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

RICARDO MACIEL LEONARCZYK

LATENCY-AWARE SELF-ADAPTIVE MICRO-BATCHING TECHNIQUES FOR GPU-ACCELERATED STREAM PROCESSING

Porto Alegre
2024

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

LATENCY-AWARE SELF-ADAPTIVE MICRO-BATCHING TECHNIQUES FOR GPU-ACCELERATED STREAM PROCESSING

RICARDO MACIEL LEONARCZYK

Master Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Dr. Dalvan Jair Griebler

Porto Alegre
2024

Ficha Catalográfica

M152L   Maciel Leonarczyk, Ricardo

Latency-aware self-adaptive micro-batching techniques for GPU-accelerated stream processing / Ricardo Maciel Leonarczyk. – 2024.

80p.

Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Dalvan Jair Griebler.

1. Stream processing. 2. Micro-batching. 3. Multicores. 4. GPUs. I. Griebler, Dalvan Jair. II. Título.

RICARDO MACIEL LEONARCZYK

LATENCY-AWARE SELF-ADAPTIVE MICRO-BATCHING TECHNIQUES FOR GPU-ACCELERATED STREAM PROCESSING

This Master Thesis has been submitted in partial fulfillment of the requirements for the degree of Master in Computer Science of the Computer Science Graduate Program, School of Technology of the Pontifical Catholic University of Rio Grande do Sul

Sanctioned on January 8, 2024.

COMMITTEE MEMBERS:

Prof. Dr. Massimo Torquati (University of Pisa)

Prof. Dr. Tiago Coelho Ferreto (PPGCC/PUCRS)

Prof. Dr. Dalvan Jair Griebler (PPGCC/PUCRS - Advisor)

Dedico este trabalho a meus pais.

ACKNOWLEDGMENTS

I want to thank everyone who, in one way or another, helped me complete this work. Your support and guidance meant a lot during my academic journey. Now for some specifics. First, a big thanks to my advisor. Your continuous support was crucial for this project, and I truly appreciate your belief in my potential and the time you spent helping me improve. I also want to thank my peers and colleagues from PUCRS/GMAP and PUCRS/LIS. Your insightful discussions made this project better, and your camaraderie made the journey through a project like this much more fulfilling. Special thanks to the evaluation committees for their helpful advice, which made this work better each time we met. Last but not least, a heartfelt thanks to my family and friends. Your constant patience and support really helped me get through it all.

This result was achieved in cooperation with HP Brasil Indústria e Comércio de Equipamentos Eletrônicos LTDA. using incentives of Brazilian Informatics Law (Law n$^\circ$ 8.248 of 1991).

TÉCNICAS DE MICRO-BATCHING AUTOADAPTÁVEIS E SENSÍVEIS À LATÊNCIA PARA PROCESSAMENTO EM STREAMING ACELERADO POR GPU

RESUMO

O processamento de streaming desempenha um papel vital em aplicações que exigem processamento contínuo de dados com baixa latência. Graças às suas extensas capacidades de processamento paralelo e custo relativamente baixo, as GPUs mostram-se adequadas para cenários nos quais tais aplicações requerem recursos computacionais substanciais. No entanto, o processamento em microlote torna-se essencial para uma computação eficiente em GPUs integradas a sistemas de processamento de streaming. O processamento em microlote introduz o desafio de encontrar tamanhos de lote apropriados para manter um nível de serviço adequado, especialmente em casos nos quais as aplicações de streaming enfrentam flutuações na taxa de entrada e carga de trabalho. Abordar esse desafio requer o ajuste do tamanho do lote em tempo de execução, o que pode ser feito por meio de autoadaptação. Nesta dissertação, avaliamos um conjunto de algoritmos autoadaptativos existentes e propostos para adaptação do tamanho do lote usando uma nova aplicação de processamento de streaming acelerada por GPU. Além disso, propomos um novo conjunto de métricas para ajudar a classificar e comparar os algoritmos de adaptação entre si em termos de qualidade de serviço sob diferentes perspectivas. Os resultados indicam que a aplicação testada e sua carga de trabalho altamente dinâmica representaram um desafio para algoritmos previamente avaliados em trabalhos relacionados, tornando-os 33% menos eficazes na manutenção da latência de microlote dentro dos requisitos de latência mais rigorosos. Os algoritmos atingiram um desempenho comparável na manutenção da latência dentro de níveis aceitáveis, na perspectiva dos elementos do stream. Além disso, com o conjunto certo de parâmetros, um de nossos algoritmos propostos pôde permanecer 40% mais próximo da latência alvo do que os outros algoritmos testados, devido à sua capacidade de combinar ajuste fino do tamanho do lote com reatividade.

Palavras-Chave: Processamento de streaming, Micro-batching, Multicores, GPUs.

LATENCY-AWARE SELF-ADAPTIVE MICRO-BATCHING TECHNIQUES FOR GPU-ACCELERATED STREAM PROCESSING

ABSTRACT

Stream processing plays a vital role in applications that require continuous, low-latency data processing. Thanks to their extensive parallel processing capabilities and relatively low cost, GPUs are well-suited to scenarios where such applications require substantial computational resources. However, micro-batching becomes essential for efficient GPU computation within stream processing systems. Micro-batching introduces the challenge of finding appropriate batch sizes to maintain an adequate level of service, particularly in cases where stream processing applications experience fluctuations in input rate and workload. Addressing this challenge requires adjusting the batch size at runtime, which can be done by using self-adaptation. In this thesis, we evaluated a set of existing and proposed self-adaptive algorithms for micro-batch size adaptation using a new GPU-accelerated stream processing application. Furthermore, we proposed a new set of metrics to help rank and compare the adaptation algorithms among themselves in terms of quality of service from different perspectives. The findings indicate that the tested application and its highly dynamic workload proved challenging for the existing algorithms previously evaluated in related work, making them 33% less effective in maintaining micro-batch latency for the most strict latency requirements. Among themselves, the algorithms attained comparable performance in maintaining latency within acceptable levels in the perspective of the stream data items. Furthermore, given the right set of parameters, one of our proposed algorithms could stay 40% closer to the target latency than others due to its ability to combine batch size fine-tuning and reactivity.

Keywords: Stream processing, Micro-batching, Multicores, GPUs.

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF ACRONYMS

AIMD – additive-increase/multiplicative-decrease

API – Application Programming Interface

B-SLH – Batched SLO Hit

CPU – Central Processing Unit

DSPS – Distributed Stream Processing System

EFC – Expert Fuzzy Control

FPGA – Field-Programmable Gate Array

GPGPU – General Purpose Graphics Processing Unit

I-SLH – Itemized SLO Hit

JVM – Java Virtual Machine

MAD-D – MAD-Based SLO Distance

MPI – Message Passing Interface

PID – Proportional-Integral-Derivative

PPI – Parallel Programming Interface

QOS – Quality of Service

SD-D – SD-Based SLO Distance

SLA – Service Level Agreement

SLO – Service Level Objective

SLR – Systematic Literature Review

SPAR – Stream Parallelism

SPA – Stream Processing Application

SPS – Stream Processing System

CONTENTS

1. INTRODUCTION

In recent decades, we have witnessed the emergence of a distinct class of applications demanding continuous processing of boundless data streams in near real-time. Stream processing has become a popular computing paradigm to address this demand [14]. In this context, a *stream* represents a constant flow of data. This data originates from diverse sources, including social media feeds, IoT sensors, and financial trading platforms [7, 42]. Specialized infrastructures known as stream processing systems (SPSs) have arisen To handle these demanding workloads.

An SPS is responsible for receiving and processing data items as they arrive, according to user-defined logic. This frees developers from the complexities of directly managing fault tolerance, parallelism, and data partitioning within their stream processing applications. SPSs such as Apache Spark [3] and Apache Flink [1] are popular options for building scalable and robust stream processing solutions. They have traditionally been deployed in commodity clusters, aiming at horizontal scalability [58, 12]. In such scenarios, the focus is typically on input/output (I/O), and the computing demands are relatively low, mainly comprising logic for data filtering and transformations. However, high-capacity computing infrastructure becomes critical for maintaining acceptable service levels for stream processing applications (SPAs) with high computational demands, such as those in computer vision [60], natural language processing [11], and robotics [17].

Accelerators such as graphics processing units (GPUs) offer significant advantages in such cases due to their capacity for massive data parallelism [33, 46]. However, integrating a GPU into an SPS can exhibit several challenges. For instance, when approaching some applications from domains such as streaming analytics or natural language processing [20, 50], data streams convey small items as attribute records (which we call *tuples*). Such tuples are often small (e.g., a few hundred bytes); thus, individually processing them on GPUs may impose an under-utilization of their computing capacity [15, 16]; in this case, it is more efficient to exploit only CPU parallelism instead of offloading tuples to the GPUs.

A potential solution to the challenge of GPU integration into SPSs is the processing of stream tuples in micro-batches. Micro-batching is a technique used in SPSs to process small batches of data simultaneously rather than processing tuples individually. The SPS collects a defined number of incoming tuples, groups them into a batch, and processes them as a computation unit. It is the default processing model for SPSs such as Spark Streaming (even without a GPU). In contrast, for tuple-at-a-time SPSs like Flink, the user can manually perform micro-batching when co-processors are present in the system or use other mechanisms that abstract this operation [15, 16].

Once micro-batching is adopted, a subsequent challenge arises in determining the optimal batch size (e.g., the number of tuples in the batch). Increasing the batch size in GPU-accelerated SPAs leads to a rise in both buffering requirements and the time it takes for the GPU to start processing. This results in a trade-off between latency and throughput, which can be managed by finding a batch size that can satisfy the particular streaming application's QoS (quality of service) requirements [46]. If a given SPA's workload and input rate are stable, finding a suitable batch size can be a one-time task, and it may be manually found with relative ease. However, streaming applications are often subjected to workload and input rate fluctuations, making the suitable batch size an evolving target.

Self-adaptation can be adopted to determine a suitable batch size at different stages of the SPA's execution, as demonstrated in Section 3. Broadly defined, self-adaptation is the capability of a system to autonomously change itself to better respond to its dynamic environment, keeping the quality of service (QoS) [53]. In practice, a self-adaptive system will collect metric data about the SPA, such as the processing time of the last tuple or batch. Based on them, it will perform adaptation actions, changing entities (e.g., batch size or parallelism degree) at run-time to achieve defined SLOs (service level objectives) like latency or throughput.

The main goal of this Master's thesis is to evaluate a set of existing and proposed self-adaptive algorithms for micro-batch size adaptation using a new GPU-accelerated stream processing application. Four out of the six evaluated algorithms come from Stein et al. [47] study (more detailed in chapter 3). They were originally applied to a GPU-accelerated data compression application. The fifth algorithm combines two of the algorithms from Stein et al. [47], created by us to fill a gap detected by our evaluation. The sixth algorithm is a proportional-integral-derivative (PID) controller, a classical control theory approach commonly used for industrial processes and automation [10]. The application we selected for the evaluation is the *Military Server Benchmark* [8] (MS benchmark), a highly configurable SPA designed to exploit stream combined with data parallelism, which allows workload variations.

We summarize our contributions as follows:

- An evaluation of the six mentioned self-adaptive algorithms with a new SPA (the MS benchmark).
- An evaluation of how latency sampling affects the adaptation. The study from Stein et al. [47] did not address this aspect, which affects the average of a set of latencies corresponding to the process variable in control theory.
- A set of three metrics that complement the metric used to compare the algorithms and workloads in the paper from Stein et al. [47], offering a novel way to analyze experiments with SLO-constrained SPAs that use micro-batching techniques.

- The introduction of dynamic batch support to the MS benchmark's architecture.
- The introduction of a new workload class to the MS benchmark, which justifies the use of micro-batching and provides a dynamic processing time for each stream tuple.

The remainder of this Master Thesis is structured as follows. Chapter 2 introduces the main concepts that form the basis of our work. Chapter 3 presents the related work and emphasizes the contributions of our work in comparison to the state-of-the-art. Chapter 4 details the development of the research, focusing on our contributions. Chapter 5 is dedicated to evaluating the performance of the self-adaptive algorithms (as measured by the SLO-centric metrics) within the context of our chosen SPA and workload. Finally, Chapter 6 provides a summary of our findings and outlines future directions for our research.

2. BACKGROUND

2.1 Stream Processing

Stream processing is a computing paradigm that concerns the processing of unbounded data streams in near real-time [7]. The term **stream** in this context refers to an ongoing flow of data, akin to how water moves in a river. The data, which may be unstructured (e.g., video frames and plain text) or structured (e.g., key-value pairs), can be derived from various sources. These sources include social media feeds, IoT sensors, and financial trading platforms [7, 42].

A single data unit in a stream may be called a data item or data tuple, among other terms [31]. A data item may consist of an entire image in a stream processing application (SPA) in fields such as computer vision. In contrast, the robotics domain could contain sensor data such as the robot's coordinates and other environmental information. The term tuple is commonly used for the latter case to specify that the data item is structured as a set of predefined attributes (like a row in a relational database) [7].

The infrastructure responsible for processing the stream is called a stream processing system (SPS). The SPS is designed to receive and process the data items according to the users' logic, freeing them from implementing fault tolerance, parallelism, and data partitioning directly in their stream processing applications. Among the most popular SPSs in industry and academia are Flink [1] and Storm [4], all of which are developed under the Apache Foundation. These SPSs are also called DSPSs, where "D" stands for distributed since they are most commonly used in clusters of commodity machines [58].

The Apache stream processing solutions are built on the JVM (Java Virtual Machine). Studies in the field of stream processing have introduced alternative SPSs that are not based on the JVM and, thus, are not subject to some of its inherent limitations, including the detrimental performance effects of garbage collection [13]. Some instances of such SPSs include FastFlow [6], WindFlow [40], and SPar [27]. These systems are typically written in C++ and tend to prioritize multi-core performance.

Stream processing evolved from batch processing, a way of processing large volumes of data in a finite and well-defined time frame. In batch processing, data is collected over a period of time, stored in a file or database, and then processed as a batch. One example of a batch processing system is the widely known Apache Hadoop [2]. Today, SPSs like Apache Flink provide support for both batching and streaming paradigms.

An intermediary computing paradigm exists between batch and stream processing, known as micro-batch processing. It is a compromise between the two purer approaches, as it provides a way to process data in small batches with a delay of usually a few seconds, rather than processing data in large batches (as in batch processing) or processing data tuple-at-a-time (as in stream processing). Spark Streaming is another popular solution from Apache Foundation, providing micro-batch processing as the paradigm for processing data streams [3].

Micro-batch can enhance performance by enabling operators to exploit batch parallelism [25]. Similarly, micro-batch is beneficial when operators offload work to accelerators like GPUs, especially if the accelerator is underutilized due to processing individual data items [41, 46]. Users of micro-batch systems often face a tradeoff between latency and throughput, which can be balanced by selecting appropriate batch sizes that meet the application's needs. Increasing the batch size can enable higher parallelism and improve throughput, but often at the cost of higher latencies when compared to processing one item at a time. The batch size can be defined as the number of items inside the batch or a predefined time interval in which items are collected. The latter definition is The one adopted by Spark Streaming, where batch interval is the term used to refer to batch size.

An application executing on an SPS is called a stream processing application (SPA). We can model an SPA as a directed graph, where the nodes correspond to operators and the edges define the data flow between operators [58]. An operator is responsible for receiving data items, processing them according to user-defined code, and possibly sending the results to another operator(s).

Certain patterns can be identified based on how nodes and edges are arranged in the SPA's graph. One of the canonical patterns is the pipeline [26, 39]. A Pipeline is created when the output of one operator serves as the input for another, possibly forming a chain of multiple operators. In this context, each operator can be considered as implementing a pipeline stage. Operators that do not receive input or do not produce output are commonly known as source and sink operators, respectively [59]. The source (first stage) is usually responsible for reading the data items and sending them downstream for processing, while the sink (last stage) collects the final processed items and produces the pipeline output. Some operators can be replicated to enable parallelism in a given pipeline stage, and each stage can asynchronously process different items.

Another notable parallel pattern is the task farm, also known as master-slave/worker or bag of tasks [26]. In this pattern, the processing is done by one or more replicated worker operators. The emitter operator sends tuples to the workers while the collector operator retrieves the processed tuples. The emitter can opt for different strategies to distribute work among the workers, such as round-robin or Priority-based distribution. The emitter and collector have the same role as the source and the sink when a given SPA comprises a single farm pattern. However, the patterns described can also be composed. A common pattern composition combines pipeline and farm patterns, resulting in a pipeline where each stage is a farm.

2.2 GPU-accelerated Stream Processing

In the past, GPUs (graphics processing units) were designed and optimized for graphics processing tasks, such as rendering complex 3D scenes and manipulating high-resolution images [21]. However, with the increasing demand for low-cost, high-performance computing solutions, especially in areas like scientific simulations, machine learning, and data analytics, researchers and developers started exploring the use of GPUs for general-purpose computing [44]. This led to the development of the GPGPU (General-Purpose GPU) programming models and frameworks, which allowed programmers to leverage the massively parallel architecture of GPUs for a wide range of compute-intensive applications. These applications include the matrix-oriented applications present in scientific computations, as well as media-oriented image and sound processing [30]. The shift towards general-purpose computing on GPUs has also driven innovations in hardware design, software optimization, and parallel algorithms, making GPUs a powerful and versatile tool for accelerating diverse workloads [44, 30].

Figure 2.1 generally depicts some important differences between a CPU and a GPU architecture. The CPU has a few powerful cores that can simultaneously execute distinct instructions and (individually) are optimized for processing serial code with complex instruction flow. This characteristic makes the CPU well-suited for tasks with unpredictable or irregular execution patterns. In contrast, the GPU has many cores (usually in the order of thousands), but these cores are individually simpler, slower, and share more chip resources (e.g., cache and instruction decoders). For these reasons, most GPU cores will simultaneously execute a single instruction over different parts of the memory. These characteristics make the GPU highly efficient at performing repetitive computations across large datasets simultaneously, making them ideal for tasks with regular and predictable execution patterns.

CPU cores tend to have bigger and faster caches inside the chip, mostly focusing on low-latency data access. Most of the data stays in the RAM memory, and the CPU leverages its sophisticated prediction mechanisms and ample cache size to accelerate computation. In comparison, a greater percentage of the cache available in the GPU is shared between multiple cores, with the number of cores varying depending on the cache level. Furthermore, GPUs typically rely on larger, high-bandwidth memory integrated into the chip, prioritizing high-throughput data access to support data parallelism.

As a co-processor, the GPU receives data and tasks from the host (the CPU). The program that runs inside the GPU is known as a kernel. Inside the kernel, the programmer partitions the data to be processed between blocks of threads. The number of blocks and threads per block depends on the application and is limited by GPU-specific hardware. Launching a GPU kernel is a four-step process:

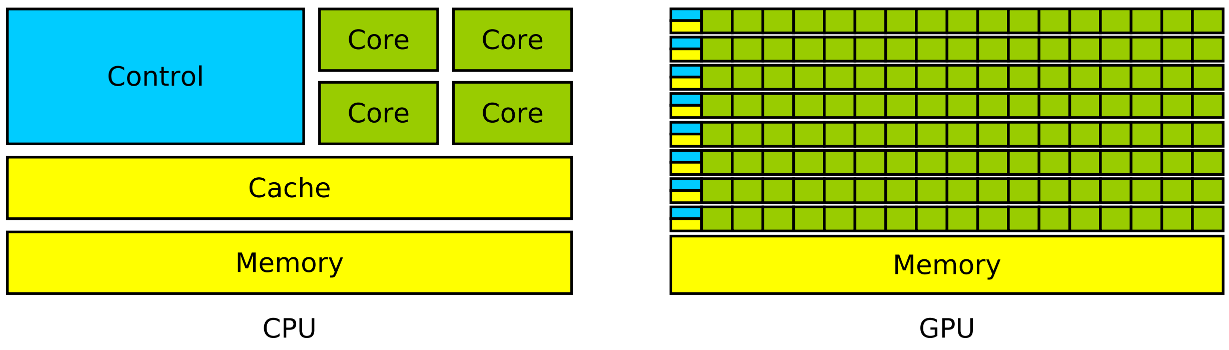

Figure 2.1 – A general comparison between the CPU and GPU architectures. Extracted from [8].

- Enough memory must be allocated for the input and output data. This memory is allocated in the GPU, which has its own memory separate from the host system.
- The input data is copied from the host memory to the allocated GPU memory.
- The kernel code is executed on the GPU, which performs the necessary computations on the input data.
- The output data is copied back from the GPU memory to the host memory for further processing or storage.

