight. Figure 8.13(a) shows a representative scenario of simulating an application with four parallel stages where there are three different levels of computational weight. In this scenario, the first parallel stage is heaviest one, followed by the second one that is lighter compared to the first stage but is still two times more intensive than the light stages (third and fourth).



(a) Non-blocking Mode.



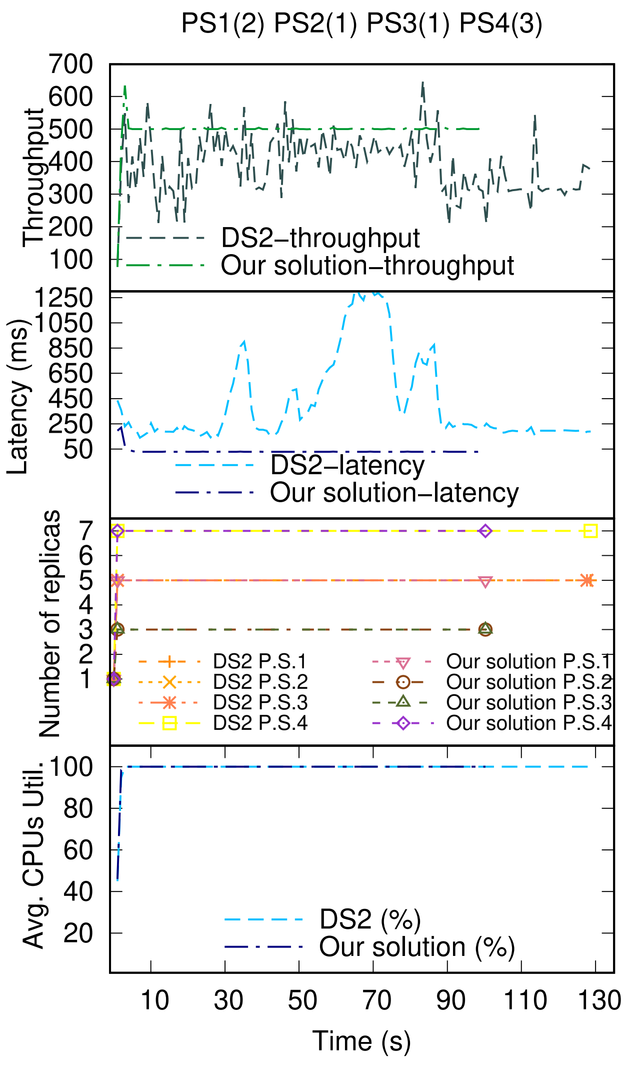
(b) Blocking Mode.

Figure 8.13: Synthetic: Unbalanced Stages. IR and Target Throughput of 500 I/s.

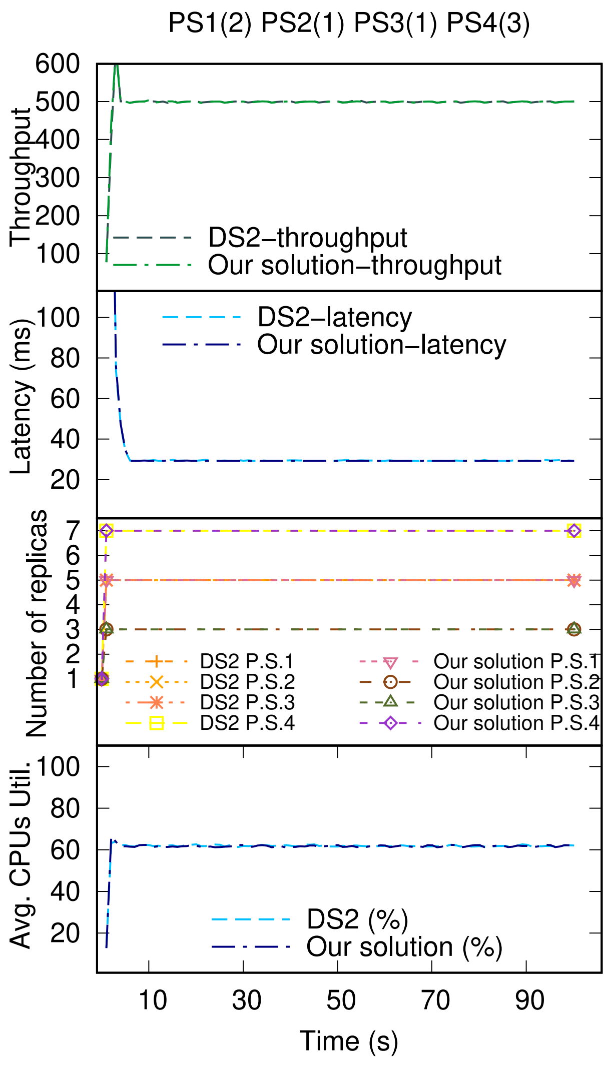
Figure 8.13(a) shows how our solution was able to correctly estimate the computational weight of the stages to set an appropriate number of replicas on each stage.

Hence, compared to DS2, our solution achieved a higher throughput, evidenced by the lower execution time and significantly lower latency.

Figure 8.14 shows a similar performance trend w.r.t. Figure 8.13. However, in Figure 8.14 the heavier stage is the last one instead of the first one. Moreover, in this scenario, the first stage is lighter than the major bottleneck stage but heavier than the lighter middle stages. Although using more replicas, DS2 achieved a competitive performance with the blocking mode (Figure 8.14(b)). However, these additional replicas degraded DS2's performance in the non-blocking mode (Figure 8.14(a)).



(a) Non-blocking Mode.

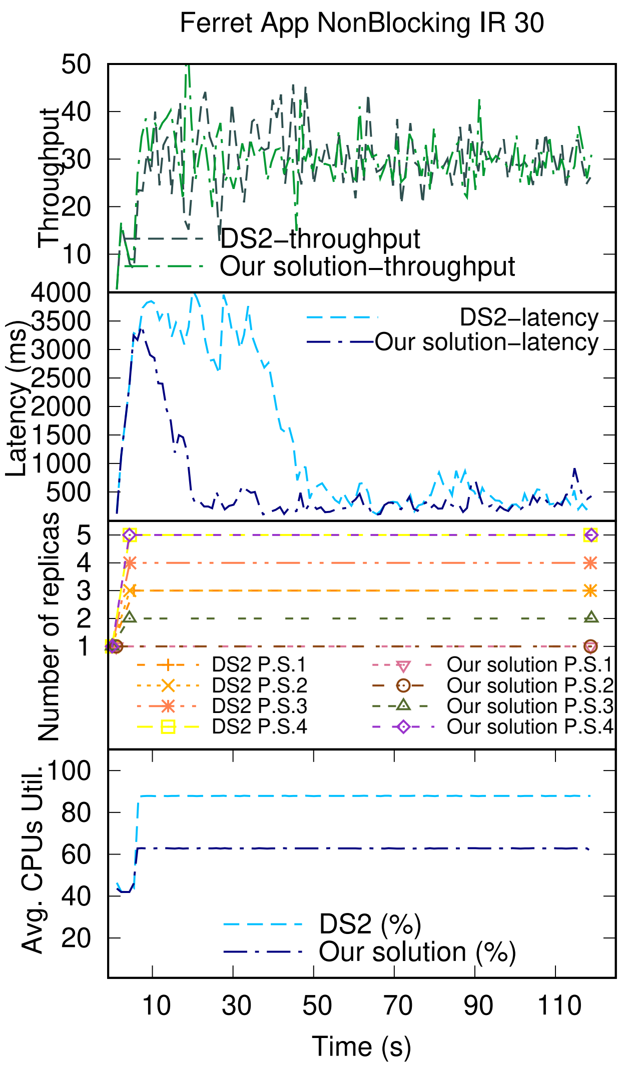


(b) Blocking Mode.

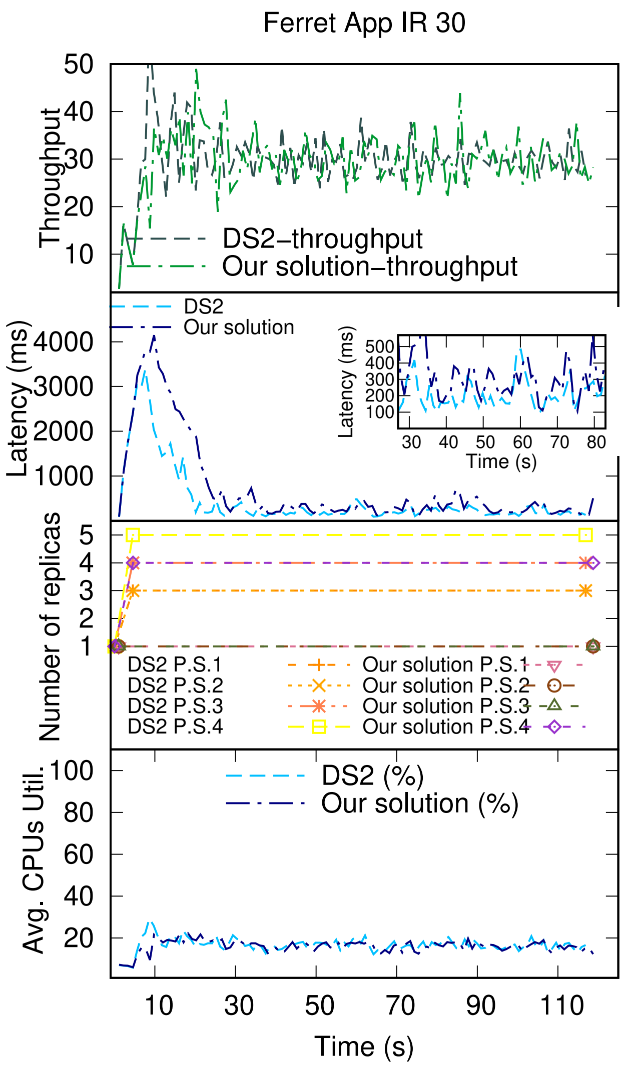
Figure 8.14: Synthetic: Unbalanced Stages. IR and Target Throughput of 500 I/s.

8.6 ES 7 - Ferret

In Section 7.3.4, Ferret was tested with IR of 10 and 20 I/s. Considering that there 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 Figure 8.15.



(a) Non-blocking Mode.