GPUs have also been found to be used in stream processing applications that benefit from their massive data parallelism. For SPSs such as FastFlow [6] and GStream [59], which allow arbitrary code inside user-defined operators, the GPU kernels can be launched in the same way as for non-streaming applications. As expected, a pipeline with multiple GPU-accelerated stages will possibly keep multiple kernels executing simultaneously, especially in the presence of replicated operators. Simultaneously running multiple kernels can help improve GPU resource utilization. Nevertheless, this approach may not suffice when the workload per item is small. Micro-batch processing can be advantageous in such cases since the overhead of launching and managing the kernel by item can be reduced, helping to increase the system's overall throughput.

There are multiple programming models available for writing GPU kernel code. For NVIDIA GPUs, the CUDA interface is the default provided by the vendor. OpenCL, a cross-platform framework for programming heterogeneous systems, is a possible alternative to CUDA. OpenCL allows users to execute the same kernel in GPUs from different manufacturers. The programming models for GPUs, CUDA, and OpenCL are low-level and allow the programmer to perform greater control of the device, with the drawback of reduced productivity. There is also a high-level alternative called OpenACC. It works through annotations to sequential code in the same fashion as OpenMP for multi-core processors. Another high-level approach is the SPar DSL [27], which was extended by Rockenbach[45] to target GPUs. The SPar compiler can generate both CUDA and OpenCL code using a lower-level library named GSParLib[45]. Direct usage of GSParLib is possible if the programmer needs greater control over the generated code.

2.3 Self-Adaptation

Self-adaptation refers to the ability of a system to modify its behavior or structure in response to changes in its environment or internal conditions [23]. By collecting additional runtime data (data not present before deployment), a self-adaptive system can better understand the uncertainties and challenges it faces [54, 53]. This information can then be used to reason about the system's goals and make the necessary reconfigurations to maintain or improve its overall performance. Ideally, a self-adaptive system should also be able to degrade gracefully in situations where it is unable to meet its goals or where performance is reduced, minimizing the impact of failures or errors.

Figure 2.2 illustrates a web server that continuously processes queued service requests. The requests enter, wait in the buffer, get processed by the server, and are dispatched (e.g., by returning an HTTP response to the client). We can use this queueing system to model the general case where the revenue is generated by subscriber transactions, such that the more requests, the better. However, we are also assuming that each request has an SLO constraint on the response time, meaning that the amount of time spent waiting and processing a given request should not exceed what was defined in the SLA (service level agreement). Given that, we aim to maximize the number of requests served while still adhering to our constraints. To stay within the SLA, we can sense when the response time is close to the limit and then simply redirect the incoming, possibly SLO-offending requests to another server (e.g., by an HTTP redirect). Therefore, we need to find a suitable buffer size for the system to operate effectively.

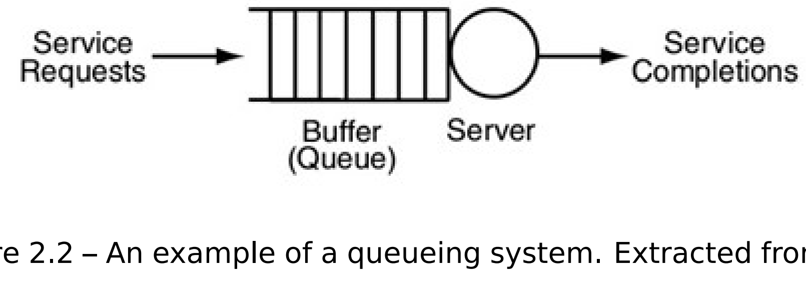

Figure 2.2 – An example of a queueing system. Extracted from [29].

The standard process of sensing and implementing adaptations is often realized as a feedback loop. In control theory, a feedback loop refers to the cyclical process of continuously measuring the target system's output, comparing it to the desired or reference value, and using the difference (error) to adjust the system's inputs [10]. When the system achieves the reference value (usually observing a tolerance) and remains at that value over an extended period, it is said to be in a steady state. This means that the system has reached a stable condition where parameters, such as the output, input, and other relevant variables, remain constant or fluctuate within a narrow range around fixed values.

Figure 2.3 demonstrates a feedback loop in the queueing system from Figure 2.2. The "Controller" component serves as the central hub for sensing and implementing adaptations. It continuously measures the system's performance (response time) and compares it to the reference. If the measured response time deviates from the reference, the controller dynamically alters the buffer size parameter within the queueing system. As the controller iteratively adjusts the buffer size based on feedback, it moves the system towards achieving a steady state where the measured response time aligns closely with the reference.

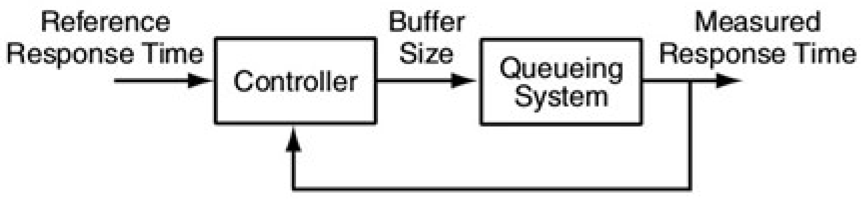

Figure 2.3 – An example of a queueing system with feedback control. Extracted from [29].

The opposite of the closed-loop control is the open-loop control. In open-loop control, also known as feedforward control, the system operates without continuous feedback regarding its output or performance. The control action is predetermined based on a model or set of rules, and it does not dynamically adjust to the system's actual behavior. This approach needs the control input to be a deterministic function of the reference value. For this reason, it is more applicable to predictable and stable systems. Otherwise, the model must be robust to predict changes and consider possible disturbances that may affect the system, which is difficult to achieve in practice [54].

It is possible to combine open-loop and closed-loop controls. In such scenarios, the open-loop control provides a predictive model for optimal operation, while the closed-loop control adapts in real-time to deviations from the expected behavior. The predictive nature of open-loop control is effective in dealing with known system behaviors, while closed-loop control excels in handling uncertainties, disturbances, and variations in real-world conditions [35].

Weyns[54] describes the SASO properties—Stability, Accuracy, Settling Time, and Overshoot. They can be used to understand how a self-adaptive system behaves at runtime.

- Stability is a fundamental property that ensures the system remains bounded and does not exhibit uncontrolled or oscillatory behavior over time.

- Accuracy refers to how closely the system output converges to the reference value.
- Settling time is the time it takes for the system to converge to its steady state value.
- Overshoot is the extent to which the system’s response exceeds the steady-state value before settling down.

3. RELATED WORK

In a recent SLR regarding self-adaptation on parallel stream processing from Vogel et al.[53] it can be noted that approximately 70% of all selected studies explored the adaptation of inter and intra-node parallelism, demonstrating that this is a common approach for maintaining QoS. However, due to the significant impact of the batch size on micro-batch streaming applications [22, 57, 25] as a parameter that can directly affect parallelism efficiency [24], and consequently latency, throughput and system stability [22], our perspective is that it has not been investigated in the current literature as thoroughly as parallelism has been. This notion is further supported by the number of studies adapting the batch size, 9 out of a total of 65 classified studies from Vogel et al.[53]. In addition, the batch size is even more relevant when an accelerator such as a GPU or an FPGA is present in the SPS (be it of micro-batching or tuple-at-a-time nature), making the investigation of its effects a profitable endeavor, however, it is not currently receiving much attention [34, 59, 55].

In the following section, we present the only two studies we could find that provide adaptive micro-batching on SPS with GPUs, as well as some other studies describing SPSs that do not adapt but provide the infrastructure to adapt batch size for efficient GPU offloading.

3.1 Micro-batching on SPSs with GPUs

Stein et al.[47] presented a control loop strategy driven by four algorithms for adapting the batch size to minimize latency on streaming compression applications targeting GPUs. The algorithms proposed expect a target latency to be informed by the user through the SPar[27] PPI's SLO mechanism[28]. Internally, they also consider a threshold indicating an acceptable percentage of variation in latency, as well as an adaptation factor that controls the step size of the adapting operation.

In terms of behavior, three of the algorithms from Stein et al.[47] are only reactive, one is proactive and will try to come as close as possible to the target latency, ignoring the threshold, and three will adjust (make elastic) the adaptation factor considering the distance from the target latency. Furthermore, some will converge faster than others according to their parameters and workload. The authors also provided a comprehensive evaluation that details each algorithm's suitability according to the workload and latency requirements. In general, the algorithms with the most elastic adaptation factor presented a better response when the workload was stable, while the proactive (with a narrower target) algorithm performed better with unbalanced workloads.

An important contribution of Stein et al.[47] was in providing a way of handling highly irregular workloads. Another characteristic that distinguishes their work from related efforts is the user's specification of a target latency instead of always optimizing for the lowest possible latency. This approach helps to avoid optimizing when the user does not need it, possibly generating some overhead. Additionally, by controlling the target latency, the user might be able to control the throughput indirectly, even though this relationship between target latency and throughput was not investigated in the study being discussed.

De Matteis et al.[24] propose Gasser, an SPS that offloads sliding windows for processing on GPUs. The batch size is measured in the number of tuples, while the authors use the term batch length to express the number of (always fully contained) windows inside a batch. The parallelism is measured as the number of distinct windows executing at a given time (inter-window parallelism). Additionally, Gasser can self-adapt the batch length and parallelism to optimize latency but focus more on throughput optimization. At the beginning of the execution, gasser calibrates a predictive model based on the throughput that a given configuration (batch length and parallelism degree) achieves on CPU and GPU to decide which is the most suitable processor to execute the Windows. The model becomes sufficiently accurate to predict the outcomes of untried configurations after some time. Only the parallelism is considered on the CPU, while the batch length is also considered for the GPU. An important drawback of Gasser's predictive model is that the calibration happens once at the beginning of the execution, and a stable input rate is assumed; thus, it does not react properly to irregular workloads (which require varying batch sizes throughout the execution).

Even when there is no self-adaptation of the batch size, the presence of an accelerator like a GPU in a SPA often requires batching for it to be efficient. For this reason, authors will usually make available means for the user to control the batch size when releasing tools for developing GPU-accelerated SPAs.

GStream[59] is one of those tools. It provides an API from which the user can set an acceptable range of sizes for the batches. The operators' code (also provided by the user) must be capable of processing any batch size in the specified range. Using a range instead of a single value allows user-provided or future GStream-provided algorithms to adapt the batch size. Another potential utility of GStream for self-adapting the batch size is that it allows the aforementioned range to be personalized for each operator. Moreover, GStream can work with clusters of GPUs through the underlying (hidden from the PPI's user) usage of MPI. Self-adaptation of batch sizes on multiple GPUs remains an unexplored topic in the literature as far as we know, with state-of-the-art studies opting for simpler architectures containing a single GPU-accelerated operator [24, 47].

In the same vein, Saber[34], as a relational SPS supporting GPUs and CPUs, utilizes the offloading of batches to the GPU. An important feature of Saber is the decoupling of the batch size from the SQL-defined window size and slide, by which it can separate the parallel processing efficiency from the query semantics. However, this decoupling implies window fragments being periodically present inside batches, and thus, Saber can only work with query operators with incremental algorithms, such as aggregations. Additionally, Saber can autonomically decide whether a given query operator will execute on the CPU or GPU through its adaptive scheduling strategy. The scheduler considers the throughput history a query operator accrued among the available heterogeneous processors and assigns the tasks according to the highest throughput values and processor load.

The introduction of GPUs into traditionally tuple-at-a-time SPSs such as Storm also requires batching for the accelerator to be effective. G-Storm[18] provides a batcher module that (in cooperation with buffering and indexing modules) is responsible for receiving the tuples, accumulating them into a buffer up to a specified batch size, and then offloading them to the GPU for data parallelism.

3.2 adaptive micro-batch SPSs

Micro-batch processing is not only useful when co-processors are present within an SPS. It can also be used to diminish the negative performance impact of fault tolerance [56] or to process data at higher rates [52]. In this section, we review the micro-batch adaptation strategies from studies that do not employ GPUs but can help us understand how this type of adaptation is being conducted in the literature and how their techniques might impact our future work. Most work of this nature makes use of micro-batch DSPSs such as Spark Streaming, which already provides static batch size [22, 57, 19]. Since by default, the batch size does not change according to operating conditions, the traditional way to deal with workload fluctuations is through resource provisioning [51].

Das et al.[22] proposed the use of the fixed point iteration method [49] (a well-known optimization technique) to find the intersection between batch processing time and batch interval (the point where the system is considered to be stable). An advantage of the algorithm is that there is no need to specify a step size, it adapts the step size according to the distance from the intersection. Nevertheless, the authors had to introduce a slack factor and a superlinear reduction factor, to (respectively) control the system's sensitivity to input workload fluctuations and to improve convergence in superlinear workloads (where increasing the batch size causes the processing time to increase at a rate that exceeds linearity). These parameters have default values, which were found to work with most evaluated workloads.

An important drawback is that the algorithm from Das et al.[22] assumes a batch interval where the processing rate can keep up with the input rate. If no such condition exists, it will not converge and will infinitely increase the batch interval. The only alternative in these cases is to employ a rate limiter or load shedder to prevent system overloading. This limitation arises mainly from the algorithm adopting a zero-prior-knowledge approach, considering only online information (the processing time of the last two completed batches) as statistics to decide the next batch interval. Another limitation is that the algorithm can present large control loop delays when the workload variation suddenly increases (such as greater than 10x from the previous batch).

Strategies to adjust the batch size produce lower overhead and fast convergence the less information needs to be collected about the system. However, with less information some limitations might arise, as Das et al.[22] have demonstrated. The work from Zhang et al.[57] explores a different strategy considering online historical statistics to adapt batch interval through isotonic regression and block interval through a heuristic approach. In Spark Streaming, the batch interval is defined as a series of block intervals. The number of blocks defines batch parallelism, although it is not a one-to-one relation. In this way, Zhang et al.[57] could adapt the batch size and parallelism with their control algorithm, entitled DyBBS (Dynamic Block and Batch Sizing). DyBBS [57] focuses on accuracy instead of higher convergence time, using the batch interval duration as the time boundary to feed the regression model and to modify the system. The historical data collected comprises the input rate, block interval, batch interval, and processing time, recorded for each batch and block interval combination. While Das et al.[22] could handle reduce, join, and window workloads with their approach, Zhang et al.[57] could only handle reduce and join workloads. However, Zhang et al.[57] obtained lower control loop delay, lower latency, and more accurate batch interval prediction. DyBBS [57] also assumes a batch interval in which the system is kept stable, otherwise requiring techniques such as load shedding or rate limiting.

Mai et al.[38] presented Chi, a framework for monitoring, feedback and reconfiguration of DSPs. Chi works by embedding its control messages into the application's data stream. This technique gives orders to the operators to reconfigure themselves and send information downstream to the controller, which stays right after the sink operators. The authors implement the tuning of batch size to illustrate Chi's functionality. In their example, the batch size starts with a default (and small) value. At every 30 seconds, the controller checks for the existence of back pressure, as well as current latency and throughput. If the system is not keeping up with the current input rate or any SLO is being violated, the batch size is doubled. Afterward, if the controller receives information indicating that the operating conditions worsen, it cuts the batch size by half. This example of tuning the batch size is simpler when compared to other presented studies; nevertheless, it demonstrates the possibilities of Chi as a framework to collect information and react to changes to the DSPS.

Studies like those presented by Das et al.[22] and Zhang et al.[57] treat the SPSs as a black box, focusing on modeling the relationship between batch size and batch processing time. Orthogonal to this, it is still possible to work within the batch to decrease latency and increase throughput, such as by devising different data partitioning schemes [5], optimizing parallelism [37, 52], or parallelizing recovery tasks[52]. Seeking to improve batch processing time is useful to avoid situations where the black box algorithms cannot find a match between the batch interval and batch processing time, thus circumventing the need for solutions like load shedders, which are unacceptable for some systems. In the following studies, the batch size is adapted to SPS components that participate in the processing time.

In some cases, the batch size is associated with the efficiency of task schedulers. Venkataraman et al.[52] propose Drizzle, an SPS that organizes the batches into groups and provides a technique to dynamically adapt the group size based on the AIMD (additive-increase/multiplicative-decrease) feedback control algorithm used on TCP congestion control [9]. This self-adapting scheme is part of a centralized scheduler that aims to decouple the batch processing time from the time spent coordinating fault tolerance and adaptability. It operates by tracking the time spent in scheduling and overall execution and using this information to keep the scheduler overhead within user-specified lower and upper bounds by multiplicatively increasing or additively decreasing the group size. As the group size increases, the parallelism of fault recovery increases, but so does the scheduling overhead. A disadvantage of this approach is that the users need to discover the best range to define the boundaries for their specific environments through experimentation.

Cheng et al.[19] propose adapting the batch interval through the expert fuzzy control (EFC) technique. The adaptive mechanism is integrated with a new scheduler for Spark Streaming named A-scheduler. The scheduler also adapts the job parallelism by a reinforcement learning algorithm. The fuzzy controller adapts the batch interval according to historic workload variations and processing times. The user-defined parameters for the controller are the step size and control period (e.g., an interval for the controller to change the batch size). Both parameters were adjusted according to the characteristics of the proposed scheduler and environment.

3.3 Summary and Comparison of Related Work

Table 3.1 summarizes the related work presented in this chapter. The column named entity refers to an adaptation action such as batch size or parallelism. The entities are generally defined so that batch size can mean batch interval or data length, and parallelism can mean inter or intra-node parallelism. When the entity and SLO columns are not being informed, the study on that row does not adapt any entities to achieve the goal described in the goal column. As for the goal column itself, it is being used to describe a specific objective of a study, selected according to what we deemed more important for inclusion as related work (usually related to the batch size).

Table 3.1 – Summary of related work

Reference	Entity	SLO	GPU support	Goal	Technology
Stein et al.[47]	Batch size	Latency	Yes	Four reactive algos. for meeting a latency SLO.	SPar and CUDA
De Matteis et al.[24]	Batch size, parallelism	Latency, throughput	Yes	Predictive model for adaptation of window-based operators.	FastFlow and CUDA
GStream[59]	-	-	Yes	API to set operator-specific batch size range.	CUDA and MPI
Saber[34]	Parallelism	Latency, throughput	Yes	Adaptively scheduling of operator execution between CPU or GPU	Java, OpenCL
G-Storm[18]	-	-	Yes	Batch module for GPU offloading in tuple-at-a-time SPS.	Storm and JCUDA
Das et al.[22]	Batch size	Latency	No	Algo to intersect batch interval and batch processing time.	Spark Streaming
Zhang et al.[57]	Batch size, parallelism	Latency	No	Adaptation of batch and block interval through a regression model.	Spark Streaming
Venkataraman et al.[52]	Batch size, parallelism	Latency, throughput	No	Scheduler that groups batches and uses algo to adapt group size.	Spark
Cheng et al.[19]	Batch size, parallelism	Latency, throughput	No	adaptive scheduler using fuzzy control for batch interval and reinf. learning for parallelism.	spark Streaming
Ours	Batch size	Latency	Yes	Six reactive algos. for meeting a latency SLO.	FastFlow and GSParLib

As demonstrated in this chapter, batch size tuning has been the route taken by a small but diverse quantity of studies in pursuit of improving QoS. We identified six studies with the batch size as an adaptation action (also called entity). Besides the batch size, four of the six studies adapt additional entities such as parallelism degree or the number of nodes. Only two of these (de Matteis et al.[24] and Stein et al.[47]) have GPUs in their pipelines, making them the most related to our work.

Most studies that do not use GPUs use micro-batch DSPSs such as Spark Streaming, while the GPU ones (like ours) use only one node and one GPU. Another notable difference between these two types of work is that when using DSPSs, the batch size is defined by a time interval (called batch interval). When GPUs are used, the batch size is usually defined by the data length (as measured in bytes, tuples, or windows).

Only the study from Stein et al.[47] (and by extension our own) uses the concept of a user-defined SLO to be achieved by the system. Using an SLO can be beneficial in preventing unnecessary optimization when the user does not require it, thereby averting potential instability. The SLO in both theirs and our case is defined in terms of a latency target and a threshold around the latency. Six of the reviewed studies pursue latency reduction, while three of them also considered the tradeoff of trying to maximize throughput. Since micro-batch streaming naturally incurs higher latencies when compared to tuple-at-a-time streaming, having latency as an SLO is a profitable option, while striking a balance between latency and throughput can be considered ideal for most streaming applications requiring micro-batching.

Turning our attention to the type of adaptation being used in the studies, we encounter both reactive and predictive approaches. Reactive strategies, adept at handling real-time changes with minimal knowledge about the system, are generally lighter in performance and simpler in code compared to predictive ones, which necessitate maintaining a model of the system. These strategies are present in the studies from Stein et al.[47], Das et al.[22], and Venkataraman et al.[52].

Predictive strategies, observed in the studies by Matteis et al.[24], Zhang et al.[57], and Cheng et al.[19], come with the drawback of finding the desired batch size only once and subsequently assuming that the system will continue to behave in accordance with the model. This assumption is often present owing to the high-performance cost associated with model recalibration. While reactive strategies require a system disturbance to trigger adaptation, predictive strategies can act proactively. To compete effectively, reactive algorithms must respond swiftly without causing additional disturbances.

Apart from the conventional latency and throughput metrics prevalent in most studies, Stein et al.[47] introduce the SLO hit metric. This metric, which we named batched SLO hit in our work (Section 4.3), measures the success of batching in maintaining latency within threshold bounds. Additionally, the Chebyshev distance is used in the study by de Matteis et al.[24] to measure how far the throughput deviates from the optimal theoretical throughput for the system.

4. CONTRIBUTIONS TO SELF-ADAPTIVE STREAMING SYSTEMS: NOVEL ALGORITHM APPLICATIONS, STREAMING WORKLOAD, AND SLO-CENTRIC PERFORMANCE METRICS

4.1 Self-adaptive Micro-batch Implementation

We adopted a feedback control strategy [29], that works by turning the SPA's pipeline into a closed loop in which the last stage (the monitor) collects and sends performance metrics to the first stage (the controller). The first stage then decides through the use of a decision algorithm whether to modify the batch size and, if so, by how many tuples. Figure 4.1 depicts the graph of the previously described pipeline. In this example, the source stage sends batches to a GPU-accelerated worker stage, which delivers the processed batches to be collected by the sink stage. The sink will produce the pipeline output and then send the batch to the monitor stage, which will collect the metrics (the latency from source to sink in this case) and then send it to the controller stage. The controller will then decide on the new batch size based on the latency and send it back to the source stage.

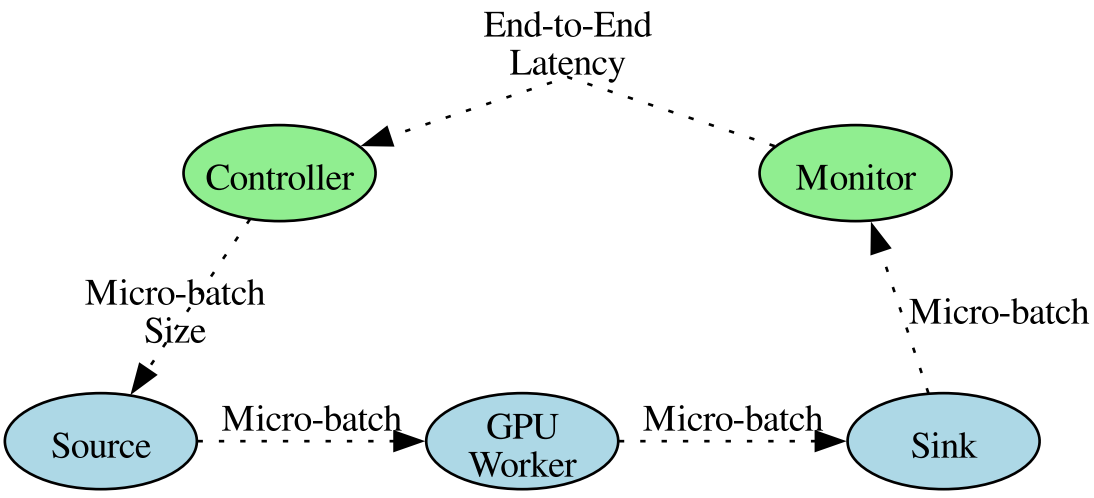

Figure 4.1 – Graph of a pipeline with a feedback loop.

Although it can be useful to think of the pipeline in figure 4.1 as having additional monitoring and control stages, it is convenient to merge the monitoring stage with the sink and the control stage with the source in the implementation, thereby avoiding extra communication costs. Moreover, communication between the last and first stages (the closed loop) should not result in any waiting conditions. If the latency from a delivered batch does not arrive in time to decide the size of the next batch, the batch is delivered with the last received size from the controller.

Since the algorithms from Stein et al. [47] are fairly lightweight, chiefly consisting of two conditional statements, we could afford to execute them for each batch. However, they are only executed once we have received a number *x* of measured latencies from the monitor, so an average of the last latencies can be used as input. This tends to avoid disturbances from specific batches containing latencies that significantly deviate from their neighbors.

As previously discussed briefly in Section 3, Stein et al.[47] presented four decision algorithms, namely Fixed Adaptation Factor (FAF), Percentage-Based Adaptation Factor (PBAF), PBAF Without Threshold (PBAF-WT), and Multiplier-based Adaptation Factor (MBAF). All algorithms accept as parameters a target latency, a threshold defining the allowed region around the target latency, and a step size (called adaptation factor by the authors). The goal is to keep the target latency inside the threshold bounds, expressed in Percentage.

4.1.1 Algorithms Currently Applied in the Self-adaptation of Batch Size for GPU-accelerated Stream Processing

The most straightforward algorithm is FAF. It works by incrementing the batch size by a fixed number if the latency passes the lower bound or decrementing it if the latency passes the upper bound.

Algorithm 4.1 – FAF (Fixed Adaptation Factor)

```
1:  UpperLim $\leftarrow$ Target $\times$ (1 + Threshold)
2:  LowerLim $\leftarrow$ Target $\times$ (1 $-$ Threshold)
3:  procedure FAF:PLAN(LastLatency, MicroBatchSize, AdaptationFactor)
4:      if LastLatency $>$ UpperLim then
5:          return MicroBatchSize $-$ AdaptationFactor
6:      end if
7:      if LastLatency $<$ LowerLim then
8:          return MicroBatchSize + AdaptationFactor
9:      end if
10:     return MicroBatchSize
11: end procedure
```

The FAF algorithm 4.1 starts by calculating the upper and lower limits, `UpperLim` and `LowerLim`, respectively, based on the target latency and a given threshold. Inside the `FAF:PLAN` procedure, lines 4 and 7 contain the conditionals necessary for knowing whether the batch size should be decreased or increased. If none of the conditions are met, we are inside the threshold bounds, and the batch size stays the same.

The other three algorithms largely follow the previously described pattern. In the case of PBAF, when it is distant from the target latency, it behaves like FAF and uses the whole adaptation factor. However, as it gets closer, it starts fractioning the adaptation factor so as to be more precise and to avoid stepping to the opposite side of the threshold bounds.

Algorithm 4.2 presents the pseudocode for PBAF. Noticed that in lines 7 and 11 PBAF uses an additional parameter called `MaxGrowBoundary` to define when to start fractionating the adaptation factor. It is assigned a value of 60% (0.6), such that when the measured latency has passed from 60% of the target, the whole adaptation factor is used, as in FAF.

Algorithm 4.2 – PBAF (Percentage-Based Adaptation Factor)

```
1:  UpperLim ← Target × (1 + Threshold)
2:  LowerLim ← Target × (1 − Threshold)
3:  MaxGrowBoundary ← 0.6
4:  **procedure** PBAF:PLAN(LastLatency, MicroBatchSize, AdaptationFactor)
5:      Perc ← LastLatency/Target
6:      **if** LastLatency > UpperLim **then**
7:          AdaptationPercentage ← min ((Perc − 1)/MaxGrowBoundary, 1)
8:          **return** MicroBatchSize − AdaptationFactor × AdaptationPercentage
9:      **end if**
10:     **if** LastLatency < LowerLim **then**
11:         AdaptationPercentage ← min ((1 − MaxGrowBoundary)/Perc, 1)
12:         **return** MicroBatchSize + AdaptationFactor × AdaptationPercentage
13:     **end if**
14:     **return** MicroBatchSize
15: **end procedure**
```

Algorithm 4.3 presents PBAF-WT (PBAF Without Threshold). The PBAF-WT algorithm is a version of PBAF in which the measured latency is compared directly with the target latency, ignoring the threshold bounds (variables `LowerLim` and `UpperLim`). Other than this, it is the same as the PBAF algorithm. The differences are on lines 5 and 9.

The narrow focus on the target makes PBAF-WT change the batch size more often. This can improve precision while avoiding being too close to the boundaries of the threshold. However, there is also the risk of going beyond the boundaries if the adaptation factor used is too large inside the threshold bounds.

The last algorithm presented by Stein et al.[47] is MBAF. In this algorithm, the goal is to get within the threshold bounds as fast as possible. To accomplish this, the adaptation factor is multiplied according to the distance from the region. Algorithm 4.4 presents MBAF's pseudocode.

Algorithm 4.3 – PBAF-WT (PBAF Without Threshold)

```
1: MaxGrowBoundary $\leftarrow$ 0.6
2: procedure PBAF-WT:PLAN(LastLatency, MicroBatchSize, AdaptationFactor)
3:     Perc $\leftarrow$ LastLatency/Target
4:     if LastLatency $>$ Target then
5:         AdaptationPercentage $\leftarrow \min((\text{Perc} - 1)/\text{MaxGrowBoundary}, 1)$
6:         return MicroBatchSize $-$ AdaptationFactor $\times$ AdaptationPercentage
7:     end if
8:     if LastLatency $<$ Target then
9:         AdaptationPercentage $\leftarrow \min((1 - \text{MaxGrowBoundary})/\text{Perc}, 1)$
10:        return MicroBatchSize $+$ AdaptationFactor $\times$ AdaptationPercentage
11:    end if
12:    return MicroBatchSize
13: end procedure
```

Algorithm 4.4 – MBAF (Multiplier-based Adaptation Factor)

```
1: UpperLim $\leftarrow$ Target $\times$ (1 $+$ Threshold)
2: LowerLim $\leftarrow$ Target $\times$ (1 $-$ Threshold)
3: procedure MBAF:PLAN(LastLatency, MicroBatchSize, AdaptationFactor)
4:     if LastLatency $>$ UpperLim then
5:         AdaptationMultiplier $\leftarrow$ LastLatency/Target
6:         return MicroBatchSize $-$ AdaptationFactor $\times$ AdaptationMultiplier
7:     end if
8:     if LastLatency $<$ LowerLim then
9:         AdaptationMultiplier $\leftarrow$ (2 $\times$ Target $-$ LastLatency)/Target
10:        return MicroBatchSize $+$ AdaptationFactor $\times$ AdaptationMultiplier
11:    end if
12:    return MicroBatchSize
13: end procedure
```

The main lines that differentiate MBAF from the other algorithms are lines 5 and 9, responsible for creating the variable `AdaptationMultiplier`. This variable will be used to scale the step size. It will always be greater than or equal to one. On line 9 from algorithm 4.4, the adaptation factor multiplier for the lower bound is defined such that it will always be between 1 and 2, meaning that the adaptation factor will at most double whenever the measured latency is small enough.

4.1.2 Proposed Algorithms to be Applied in the Self-adaptation of Batch Size for GPU-accelerated Stream Processing

The first algorithm we propose is the Percentage and Multiplier-Based Adaptation Factor (PMBAF). It takes inspiration from both PBAF and MBAF. From PBAF it inherits the capability of reducing the adaptation factor when close enough to the target, and from MBAF it inherits the capability of increasing the adaptation factor when far enough from the target. Algorithm 4.5 presents the pseudocode for PMBAF.

Algorithm 4.5 – PMBAF (Percentage and Multiplier-Based Adaptation Factor)

```
1: UpperLimPBAF $\leftarrow$ 0.8
2: LowerLimPBAF $\leftarrow$ 0.3
3: **procedure** PMBAF:PLAN(LastLatency, MicroBatchSize, AdaptationFactor)
4:     **if** LastLatency $>=$ Target $\times$ (1 $-$ LowerLimPBAF) $\wedge$ LastLatency $<=$ Target $\times$ (1 + UpperLimPBAF) **then**
5:         **return** PBAF:PLAN(LastLatency, MicroBatchSize, AdaptationFactor)
6:     **end if**
7:     **return** MBAF:PLAN(LastLatency, MicroBatchSize, AdaptationFactor)
8: **end procedure**
```

We are reusing (calling) the procedures from PBAF and MBAF defined in algorithms 4.2 and 4.4. The procedures are called according to the `UpperLimPBAF` and `LowerLimPBAF` variables. Those variables define the region around the target where the PBAF algorithm should be applied. If we are not in this condition, then MBAF is called.

There is a difference between the MBAF from Algorithm 4.4, and the MBAF implemented within PMBAF's real (C++) code. This is not shown in Algorithm 4.4 for the sake of simplicity. The difference is that when MBAF is inside the conditional for the threshold lower bound, like on line 9 in Algorithm 4.4, it does not have the limitation of doubling the step size at most. Its reaction is always proportional to the distance from the lower threshold, so the greater the distance, the greater the step size. Mathematically, we go from the formula $(2 \times \text{Target} - \text{LastLatency})/\text{Target}$ to the formula $1/(\text{LastLatency}/\text{Target})$.

The PID controller is the second algorithm we propose for the evaluation with the MS benchmark. The Proportional-Integral-Derivative (PID) controller represents a classical control theory approach that is widely employed in various industrial processes and automation systems [10]. In our scenario, the PID controller is used to dynamically adapt the batch size in a stream processing system, where the process value is the measured latency, the setpoint is the target latency, and the controller (or PID) output is the batch size.

The PID controller is usually used for regulating continuous variables. Since the batch size is a discrete value, our approach was to tune the PID parameters with integer values and reset the integral term to the minimum batch size whenever it became less than one. Besides this, the PID output also had to be truncated to the nearest integer value.

The equation for the PID controller is as follows:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\, d\tau + K_d \frac{de(t)}{dt} \tag{4.1}$$

In equation 4.1, $u(t)$ denotes the control output at time t. The three PID terms can be identified by having the + symbol as a separator. The gains K_p, K_i, and K_d are the tuning parameters that should be adjusted according to the specific dynamics of the system being controlled.

The Proportional (P) term immediately responds to the current latency error. It scales the correction signal based on the magnitude of the error, allowing for a quick reaction (depending on the gain) according to the distance from the target latency.

The Integral (I) term addresses any persistent steady-state latency error by accumulating the past errors over time. From the perspective of the previously discussed self-adaptive algorithms, the integral term operation can roughly be compared to the changes in the batch size by incrementing or subtracting a step size without a notion of time.

Finally, the Derivative (D) term anticipates future trends in the system's latency by considering the rate of change of the error. The goal is to prevent overshooting or oscillations in the latency.

Algorithm 4.6 presents the pseudocode for the PID controller.

In the PID:PLAN procedure, the error is calculated based on the difference between the target and the last latency (line 4). The proportional term (`PTerm`), integral term (`ITerm`), and derivative term (`DTerm`) are then computed on lines 5, 6, and 9, respectively.

The PID result is obtained by summing the three terms on line 11. After this, , the integral term is reset to prevent windup if the result falls below the minimum batch size. Finally, the new batch size is calculated based on the truncated PID result

Algorithm 4.6 – Proportional-Integral-Derivative (PID) Controller

```
1: Integral $\leftarrow$ 0
2: PreError $\leftarrow$ 0
3: **procedure** PID:PLAN(LastLatency, Kp, Ki, Kd, dt)
4:     Error $\leftarrow$ (Target $-$ LastLatency)/Target
5:     PTerm $\leftarrow$ Kp $\times$ Error
6:     Integral $\leftarrow$ Integral + Error $\times$ dt
7:     ITerm $\leftarrow$ Ki $\times$ Integral
8:     Derivative $\leftarrow$ (Error $-$ PreError)/dt
9:     DTerm $\leftarrow$ Kd $\times$ Derivative
10:    PreError $\leftarrow$ Error
11:    PIDResult $\leftarrow$ PTerm + ITerm + DTerm
12:    **if** PIDResult $<$ 1 **then**
13:        Integral $\leftarrow$ 0
14:    **end if**
15:    NewBatchSize $\leftarrow$ floor(PIDResult)
16:    **Return** min (NewBatchSize, 1)
17: **end procedure**
```

4.1.3 Summary of the Self-adaptive Algorithms Based on the Adaptation Factor (Step Size)

The charts shown in Figure 4.2 summarize and illustrate the behaviors discussed regarding the algorithms. On the x-axis, we depict the latencies provided as input to the algorithms, while the y-axis represents the step sizes chosen by the algorithms based on their distance from the target. A positive step size implies an increase in batch size by the specified value, a negative step size indicates a decrease, and a step size of 0 denotes no adaptation, maintaining the current batch size.

Examining the FAF chart allows us to deduce the default step size since FAF consistently maintains the same step size, evident as 10 in this case. For other algorithms, the PBAF graph illustrates a decreasing step size as it approaches the target, stopping adaptation once inside the threshold. PBAF-WT continues adjusting the step size even within the threshold. Meanwhile, MBAF increases the step size when distant from the target and reduces it to a value approximating the default when nearing the target.

At last, PMBAF behaves similarly to MBAF when distant from the target, and it behaves similarly to PBAF when close enough to the target. However, it noticeably differs from MBAF in the choice of step size, opting for large values when the latency is below the threshold. In Algorithm 4.5, the lines 1 and 2 define the upper and lower limits for applying PBAF.

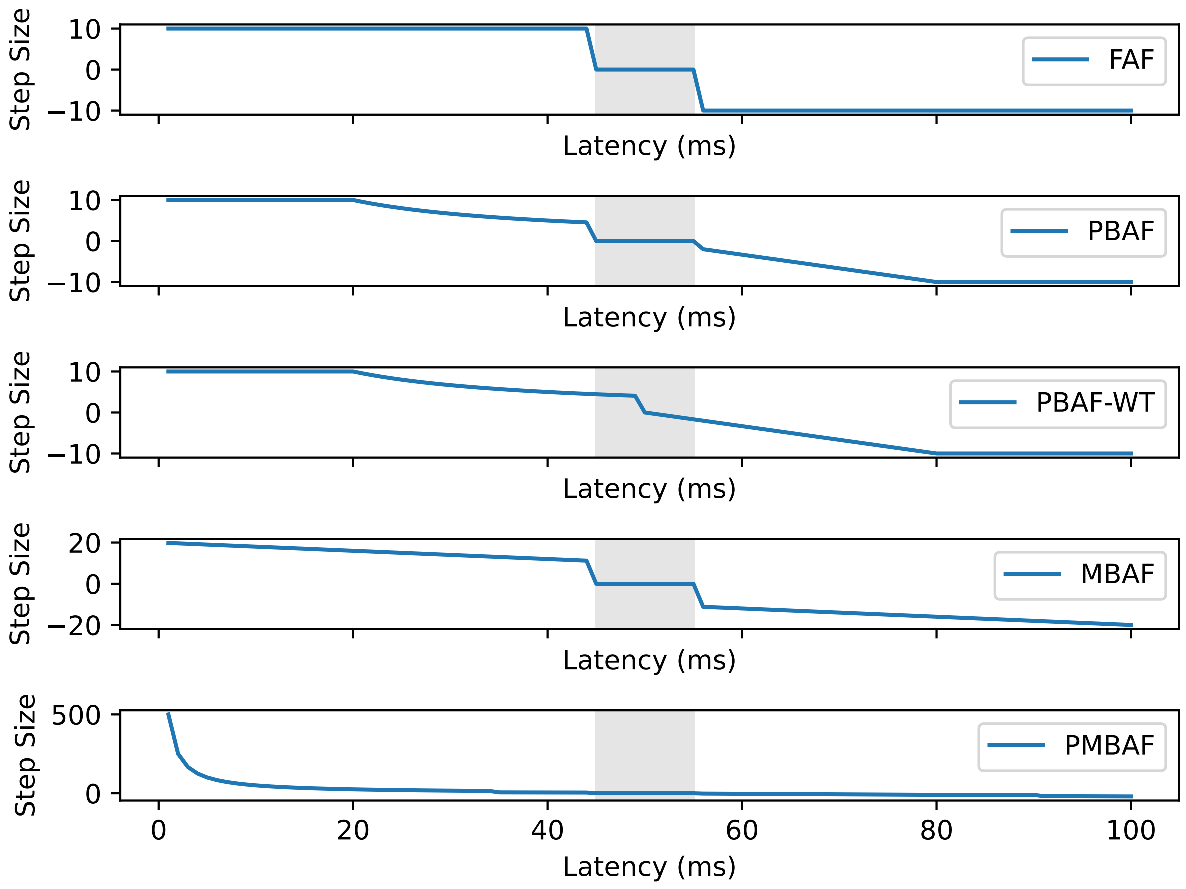

Figure 4.2 – A representation of the adaptive algorithms' behavior in a synthetic system. The target latency is set at 50ms, with a 10% threshold around the target, defining an acceptable latency range of 45 to 55ms.

Given that the lower limit is 30% and the upper limit is 80% of the target, we start applying PBAF in figure 4.2 when the latency is at 35 milliseconds, and we stop applying it when the latency grows beyond 90 milliseconds.