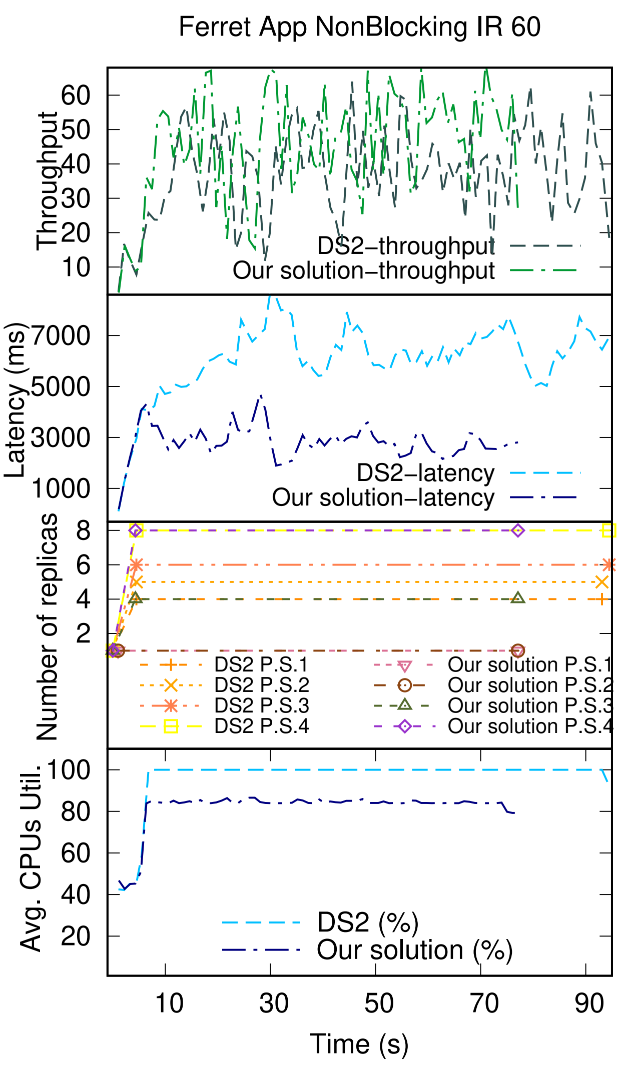


(b) Blocking Mode.

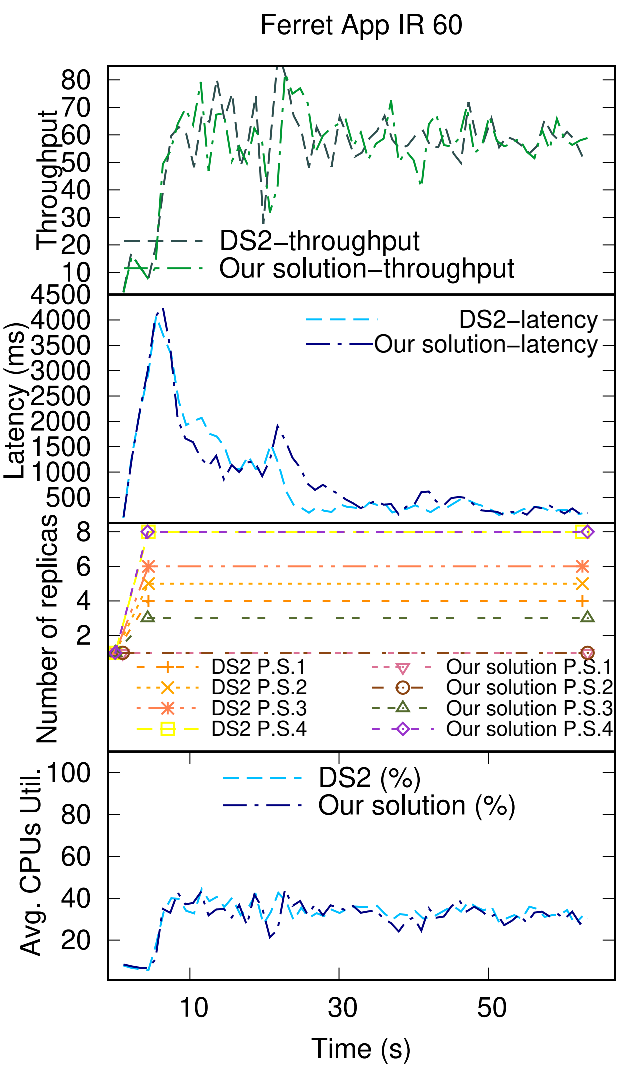
Figure 8.15: Ferret with IR and Target Throughput 30 I/s.

In the non-blocking mode shown in Figure 8.15(a), it is notable that both strategies enforced more replicas to the last stage Rank stage, which is the heavier one. Moreover, compared to our solution, DS2's decision-making enforced more replicas in other lighter stages. Considering a QoS perspective, both solutions present a fluctuating throughput due to Ferret's unstable behavior. Our solution achieved a more consistent and mostly lower latency and consumed fewer resources.

In the Blocking mode from Figure 8.15(b), both strategies achieved a similar performance and resources consumption. Noteworthy, there are instants where DS2 has lower latency and others where our solution has a lower one. Still, it seems inconclusive where the contrasts are potentially due to specific workload peaks.



(a) Non-blocking Mode.



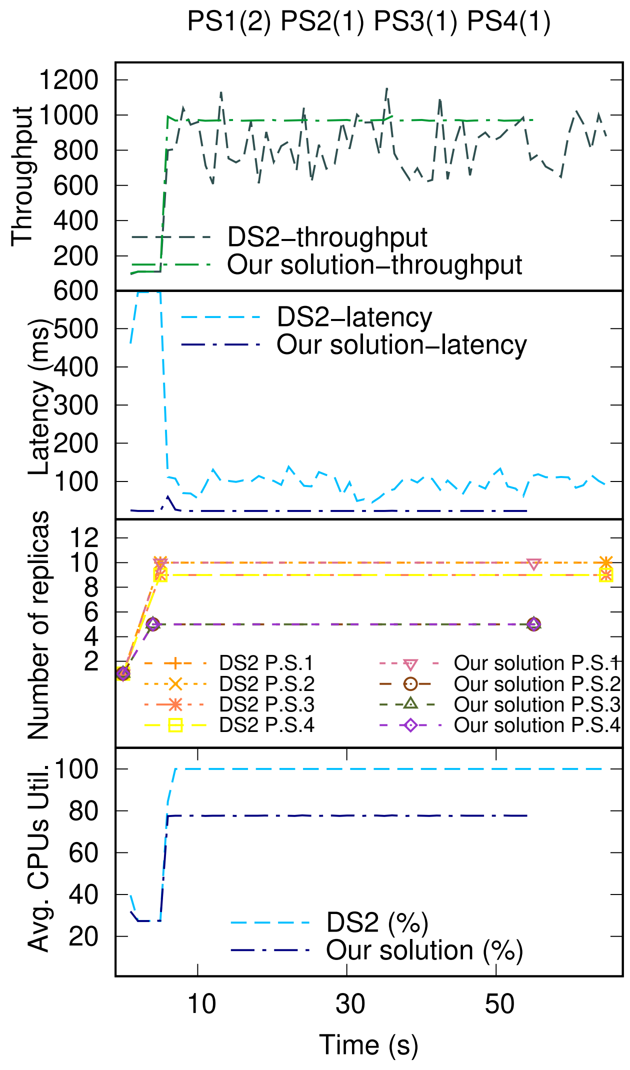
(b) Blocking Mode.

Figure 8.16: Ferret with IR and Target Throughput 60 I/s.

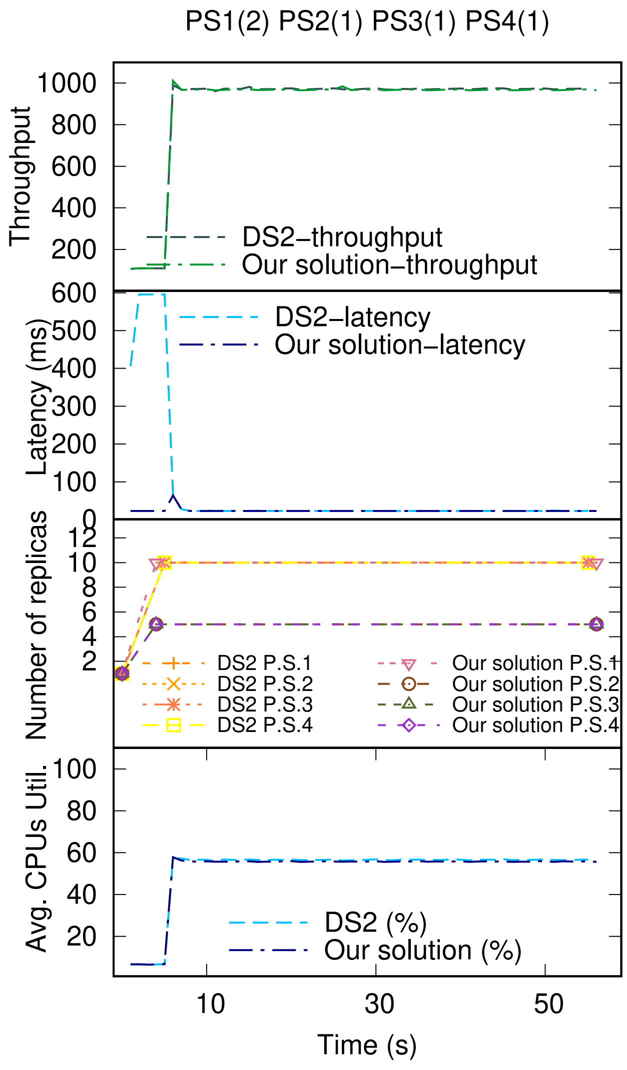
In Figure 8.16, the input rate is doubled to simulate a scenario that demands a higher throughput and utilizes more resources. Considering the number of replicas enforced by the strategies, the outcome is similar to the one seen with IR 30 in Figure 8.15(b), where DS2 estimates to use more replicas mainly in lighter stages. However, under a higher workload like IR 60, unnecessary resource consumption caused more contention, reducing DS2's throughput and increased latency. Hence, in Ferret, it is also notable that our solution is more accurate in determining the appropriate number of replicas for the parallel stages and can significantly outperform DS2's performance in representative scenarios.

8.7 Complementary results with Machine 2

This section presents insightful results from running the evaluation scenarios in another machine, called M2. As explained in Section 8.3.3, for the sake of conciseness, here we present only the most relevant results to assess if the decision-making strategies work consistently in different machines architectures. M2 is a more recent and powerful machine compared to M1. Hence, we expect some contention seen in previous sections to be avoided in a better machine. Moreover, in M2, all the experiments utilized a training step time of five seconds. Here, we increased (from one to five seconds) in the synthetic application for testing if the DS2's (poor) decision-making accuracy was being impacted by the training step time.



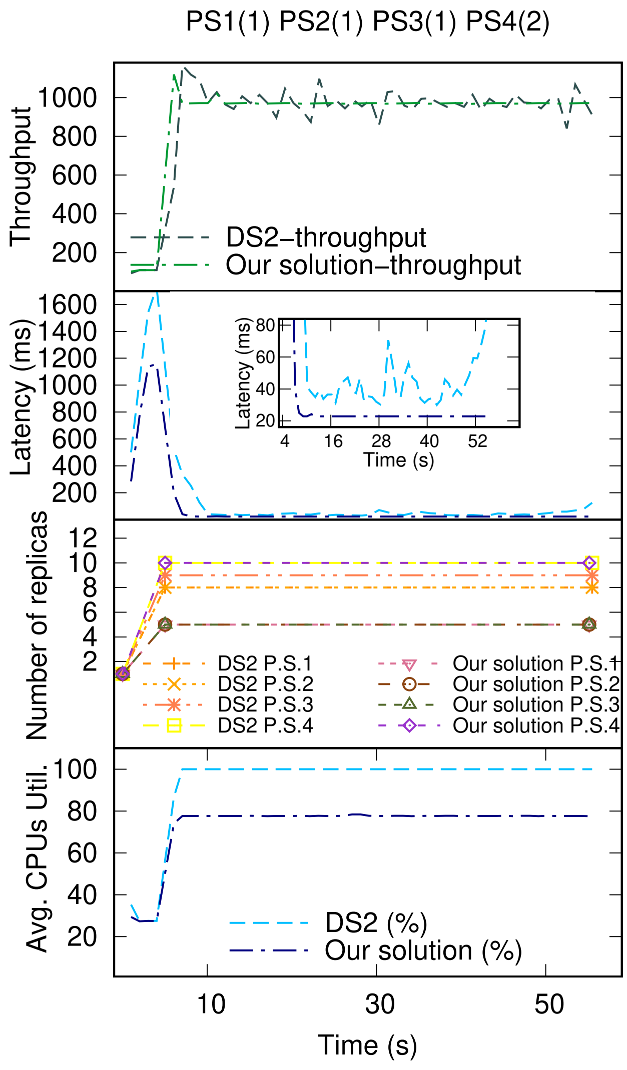
(a) Non-blocking Mode.