4.2 The Military Server Benchmark

The Military Server Benchmark (MS Benchmark) is a synthetic application developed by Araujo et al.[8] designed to leverage GPU-accelerated computing in stream processing. The benchmark comprises heavy computations and allows for the exploitation of data parallelism in each tuple. The problem domain involves allocating military units on a map while considering the location requirements specific to each unit type. A server is employed to process data continuously transmitted by drones, which fly over designated coordinates of the map. A list of coordinates and a list of military units is assigned to each drone. For each (x, y) coordinate, a drone explores a squared area in front of it defined by a coordinate width w. The drone will also perform rotations for each coordinate, to cover the four cardinal directions. The data collected about the explored regions will be used to find the right coordinate for each drone's assigned military unit. The server performs the required computations to allocate the military units efficiently, a task that would place too much strain on a drone's embedded system.

The five stages composing the MS application are as follows:

- Stage A: A CPU lightweight I/O stage responsible for loading the input data.
- Stage B: A computational intensive stage performed by the GPU. It processes irregular computations using the parallel pattern Map and is responsible for extracting information from each drone's coordinates, such as the average height.
- Stage C: Another computational intensive stage performed by the GPU. It uses the parallel pattern MapReduce with multiple output variables to find the best coordinate for each military unit, relying on extracted data from stage B.
- Stage D: A lightweight stage performed by the GPU. It uses the parallel pattern Reduce with a single output variable to validate the results coming from stage C.
- Stage E: A lightweight I/O stage performed by the CPU. It simulates the output by writing the results of each tuple to the disk. The output is a file containing a list of military units with their respective most suitable coordinates.

Figure 4.3 visually represents the Ms Benchmark's pipeline.

MS Benchmark has been parallelized with some PPIs such as Pthreads [43] and SPar [27]. The MS implementation used in this work is parallelized with FastFlow [6] and

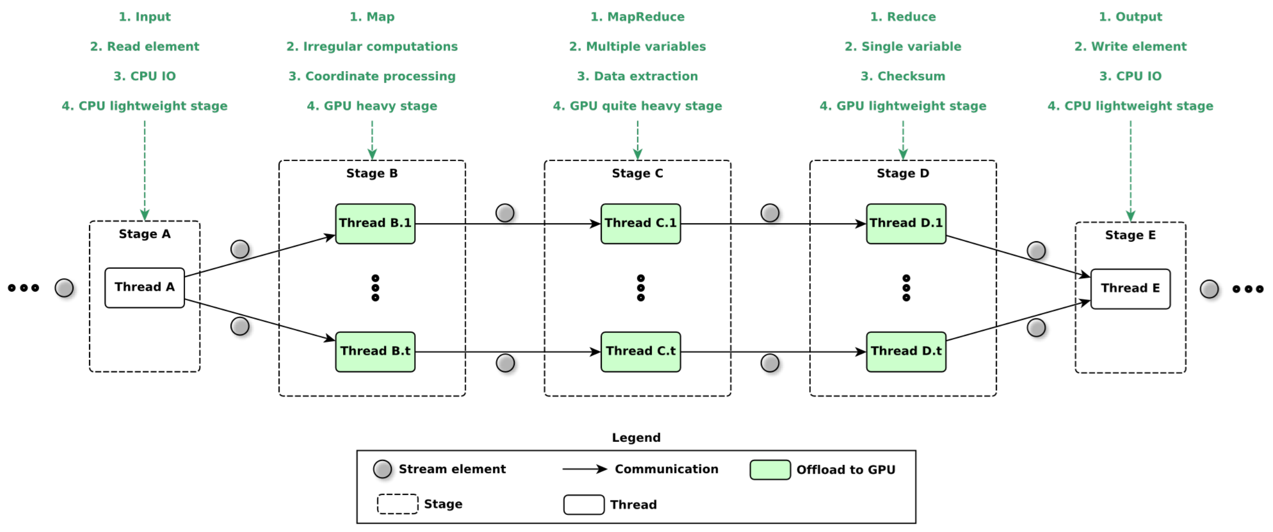

Figure 4.3 – A flow chart of the MS Benchmark's pipeline. Extracted from [8].

GSParLib [45]. FastFlow assembles and coordinates the pipeline's stages, while GSParLib is responsible for the GPU offloading inside the stages.

4.2.1 Changes to the Military Server Benchmark: Batch Support and New Workload

We introduced two notable changes to the MS benchmark before integrating the self-adaptation system described in Section 4.1. The first change was the addition of support for batching. To incorporate batch support, the FastFlow-related code was refactored to allow each tuple to carry data from multiple drones, thus composing the stream with batches. In GSParLib–related code, we implemented copying of multiple tuples (batch) at once to the GPU memory, making necessary adjustments to the code inside the GPU kernels to handle the tuples. Instead of iterating over each batch tuple using a sequential for loop, we assign each to a GPU block. If the tuple itself is assigned multiple blocks, the tuple blocks are kept as they are, and the number of blocks per tuple is multiplied by the number of tuples in the batch. The batches are allocated in the GPU's memory at the source (stage A) and only deallocated at the sink (stage E). Consequently, the batch sizes are defined for the whole pipeline; they are not defined individually by stage.

The second change consisted of introducing a new workload class that justifies the use of batching for the MS benchmark. Initially, MS had workload classes in which the tuples contained large payloads, to justify using a GPU without batching. There were 2048 tuples in the largest class, and each drone (the tuple) had to explore 6,144 coordinates in the map to allocate 1,536 military units. The combination of batching with such large payloads resulted in minimal throughput gains as the latency increased. Therefore, we introduced a new class with one million tuples, in which each drone (tuple) was assigned 32 coordinates and 16 units. This class contains varying processing costs associated with its tuples (e.g., processing one tuple can potentially take shorter or longer processing time than the next one).

Runtime performance variability in a SPA can be attributed to variables such as tuple inter-arrival time and tuple processing time. The time series generated by the moving average of these variables often demonstrates non-stationarities in real-world scenarios, such as increasing or decreasing trends, and cyclic behaviors such as seasonal patterns. The overall software architecture of MS Benchmark can reproduce workload variations called *computation patterns* applied to the computation time per tuple by the system (while the input rate is kept fixed for each execution). This is a realistic scenario for MS Benchmark since drones generate data and transmit them at a fixed rate. Some studies reviewed in Section 3, specifically by [22] and [57], similarly introduce the concept of variations in workload or input rates. These variations adhere to patterns, such as waveform or binary.

The algorithms to generate the computation patterns come from Garcia et al. [25] and are included in SPBench, a framework for creating benchmarks of SPAs. We adapted them to vary computation, although they were originally designed for generating frequency patterns. Our goal was to start experimentation with a simpler scenario in which the frequency was stable, and thus, there would be no time-related concerns in batching. The difference in processing cost among the tuples was regulated through the drone's coordinate width parameter described at the beginning of this section. As the coordinate width increases, so does the processing cost for the specific tuple. Nevertheless, it is difficult to determine the precise relationship between the width and the tuple's measured latency, since the latency will vary according to the computational environment used. The patterns in terms of computational load are presented in figure 4.4, in which the tuple's id on the x-axis is associated with the tuple's computational load on the y-axis.

Figure 4.4 depicts five patterns from top to bottom, namely increasing, spike, decreasing, binary, and waveform. Besides the self-evident increasing and decreasing patterns, the spike pattern will mostly keep the minimum width and occasionally increase it to the maximum value, while the binary pattern will alternate between intervals of minimum and maximum width. The waveform pattern can be seen as a more challenging version of the increasing and decreasing patterns, where these patterns alternate quickly.

Table 4.1 summarizes the patterns, informing for each one the duration in percentage of tuples, the number of cycles (how many times the pattern repeats), the number of tuples the duration percentage represents, and the number of tuples contained in a cycle.

Once we had batching support and a workload that justified using batches in the MS benchmark, the next step was to integrate the self-adaptive system described in Section 4.1. To achieve this, All that was required was to call the controller in the source to decide the batch size before sending the batch to the next stage of the pipeline, and

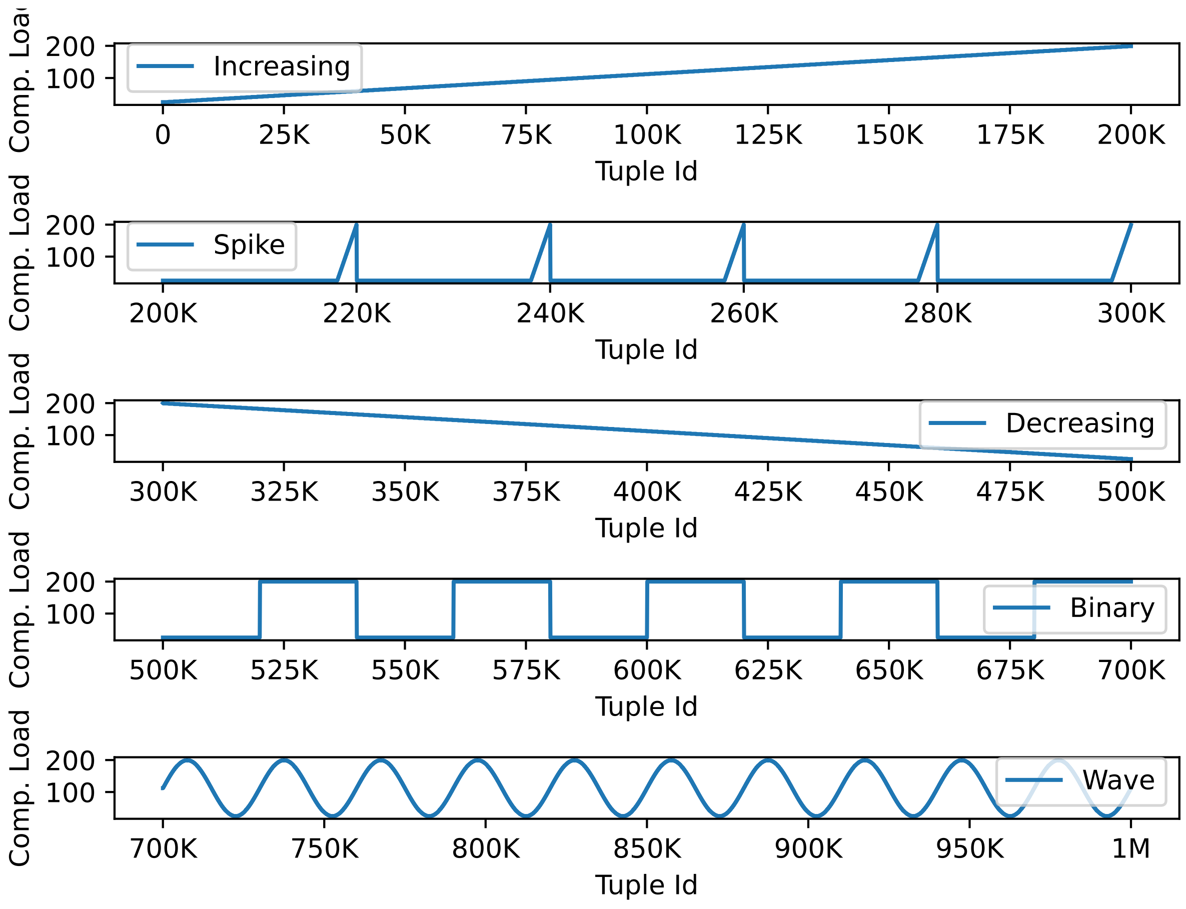

Figure 4.4 – Computational load per tuple.

Table 4.1 – Summary of computation patterns.

Pattern	Duration	Tuples	Tuples/Cycle	Num. of Cycles
increasing	20%	200K	200K	1
spike	10%	100K	20K	5
decreasing	20%	200K	200K	1
binary	20%	200K	40K	5
wave	30%	300K	30K	10

then to connect the sink to the source so that the source could receive the batches end-to-end latency (the latency measured from source to sink). The feedback loop delivering measurements from the sink to the source is implemented by a particular kind of stack data structure having bounded capacity and providing the possibility to peek (read without extracting) multiple items in the front of the stack. Therefore, the sink pushes the measured latencies, while the source peeks a number of last latency values. These values are then used to calculate a moving average, which is used as input for the adaptation algorithm. The number of the last measured latencies used for computing the average (which is also the size of the stack) is controlled by a parameter named *latency sample size*. This sampling size can be used to try reducing interferences from specific batches containing latencies that significantly deviate from their neighbors.

4.3 Evaluation Metrics

Our evaluation proposed a new set of metrics: the *SLO hit* and the *SLO distance* metric. They compare the adaptation algorithms among themselves regarding quality and effectiveness from different perspectives. Each metric has both a less and a more sensitive version.

4.3.1 SLO Hit Metrics

We propose a *batched* and an *itemized* definition for the SLO hit. The batched SLO hit is formally described as:

Let t be the target SLO, h be a threshold percentage such that $0 < h < 1$, B be the set of batches processed during the whole or part of the application execution, and $\omega : B \to \mathbb{R}$ be a mapping of a batch $b_i \in B$ to its measured performance metric value (e.g., in terms of latency or throughput). The set $B' \subseteq B$ containing the batches which fell within threshold bounds is defined as $B' = \{b \in B \mid t * (1 - h) \leq \omega(b) \leq t * (1 + h)\}$. The batched SLO hit is defined as the percentage of batches that fall within threshold bounds, formally as:

$$\text{B-SLH} = |B'|/|B| \tag{4.2}$$

What we refer to as the batched SLO hit was the metric chosen by Stein et al.[47] to evaluate the adaptation algorithms they proposed. This metric is useful to understand the effectiveness of the adaptation algorithms for achieving a defined SLO. However, it presents a notable limitation resulting from the focus on the batch level. Specifically, batches containing large quantities of items will have the same weight as batches containing only a few items. This can produce situations where the adaptation algorithm fails to achieve the SLO for the majority of the items processed by the application, but the resultant SLO hit still remains greater than 50%. The problem described with the batched SLO hit becomes more pronounced as the batch size range increases.

To solve this problem, we propose the itemized SLO hit. It is fundamentally the same as the batched SLO hit, with the additional consideration of the batch sizes. It is defined as:

Let the definition of the sets B and $B' \subseteq B$ be the same as in the definition of the batched SLO hit, and let $\sigma : B \rightarrow \mathbb{N}$ be the mapping of a batch $b_i \in B$ to its size. The itemized SLO hit is defined as follows:

$$\text{I-SLH} = \frac{\sum_{b' \in B'} \sigma(b')}{\sum_{b \in B} \sigma(b)} \tag{4.3}$$

Table 4.2 presents a situation where the itemized SLO hit can complement the batched SLO hit. It provides details regarding batches generated during executions of a dummy stream processing system. The table includes a total of 10 batches. The first 5 batches (first row) have a size of 6, while the subsequent 5 batches (second row) have a size of 14. The table reading should be done so that when a row is regarded as an SLO hit, the corresponding row should be interpreted as an SLO miss. This way, when we choose a row as an SLO hit, we choose one of two possible system executions.

Table 4.2 – SLO hit example.

Num. of Batches	Batch Size	B-SLH	I-SLH
5	6	50%	30%
5	14	50%	70%

Analyzing Table 4.2 from the perspective of batched SLO hits, we can conclude that the row representing the SLO hit is inconsequential, always resulting in a consistent 50% SLO hit rate. Conversely, from the perspective of the itemized SLO hits, we conclude that selecting the last row is more profitable, yielding a 40% higher SLO hit rate than choosing the first row. This suggests that the batched SLO hit metric does not differentiate between the two executions represented by the first and second rows. This demonstrates a significant drawback for systems with varying batch sizes, such as those that utilize self-adaptive algorithms for dynamic batch size adaptation.

The two SLO hit metrics are binary because a given batch is inside or outside threshold bounds. Such metrics can work perfectly for users whose only concern is knowing whether the application meets the SLO. However, they fail to provide information for the researcher/practitioner who wants to know how much the SLO is being met or not. In the latter case, proper SLO distance metrics can be defined to supplement the SLO hit metrics by measuring how far the batches were from the target SLO.

4.3.2 SLO Distance Metrics

We propose two distance metrics: the MAD-based SLO distance and the SD-based SLO distance. They are calculated in the same fashion as the population's mean absolute deviation (MAD) and the population's standard deviation (SD). The only difference from the standard statistical forms of MAD and SD is that the target SLO value is used instead of the mean. As the second and final step, we divide the value obtained in the first step by the target SLO value. We found that expressing the values in percentage helps with the interpretability because it is expected that the distance values will not surpass more than one time the target SLO. Furthermore, the percentage format fits naturally with the way the thresholds are specified (as a percentage of the target SLO).

We define below the MAD-based SLO distance and the SD-based SLO distance.

Let t be the target SLO, and let the set B and the function ω be as defined for the batched SLO hit metric. The MAD-based SLO distance is defined as:

$$\text{MAD-D} = \frac{\sum_{b \in B}(|t - \omega(b)|)}{|B| \cdot t}. \tag{4.4}$$

Let t be the target SLO, and let the set B and the function ω be as defined for the batched SLO hit metric. The SD-based SLO distance is defined as:

$$\text{SD-D} = \frac{\sqrt{\sum_{b \in B}(|t - \omega(b)|^2)}}{|B| \cdot t} \tag{4.5}$$

The MAD-based SLO distance arguably provides the most intuitive results when compared to the SD-based SLO distance. When applied over a single batch, it provides a value that can be directly compared with the threshold. In fact, the batched SLO hit metric can be derived from the MAD-based SLO distance by checking if every batch's distance value is less or equal to the threshold. The MAD-based SLO distance also exhibits the property of not being affected by a few large batch distance values (outliers). In contrast, the SD-based SLO distance is more sensitive to such values, given that it is based on squaring distances from the target.

Figure 4.5 shows a synthetic example where we depict a system producing batches, with the x-axis representing the batches by their associated order IDs. The y-axis illustrates that the measured latency generally increases as the execution progresses, without significant disturbances. The specific cause for this latency increase, whether due to escalating batch sizes or increased computation for each batch, is not a current concern for the example.

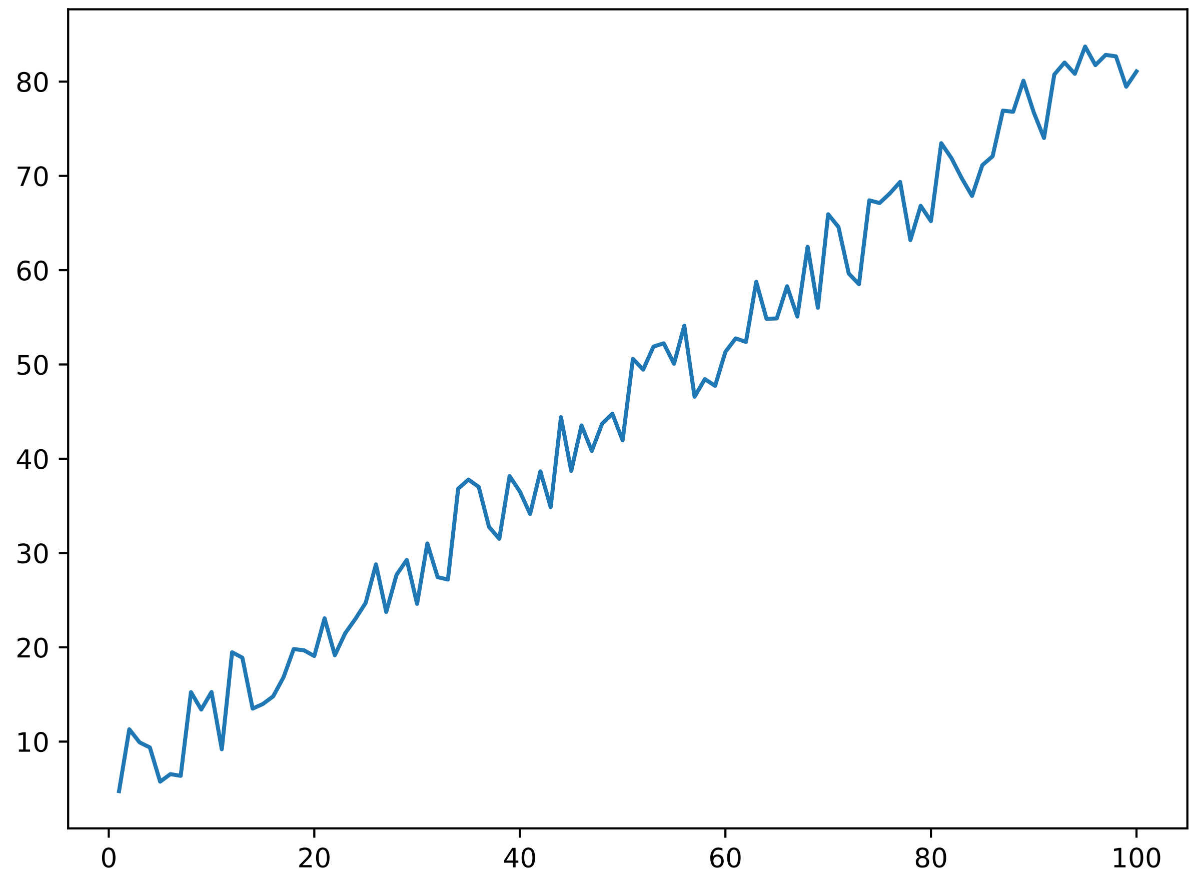

Figure 4.5 – Example of the distance metrics behavior when applied to a stable system.