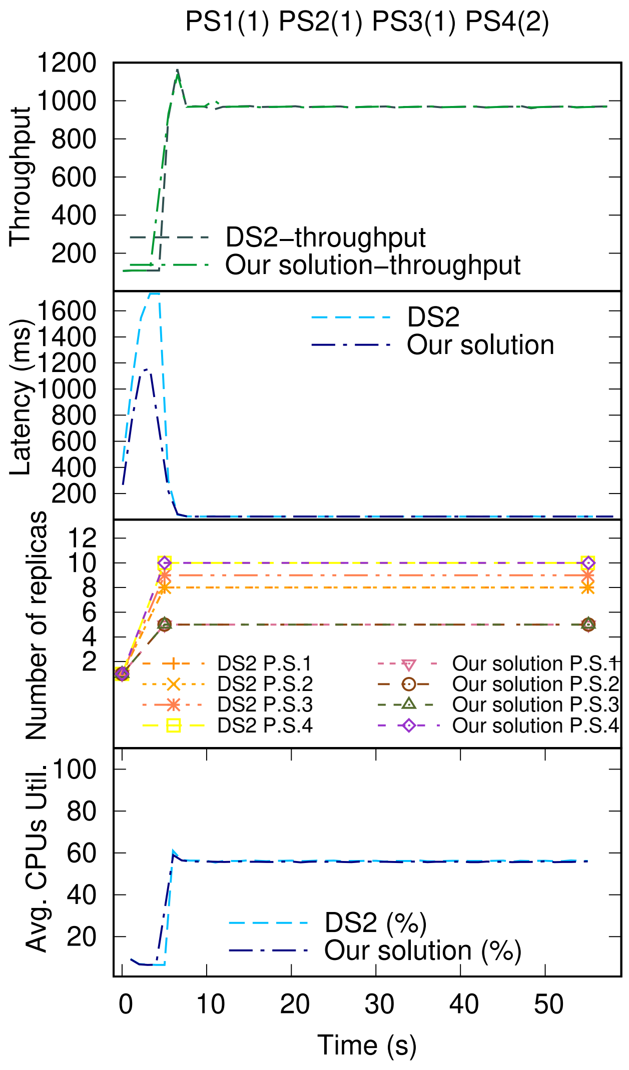
(b) Blocking Mode.

Figure 8.17: M2 - Synthetic: Unbalanced Stages. IR and Target Throughput of 1000 I/s.

Section 8.5.1 evinced the ES 4 where our solution did a seemingly optimal decision-making. However, in Figure 8.11(a) the latency outcome was not very intuitive because DS2 enforced more replicas and still achieved a lower latency. Hence, we repeated this experiment in M2 to assess the strategies' consistency and verify if the higher latency in our solution occurs again. Figure 8.17(a) evinces the better decision-making of our solution compared to DS2, which resulted in our solution achieving a higher throughput compatible with the IR. This results in a lower latency and a more efficient resources usage. Hence, the higher latency of our solution seen in Figure 8.11(a) occurred because the IR was too high for M1, which caused more instability and buffering that increased the latency of items. Moreover, another relevant outcome from Figure 8.17 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 due to its limited generalizability to multi-core machines instead of being due to instabilities during the training step.



(a) Non-blocking Mode.

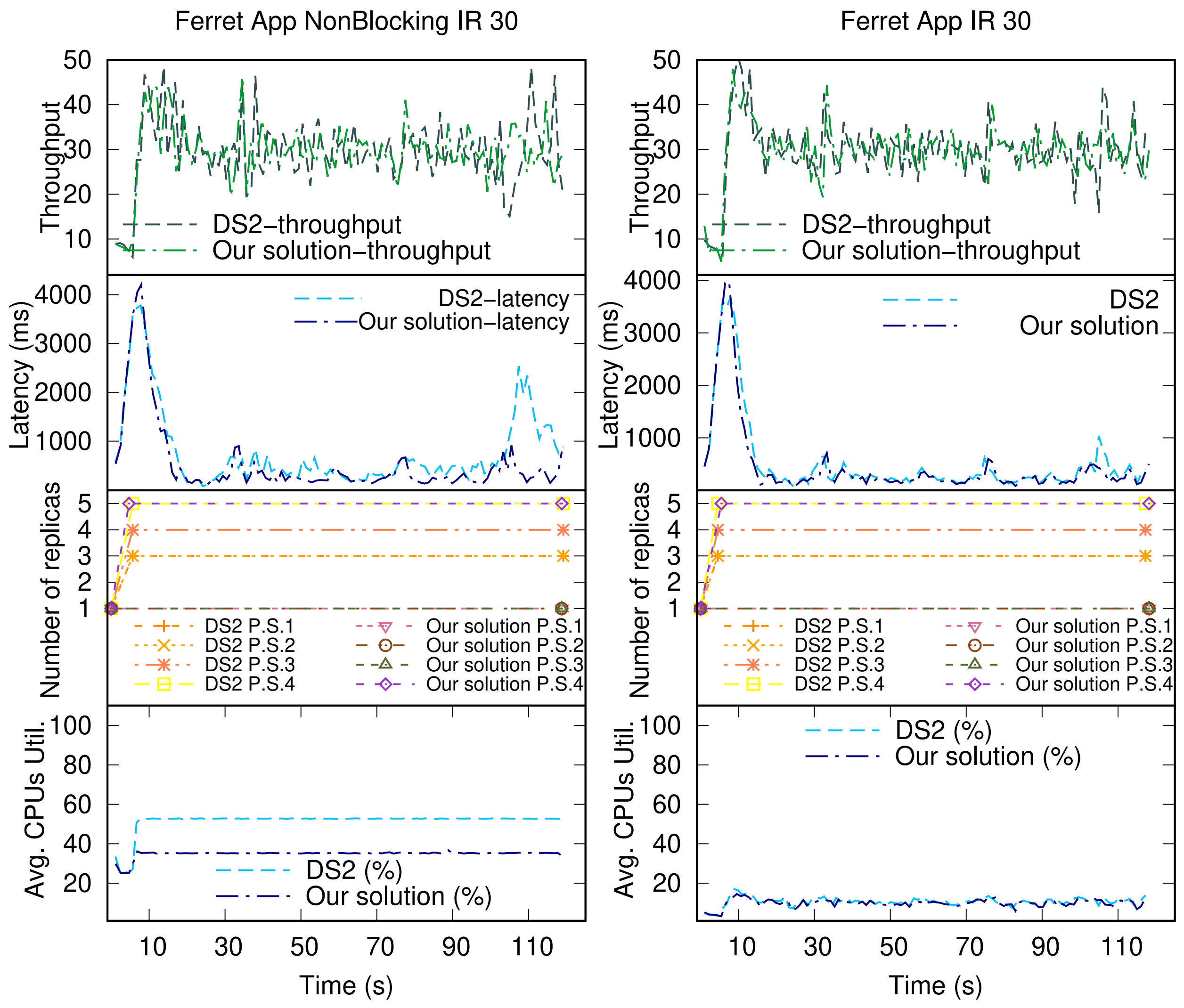


(b) Blocking Mode.

Figure 8.18: M2 - Synthetic: Unbalanced Stages. IR and Target Throughput of 1000 I/s.

Figure 8.18 complements the results from the ES 5 shown in Figure 8.12. Although a different machine is used, the overall performance trend is similar. Our solution achieved slight gains in throughput and lower latency than DS2. Once again, the gains are due to the optimal decision-making that consumes fewer resources, which avoids contention that degrades the QoS in DS2. On the other hand, Figure 8.18(b) shows the results regarding the efficient FastFlow's Blocking mode, where the additional replicas employed in DS2 did not significantly degrade the QoS.

Considering that in Figures 8.17 and 8.18 we addressed the only non-intuitive outcome from M1, in the rest of this section, we focus on additional results from the real-world Ferret application executed in M2.

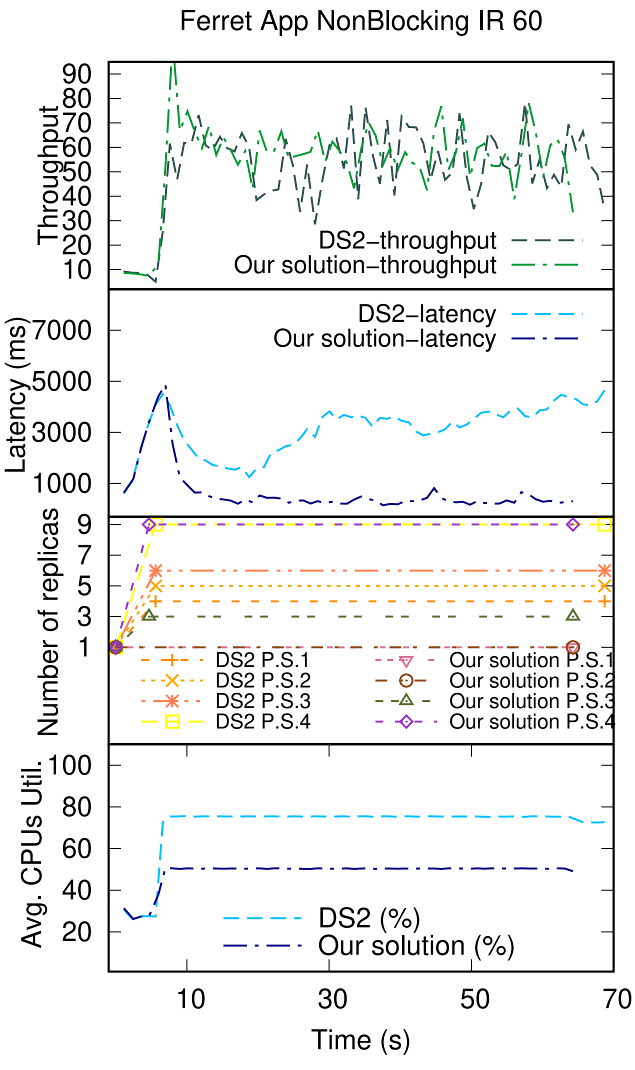


(a) Non-blocking Mode. (b) Blocking Mode.

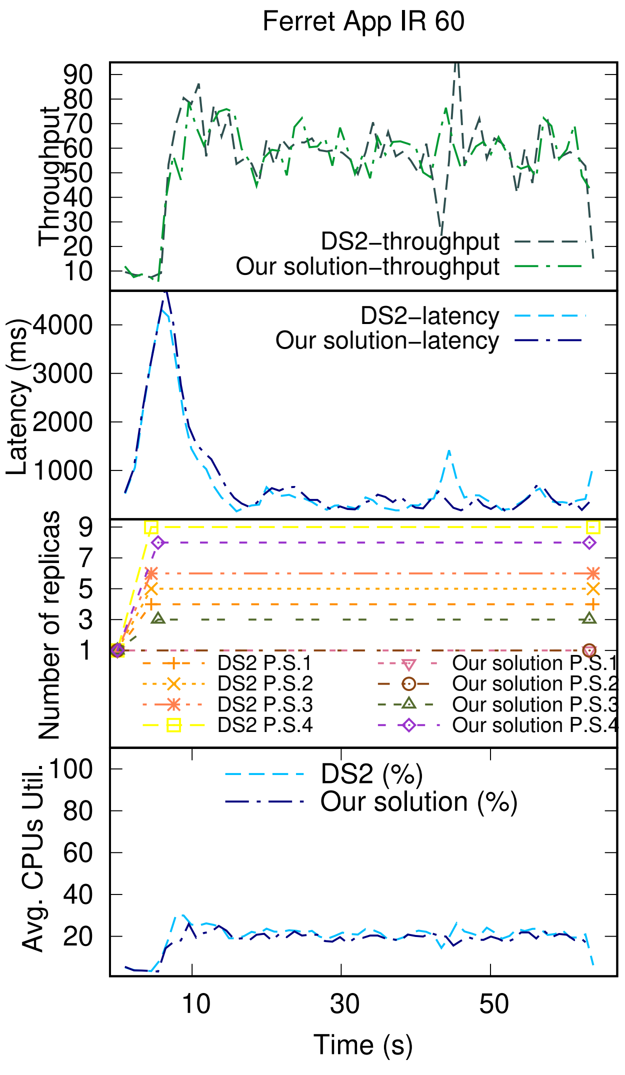
Figure 8.19: M2 - Ferret with IR and Target Throughput 30 I/s.

The results provided in Figure 8.19 extend the evaluation from Figure 8.15 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 Figure 8.19(a) it is notable that our solution consumed significantly fewer resources and yet achieved stabler throughput and lower latency.

Figure 8.20 highlights Ferret with IR 60 I/s executed in M2, which complements the same experiment shown in Figure 8.16 executed in M1. Here, the optimal decision-making of our solution again outperformed DS2 in terms of resources efficiency and QoS. Contrasting with the outcome shown in Figure 8.16(a), in M2, the non-blocking mode shown in Figure 8.20(a) did not consume 100% of the machine's resources in DS2. However, the higher (and unnecessary) resources consumption due to DS2's suboptimal decision-making still caused contention and performance degradation, e.g., the high latency notable in Figure 8.20(a).



(a) Non-blocking Mode.

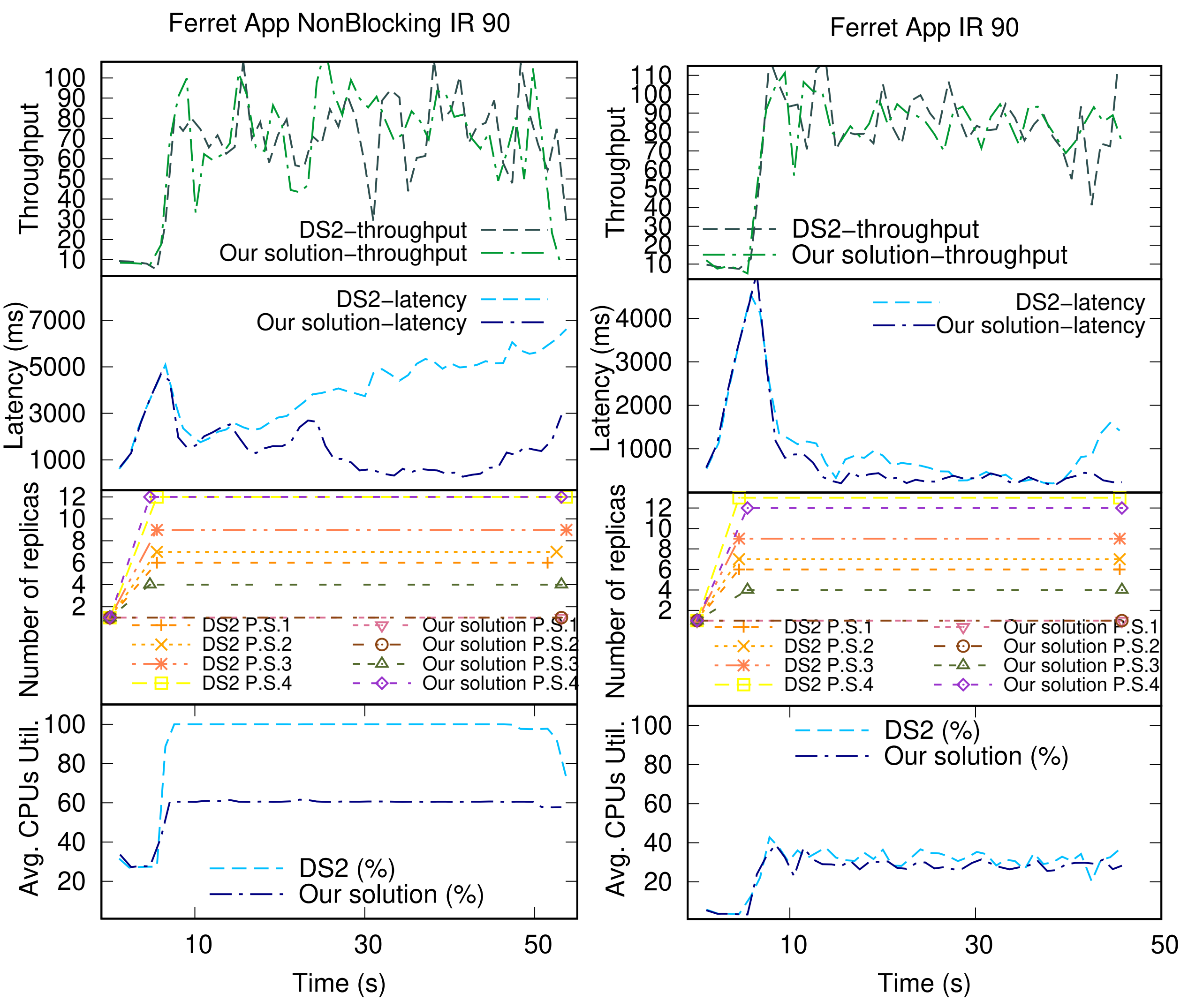


(b) Blocking Mode.

Figure 8.20: M2 - Ferret with IR and Target Throughput 60 I/s.

Considering the Blocking mode evaluation demonstrated in Figure 8.20(b), it is notable a similar resource consumption, but our solution enforced fewer replicas. Hence, it is possible to note that our solution's throughput and latency are stablers. Moreover, the latency achieved in our solution is slightly lower than DS2's latency.

Considering that in Figure 8.20 where an IR 60 I/s running in M2 did not demand all the machine's processing capacity, in Figure 8.21 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 and potentially achieve relevance only in M2 due to its additional processing capacity compared to M1. With empirical characterization tests, M1 did not cope with the high IR on 90, which compromised the stability of the decision-making strategies.



(a) Non-blocking Mode.

(b) Blocking Mode.

Figure 8.21: M2 - Ferret with IR and Target Throughput 90 I/s.

Figure 8.21 provides the results from the executions with IR 90 I/s, which compared to Figure 8.20 (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 while our solution is stabler, it is notable significant latency gains of our solution both in Blocking and the non-blocking executions.

Moreover, with the IR 90 (I/s), the DS2 execution in the non-blocking mode (Figure 8.21(a)) fully utilized the machine's resources, which further increased the latency. In short, the additional results from Ferret running in M2 further confirmed that our solution significantly outperforms DS2 in multi-core machines.

8.8 Closing remarks

The following are the key insights of self-adapting the number of replicas in complex compositions:

- The mechanism integrated within FastFlow show to be effective.
- The decision-making strategies were able to very fast (in a order of milliseconds) decide and find configurations. However, in this work, we are more interested in evaluating how accurate are the configurations found in the decision-making step.
- The solution proposed here goes beyond reactive solutions. It enables to react but at the same time it is proactive by estimating the number of replicas needed to meet the SLOs in the future. This allows a given proactive model to be extended in the future to cope with even more dynamic scenarios than the ones considered here. For instance, implementing additional training steps with configurations shortlisting and trials (see Section 7.2) could handle additional uncertainties.
- The usage of an optimal number of replicas on multi-core 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 amount of resources available, which reduces contention and improves the performance in several scenarios.
- A notable rationale for DS2's poor decision-making is the lack of coordination; there is a need for coordinated decision-making in applications with many parallel stages (or other complex characteristics). One example of such a need is when the availability of resources is a constraint. Coordination combined with optimal profiling is a potential solution to find a proper configuration.
- The usage of an optimal number of replicas has several other implications that are intuitive but were not explored here. For instance, highly efficient executions achiev-

able with improved decision-making consume less energy [119, 36], which has positive implications on sustainability and cost savings.

- The FastFlow blocking mode is nowadays a very efficient solution. However, other runtime libraries/programming frameworks (e.g., C++ threads, OpenMP) and distributed frameworks do not support such a dynamic execution mode. Consequently, use an optimal number of threads/replicas is also very relevant when looking from a more generic perspective.

- 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, different number of parallel stages, different location and numbers of bottleneck stages, and most importantly, running in different configurations of shared memory multi-core machines. This is one more step towards addressing the research challenges (see Section 3.4.4) to better evaluate the self-adaptive solutions applied to parallel computing.

- The conceptual decision-making framework (Chapter 4) was effectively applied here. This facilitated generalization and flexibility, enabling to 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).

- Yet regarding generalization, we expect the decision-making is applicable to other execution environments, such as clusters with distributed memory. On the one hand, the decision-making in cluster environments could be even simpler without considering the availability of the resources because it is less constrained in clusters. On the other hand, applying the decision-making to clusters could require additional steps, e.g., to detect and handle resources heterogeneity. Moreover, considering that the decision-making using an online profiler accurately estimates the number of replicas, we expect that the decision-making could also be integrated with other programming frameworks that have mechanisms to apply (at run-time) the adaptations actions.

- The encouraging results achieved here demonstrated that our solution for self-adaptive replicas is ready to be integrated with larger adaptation spaces. For instance, in Chapters 6 and 7 the entire applications' compositions structures was self-adapted. Adapting the entire structure is a more powerful but more intrusive adaptation that currently 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. Still, it is empowered with more flexible mechanisms that do not require compiling additional configurations. In short, this paves the way to combine both the self-adaptable entities.

9. CONCLUSION

This work discusses opportunities, concepts, and techniques for applying self-adaptation in parallel computing. Hence, we provided efforts for additional parallelism abstractions and for making the abstractions more efficient.

Our understanding is that self-adaptation applied to parallel systems is a relevant evolving area due to the enormous demand to make the systems more flexible and dynamic. This demand is combined with self-adaptation being a promising area that is already being successfully applied (e.g., see below in Section 9.1 some implications of applying self-adaptation). However, self-adaptation is complex due to the challenges in designing, implementing, and validating it. On the one hand, the great potential of self-adaptation and its complexities motivates the area to expand and evolve. On the other hand, we acknowledge existing complexities that are opportunities for future research and practical solutions. Therefore, in Section 9.2 we discuss relevant existing limitations and put them into perspective for future works.