We can observe in Figure 4.5 that the MAD distance is approximately 20% from the target, while the SD distance closely aligns at 23%. In scenarios characterized by gradual changes, such as the one presented, the choice between using one distance metric or the other has minimal impact.

Figure 4.6 provides another example in the same vein as in Figure 4.5, now featuring two latency spikes, one occurring near the beginning and the other closer to the end of the execution.

We can observe that in Figure 4.6 the MAD distance exhibits minimal change, increasing from 20% to 33%. In contrast, the SD distance shows a more pronounced shift,

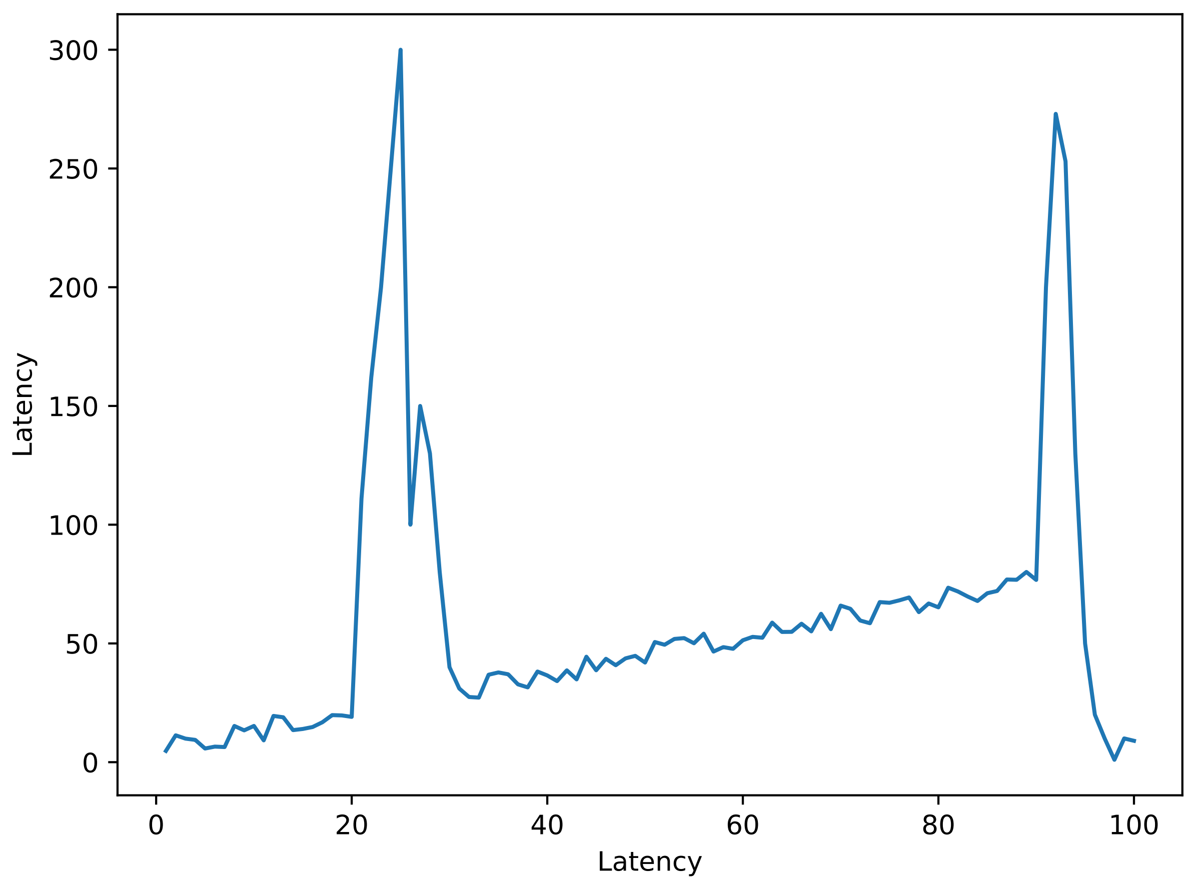

Figure 4.6 – Example of the distance metrics behavior when applied to an unstable system.

escalating from 23% to 57%. These examples demonstrate that the SD distance is more sensible to spikes.

Given that these metrics are numerical representations of various aspects of a SPA’s execution, they lend themselves to experiment sorting. For instance, by conducting multiple experiments with diverse adaptation parameters and subsequently sorting them based on the SD distance, we can discern which parameter combinations lead to more significant latency spikes.

5. EVALUATION

As a first step in addressing the problem of improving self-adaptation on GPU-accelerated SPSs through batch size adaptation, we evaluated some approaches from the literature that presented results we deemed relevant to our scenario, namely, SPAs executing on a single multi-core machine with one or more GPUs. According to the related work reviewed in Section 3, the most common approach for self-adaptation in SPSs is the use of online decision algorithms, which consider previous measurements of parameters such as latency or throughput to adapt the batch size and sometimes other adaptation parameters, such as parallelism degree. Even though we were able to find about ten studies that provide self-adaptive approaches concerning batch size, only two of them (Stein et al.[47] and De Matteis et al.[24]) provide support for GPU offloading.

De Matteis et al.[24] presented two algorithms capable of adapting the batch size to minimize latency and maximize throughput. However, a stable workload is assumed to never change once a suitable batch size is found. We consider this an important limitation (besides the complexity and computation time of the algorithms), and for this reason, we opted not to evaluate these algorithms in the present study.

Stein et al.[47] proposed four distinct but related algorithms to adapt the batch size to keep the latency within a user-specified SLO threshold. The algorithms are simple and lightweight, so they can be easily implemented in various SPAs and SPSs. In addition, they assume a workload that may vary with time. Given such characteristics, we evaluated the same strategy and algorithms with a different application. The authors originally performed experiments with a streamed data compression application implementing the LZSS algorithm [48]. Their chosen application was parallelized with SPar [27] and CUDA, using the farm parallel pattern with GPU-accelerated workers. Our evaluation was performed using the Military Server (MS) application [8], a synthetic stream processing benchmark simulating a server that continuously receives data from drones, and which was designed to facilitate data parallelization within tuples. This SPA has been parallelized with FastFlow [6] and GSParLib [45], and it is structured as a pipeline of three farms between the main source and emitter stages. Compared to the LZSS SPA, the MS SPA has more stages, and the batch size is more coarse-grained, being expressed as structure data (records) instead of raw bytes. We believe that understanding how the algorithms proposed by Stein et al.[47] behave with an entirely different application sheds more light on their generalizability and usefulness beyond the scenario in which they were originally evaluated.

Besides the four algorithms from Stein et al.[47], we evaluated other two algorithms focusing on the same reactive approach. The intention was to address specific limitations identified in the four mentioned algorithms during the evaluation process, namely the limitation in fine-tuning coming from MBAF and the limitation in reactivity coming from PBAF.

5.1 Evaluation Environment

The experiments were executed in a computer equipped with an AMD Ryzen 5 processor (6 cores and 12 threads) and 32 GB of RAM. The GPU was an NVIDIA GeForce RTX 3090 (Ampere architecture) with 24 GB of VRAM and 10,496 CUDA cores. The operating system was Ubuntu 20.04 LTS. The software used was GCC 9.0.5, CUDA 11, FastFlow 3, and an optimized version of GSParLib provided by [8]. We used the GCC compiler-level optimization 3 (flag 03). There was no replicated stage in our stream processing pipeline, so each stage is run by a dedicated host thread offloading computation on GPU.

5.2 Experiments with Fixed Batching

Our first goal for the evaluation was to understand how the batch size affected the latency and throughput in the MS benchmark. Some specific points we wanted to determine included the batch size cutoff for improving throughput and whether a higher batch size would correspond to a higher latency. To accomplish this, we executed MS Benchmark ten times for each batch size from 1 to 500 (that is, ten times for the same batch size). We then collected the average latency and throughput for each execution. For executions having the same batch size, we averaged the previously collected average latency and throughput values among the (ten) executions.

Since the batch size is fixed during the execution, we did not use self-adaptation in this phase. The results are shown in figure 5.1, in which the batch size is on the x-axis, the latency is on the primary y-axis, and the throughput is on the secondary y-axis.

Based on the experiments with fixed batch sizes, we can determine that the throughput cutoff is achieved with a batch size of around 115. Therefore, incrementing the batch size beyond this limit does not seem very productive since the latency will increase while the throughput will mostly stay the same.

However, we are dealing with average latency and throughput measurements. It can still be beneficial to increase the batch size beyond 115 at specific execution points. Therefore, there is no correct fixed batch size for dynamic workloads like ours. The manual method of finding the fixed batch size through iteration limits us to approximating a batch size suitable only for average scenarios.

Ignoring occasional irregularities, it is possible to state that the latency tends to increase with the batch size, roughly following the proportion of 1 millisecond for an

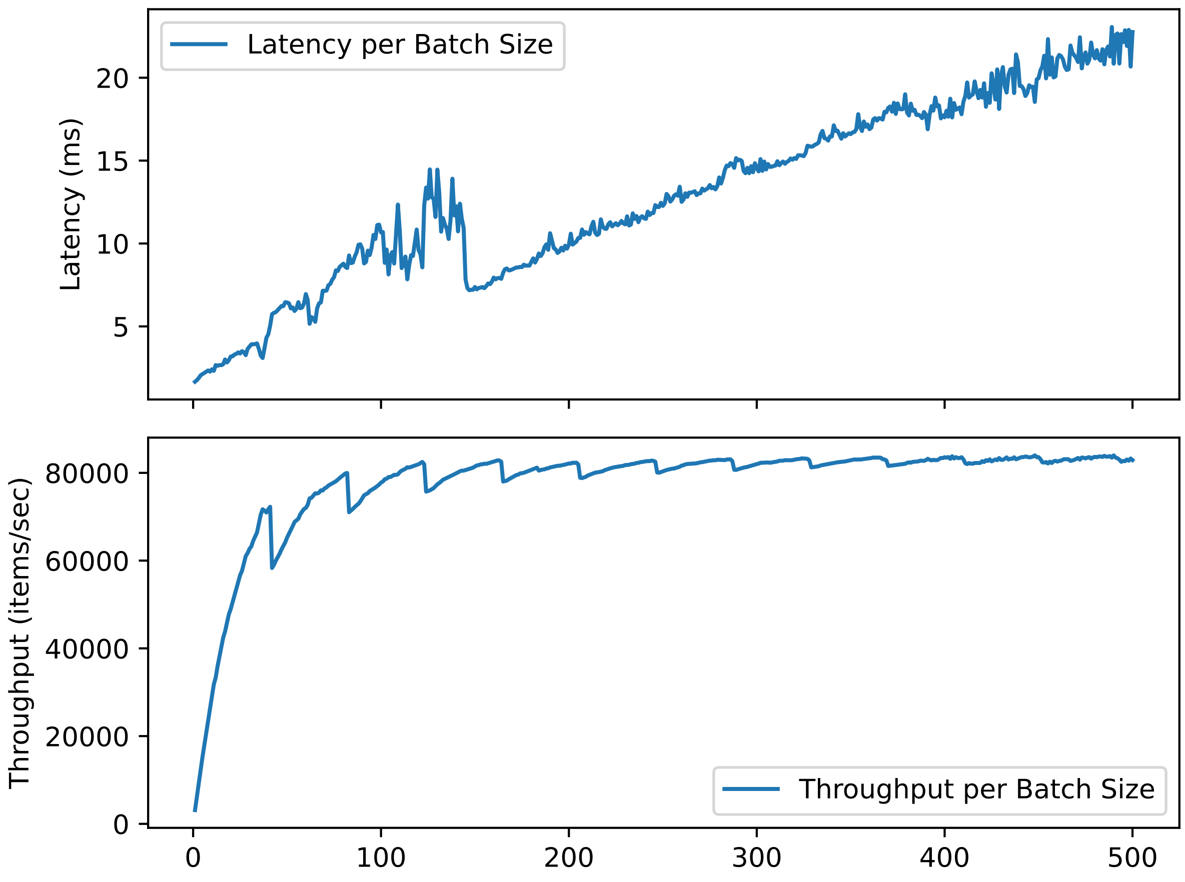

Figure 5.1 – Latency and throughput with respect to batch size.

increment of ten in the batch size. However, this relationship is less accurate for batch sizes greater than 100, where the latency increases more slowly.

Overall, it is noticeable that the use of batching yields benefits for the tested workload about throughput. Specifically, running the application with batching resulted in up to 25 times higher throughput than running it without. However, using a fixed batch size results in latency variations during the execution due to the varying processing costs of the tuples (as discussed in Section 4.2), and the bigger the batch size, the bigger the latency.

Figure 5.2 demonstrates the latency in time (during MS Benchmark's execution) when running with batch sizes 1, 50, and 100. The chart's legend abbreviates the batch size as *b-size*.

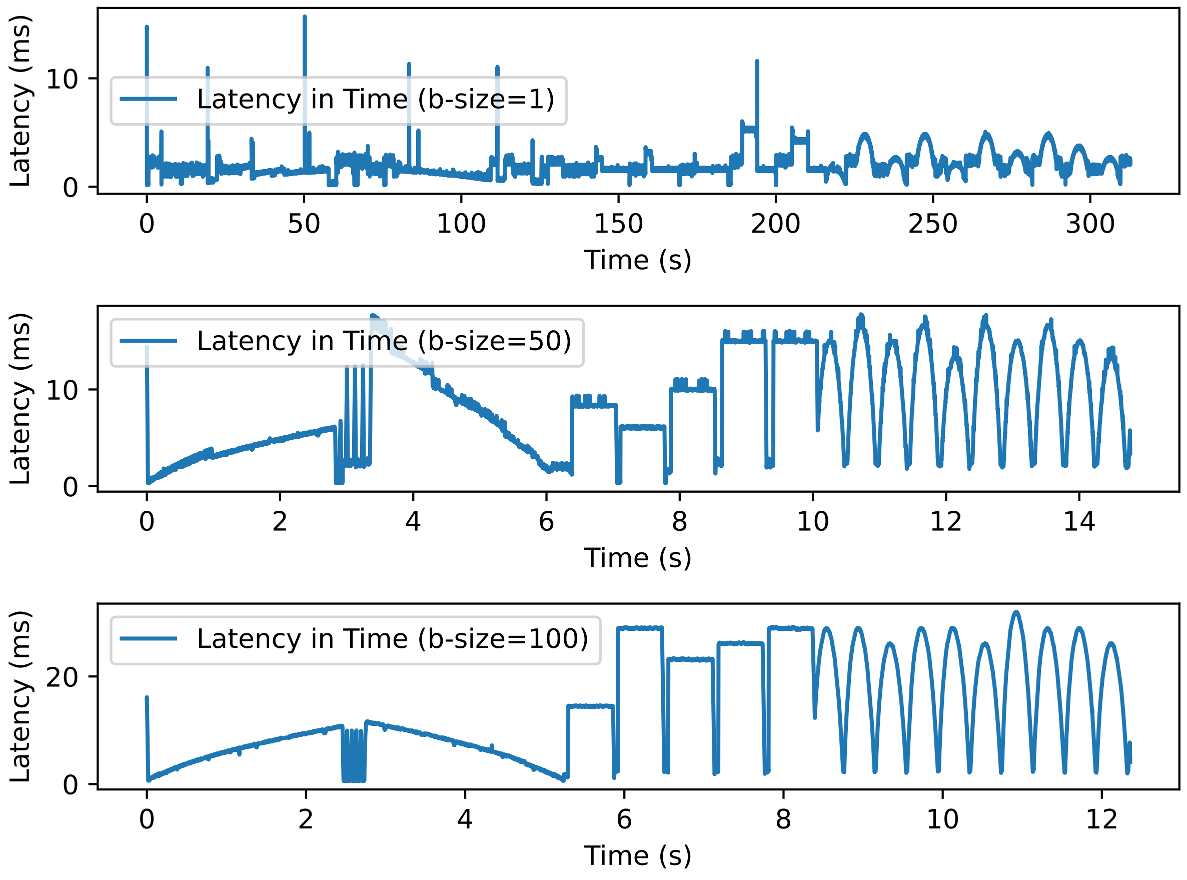

Figure 5.2 – Latency in time with fixed batch sizes.

In Figure 5.2, we can easily notice the aforementioned throughput improvement. The execution with a batch size of 1 takes around 5 minutes to finish, while with a batch size of 100, the execution finishes in approximately 12 seconds. It is also possible to notice the patterns in latency variation. From left to right and independently of the subplot, we have the increasing, spike, decreasing, binary, and wave patterns. The patterns can be more easily observed as the batch size (and consequently the latency) increases. In cases where such latency variations are not desirable, the self-adaptive algorithms presented can be used to adjust the batch size and keep the latency within a user-defined threshold.

During execution without batching, we observed that the highest latencies are up to 30 times greater than the lowest latencies, ranging from 0.15ms to around 5ms (excluding some rare extreme values). Considering this observation, we can conclude that choosing a latency target much less than 5 ms (like 1 ms) will generate situations in which, even with a batch size of 1, achieving the latency target is impossible. This will inevitably result in lower SLO hit and distance metrics gains. A possible solution for this problem is to filter these situations out before computing the metrics. Alternatively, we can just accept that it will be impossible to attain a perfect score in the metrics (e.g., 100% SLO hit). We opted for the latter, since we did not observe significant differences in the SLO hit metrics for our workload after filtering.

Another important consideration is that choosing an excessively high latency target may result in large batch sizes that just a few of them will be enough to consume the whole dataset (one million items). Balancing the factors presented regarding the issue under discussion, we arrived at the latency target of 3 ms as a good candidate for the experiments with dynamic batching. Ideally, we would have experimented with multiple latency targets. However, we did not run the whole battery of experiments detailed in Section 5.3 with other latency targets due to resource and time constraints.

The following experiments present the results of applying the adaptation strategy and algorithms described in Section 4.1 to maintain the latency within a specified SLO threshold.

5.3 Experiments with Dynamic Batching

For the experiments with dynamic batching, we executed the MS benchmark 10 times for each parameter combination to obtain the means and standard deviations. The parameters are as follows:

- Target latency: 3 milliseconds
- Threshold values: 5%, 10%, 15%, and 20%
- Step sizes (adaptation factors): 1, 5, 10, 15, and 20
- Latency sample sizes: 1, 5, 10, and 20
- Self-adaptive algorithms: FAF, PBAF, PBAF-WT, MBAF, PMBAF

There are a total of 400 possible combinations resulting from the listed parameters. All of them were considered in the experiments. The total number of executions of the MS benchmark was 4,000, factoring in the 10 executions for each parameter combination. The reason for considering all possible parameter combinations is to enhance our comprehension of the behavior of the adaptation algorithms across diverse scenarios. Furthermore, the SLO metrics outlined in Sect. 4.3 facilitate this process by enabling us to rank the experiments based on them and identify important results before performing an in-depth analysis of data from individual executions. We opted not to delve into details about most parameter combinations as they are highly specific and lack broad generalization.

The execution time for processing our workload (containing one million tuples) varied from twelve and a half seconds to around two minutes and a half, depending on the parameter combination. Note that these numbers regard the executions using the GPU. The sequential CPU-only version finishes the execution in about half an hour. We consider the whole execution of the experiments without warm-up or cool-down periods. We observed that only the first three to five batches launched at the beginning of MS Benchmark's execution present a higher latency than expected, which does not significantly affect the metrics.

We chose the values for the parameters based on the workload used, described in Section 4.2 as the new workload class introduced to the MS benchmark. The 3-millisecond target latency was chosen because, empirically, it is achievable at almost any point in the execution, given the right batch size. It is close to the highest tuple latency encountered when executing without batching, so it can be achieved with a batch size close to 1, while the lowest tuple latencies require batch sizes around 300. The threshold values are the same as in Stein et al.[47]. We concentrate primarily on analyzing results based on the 5% threshold in our evaluation. This threshold is the most stringent, providing a more concise worst-case-scenario perspective. The step size and latency sample size parameter values were set empirically by observing the values in which the adaptation started to present diminishing results. For the step size, we also considered that usually changing the batch size by 10 in the fixed batching experiments (Section 5.2) resulted in a change of 1 millisecond in the average latency.

5.3.1 Result Analysis by the SLO Hit Metrics

Table 5.1 presents the best SLO hit metrics achieved by each algorithm and the configuration used to achieve the metric value.