9.1 Implications

9.1.1 Advances in self-adaptation applied to parallel computing

This research started with an empirical exploratory work to evaluate the potential of making parallel parameters transparent (abstracted). We have seen that there are many theoretical concepts applicable and a vast inconsistent terminology in the field, e.g., in reference [127] and Chapter 3 we attempt to catalog and unify the terms of the area. Moreover, we noted that many entities could be adapted to provide abstractions, but not all entities are being self-adapted at run-time without the need to recompile or rerun the entire application. Over the years, we noticed that the impact on QoS and potential self-adaptation overheads were mainly not measured and discussed.

Although acknowledging the limitations that were making difficult the acceptance and use of self-adaptation by practitioners, our perspective on applying self-adaptation to parallel computing has been positive due to the great potential of self-adaptation. Therefore, first, we contributed in more simplistic (but representative) scenarios with applications with one parallel stage (Farm in structured parallel programming). Noteworthy, we proposed and contributed with mechanisms and optimized decision-making strategies (shown in Chapter 5). In [125] we investigated the limits of self-adaptation applied to parallel computing, where it was manifested that a fully transparent execution benefits from users and programmers defining their SLO to indicate to the self-adaptive strategies which QoS is relevant declaratively.

Moreover, considering the parallelism configurations complexities and the need to increase the adaptation space for the sake of flexibility and abstractions, in Chapter 6 we provided a mechanism to adapt the entire application's structure dynamically. Then, in Chapter 7, we improved the decision-making of the self-adaptive strategy to avoid the try-all mode that tested suboptimal configurations. Moreover, the decision-making proposed in Chapter 8 was compared to the state-of-the-art solution called DS2, where our solution significantly outperforms DS2 in terms of accuracy, performance, and efficiency.

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 is demonstrated to be effective in making the applications' executions more dynamic, flexible, and responsive. Dynamic and flexible in adapting larger adaptation spaces and supporting multiple user's goals. Moreover, bringing responsiveness relates to autonomously responding to changes without user intervention, which is a facet of abstractions. These characteristics are essential to cope with modern environments, e.g., providing flexibility for more dynamic executions that can assist in bridging cloud and High Performance Computing (HPC).

Another implication relates to the relevant concerns about the overhead caused by self-adaptation. We have demonstrated that the instrumentation and decision-making overheads can be managed (minimized) throughout this work. For instance, a recurrent noticed concern was the overhead that monitoring could cause, where in [125] (highlighted in Section 5.2.6) the monitoring costs were measured and shown to be very low (hundreds of nanoseconds). Additionally, in reference [126] (highlighted in Section 5.2.5), we showed that the decision-making overhead could be minimized, implying insightful good practices that were incorporated in recent advancements.

Comparing this work to the state-of-the-art solutions, the framework introduced in Chapter 4 diverges from related solutions like NORNIR [36] in the sense that our framework is intended to be conceptual, focusing on more generic and flexible decision-making. Moreover, the strategies for self-adaptive replicas discussed in 5 extend the state-of-the-art by providing additional SLO and abstraction that achieve performance gains even when compared to OS level tools like CPUlimit. The new mechanism that self-adapts the Parallel Patterns to dynamically change the applications' graphs topologies from Chapter 6 and the optimized decision-making strategy from Chapter 7 are novel contributions to the research area. Hence, there are no solutions available to provide a fair comparison to the best of our knowledge. Moreover, comparing the solution for self-adaptation in complex composition structures evinced in Chapter 8 to the state-of-the-art solution called DS2 [64], our solution achieved significant gains in terms of performance and systems efficiency.

9.1.2 Self-adaptation generalizability

In addition to the practical and scientific implications of our contributions explained above, we argue that there is a important demand to make self-adaptation more generic. To be a crucial part of providing solutions for future computational challenges, the self-adaptive approaches are expected to be better designed and implemented, focusing on decoupling and generalization.

Therefore, we proposed a new way to design and implement self-adaptation that encapsulates and decouples the decision-making from the specific mechanisms to apply adaptation actions. By doing this, the decision-making of the self-adaptation can become more generic. For instance, we demonstrated that modules like the data generator (see Chapter 4) were successfully applied in Chapters 6 and 7. Moreover, it was possible to employ the profiler module for making adaptation actions w.r.t. different entities adapted, which was shown in Chapters 7 and 8.

The solutions for self-adaptive parallelism provided in this work have been integrated into two programming frameworks. In Chapter 5 we demonstrated how it was possible to generate self-adaptive code with the DSL SPar [47]. Moreover, the other entities' adaptations are currently applied directly to the FastFlow framework [2]. These integrations indicate that our solution can be flexible enough to be integrated with other programming frameworks and scenarios. From an abstraction perspective, SPar is more suitable for domain experts (application programmers), and FastFlow tends to be more appropriate for system expert programmers. Importantly, SPar's compiler can easily generate the self-adaptive code manually included in the strategies implemented in FastFlow, as already done with the strategies from Chapter 5. Such a code generation would allow users/programmers to utilize additional programming abstractions and even provide new use cases. Moreover, considering that FastFlow is the runtime library of WindFlow [90], our practical contributions can potentially be used there.

Another relevant real-world implication is that the solutions proposed here were 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 facilitates new use-cases and integrations in a modular way. From a technical perspective, we expect it would not be easy to implement our self-adaptive solution in runtime libraries based on task processing, i.e., Intel TBB. However, our understanding is that one can apply this work's solutions to other software systems with the popular threading model based on nodes, for instance, to the vast majority if distributed SPE 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 the self-adaptive strategies using the generic decision-making modules.

Finally, the implications discussed in this section can enable to transfer of knowledge to industry practitioners. For instance, we envision that the conceptual framework can help in the design and implementation of self-adaptive solutions that can be a part of real-world Information Technology (IT) systems. Moreover, the technical mechanisms and systems modules that are part of the scientific contributions can potentially be applied to real-world applications and runtime systems based on the C++ programming language.

9.2 Limitations and future work

The demand for mechanisms to apply adaptation action in a given software system is a known challenge in the field. Hence, considering that generalization is a relevant aspect in this work, the availability of mechanisms can be viewed as a partial limitation to our self-adaptive solutions' broad usage/applicability. However, considering that the 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, the concepts, frameworks, mechanisms, strategies, and use-cases provided in this work can help in inspiring future solutions.

For instance, highly efficient executions achievable with improved mechanisms and decision-making strategies tend to consume less energy [36]. However, due to a considerable amount of work in terms of machinery and experiments, it was not possible to cover energy consumption aspects in this work. Hence, we believe that, in the future, one can extend the self-adaptive strategies proposed to support potential abstractions on performance and energy trade-offs. Moreover, another relevant trade-off to be considered in future strategies and SLO is the relation between performance and cost in pay-per-use environments such as cloud computing.

Although we considered and proposed comprehensive evaluation methodologies to validate our solutions, the experiments can still be limited in some aspects. Additional statistical properties of the results can be considered in the future. Moreover, other applications like the stateful ones could even motivate further enhancements for the decision-making strategies. For instance, additional training steps could be included in the decision-making for better accuracy in scenarios with nonlinear relations between resources and QoS (e.g., adding more processing capacity is not a guarantee of performance gains or QoS increase).

In practice, we expect that the management of stateful applications' state executing in shared-memory multi-core 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 could implement such a step within the transitioning model of the conceptual framework described in Chapter 4.

We are in the process of documenting and open-sourcing the components of the self-adaptive solutions, which could pave the way towards more significant support and provisioning of self-adaptation in other software systems, programming frameworks, and runtime libraries. Our perspectives about the applications and utilized systems showed that the proposed solutions are practical and efficient, where more applications can provide new insights and harness the solutions' applicabilities. Moreover, we intend to make the usability of the self-adaptive strategies more accessible to non-experts programmers.

Considering that it is not trivial to measure and estimate the impact that abstractions can provide in real-world applications, another relevant future endeavor is to study the effect of what we call execution abstractions, which are the ones that abstract complexities and can provide optimizations while the systems are executing. This work provides many examples of execution abstractions achieved through self-adaptation, such as enabling the user to define a non-functional goal instead of low-level error-prone system configurations. Other abstractions, such as the programming abstractions [5] are being improved with methodologies like structured parallel programming. However, in our understanding, execution abstractions can be more impacting because they can abstract and optimize during the entire (potentially long) executions. On the other hand, programming abstractions are usually one-time used abstractions for facilitating code development. Future work could be a software engineering inspired work to contribute with new modern methodologies for quantifying the impact of execution abstractions provided and providing new API for designing self-adaptive strategies.

9.3 Publications

The following are the research articles accepted during the doctorate period that are directly related to this study:

- Title: **Autonomic and Latency-Aware Degree of Parallelism Management in SPar**. Reference: [128].
- Title: **Seamless Parallelism Management for Video Stream Processing on Multi-cores**. Reference: [125].
- Title: **Minimizing Self-Adaptation Overhead in Parallel Stream Processing for Multi-Cores**. Reference: [126].
- Title: **Simplifying and Implementing Service Level Objectives for Stream Parallelism**. Reference: [51].

- Title: **Towards On-the-fly Self-Adaptation of Stream Parallel Patterns**. Reference: [131].
- Title: **Online and Transparent Self-adaptation of Stream Parallel Patterns**. Reference: [132].
- Title: **Self-adaptation on Parallel Stream Processing: A Systematic Review**. Reference: [127].

REFERENCES

[1] Abdelhamid, A.; Mahmood, A.; Daghistani, A.; Aref, W. “Prompt: Dynamic Data-Partitioning for Distributed Micro-Batch Stream Processing Systems”. In: Proceedings of the International Conference on Management of Data, 2020, pp. 2455–2469.

[2] Aldinucci, M.; Danelutto, M. “Skeleton-based Parallel Programming: Functional and Parallel Semantics in a Single Shot”, *Computer Languages, Systems & Structures*, vol. 33–3-4, October 2007, pp. 179–192.

[3] Aldinucci, M.; Danelutto, M.; Kilpatrick, P.; Torquati, M. “Fastflow: High-Level and Efficient Streaming on Multicore”. Wiley, 2017, chap. 13, pp. 261–280.

[4] Amdahl, G. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”. In: Proceedings of the Spring Joint Computer Conference, 1967, pp. 483–485.

[5] Andrade, G.; Griebler, D.; Santos, R.; Danelutto, M.; Fernandes, L. G. “Assessing Coding Metrics for Parallel Programming of Stream Processing Programs on Multi-cores”. In: Proceedings of the Euromicro Conference on Software Engineering and Advanced Applications, 2021, pp. 291–295.

[6] Andrade, H.; Gedik, B.; Turaga, D. “Fundamentals of Stream Processing: Application Design, Systems, and Analytics”. Cambridge University Press, 2014, 558p.

[7] Bacci, B.; Danelutto, M.; Pelagatti, S. “Resource Optimisation via Structured Parallel Programming”. In: *Programming Environments for Massively Parallel Distributed Systems*, Springer, 1994, pp. 13–25.

[8] Balkesen, C.; Tatbul, N.; Özsu, M. T. “Adaptive Input Admission and Management for Parallel Stream Processing”. In: Proceedings of the ACM International Conference on Distributed Event-based Systems, 2013, pp. 15–26.

[9] Bartnik, A.; Monte, B. D.; Rabl, T.; Markl, V. “On-the-fly Reconfiguration of Query Plans for Stateful Stream Processing Engines”. In: Datenbanksysteme für Business, Technologie und Web, 2019, pp. 127–146.

[10] Calasanz, R. T.; Montes, J. D.; Rana, O.; Parashar, M. “Feedback-Control & Queueing Theory-Based Resource Management for Streaming Applications”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 28–4, April 2017, pp. 1061–1075.

[11] Carbone, P.; Katsifodimos, A.; Ewen, S.; Markl, V.; Haridi, S.; Tzoumas, K. “Apache Flink: Stream and Batch Processing in Single Engine”, *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36–4, December 2015, pp. 1–11.

[12] Cardellini, V.; Grassi, V.; Presti, F. L.; Nardelli, M. “Optimal Operator Replication and Placement for Distributed Stream Processing Systems”, *ACM SIGMETRICS Performance Evaluation Review*, vol. 44–4, May 2017, pp. 11–22.

[13] Cardellini, V.; Lo Presti, F.; Nardelli, M.; Russo Russo, G. “Optimal Operator Deployment and Replication for Elastic Distributed Data Stream Processing”, *Concurrency and Computation: Practice and Experience*, vol. 30–9, April 2018, pp. e4334.

[14] Cardellini, V.; Presti, F. L.; Nardelli, M.; Russo, G. R. “Towards Hierarchical Autonomous Control for Elastic Data Stream Processing in the Fog”. In: Proceedings of European Conference on Parallel Processing, 2017, pp. 106–117.

[15] Cardellini, V.; Presti, F. L.; Nardelli, M.; Russo, G. R. “Decentralized Self-adaptation for Elastic Data Stream Processing”, *Future Generation Computer Systems*, vol. 87, October 2018, pp. 171–185.

[16] Chakravarthy, S.; Jiang, Q. “Stream Data Processing: A Quality of Service Perspective: Modeling, Scheduling, Load Shedding, and Complex Event Processing”. Springer US, 2009, 324p.

[17] Chatzistergiou, A.; Viglas, S. D. “Fast Heuristics for Near-Optimal Task Allocation in Data Stream Processing over Clusters”. In: Proceedings of International Conference on Conference on Information and Knowledge Management, 2014, pp. 1579–1588.

[18] Cheng, B.; de Lemos, R.; Giese, H.; Inverardi, P.; Magee, J.; Andersson, J.; Becker, B.; Bencomo, N.; Brun, Y.; Cukic, B.; et al.. “Software Engineering for Self-adaptive Systems: A Research Roadmap”. In: *Software engineering for self-adaptive systems*, Springer, 2009, pp. 1–26.

[19] Cheng, D.; Chen, Y.; Zhou, X.; Gmach, D.; Milojicic, D. “Adaptive Scheduling of Parallel Jobs in Spark Streaming”. In: Proceedings of the IEEE INFOCOM Conference on Computer Communications, 2017, pp. 1–9.

[20] Cheng, D.; Zhou, X.; Wang, Y.; Jiang, C. “Adaptive Scheduling Parallel Jobs with Dynamic Batching in Spark Streaming”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 29–12, December 2018, pp. 2672–2685.

[21] Chis, A.; González-Vélez, H. “Design Patterns and Algorithmic Skeletons: A Brief Concordance”. In: *Modeling and Simulation in HPC and Cloud Systems*, Springer, 2018, pp. 45–56.

[22] Choi, Y.; Li, C.-H.; Silva, D. D.; Bivens, A.; Schenfeld, E. “Adaptive Task Duplication Using On-line Bottleneck Detection for Streaming Applications”. In: Proceedings of the Conference on Computing Frontiers, 2012, pp. 163–172.

[23] Cole, M. “Bringing Skeletons Out Of The Closet: A Pragmatic Manifesto For Skeletal Parallel Programming”, *Parallel Computing*, vol. 30–3, March 2004, pp. 389–406.

[24] Costa, V. G.; Hidalgo, N.; Rosas, E.; Marin, M. “A Dynamic Load Balance Algorithm for the S4 Parallel Stream Processing Engine”. In: Proceedings of the International Symposium on Computer Architecture and High Performance Computing Workshops, 2016, pp. 19–24.

[25] CPUlimit. “CPU Usage Limiter for Linux”. Source: http://cpulimit.sourceforge.net/, Last access march, 2022.

[26] da Rosa Righi, R.; Rodrigues, V. F.; Rostirolla, G.; da Costa, C. A.; Roloff, E.; Navaux, P. O. A. “A Lightweight Plug-and-play Elasticity Service for Self-organizing Resource Provisioning on Parallel Applications”, *Future Generation Computer Systems*, vol. 78, January 2018, pp. 176–190.

[27] Dagum, L.; Menon, R. “OpenMP: An Industry Standard API for Shared-memory Programming”, *IEEE Computational Science and Engineering*, vol. 5–1, January 1998, pp. 46–55.

[28] Danelutto, M.; Mencagli, G.; Torquati, M.; González Vélez, H.; Kilpatrick, P. “Algorithmic Skeletons and Parallel Design Patterns in Mainstream Parallel Programming”, *International Journal of Parallel Programming*, vol. 49, November 2020, pp. 177–198.

[29] Das, T.; Zhong, Y.; Stoica, I.; Shenker, S. “Adaptive Stream Processing using Dynamic Batch Sizing”. In: Proceedings of the ACM Symposium on Cloud Computing, 2014, pp. 1–13.

[30] de Assuncao, M. D.; da Silva Veith, A.; Buyya, R. “Distributed Data Stream Processing and Edge Computing: A Survey on Resource Elasticity and Future Directions”, *Journal of Network and Computer Applications*, vol. 103, February 2018, pp. 1–17.

[31] De Matteis, T.; Mencagli, G. “Keep Calm and React with Foresight: Strategies for Low-latency and Energy-efficient Elastic Data Stream Processing”, *ACM SIGPLAN Notices*, vol. 51–8, February 2016, pp. 13:1–13:12.

[32] De Matteis, T.; Mencagli, G. “Elastic Scaling for Distributed Latency-sensitive Data Stream Operators”. In: Proceedings of Euromicro International Conference on Parallel, Distributed and Network-based Processing, 2017, pp. 61–68.

[33] De Matteis, T.; Mencagli, G. “Proactive Elasticity and Energy Awareness in Data Stream Processing”, *Journal of Systems and Software*, vol. 127, May 2017, pp. 302–319.

[34] De Matteis, T.; Mencagli, G.; De Sensi, D.; Torquati, M.; Danelutto, M. “GASSER: An Auto-Tunable System for General Sliding-Window Streaming Operators on GPUs”, *IEEE Access*, vol. 7, April 2019, pp. 48753–48769.

[35] De Sensi, D.; De Matteis, T.; Danelutto, M. “Nornir: A Customisable Framework for Autonomic and Power-Aware Applications”, *Lecture Notes in Computer Science*, vol. 10659, December 2018, pp. 42–54.

[36] De Sensi, D.; De Matteis, T.; Danelutto, M. “Simplifying Self-adaptive and Power-aware Computing with Nornir”, *Future Generation Computer Systems*, vol. 87, October 2018, pp. 136–151.

[37] De Sensi, D.; De Matteis, T.; Torquati, M.; Mencagli, G.; Danelutto, M. “Bringing Parallel Patterns Out of the Corner: The P^{3}ARSEC Benchmark Suite”, *ACM Transactions on Architecture and Code Optimization*, vol. 14–4, December 2017, pp. 1–26.

[38] El Maghraoui, K.; Desell, T. J.; Szymanski, B. K.; Varela, C. A. “Dynamic Malleability in Iterative MPI Applications”. In: Proceedings of the IEEE International Symposium on Cluster Computing and the Grid, 2007, pp. 591–598.

[39] Fardbastani, M. A.; Sharifi, M. “Scalable Complex Event Processing Using Adaptive Load Balancing”, *Journal of Systems and Software*, vol. 149, March 2019, pp. 305–317.

[40] Floratou, A.; Agrawal, A.; Graham, B.; Rao, S.; Ramasamy, K. “Dhalion: Self-Regulating Stream Processing in Heron”, *Proceedings of the VLDB Endowment*, vol. 10, August 2017, pp. 1825–1836.

[41] Gad, R.; Kappes, M.; Medina-Bulo, I. “Local Parallelization of Pleasingly Parallel Stream Processing on Multiple CPU Cores”. In: Proceedings of the IEEE Annual Information Technology, Electronics and Mobile Communication Conference, 2016, pp. 1–8.

[42] Garcia, A. M.; Griebler, D.; Schepke, C.; Fernandes, L. G. “SPBench: A Framework for Creating Benchmarks of Stream Processing Applications”, *Computing*, In press 2022, pp. 1–23.

[43] Gedik, B.; Andrade, H.; Wu, K.-L.; Yu, P. S.; Doo, M. “SPADE: The System S Declarative Stream Processing Engine”. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2008, pp. 1123–1134.

[44] Gedik, B.; Schneider, S.; Hirzel, M.; Wu, K.-L. “Elastic Scaling for Data Stream Processing”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 25–6, June 2014, pp. 1447–1463.

[45] Gheibi, O.; Weyns, D.; Quin, F. “Applying Machine Learning in Self-adaptive Systems: A Systematic Literature Review”, *ACM Transactions on Autonomous and Adaptive Systems*, vol. 15–3, August 2021, pp. 1–37.

[46] Griebler, D. “Domain-Specific Language & Support Tool for High-Level Stream Parallelism”, Ph.D. Thesis, Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil, 2016, 243p.

[47] Griebler, D.; Danelutto, M.; Torquati, M.; Fernandes, L. G. “SPar: A DSL for High-Level and Productive Stream Parallelism”, *Parallel Processing Letters*, vol. 27–01, March 2017, pp. 1740005.

[48] Griebler, D.; De Sensi, D.; Vogel, A.; Danelutto, M.; Fernandes, L. G. “Service Level Objectives via C++11 Attributes”, *Lecture Notes in Computer Science*, vol. 11339, August 2019, pp. 745–756.

[49] Griebler, D.; Hoffmann, R. B.; Danelutto, M.; Fernandes, L. G. “Higher-Level Parallelism Abstractions for Video Applications with SPar”. In: Proceedings of the International Conference on Parallel Computing, 2017, pp. 698–707.

[50] Griebler, D.; Hoffmann, R. B.; Danelutto, M.; Fernandes, L. G. “High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2”, *International Journal of Parallel Programming*, vol. 47–1, February 2018, pp. 253–271.

[51] Griebler, D.; Vogel, A.; De Sensi, D.; Danelutto, M.; Fernandes, L. G. “Simplifying and Implementing Service Level Objectives for Stream Parallelism”, *The Journal of Supercomputing*, vol. 76–6, June 2020, pp. 4603–4628.

[52] Gulisano, V.; Peris, R. J.; Martinez, M. P.; Soriente, C.; Valduriez, P. “StreamCloud: An Elastic and Scalable Data Streaming System”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 23–12, January 2012, pp. 2351–2365.