In the most strict (5%) threshold the algorithms were able to meet the latency SLO for 30% (FAF) to 37% (PMBAF) of the one million processed tuples. The SLO hit per batch ranged between 26% and 57%, with MBAF and PBAF-WT defining the lower end, and PMBAF and PBAF defining the higher end of the range.

Table 5.1 – Best SLO hit metrics and configurations by algorithm and threshold. Each metric value is the average of 10 executions. The *step size* and *latency sample size* parameters are abbreviated as *step* and *sample*, respectively.

Threshold	Algo.	Batched SLO Hit			Itemized SLO Hit		
		Value	Step	Sample	Value	Step	Sample
	FAF	37.25	5	5	30.05	5	1
	PBAF	36.55	20	1	34.19	10	1
5%	PBAF-WT	26.25	5	1	30.55	5	1
	MBAF	27.18	15	10	31.95	5	1
	PMBAF	57.78	10	1	37.06	10	1
	FAF	49.09	10	5	45.59	10	1
	PBAF	52.79	15	5	47.68	20	1
10%	PBAF-WT	37.98	5	1	44.39	5	1
	MBAF	44.9	15	10	45.38	5	1
	PMBAF	60.95	20	5	47.37	15	1
	FAF	54.72	20	10	54.44	10	1
	PBAF	59.13	20	1	54.25	20	1
15%	PBAF-WT	47.2	20	5	54.8	10	1
	MBAF	57.73	15	5	55.65	10	1
	PMBAF	70.62	20	5	55.42	20	1
	FAF	65.09	20	1	62.62	10	1
	PBAF	67.32	20	5	59.05	20	1
20%	PBAF-WT	54.14	20	5	64.28	15	1
	MBAF	74.46	15	5	62.66	10	1
	PMBAF	77.35	15	1	63.17	20	1

Consistent with our intuitions, the SLO hits are affected by the strictness of the threshold. Considering the itemized SLO hit and a 10% threshold, the algorithms secured the SLO for 44% to 47% of the tuples, demonstrating very similar results. This trend continues for higher thresholds, with the 15% threshold achieving the 50s range and the 20% threshold achieving the 60s range.

The best batched SLO hit results achieved in [47] for the 5% threshold range from 50% to 90%. The highest batched SLO hit we achieve in MS Benchmark with the same threshold is 57%, which is comparable with the lowest result from [47]. Even our highest result from the 10% threshold (60%) does not go much further than their lowest result. These results demonstrate that our workload is more challenging for those self-adaptive algorithms than the workloads presented in [47].

In Table 5.1 we can observe that as the threshold grows, the trend is that all algorithms start achieving better SLO hit results with greater step sizes. The reason for the trend is that a coarser tuning of the batch size is sufficient to enter the threshold bounds. With the fine-tuning requirements alleviated, using a greater step size becomes an advantage due to the gains in reactivity, which accelerates the algorithm convergence towards the threshold bounds.

SLO Hit Analysis by the Step Size Perspective

Figure 5.3 depicts the evolution of the itemized SLO hit for every algorithm as the step size increases. We selected the executions with the best itemized SLO hit for each step size. Therefore, the value of the latency sample size parameter (which is not shown) is always the one that achieved the best itemized SLO hit in combination with a given step size.

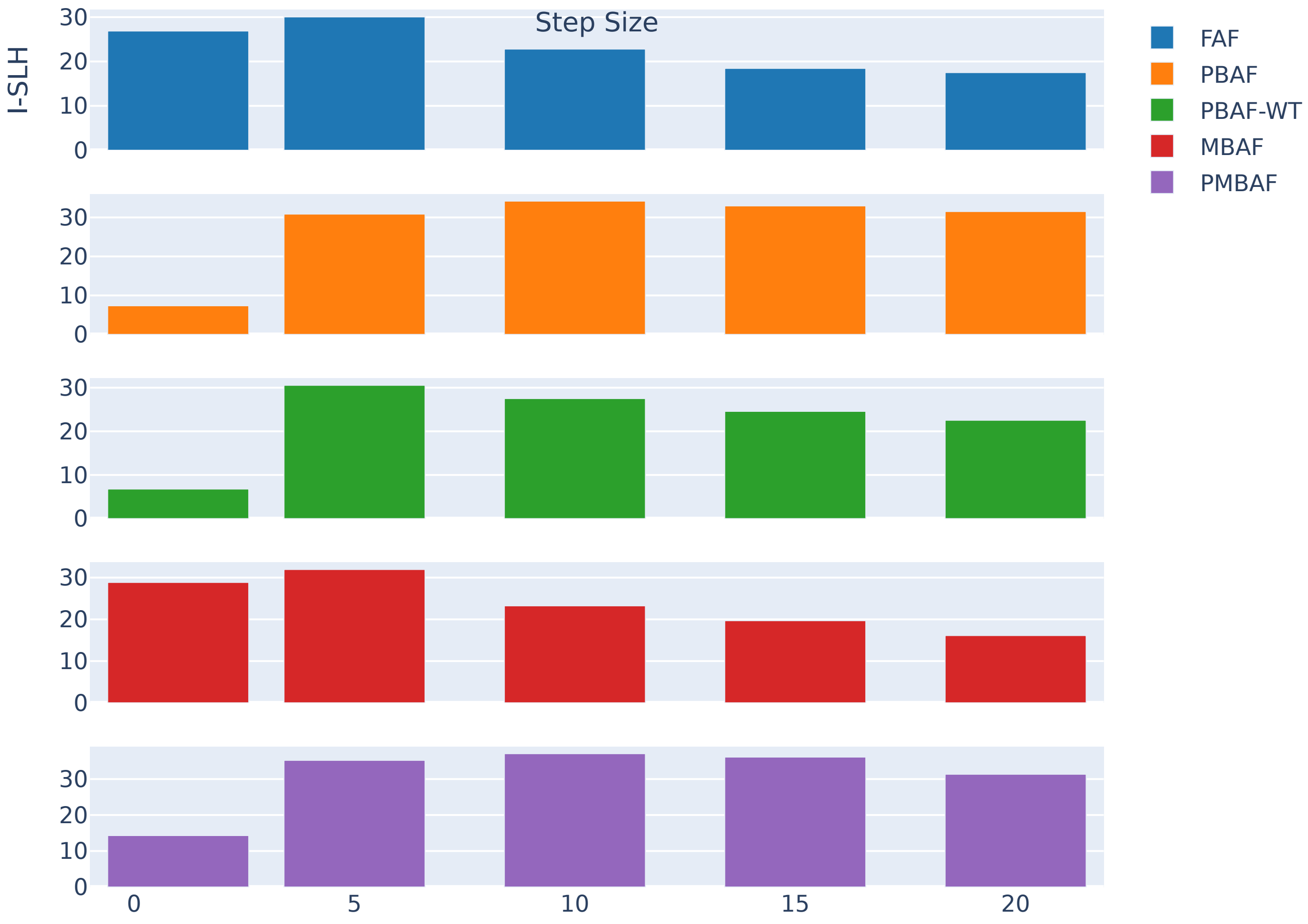

Figure 5.3 – Itemized SLO hit by step size and algorithm with a 5% threshold.

Overall, the algorithms achieve comparable results across all SLO hit metrics. However, their configuration to achieve their best results is not homogeneous. PBAF and PBAF-WT tend to achieve better results with a slightly greater step size than FAF and MBAF, owing to their ability to use a fraction of the step size as they approach the threshold bounds. PBAF-WT mostly keeps the step size smaller than PBAF's to achieve its best results, since coarse-grained batch size adjustments once inside the threshold bounds are more likely to cause SLO violations.

In most cases, the best itemized SLO hit results for FAF and MBAF are achieved with a step size of 5. This happens because larger step sizes for these algorithms, while allowing a faster reaction, also prevent them from fine-tuning the batch size when latencies are close to the target. In Figure 5.3 we can verify that the PBAF and PMBAF algorithms (which use a percentage of the step size) tend to not significantly decay in performance (itemized SLO hit) as the step size increases, in contrast to FAF and MBAF. The tradeoff regarding the step size is mitigated by PMBAF, which was designed to incorporate both

PBAF's fine-tuning and MBAF's reactivity. For this reason, PMBAF tends to perform better in terms of itemized SLO hit using a step size predominantly larger than the other algorithms in the same threshold, even though the improvement is not significantly larger than PBAF and MBAF alone.

Even though PMBAF did not attain an itemized SLO hit significantly larger than the other algorithms, it attained a significant increase in the batched SLO hit compared to the others for all thresholds. This characteristic arises mainly due to PMBAF's high reactivity. When the batch size is high (around 300 items) and the latency spikes abruptly, PMBAF can drastically reduce the batch size quicker than algorithms like FAF and PBAF. Due to this behavior, PMBAF frequently launches many smaller batches in patterns like binary and waveform, inflating the batched SLO hit metric.

Figure 5.4 depicts on the y-axis the batched and itemized SLO hit as the execution evolves (cumulative SLO hit). In the x-axis, *time* is normalized, progressing based on a numerical identifier (id) associated with each tuple, according to the order in which it was produced (which is fixed). The parameters for the execution can be found in Table 5.1 under the 5% threshold, on the row corresponding to PMBAF. Note that the final batched SLO hit value is not exactly the same as in the table, since each metric value in the table is an average of ten executions. We demonstrate the computation patterns in Figure 5.4 using a subplot below the main one showing the SLO hit metrics. This subplot presents the patterns horizontally in the same fashion as in Figure 4.4, with the difference that the patterns were organized vertically in that figure.

In Figure 5.4 we can observe that the itemized and batched SLO hit metrics are very similar until the algorithm enters the workload segment belonging to the binary pattern. Every time the computational load (per consequence, the latency) rises in the binary and waveform patterns, PMBAF reacts by reducing the batch size, and expanding the space between the lines associated with the two SLO hit metrics.

In general, the itemized SLO hits were consistently lower than the batched SLO hits, indicating that relying solely on the latter metric to assess SLO compliance can be misleading. Another disadvantage of the batched SLO hit uncovered by our evaluation is its high variability caused by specific (and sometimes unpredictable) runtime and execution characteristics. Many external factors can affect the latency in a SPA like the MS Benchmark, one of them being the current GPU occupancy. These factors commonly lead the self-adaptive algorithms to choose different sets of batch sizes even in repeating executions (multiple executions with the same set of parameters). Table 5.2 compares the standard deviations between the batched and itemized SLO hit metrics. The standard deviations were computed using the ten execution samples from each of the best results in Table 5.1 under the 5% threshold. We did not include the remaining thresholds in Table 5.2 for the sake of conciseness and because there is no significant difference among them.

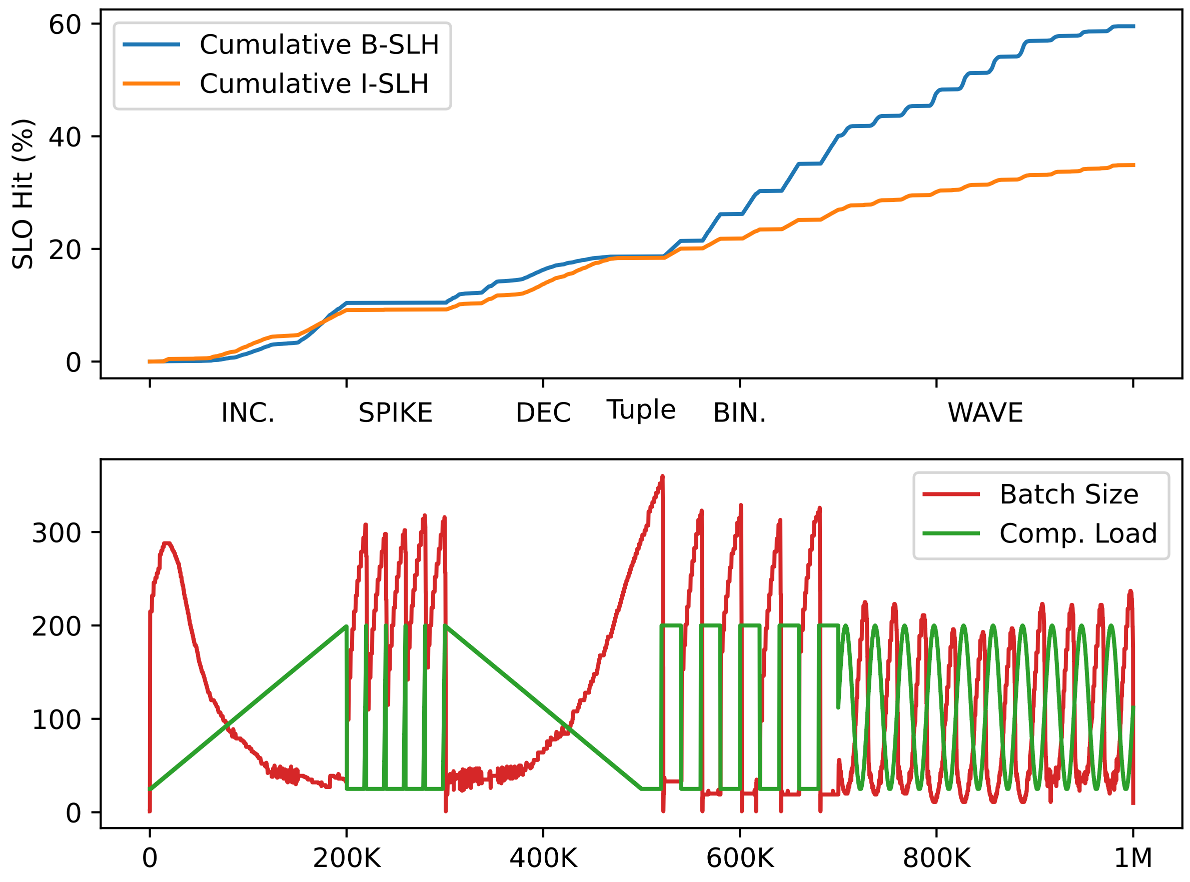

Figure 5.4 – Cumulative batched and itemized SLO hit (B-SLH and I-SLH) for the PMBAF algorithm under the 5% threshold.

Table 5.2 – Standard deviations for the best SLO hit metrics presented in Table 5.1 under the 5% threshold.

Algorithm	σ (**Batched SLO Hit**)	σ (**Itemized SLO Hit**)
FAF	14.66	2.78
PBAF	11.65	3.45
PBAF-WT	8	1.8
MBAF	4.5	2.54
PMBAF	12.73	1.39

SLO Hit Analysis by the Latency Sample Size Perspective

In Table 5.1, a consistent pattern can be discerned where the latency sample sizes are always 1 for the itemized SLO hit, while for the batched SLO hit, they are greater than one (usually five) in 13 out of the 20 instances (rows). This behavior stems from a trade-off between two conflicting situations that favor one metric over the other. The first situation is that a latency sampling greater than 1 increases the SLO hit in regions with frequent latency variations, such as in the workload segments belonging to the (gradual) increasing and decreasing computation patterns. However (as in the second situation), this increased sampling reduces reactivity since five latency samples must be collected before re-evaluating the strategy again. Consequently, the SLO hit will be lower in regions with abrupt latency changes necessitating high reactivity, e.g., spike and binary patterns. Furthermore, these patterns contain extensive regions with minimal computation, where large batch sizes (close to 300) are needed to keep the SLO. Specifically for the itemized SLO hit, batches missing the threshold bounds in these regions result in a greater cost than for the batched SLO hit. Consequently, the best results for the itemized SLO hit do not incorporate latency sample sizes greater than 1.

Figure 5.5 presents an example of the latency sample size parameter's influence in the batched SLO hit. The five computation patterns are on the x-axis, while the batched SLO hit values are on the y-axis. There is one subplot per sample size. The term latency sample size is abbreviated as *samp* in the figure. To compute the batched SLO hit for each pattern, we isolate the latencies from the time when the pattern occurs, and then divide the latencies that fell within threshold bounds by the total number of batches launched during the pattern's duration.

We can observe in Figure 5.5 that as the latency sample size grows, the batched SLO hit improves for the increasing and decreasing computation patterns. The exception appears with a sample size of 20, which causes an SLO miss of about 7% and indicates that further increasing the sample size is not advantageous. We can also verify a significant improvement in the SLO hit for the spike pattern. This improvement is not caused by the same behavior as in the increasing and decreasing patterns. The reason is that a greater latency sample size makes the adaptation less reactive to spikes. We observed that it is more advantageous for the spike pattern when the algorithm avoids reacting to the spikes, just keeping the batch size as before the spike. The spikes are short-lived, and when the

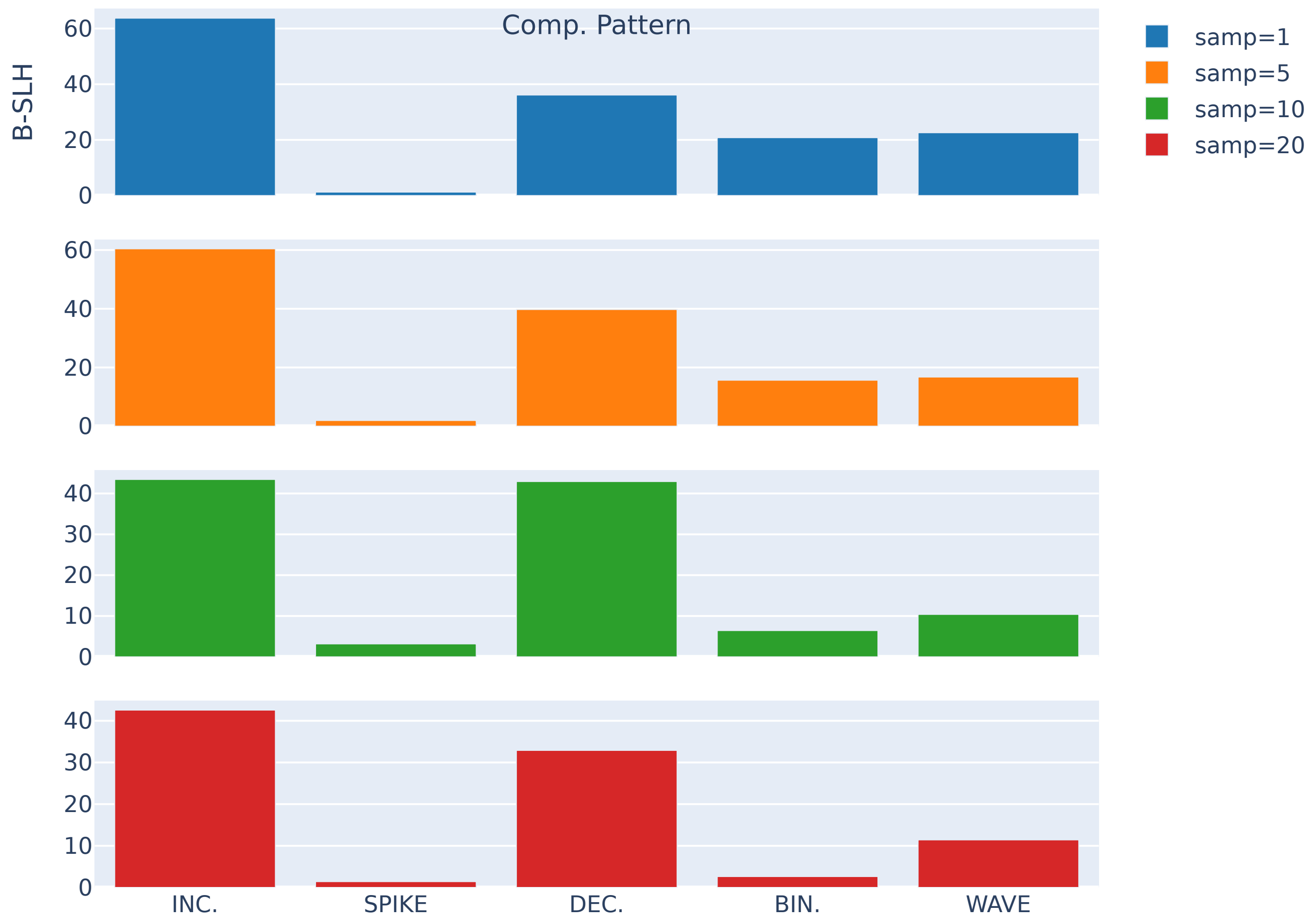

Figure 5.5 – Batched SLO hit (B-SLH) by pattern and latency sample size for the FAF algorithm under the 5% threshold and having a step size of 5.

algorithms try to react, the reaction (batch size changes) causes latency instability after the spike has subsided, taking longer for the latency to normalize.

SLO Hit Analysis by the Computation Patterns Perspective

Table 5.3 provides a breakdown of the executions according to the computation patterns introduced in Section 4.1. Specifically, the itemized SLO hit was computed for each pattern, being averaged across all threshold, step size, and latency sample size parameters. The table is organized such that each column corresponds to a pattern (denoted by its abbreviation), and each row corresponds to an algorithm.