[53] Heinze, T.; Jerzak, Z.; Hackenbroich, G.; Fetzer, C. “Latency-aware Elastic Scaling for Distributed Data Stream Processing Systems”. In: Proceedings of International Conference on Distributed Event-Based Systems, 2014, pp. 13–22.

[54] Heinze, T.; Pappalardo, V.; Jerzak, Z.; Fetzer, C. “Auto-scaling Techniques for Elastic Data Stream Processing”. In: Proceedings of International Conference on Data Engineering Workshops, 2014, pp. 296–302.

[55] Heinze, T.; Roediger, L.; Meister, A.; Ji, Y.; Jerzak, Z.; Fetzer, C. “Online Parameter Optimization for Elastic Data Stream Processing”. In: Proceedings of ACM Symposium on Cloud Computing, 2015, pp. 276–287.

[56] Hellerstein, J.; Diao, Y.; Parekh, S.; Tilbury, D. “Feedback Control of Computing Systems”. Wiley, 2004, 456p.

[57] Hidalgo, N.; Rosas, E.; Vasquez, C.; Wladdimiro, D. “Measuring Stream Processing Systems Adaptability under Dynamic Workloads”, *Future Generation Computer Systems*, vol. 88, November 2018, pp. 413–423.

[58] Hirzel, M.; Soulé, R.; Schneider, S.; Gedik, B.; Grimm, R. “A Catalog of Stream Processing Optimizations”, *ACM Computing Surveys*, vol. 46–4, April 2014, pp. 46.

[59] Hoffmann, R. B.; Griebler, D.; Danelutto, M.; Fernandes, L. G. “Stream Parallelism Annotations for Multi-Core Frameworks”. In: Proceedings of the Brazilian Symposium on Programming Languages, 2020, pp. 48–55.

[60] Hoffmann, R. B.; Löff, J.; Griebler, D.; Fernandes, L. G. “OpenMP as Runtime for Providing High-level Stream Parallelism on Multi-cores”, *The Journal of Supercomputing*, vol. 78, April 2022, pp. 1–22.

[61] Jacobson, V. “Congestion Avoidance and Control”, *ACM SIGCOMM computer communication review*, vol. 18–4, August 1988, pp. 314–329.

[62] Janjic, V.; Brown, C.; Mackenzie, K.; et al. “RPL: A Domain-Specific Language for Designing and Implementing Parallel C++ Applications”. In: Proceedings of Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2016, pp. 288–295.

[63] Kahveci, B.; Gedik, B. “Joker: Elastic Stream Processing with Organic Adaptation”, *Journal of Parallel and Distributed Computing*, vol. 137, March 2020, pp. 205–223.

[64] Kalavri, V.; Liagouris, J.; Hoffmann, M.; Dimitrova, D.; Forshaw, M.; Roscoe, T. “Three Steps is All You Need: Fast, Accurate, Automatic Scaling Decisions for Distributed Streaming Dataflows”. In: Proceedings of USENIX Symposium on Operating Systems Design and Implementation, 2018, pp. 783–798.

[65] Karavadara, N.; Zolda, M.; Nguyen, V. T. N.; Knoop, J.; Kirner, R. “Dynamic Power Management for Reactive Stream Processing on the SCC Tiled Architecture”, *EURASIP Journal on Embedded Systems*, vol. 2016–1, June 2016, pp. 14.

[66] Kehrer, S.; Blochinger, W. “Elastic Parallel Systems for High Performance Cloud Computing: State-of-the-Art and Future Directions”, *Parallel Processing Letters*, vol. 29–02, July 2019, pp. 1–20.

[67] Kephart, J.; Chess, D. “The Vision of Autonomic Computing”, *Computer*, vol. 36–1, January 2003, pp. 41–50.

[68] Kitchenham, B.; Charters, S. “Guidelines for Performing Systematic Literature Reviews in Software Engineering”, Technical Report, EBSE, 2007, 65p.

[69] Kombi, R. K.; Lumineau, N.; Lamarre, P. “A Preventive Auto-parallelization Approach for Elastic Stream Processing”. In: Proceedings of International Conference on Distributed Computing Systems, 2017, pp. 1532–1542.

[70] Kombi, R. K.; Lumineau, N.; Lamarre, P.; Rivetti, N.; Busnel, Y. “DABS-storm: A Data-aware Approach for Elastic Stream Processing”, *Lecture Notes in Computer Science*, vol. 11360, December 2019, pp. 58–93.

[71] Krupitzer, C.; Roth, F. M.; VanSyckel, S.; Schiele, G.; Becker, C. “A Survey on Engineering Approaches For Self-adaptive Systems”, *The Pervasive and Mobile Computing Journal*, vol. 17, February 2015, pp. 184–206.

[72] Li, J.; Pu, C.; Chen, Y.; Gmach, D.; Milojicic, D. “Enabling Elastic Stream Processing in Shared Clusters”. In: Proceedings of the IEEE International Conference on Cloud Computing, 2016, pp. 108–115.

[73] Liu, X.; Dastjerdi, A. V.; Calheiros, R. N.; Qu, C.; Buyya, R. “A Stepwise Auto-Profiling Method for Performance Optimization of Streaming Applications”, *ACM Transactions on Autonomous and Adaptive Systems*, vol. 12–4, December 2018, pp. 24.

[74] Liu, Y.; Shi, X.; Jin, H. “Runtime-aware Adaptive Scheduling in Stream Processing”, *Concurrency and Computation: Practice and Experience*, vol. 28–14, September 2016, pp. 3830–3843.

[75] Loff, J.; B. Hoffman, R.; Griebler, D.; G. Fernandes, L. “High-Level Stream and Data Parallelism in C++ for Multi-Cores”. In: Proceedings of Brazilian Symposium on Programming Languages, 2021, pp. 41–48.

[76] Lohrmann, B.; Janacik, P.; Kao, O. “Elastic Stream Processing with Latency Guarantees”. In: Proceedings of International Conference on Distributed Computing Systems, 2015, pp. 399–410.

[77] Lombardi, F.; Aniello, L.; Bonomi, S.; Querzoni, L. “Elastic Symbiotic Scaling of Operators and Resources in Stream Processing Systems”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 29–3, March 2018, pp. 572–585.

[78] Lombardi, F.; Muti, A.; Aniello, L.; Baldoni, R.; Bonomi, S.; Querzoni, L. “PASCAL: An Architecture for Proactive Auto-scaling of Distributed Services”, *Future Generation Computer Systems*, vol. 98, September 2019, pp. 342–361.

[79] Lv, Q.; Josephson, W.; Wang, Z.; Charikar, M.; Li, K. “Ferret: A Toolkit for Content-based Similarity Search of Feature-rich Data”. In: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems, 2006, pp. 317–330.

[80] Mai, L.; Zeng, K.; Potharaju, R.; et al.. “Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems”, *Proceedings of the VLDB Endowment*, vol. 11–10, June 2018, pp. 1303–1316.

[81] Marangozova-Martin, V.; Palma, N.; Rheddane, A. “Multi-Level Elasticity for Data Stream Processing”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 30–10, October 2019, pp. 2326–2337.

[82] Maron, C. A. F.; Vogel, A.; Griebler, D.; Fernandes, L. G. “Should PARSEC Benchmarks be More Parametric? A Case Study with Dedup”. In: Proceedings of Euromicro International Conference on Parallel, Distributed and Network-based Processing, 2019, pp. 217–221.

[83] Martin, A.; Brito, A.; Fetzer, C. “Scalable and Elastic Realtime Click Stream Analysis Using StreamMine3G”. In: Proceedings of International Conference on Distributed Event-Based Systems, 2014, pp. 198–205.

[84] Martin, A.; Smaneoto, T.; Dietze, T.; Brito, A.; Fetzer, C. “User-constraint and Self-adaptive Fault Tolerance for Event Stream Processing Systems”. In: Proceedings of International Conference on Dependable Systems and Networks, 2015, pp. 462–473.

[85] Mattson, T. G.; Sanders, B. A.; Massingill, B. L. “Patterns for Parallel Programming”. Addison-Wesley, 2005, 355p.

[86] Mayer, R.; Koldehofe, B.; Rothermel, K. “Meeting Predictable Buffer Limits in the Parallel Execution of Event Processing Operators”. In: Proceedings of International Conference on Big Data, 2014, pp. 402–411.

[87] Mayer, R.; Koldehofe, B.; Rothermel, K. “Predictable Low-Latency Event Detection With Parallel Complex Event Processing”, *IEEE Internet Things J*, vol. 2–4, August 2015, pp. 274–286.

[88] McCool, M.; Reinders, J.; Robison, A. “Structured Parallel Programming: Patterns for Efficient Computation”. Elsevier Science, 2012, 406p.

[89] Mencagli, G. “A Game-Theoretic Approach for Elastic Distributed Data Stream Processing”, *ACM Transactions on Autonomous and Adaptive Systems*, vol. 11–2, July 2016, pp. 13.

[90] Mencagli, G.; Torquati, M.; Cardaci, A.; Fais, A.; Rinaldi, L.; Danelutto, M. “WindFlow: High-Speed Continuous Stream Processing with Parallel Building Blocks”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, November 2021, pp. 2748 – 2763.

[91] Mencagli, G.; Torquati, M.; Danelutto, M. “Elastic-PPQ: A Two-level Autonomic System for Spatial Preference Query Processing Over Dynamic Data Streams”, *Future Generation Computer Systems*, vol. 79, February 2018, pp. 862–877.

[92] Metzger, P.; Cole, M.; Fensch, C.; Aldinucci, M.; Bini, E. “Enforcing Deadlines for Skeleton-based Parallel Programming”. In: Proceedings of the IEEE Symposium on Real-Time and Embedded Technology and Applications, 2020, pp. 188–199.

[93] Miller, J.; Trümper, L.; Terboven, C.; Müller, M. S. “A theoretical model for global optimization of parallel algorithms”, *Mathematics*, vol. 9–14, July 2021, pp. 1685.

[94] Misale, C.; Drocco, M.; Tremblay, G.; Martinelli, A.; Aldinucci, M. “PiCo: High-performance Data Analytics Pipelines in Modern C++”, *Future Generation Computer Systems*, vol. 87, October 2018, pp. 392–403.

[95] Moore, G. “Moore’s Law”, *Electronics Magazine*, vol. 38–8, April 1965, pp. 114.

[96] Nardelli, M.; Russo, G. R.; Cardellini, V.; Lo Presti, F. “A Multi-level Elasticity Framework for Distributed Data Stream Processing”. In: Proceedings of the European Conference on Parallel Processing, 2018, pp. 53–64.

[97] Ni, X.; Schneider, S.; Pavuluri, R.; Kaus, J.; Wu, K.-L. “Automating Multi-level Performance Elastic Components for IBM Streams”. In: Proceedings of International Middleware Conference, 2019, pp. 163–175.

[98] Pieper, R. L. “High-level Programming Abstractions for Distributed Stream Processing”, Master’s thesis, School of Technology - PPGCC - PUCRS, Porto Alegre, Brazil, 2020, 170p.

[99] Pusukuri, K. K.; Gupta, R.; Bhuyan, L. N. “Thread reinforcer: Dynamically determining number of threads via os level monitoring”. In: Proceedings of the IEEE International Symposium on Workload Characterization, 2011, pp. 116–125.

[100] Qin, C.; Eichelberger, H.; Schmid, K. “Enactment of Adaptation in Data Stream Processing with Latency Implications—A Systematic Literature Review”, *Information and Software Technology*, vol. 111, July 2019, pp. 1–21.

[101] Rajadurai, S.; Bosboom, J.; Wong, W.-F.; Amarasinghe, S. “Gloss: Seamless Live Reconfiguration and Reoptimization of Stream Programs”, *ACM SIGPLAN Notices*, vol. 53–2, February 2018, pp. 98–112.

[102] Rockenbach, D. A. “High-Level Programming Abstractions for Stream Parallelism on GPUs”, Master’s thesis, School of Technology - PPGCC - PUCRS, Porto Alegre, Brazil, 2020, 163p.

[103] Röger, H.; Mayer, R. “A Comprehensive Survey on Parallelization and Elasticity in Stream Processing”, *ACM Computing Surveys*, vol. 52–2, March 2019, pp. 36.

[104] Russo, G. R.; Cardellini, V.; Lo Presti, F. “Reinforcement Learning Based Policies for Elastic Stream Processing on Heterogeneous Resources”. In: Proceedings of International Conference on Distributed and Event-based Systems, 2019, pp. 31–42.

[105] Sahin, S.; Gedik, B. “C-Stream: A Co-routine-Based Elastic Stream Processing Engine”, *ACM Transactions on Parallel Computing*, vol. 4–3, September 2018, pp. 15.

[106] Schneider, S.; Andrade, H.; Gedik, B.; Biem, A.; Wu, K.-L. “Elastic Scaling of Data Parallel Operators in Stream Processing”. In: Proceedings of the IEEE International Symposium on Parallel & Distributed Processing, 2009, pp. 1–12.

[107] Schneider, S.; Hirzel, M.; Gedik, B.; Wu, K.-L. “Auto-parallelizing Stateful Distributed Streaming Applications”. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 2012, pp. 53–64.

[108] Schneider, S.; Wu, K.-L. “Low-Synchronization, Mostly Lock-Free, Elastic Scheduling for Streaming Runtimes”, *ACM SIGPLAN Notices*, vol. 52–6, June 2017, pp. 648–661.

[109] Schor, L.; Bacivarov, I.; Yang, H.; Thiele, L. “AdaPNet: Adapting Process Networks in Response to Resource Variations”. In: Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems, 2014, pp. 22.

[110] Selva, M.; Morel, L.; Marquet, K.; Frenot, S. “A Monitoring System for Runtime Adaptations of Streaming Applications”. In: Proceedings of Euromicro International Conference on Parallel, Distributed and Network-Based Processing, 2015, pp. 27–34.

[111] Shevtsov, S.; Berekmeri, M.; Weyns, D.; Maggio, M. “Control-Theoretical Software Adaptation: A Systematic Literature Review”, *IEEE Transactions on Software Engineering*, vol. 44–8, August 2017, pp. 784–810.

[112] Stein, C.; Rockenbach, D.; Griebler, D.; et al.. “Latency-aware Adaptive Micro-batching Techniques for Streamed Data Compression on Graphics Processing Units”, *Concurrency and Computation: Practice and Experience*, vol. 33–11, June 2020, pp. e5786.

[113] Su, Y.; Shi, F.; Talpur, S.; Wang, Y.; Hu, S.; Wei, J. “Achieving self-aware Parallelism in Stream Programs”, *Cluster Computing*, vol. 18–2, December 2015, pp. 949–962.

[114] Talebi, M.; Sharifi, M.; Kalantari, M. “ACEP: An Adaptive Strategy for Proactive and Elastic Processing of Complex Events”, *The Journal of Supercomputing*, vol. 77–5, May 2021, pp. 4718–4753.

[115] Tang, Y.; Gedik, B. “Autopipelining for Data Stream Processing”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 24–12, December 2012, pp. 2344–2354.

[116] Theis, T.; Wong, P. “The End Of Moore’s Law: A New Beginning for Information Technology”, *Computing in Science & Engineering*, vol. 19–2, March 2017, pp. 41.

[117] Thies, W.; Karczmarek, M.; Amarasinghe, S. “StreamIt: A Language for Streaming Applications”. In: Proceedings of the International Conference on Compiler Construction, 2002, pp. 179–196.

[118] Torquati, M. “Harnessing Parallelism in Multi/Many-Cores with Streams and Parallel Patterns”, Ph.D. Thesis, Computer Science Dept. - University of Pisa, Italy, 2019, 378p.

[119] Torquati, M.; Sensi, D. D.; Mencagli, G.; Aldinucci, M.; Danelutto, M. “Power-aware Pipelining with Automatic Concurrency Control”, *Concurrency and Computation: Practice and Experience*, vol. 31–5, March 2019, pp. e4652.

[120] Toshniwal, A.; Taneja, S.; Shukla, A.; et al.. “Storm Twitter”. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2014, pp. 147–156.

[121] Tudoran, R.; Nano, O.; Santos, I.; et al.. “Jetstream: Enabling High Performance Event Streaming Across Cloud Data-centers”. In: Proceedings of International Conference on Distributed Event-Based Systems, 2014, pp. 23–34.

[122] Venkataraman, S.; Panda, A.; Ousterhout, K.; Armbrust, M.; Ghodsi, A.; Franklin, M. J.; et al. “Drizzle: Fast and Adaptable Stream Processing at Scale”. In: Proceedings of the Symposium on Operating Systems Principles, 2017, pp. 374–389.

[123] Vilches, A.; Navarro, A.; Asenjo, R.; Corbera, F.; Gran, R.; Garzaran, M. J. “Mapping Streaming Applications on Commodity Multi-CPU and GPU On-Chip Processors”,

IEEE Transactions on Parallel and Distributed Systems, vol. 27–4, April 2015, pp. 1099–1115.

[124] Vogel, A. “Adaptive Degree of Parallelism for the SPar Runtime”, Master’s Thesis, School of Technology - PPGCC - PUCRS, Porto Alegre, Brazil, 2018, 100p.

[125] Vogel, A.; Griebler, D.; Danelutto, M.; Fernandes, L. G. “Seamless Parallelism Management for Video Stream Processing on Multi-cores”. In: Proceedings of the International Conference on Parallel Computing, 2019, pp. 533–542.

[126] Vogel, A.; Griebler, D.; Danelutto, M.; Fernandes, L. G. “Minimizing Self-Adaptation Overhead in Parallel Stream Processing for Multi-Cores”, *Lecture Notes in Computer Science*, vol. 11997, May 2020, pp. 30–41.

[127] Vogel, A.; Griebler, D.; Danelutto, M.; Fernandes, L. G. “Self-adaptation on Parallel Stream Processing: A Systematic Review”, *Concurrency and Computation: Practice and Experience*, vol. 34, March 2022, pp. e6759.

[128] Vogel, A.; Griebler, D.; De Sensi, D.; Danelutto, M.; Fernandes, L. G. “Autonomic and Latency-Aware Degree of Parallelism Management in SPar”, *Lecture Notes in Computer Science*, vol. 11339, August 2019, pp. 28–39.

[129] Vogel, A.; Griebler, D.; Fernandes, L. G. “Providing High-level Self-adaptive Abstractions for Stream Parallelism on Multicores”, *Software: Practice and Experience*, vol. 51–6, June 2021, pp. 1194–1217.

[130] Vogel, A.; Griebler, D.; Maron, C. A.; Schepke, C.; Fernandes, L. G. “Private IaaS Clouds: A Comparative Analysis of OpenNebula, CloudStack and OpenStack”. In: Proceedings of Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2016, pp. 672–679.

[131] Vogel, A.; Mencagli, G.; Griebler, D.; Danelutto, M.; Fernandes, L. G. “Towards On-the-fly Self-Adaptation of Stream Parallel Patterns”. In: Proceedings of Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2021, pp. 89–93.

[132] Vogel, A.; Mencagli, G.; Griebler, D.; Danelutto, M.; Fernandes, L. G. “Online and Transparent Self-adaptation of Stream Parallel Patterns”, *Computing*, In press 2022, pp. 1–19.

[133] Vogel, A.; Rista, C.; Justo, G.; Ewald, E.; Griebler, D.; Mencagli, G.; Fernandes, L. G. “Parallel Stream Processing with MPI for Video Analytics and Data Visualization”. In: Proceedings of High Performance Computing Systems, 2020, pp. 102–116.

[134] Voss, M.; Asenjo, R.; Reinders, J. “Pro TBB: C++ Parallel Programming with Threading Building Blocks”. Apress, 2019, 754p.

[135] Wang, L.; Fu, T. Z.; Ma, R. T.; Winslett, M.; Zhang, Z. “Elasticutor: Rapid Elasticity for Realtime Stateful Stream Processing”. In: Proceedings of the International Conference on Management of Data, 2019, pp. 573–588.

[136] Weyns, D. “Software Engineering of Self-adaptive Systems: An Organised Tour and Future Challenges”. In: *Chapter in Handbook of Software Engineering*, Springer, 2017, pp. 1–26.

[137] Weyns, D. “An Introduction to Self-adaptive Systems: A Contemporary Software Engineering Perspective”. Wiley, 2020, 288p.

[138] Weyns, D.; Usman Iftikhar, M.; De La Iglesia, D. G.; Ahmad, T. “A Survey of Formal Methods in Self-adaptive Systems”. In: Proceedings of International Conference on Computer Science and Software Engineering, 2012, pp. 67–79.

[139] Wu, X.; Liu, Y. “Enabling a Load Adaptive Distributed Stream Processing Platform on Synchronized Clusters”. In: Proceedings of the IEEE International Conference on Cloud Engineering, 2014, pp. 627–630.

[140] Yuan, E.; Esfahani, N.; Malek, S. “A Systematic Survey if Self-protecting Software Systems”, *ACM Transactions on Autonomous and Adaptive Systems*, vol. 8–4, January 2014, pp. 1–41.

[141] Zacheilas, N.; Kalogeraki, V.; Zygouras, N.; Panagiotou, N.; Gunopulos, D. “Elastic Complex Event Processing Exploiting Prediction”. In: Proceedings of International Conference on Big Data, 2015, pp. 213–222.

[142] Zaharia, M.; Xin, R.; Wendelland, P.; Das, T.; Armbrust, M.; et al.. “Apache Spark: A Unified Engine for Big Data Processing”, *Communications of the ACM*, vol. 59–11, November 2016, pp. 56–65.

[143] Zeuch, S.; Monte, B. D.; Karimov, J.; Lutz, C.; Renz, M.; Traub, J.; Breß, S.; Rabl, T.; Markl, V. “Analyzing Efficient Stream Processing on Modern Hardware”, *Proceedings of the VLDB Endowment*, vol. 12–5, January 2019, pp. 516–530.

[144] Zhang, Q.; Song, Y.; Routray, R. R.; Shi, W. “Adaptive Block and Batch Sizing for Batched Stream Processing System”. In: Proceedings of the IEEE International Conference on Autonomic Computing, 2016, pp. 35–44.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Graduação
Av. Ipiranga, 6681 - Prédio 1 - 3º. andar
Porto Alegre - RS - Brasil
Fone: (51) 3320-3500 - Fax: (51) 3339-1564
E-mail: prograd@pucrs.br
Site: www.pucrs.br