Table 5.3 – Itemized SLO hits by algorithm and computation pattern averaged accross all combinations of thresholds, step sizes and latency sample sizes.

Algo.	Inc	Spike	Dec	Bin	Wave
FAF	60.46	28.09	47.97	14.58	18.36
PBAF	55.32	16.53	41.82	16.84	20.76
PBAF-WT	54.37	20.82	48.69	16.03	21.94
MBAF	57.99	19.32	51.66	28.73	22.76
PMBAF	53.74	9.83	46.12	29.55	23.26

The algorithms were generally more effective for workload segments where the latency changes gradually, as found in the increasing and decreasing patterns. In Table 5.3, the average itemized SLO hit for these patterns ranged from 40% in the decreasing pattern to 60% in the increasing pattern.

For the binary pattern, the FAF, PBAF, and PBAF-wt algorithms achieved an itemized SLO hit of around 14%, while the MBAF and PMBAF achieved a value of 29% in the same metric. The predominantly low SLO hits for this pattern may be attributed to the tuple latencies being always close to the extremes, requiring a batch size close to 1 or around 300 to stay within the threshold region. MBAF and PMBAF provided the highest itemized SLO hit for the binary pattern due to their high reactivity, being able to respond quickly to the transitions between regions of lower and higher latency.

The spike pattern was used to test the algorithms' reactions for quick bursts of high latency. In this pattern, the tuples' latencies stay at their lowest most of the time, ascending to their highest for brief periods. This was the pattern with the lowest itemized SLO hit. The more successful executions kept the batch size close to 300 at all times, even when the latency spiked. As the high latency periods are short-lived, excessively reducing the batch size requires the algorithm to increase it again almost immediately, which is counterproductive. Keeping the batch size unchanged during the spike is more effective than trying to react. For this reason, the same reactivity that guaranteed the highest SLO hit for PMBAF in the binary pattern produced the lowest SLO hit (close to 10%) in the spike pattern. The second lowest SLO hit belongs to MBAF. A possible cause for MBAF's result is that MBAF is not as reactive as PMBAF; thus, it does not suffer as much with the latency spikes.

The wave pattern is similar to the (gradual) increasing and decreasing patterns. However, the wave pattern is steeper than the two mentioned patterns, requiring a faster reaction from the algorithms. For this reason, the SLO was achieved less often than for the other gradual patterns, staying around a 20% value. Little difference exists among the algorithms in the average itemized SLO hit for the wave pattern, indicating that none stand out prominently.

5.3.2 Result Analysis by the SLO Distance Metrics

Table 5.4 presents the best distance metrics achieved by each algorithm and the configuration used to achieve the metric value. This table has the same structure as Table 5.1.

Table 5.4 – Best SLO Distance metrics and configurations by algorithm and threshold. Each metric value is the average of 10 executions. The *step size* and *latency sample size* parameters are abbreviated as *step* and *sample*, respectively.

Threshold	Algo.	MAD-based Distance			SD-based Distance		
		Value	Step	Sample	Value	Step	Sample
	FAF	30.79	5	1	52.89	5	1
	PBAF	24.28	20	1	39.37	1	1
5%	PBAF-WT	29.87	10	1	39.19	1	1
	MBAF	25.68	10	1	41.16	1	1
	PMBAF	15.3	10	1	37.98	1	1
	FAF	30.37	15	1	49.06	10	1
	PBAF	29.47	20	1	38.68	1	1
10%	PBAF-WT	29.87	10	1	39.19	1	1
	MBAF	26.85	15	5	41.96	1	1
	PMBAF	20.86	5	1	39.01	1	1
	FAF	27.81	5	1	45.45	1	1
	PBAF	24.92	15	1	40.16	1	1
15%	PBAF-WT	29.87	10	1	39.19	1	1
	MBAF	24.94	15	1	38.4	1	1
	PMBAF	19.24	5	1	40.32	1	1
	FAF	22.86	10	1	39.47	10	1
	PBAF	26.34	20	1	40.78	1	1
20%	PBAF-WT	29.87	10	1	39.19	1	1
	MBAF	22.62	15	5	37.52	1	1
	PMBAF	17.98	15	1	38.27	5	1

The best results for the MAD-based distance (shown in Table 5.4) are similar for all algorithms except PMBAF, predominantly staying between 25 and 30. PMBAF presented MAD-based distances around 40% smaller than this metric's second-best algorithm (MBAF). We attribute this behavior mainly to its high reactivity to spikes in latency. As explained in Section 4.1.2, MBAF, proposed by [47], can use at most double the step size when the target is below the threshold. We removed this characteristic for PMBAF so that its reaction is always proportional to the distance from the lower threshold.

The most frequent strategy in Table 5.4 that achieved the best results for the SD-based distance is formed by setting both the step size and latency sample size to one. This parameter combination results in executions where the batch latencies converge towards the target from the lower threshold bound, albeit failing to cross into the threshold bounds most of the time. Keeping the latency closer to the lower threshold bound requires smaller batch sizes, which consequently contributes to avoiding extreme latency spikes caused by large-sized, computation-heavy batches. We named this strategy *slow convergence strategy*.

Another successful strategy for mitigating spikes is using large step sizes combined with small latency sample sizes to increase reactivity. This strategy frequently appears under the MAD-based distance in Table 5.4. Although this strategy has high latency spikes, they are reduced more quickly to a latency closer to the target. We named this strategy *reactive strategy*. Even though the reactive strategy does not appear among the best results for the SD-based distance due to the metric's sensitivity to spikes, we observed that, in general, smaller values of the MAD-based distance are more indicative of greater SLO hits.

Figure 5.6 and Figure 5.7 depict the latency and the choice of batch sizes for execution using the aforementioned *reactive* and *slow convergence* strategies. In the x-axis, *time* is normalized across all executions, progressing based on a numerical identifier (id) associated with each tuple, according to the order in which it was produced (which is fixed). Regarding the y-axis for the top subplot, the blue line represents the strategy, and the orange line represents the target latency. In the bottom subplot, the green line (named *Comp. Load*) is meant to represent the computational load of the item, while the red line depicts the choice of batch sizes. They are the same as in Figure 5.4.

Comparing Figure 5.6 with Figure 5.7, it is possible to notice that the batch sizes (red line) from the *slow convergence strategy* are significantly lower than the ones chosen by the *reactive strategy*. Consequently, the latencies from the former are lower (approaching the target from below) and do not suffer as much with spikes. Even though attaining a latency lower than the specified SLO may not be inherently undesirable, it can lead to an undesired impact on throughput. While the *slow convergence strategy* achieved an average throughput of approximately 31,000 items per second, the *reactive strategy* nearly doubled the metric, reaching 59,000 items per second.

A latency sample size greater than one cannot be found in Table 5.4 under the SD-based distance. This condition happens due to the loss of reactivity caused by increasing the mentioned parameter. When the batch size is increased to approximate the lower threshold in regions of very low latency (a result of tuples requiring low computation), a large sample size makes the batch size accumulation slow. Then, when tuples with heavy computational requirements suddenly enter the pipeline, the latency will increase significantly (creating a spike), and the previously large batch size will be reduced at about the same speed that it was increased. This circumstance leads to an extended latency spike, resulting in a penalty to the execution as indicated by the SD-based distance metric.

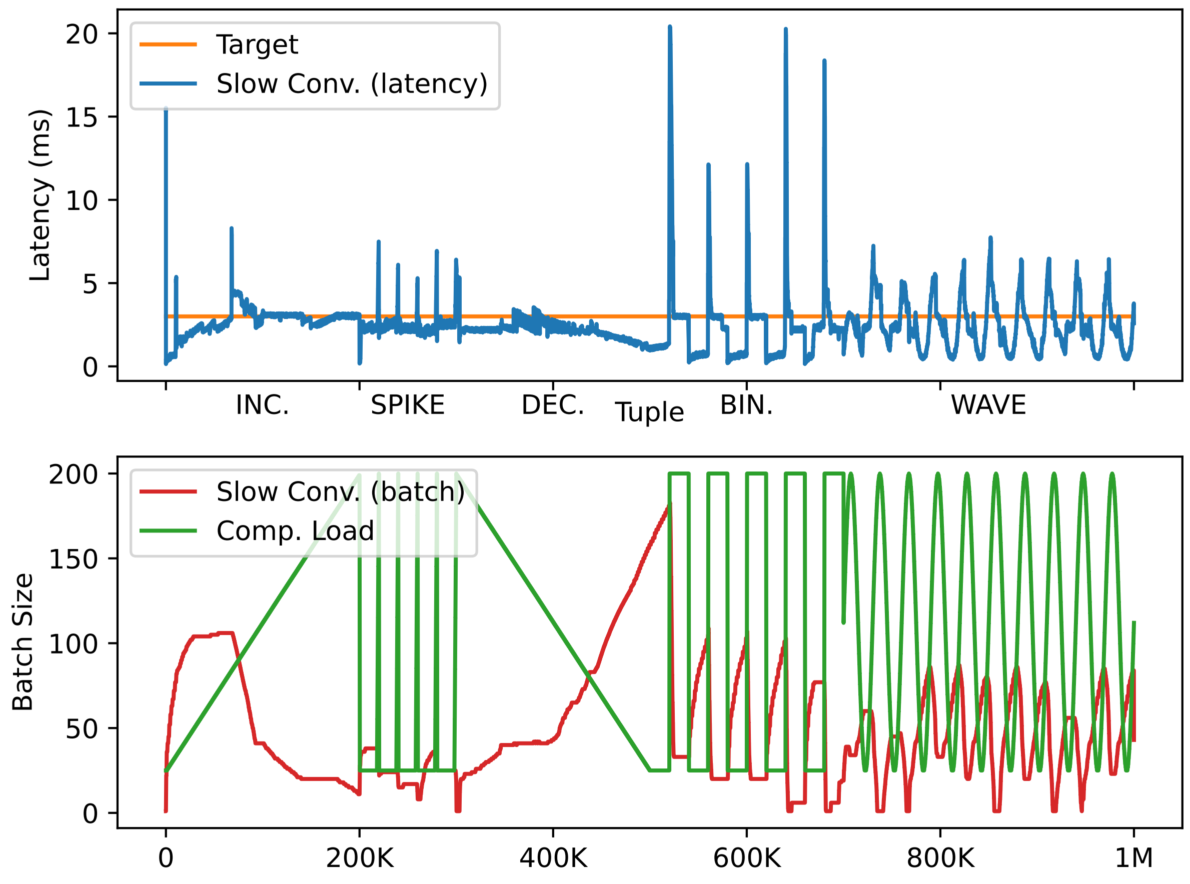

Figure 5.6 – Execution using the Slow Convergence Strategy. The latency, batch size and computational load per item are shown.

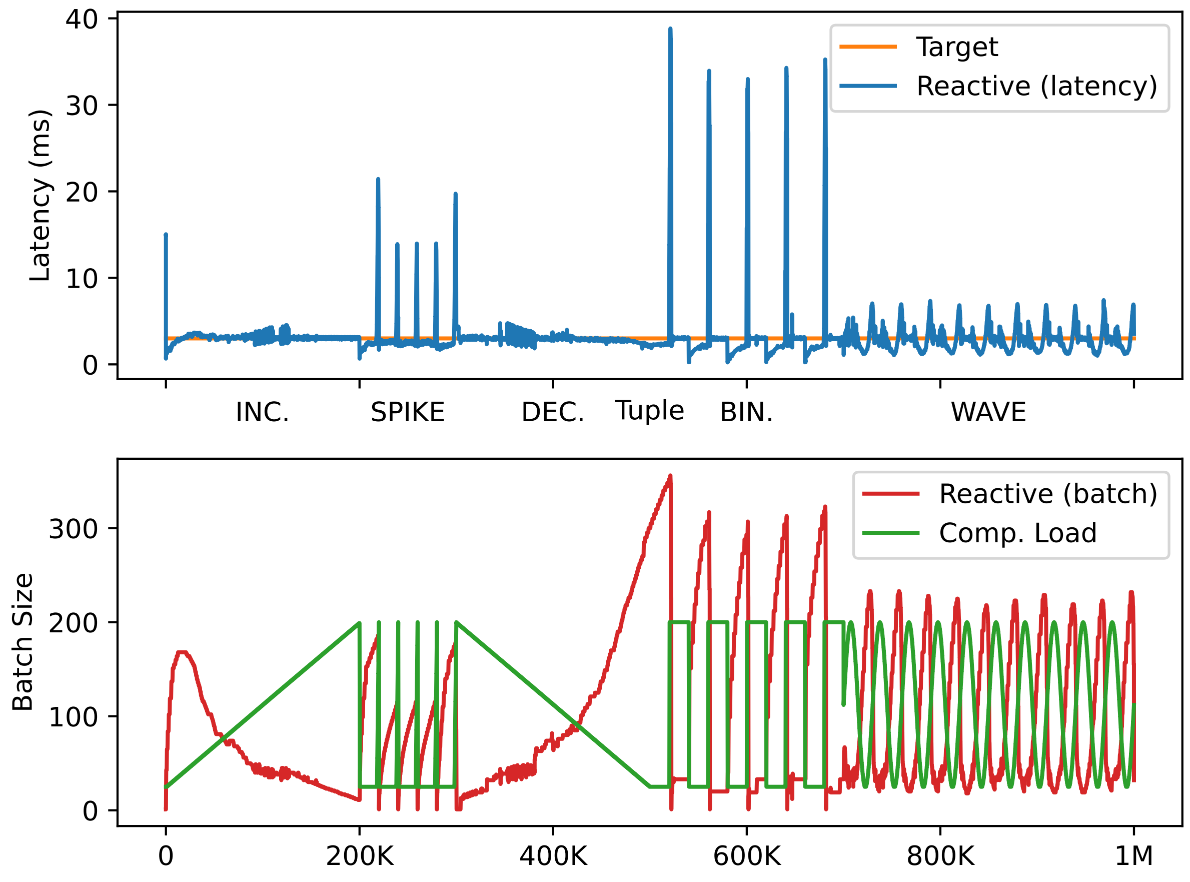

Figure 5.7 – Execution using the Reactive Strategy. The latency, batch size and computational load per item are shown.

5.3.3 Experiments with the Proportional-Integral-Derivative Controllre (PID)

This section presents the evaluation of the PID controller applied to the batch size adaptation for the MS Benchmark SPA. The experiment parameters that the PID controller has in common with the evaluated step-based algorithms are the sample size, threshold, and target latency. All parameters in common have the same values in the two sets of experiments. We show their values again in this section for completeness. The parameters for the PID controller evaluation are as follows:

- Target latency: 3 milliseconds
- Threshold values: 5%, 10%, 15%, and 20%
- Latency sample sizes: 1, 5, 10, and 20
- Proportional gain (K_p): 0, 5, 10, 15, 20, and 30
- Integral gain (K_i): 5, 15, 25, 35, and 50
- Derivative gain (K_d): 0, 0.5, 3, and 5

The threshold parameter does not affect the PID controller's runtime behavior (in the same vein as PBAF-WT). It is used to compute the SLO hit metrics. The PID gains were selected by manually varying each term and observing its effect while other terms were kept fixed.

We began the selection by setting K_p and K_d to 0 while increasing K_i from 5 to 50, as shown in the parameter list above. We observed that the SLO metrics started to degrade after K_i increased beyond 10. However, we kept the higher values of K_i for the complete experiment battery since they could still present better results in combination with other gains. Next, we found the set of values for K_p by using K_i = 10 and K_d = 0 (the setting with the highest SLO metrics so far). Degraded results were observed when $K_p > 10$. Finally, given that K_p = 10 and K_i = 15 achieved the best result having K_d = 0, we explored some values of K_d from 0 to 5. We found that diminishing results appeared after K_d = 3.

Table 5.5 demonstrates that the PID gains achieving the best results across most metrics are K_p = 10, K_i = 15, and K_d = 3. Exploring different gains did not result in better metrics, indicating that each Gaine has a single peak value. Since we skipped some gain values to complete our experiments in a timely manner (e.g., K_i going from 5 to 15), the actual peak gains can be slightly different.

Figure 5.8 complements Table 5.5 by comparing the PID controller's performance against all evaluated step-based algorithms across each metric within the 5% threshold.

Table 5.5 – Best PID controller SLO metrics and configurations by threshold. Each metric value is the average of 10 executions. The word *Any* is used when the threshold does not matter (for the distance metrics).

SLO Hit Metrics						
Threshold	Metric	Metric Value	Sample Size	K_p	K_i	K_d
5%	B-SLH	51.21	1	10	15	3
	I-SLH	38.5	1	10	15	3
10%	B-SLH	62.89	1	10	15	3
	I-SLH	52.07	1	10	15	3
15%	B-SLH	67.69	1	10	15	3
	I-SLH	59.85	1	10	15	3
20%	B-SLH	72.2	1	10	15	3
	I-SLH	66.32	1	20	15	3
SLO Distance Metrics						
Any	MAD-D	17.03	1	10	15	3
Any	SD-D	36.86	1	10	5	5

As a summary, Figure 5.8 excludes the parameters used to achieve the metric values, which can be found in Table 5.5, Table 5.4, and Table 5.1.

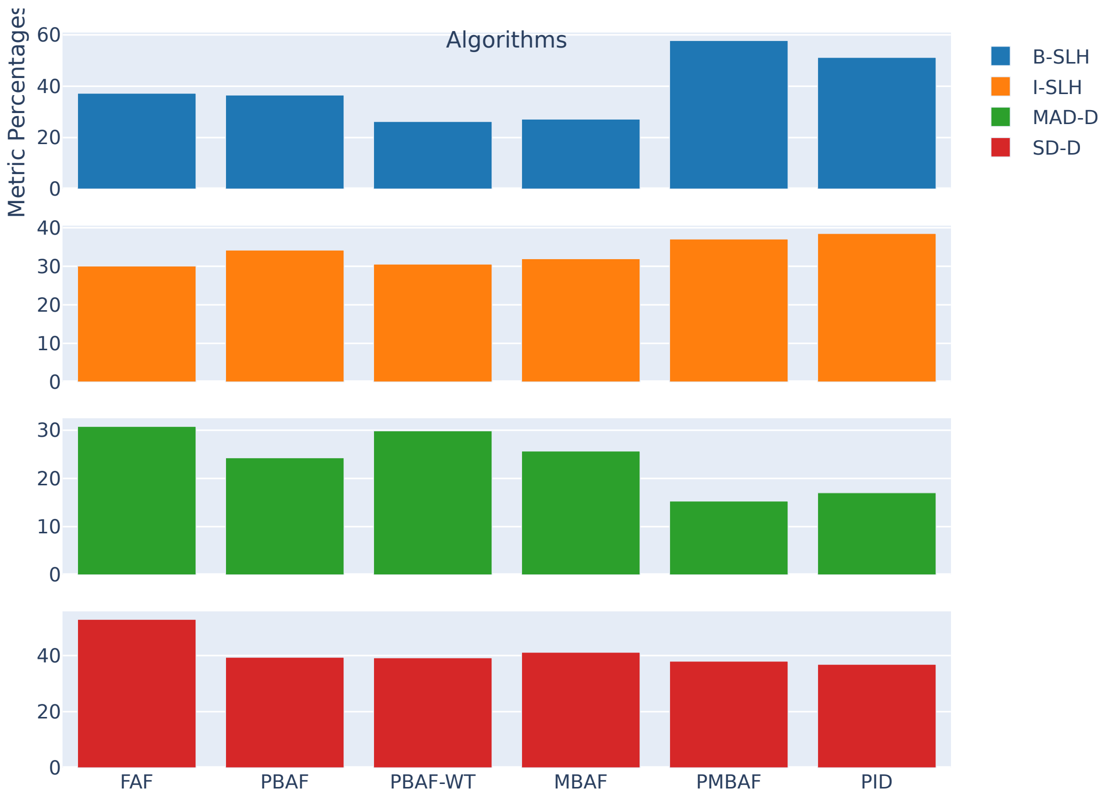

Figure 5.8 – Best SLO hit and distance metrics per algorithm. The values are averages of ten executions. In the labels, the batched and itemized SLO hits are shortened to B-SLH and I-SLH, while the MAD and SD-based distances are shortened to MAD-D and SD-D.

A comparison between Table 5.5 and Table 5.1 evinces that the PID controller can achieve comparable and slightly higher results for the SLO hit metrics in most thresholds. The PID controller results are comparable to PMBAF's, the algorithm achieving the best results for the metrics in Table 5.1. It achieved an itemized SLO hit from 1.4 (5% threshold) to 4.4 (15% threshold) metric points higher than PMBAF.

The PID controller also achieved a MAD-based distance comparable with PMBAF, with the distance being only 1.7 metric points higher than PMBAF's best result from Table 5.4 (17.03 − 15.3). The only row in Table 5.5 showing differing PID gains corresponds to the SD-based distance. In this row, K_i is reduced from 15 to 5, while K_d is increased from 3 to 5. This renders the PID controller-less reactive, as the adjustments in batch size made by K_p and K_d are minimal compared to K_i, resulting in a strategy similar to the *slow convergence strategy* presented in Figure 5.6.

The integral gain was found to be the more influential overall, while the proportional and derivative gains improved the metrics minimally. The highest average itemized SLO hit achieved having the proportional and derivative gains as 0 (integral-only result) is 35.79%, only 2.71 metric points below the best PID result from Table 5.5. The difference between the integral-only result and the table's result was mainly caused by two (out of the ten) executions that deviated significantly from the average, achieving 29% and 31% in the metric. We can reduce the difference to 1.38 metric points by not considering these executions. A proportional gain of 15 was more effective than the other tested values because it is large enough to increase reactivity when the latency is far from the target and smaller enough not to cause disturbances that result in SLO misses when close to the target. The same behavior was observed for the derivative gain in very reduced proportions.

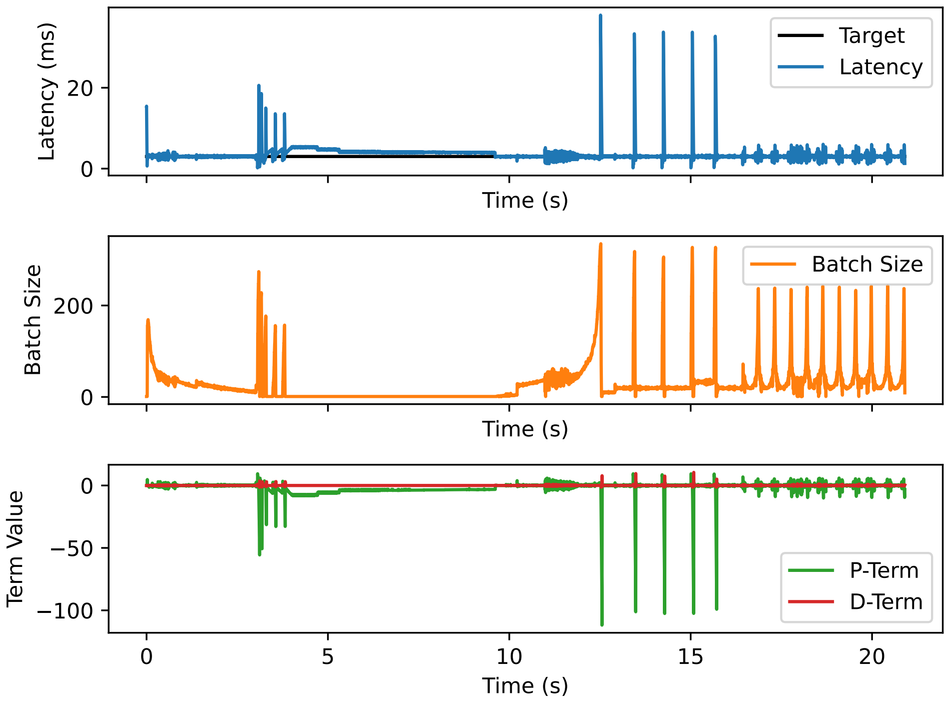

Figure 5.9 – The influence of the proportional and derivative terms in batch size and latency over time.

Figure 5.9 shows how the proportional and derivative terms influence the changes in batch size and latency over time. The data comes from the execution having the itemized SLO hit closest to the best average result (Table 5.5) from below. The exact metric value for the execution is 38.1%, while the best average is 38.5%. We can observe that the derivative term has minimal effect on the batch size. Its effect is higher when the latency falls abruptly from above to below the target, generating a positive value. An average of the highest 200 d-term values reveals an increment of 2 in the batch size. The proportional term is more noticeable when the latency is above the target, which generates a negative value. This happens because the latency can be many times higher than the target, while it can only be a fraction of the target lower. An average of the highest 200 p-term values reveals a 31 (-31) decrease in the batch size.

5.4 Key Findings and Parameter Tuning Recommendations

Our key findings can be summarized as follows.

- The self-adaptive algorithms mostly achieve comparable results (although using different parameter combinations) for the SLO hit metrics, and the results become even more homogeneous as the threshold increases. From the SLO distance metrics perspective, PMBAF can stay 40% closer to the target latency than other algorithms.
- The results achieved with the evaluated application (MS benchmark) and workload are significantly lower for the algorithms originating from the work of Stein et al.[47], even when using the same metric. In the original paper, the algorithms were evaluated by using the batched SLO hit metric.
- The itemized SLO hits were consistently lower than the batched SLO hits, indicating that relying solely on the latter metric to assess SLO compliance can be misleading. Furthermore, the batched SLO hit can vary widely for a set of executions using the same configuration. With a standard deviation of 12, the batched SLO hit varied from 22% to 71% among ten execution samples. The standard deviations are shown in table 5.2.
- The SLO hit and distance metrics demonstrate that more reactive algorithms (such as PMBAF and MBAF) and parameters produce superior results overall. This is due to the highly unstable latency in the proposed workload.
- The SLO distance metrics indicate that less responsive parameter combinations attaining low SLO hits can still be close to the threshold and mitigate large latency spikes. The downside is that this can negatively impact throughput.
- The PID controller attained comparable results to PMBAF across all metrics, especially on the itemized SLO hit metric, in which PMBAF was the overall most successful

step-based algorithm. The integral gain dominated the PID performance in all metrics, while the proportional and derivative gains presented minimal improvements.

If we were to offer a general recommendation for one of the evaluated self-adaptive algorithms based on our experience performing the evaluation, we would suggest PMBAF because it is generally applicable across most of the tested computation patterns. It is also (subjectively) easier to tune than the PID controller. As for the step size parameter in PMBAF, we would suggest a value between 10 and 15 – 10 for the 5% threshold and 15 for any higher threshold. For the latency sample size parameter, the more general and conservative suggestion would be a quantity of five samples to account for possible fluctuations in latency caused by the hardware and software environment, even though we did not observe a substantial gain (as per the metrics) for the tested application and environment when using a latency sample size greater than one. There were gains in the batched sLO hit metric for using a sample size greater than one, as can be verified in Figure 5.4. However, we consider the batched SLO hit unreliable as a metric due to its high standard deviation and the problem that all batches have the same weight.

6. CONCLUSION

In this work, we presented an evaluation of four self-adaptive algorithms for stream processing with GPUs from Stein et al.[47]. We also proposed a merge of the PBAF and MBAF algorithms from [47] into a new one called PMBAF. Furthermore, our evaluation included experiments with the PID controller, which proved to be comparable to PMBAF across all metrics. Four metrics guided the evaluation. One of them (the batched SLO hit) was used in the paper by Stein et al.[47], and the other three (itemized SLO hit, MAD-based SLO distance and SD-based SLO distance) were proposed in the course of this research as a way to compare the self-adaptation algorithms among themselves from different perspectives. We implemented the self-adaptive strategy and algorithms with FastFlow [6] and GSParLib [45] in the Military Server benchmark application [8].

In general, we conclude that the self-adaptive algorithms mostly achieve comparable results (although using different parameter combinations) for the SLO hit metrics, and the results become even more homogeneous as the threshold increases. PMBAF achieved the best results overall due to its ability to combine fine-tuning and reactivity to pursue the latency target. The PID controller was comparable to PMBAF in the itemized SLO hit metric, with a slight advantage of up to 4.4% metric points depending on the threshold.

Though there are differences in how the algorithms handle latency variations, including some inherent to their functioning, these differences did not produce significant deviations in most of the evaluation results. However, individually looking at the metrics for specific workload regions (computation patterns) makes the algorithms' differences more pronounced than when looking at the metrics for the whole execution. It is possible to observe that highly reactive algorithms like PMBAF achieve the highest itemized SLO hits in the binary pattern while also achieving the lowest values for the same metric in the spike pattern. Moreover, we can discern some effects of the parameter combination over the algorithms, such as a latency sample size of five (instead of one) causing a greater batched SLO hit for gradual patterns like increasing or decreasing.

We have identified mainly two classes of limitations, one related to the algorithms and the other related to the application and workload used for the evaluation. One of the current algorithms' limitations is the exclusive focus on latency. Having only the latency as a target creates situations in which the batch size is increased with the sole purpose of artificially increasing latency, e.g., where there is no gain in terms of throughput. Another notable limitation concerns the choice of a suitable step size. Although the MBAF, PMBAF, PBAF, and PBAF-WT algorithms are able to change the step size at runtime, they rely on a default value set by the users, that need to (experimentally or by intuition) find the appropriate value for their specific applications.

Regarding the limitations coming from our choice of application and workload, it is important to highlight that the application is synthetic and might not closely represent the real-world SPAs that could benefit from the adaptive algorithms evaluated in the current study. Further analysis can be done to understand if this is the case, and if so, choose more realistic applications and workloads, or adapt the synthetic application to better reflect a realistic scenario.

In the experiments performed for this study, the target latency was 3 milliseconds with a minimum threshold of 5%, which results in a tolerance of 0.1 milliseconds (100 microseconds) around the target. This can be considered a too-strict scenario, and may not be as prevalent in practice. However, this was the threshold in which the algorithms presented less homogeneous results and could be more easily differentiated in terms of their adaptation behavior.

In future research, we plan to:

- Explore new approaches to overcome the aforementioned limitations of the evaluated algorithms. Some alternatives may comprise predictive algorithms that learn the characteristics of the workload at runtime. Another possibility is to combine predictive and reactive algorithms. For instance, a predictive model can detect that the batch latencies are changing as per the binary pattern and then adjust the reactive algorithm's parameters to deal better with this type of workload. One helpful approach to spare users from finding the proper algorithm parameters is employing a PID controller paired with another algorithm, such as the extremum-seeking method, to adjust the PID parameters automatically [32].

- Investigate viable real-world applications to complement the synthetic application already in use. We are currently working on assessing the financial applications Spike Detection and Fraud Detection, which were implemented using Windflow with GPU support [41] and evaluated with a focus on throughput using fixed batching. We implemented self-adaptive batch size control directly in the Windflow library so that we can evaluate Windflow applications in the future with minimal modifications to their business domain code.

- Introduce a workload based on frequency patterns, in addition to the existing computation patterns with a fixed frequency.

- Target SLOs beyond latency such as throughput.

- Develop a multi-target self-adaptation algorithm or modify existing ones to accommodate multiple objectives. For example, an algorithm capable of balancing both latency and throughput targets.

We presented a paper [36] with our partial results in the International Workshop on Scalable Compute Continuum (WSCC), a workshop Colocated with the 29th International European Conference on Parallel and Distributed Computing (Euro-Par 2023). The paper is titled *Evaluation of Adaptive Micro-batching Techniques for GPU-accelerated Stream Processing*. It presents our proposed SLO metrics and uses them to evaluate the four algorithms from Stein et al. [47].

REFERENCES

[1] “Apache flink: Stateful computations over data streams”. Source: https://flink.apache.org, Feb 2023.

[2] “Apache hadoop”. Source: https://hadoop.apache.org, Feb 2023.

[3] “Apache spark: Unified engine for large-scale data analytics”. Source: https://spark.apache.org, Feb 2023.

[4] “Apache storm”. Source: https://storm.apache.org, Feb 2023.

[5] Abdelhamid, A. S.; Mahmood, A. R.; Daghistani, A.; Aref, W. G. “Prompt: Dynamic data-partitioning for distributed micro-batch stream processing systems”. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, 2020, pp. 2455–2469.

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

[7] Andrade, H. C. M.; Gedik, B.; Turaga, D. S. “Fundamentals of Stream Processing: Application Design, Systems, and Analytics”. Cambridge University Press, 2014, 529p.

[8] Araujo, G. A. d.; et al.. “Data and stream parallelism optimizations on gpus”, Master’s Thesis, Pontifícia Universidade Católica do Rio Grande do Sul, 2022.

[9] Brakmo, L.; Peterson, L. “Tcp vegas: end to end congestion avoidance on a global internet”, *IEEE Journal on Selected Areas in Communications*, vol. 13–8, 1995, pp. 1465–1480.

[10] Brunton, S. L.; Kutz, J. N. “Data-driven science and engineering: machine learning, dynamical systems, and control”. Cambridge University Press, 2021, 472p.

[11] Canuto, S.; Gonçalves, M.; Santos, W.; Rosa, T.; Martins, W. “An efficient and scalable metafeature-based document classification approach based on massively parallel computing”. In: Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval, 2015, pp. 333–342.

[12] Carbone, P.; Fragkoulis, M.; Kalavri, V.; Katsifodimos, A. “Beyond analytics: The evolution of stream processing systems”. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, 2020, pp. 2651–2658.

[13] Carpen-Amarie, M.; Marlier, P.; Felber, P.; Thomas, G. “A performance study of java garbage collectors on multicore architectures”. In: Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores, 2015, pp. 20–29.

[14] Chakravarthy, S.; Jiang, Q. “Stream Data Processing: A Quality of Service Perspective - Modeling, Scheduling, Load Shedding, and Complex Event Processing”. Springer, 2009, 340p.

[15] Chen, C.; Li, K.; Ouyang, A.; Li, K. “Flinkcl: An opencl-based in-memory computing architecture on heterogeneous cpu-gpu clusters for big data”, *IEEE Transactions on Computers*, vol. 67–12, 2018, pp. 1765–1779.

[16] Chen, C.; Li, K.; Ouyang, A.; Zeng, Z.; Li, K. “Gflink: An in-memory computing architecture on heterogeneous cpu-gpu clusters for big data”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 29–6, 2018, pp. 1275–1288.

[17] Chen, Y.; Lai, S.; Wang, B.; Lin, F.; Chen, B. M. “A gpu mapping system for real-time robot motion planning”. In: 2021 IEEE International Conference on Real-time Computing and Robotics (RCAR), 2021, pp. 762–768.

[18] Chen, Z.; Xu, J.; Tang, J.; Kwiat, K. A.; Kamhoua, C. A.; Wang, C. “Gpu-accelerated high-throughput online stream data processing”, *IEEE Transactions on Big Data*, vol. 4–2, 2018, pp. 191–202.

[19] 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, 2018, pp. 2672–2685.

[20] da Silva Veith, A.; de Assunção, M. D.; Lefèvre, L. “Latency-aware placement of data stream analytics on edge computing”. In: Service-Oriented Computing, 2018, pp. 215–229.

[21] Dally, W. J.; Keckler, S. W.; Kirk, D. B. “Evolution of the graphics processing unit (gpu)”, *IEEE Micro*, vol. 41–6, 2021, pp. 42–51.

[22] 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.

[23] de Lemos, R.; Giese, H.; Müller, H. A.; Shaw, M.; Andersson, J.; Litoiu, M.; Schmerl, B.; Tamura, G.; Villegas, N. M.; Vogel, T.; Weyns, D.; Baresi, L.; Becker, B.; Bencomo, N.; Brun, Y.; Cukic, B.; Desmarais, R.; Dustdar, S.; Engels, G.; Geihs, K.; Göschka, K. M.; Gorla, A.; Grassi, V.; Inverardi, P.; Karsai, G.; Kramer, J.; Lopes, A.; Magee, J.; Malek, S.;

Mankovskii, S.; Mirandola, R.; Mylopoulos, J.; Nierstrasz, O.; Pezzè, M.; Prehofer, C.; Schäfer, W.; Schlichting, R.; Smith, D. B.; Sousa, J. P.; Tahvildari, L.; Wong, K.; Wuttke, J. “Software engineering for self-adaptive systems: A second research roadmap”. In: Software Engineering for Self-Adaptive Systems, 2013, pp. 1–32.

[24] 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, 2019, pp. 48753–48769.

[25] Garcia, A. M.; Griebler, D.; Schepke, C.; Fernandes, L. G. L. “Evaluating micro-batch and data frequency for stream processing applications on multi-cores”. In: 2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), 2022, pp. 10–17.

[26] González-Vélez, H.; Leyton, M. “A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers”, *Software: Practice and Experience*, vol. 40–12, 2010, pp. 1135–1160.

[27] 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, 2017, pp. 1740005.

[28] 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, 2020, pp. 4603–4628.

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

[30] Hennessy, J.; Patterson, D. “Computer Architecture: A Quantitative Approach”. Elsevier Science, 2011, 856p.

[31] Hirzel, M.; Soulé, R.; Schneider, S.; Gedik, B.; Grimm, R. “A catalog of stream processing optimizations”, *ACM Computing Surveys*, vol. 46–4, 2014, pp. 1–34.

[32] Killingsworth, N.; Krstic, M. “Pid tuning using extremum seeking: online, model-free performance optimization”, *IEEE Control Systems Magazine*, vol. 26–1, 2006, pp. 70–79.

[33] Kirk, D. B.; Hwu, W. W. “Programming massively parallel processors: a hands-on approach”. Morgan Kaufmann Publishers Inc., 2010, 435p.

[34] Koliousis, A.; Weidlich, M.; Castro Fernandez, R.; Wolf, A. L.; Costa, P.; Pietzuch, P. “Saber: Window-based hybrid stream processing for heterogeneous architectures”.

In: Proceedings of the 2016 International Conference on Management of Data, 2016, pp. 555–569.

[35] Kolter, J. Z.; Plagemann, C.; Jackson, D. T.; Ng, A. Y.; Thrun, S. “A probabilistic approach to mixed open-loop and closed-loop control, with application to extreme autonomous driving”. In: 2010 IEEE International Conference on Robotics and Automation, 2010, pp. 839–845.

[36] Leonarczyk, R.; Griebler, D.; Mencagli, G.; Danelutto, M. “Evaluation of adaptive micro-batching techniques for gpu-accelerated stream processing”. In: Euro-Par 2023: Parallel Processing Workshops, 2023.

[37] Li, W.; Zhang, Z.; Shu, Y.; Liu, H.; Liu, T. “Toward optimal operator parallelism for stream processing topology with limited buffers”, *The Journal of Supercomputing*, vol. 78–11, 2022, pp. 13276–13297.

[38] Mai, L.; Zeng, K.; Potharaju, R.; Xu, L.; Suh, S.; Venkataraman, S.; Costa, P.; Kim, T.; Muthukrishnan, S.; Kuppa, V.; Dhulipalla, S.; Rao, S. “Chi: A scalable and programmable control plane for distributed stream processing systems”, *Proceedings of the VLDB Endowment*, vol. 11, 2018, pp. 1303–1316.

[39] Mattson, T.; Sanders, B.; Massingill, B. “Patterns for Parallel Programming”. Pearson Education, 2004, 384p.

[40] 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. 32–11, 2021, pp. 2748–2763.

[41] Mencagli, G.; Torquati, M.; Griebler, D.; Fais, A.; Danelutto, M. “General-purpose data stream processing on heterogeneous architectures with windflow”, *Journal of Parallel and Distributed Computing*, vol. 184, 2024, pp. 104782.

[42] Nasiri, H.; Nasehi, S.; Goudarzi, M. “A survey of distributed stream processing systems for smart city data analytics”. In: Proceedings of the International Conference on Smart Cities and Internet of Things, 2018.

[43] Nichols, B.; Buttlar, D.; Farrell, J.; Farrell, J. “PThreads Programming: A POSIX Standard for Better Multiprocessing”. O’Reilly Media, Incorporated, 1996, 284p.

[44] Owens, J. D.; Luebke, D.; Govindaraju, N.; Harris, M.; Krüger, J.; Lefohn, A. E.; Purcell, T. J. “A survey of general-purpose computation on graphics hardware”, *Computer Graphics Forum*, vol. 26–1, 2007, pp. 80–113.

[45] Rockenbach, D. A. “High-level programming abstractions for stream parallelism on gpus”, Master’s Thesis, Pontifícia Universidade Católica do Rio Grande do Sul, 2020.

[46] Rockenbach, D. A.; Stein, C. M.; Griebler, D.; Mencagli, G.; Torquati, M.; Danelutto, M.; Fernandes, L. G. “Stream processing on multi-cores with gpus: Parallel programming models’ challenges”. In: 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2019, pp. 834–841.

[47] Stein, C. M.; Rockenbach, D. A.; Griebler, D.; Torquati, M.; Mencagli, G.; Danelutto, M.; Fernandes, L. G. “Latency-aware adaptive micro-batching techniques for streamed data compression on graphics processing units”. In: Concurrency and Computation: Practice and Experience, 2021, pp. 5786.

[48] Storer, J. A.; Szymanski, T. G. “Data compression via textual substitution”, *Journal of the ACM (JACM)*, vol. 29–4, 1982, pp. 928–951.

[49] Süli, E.; Mayers, D. F. “An introduction to numerical analysis”. Cambridge university press, 2003, 433p.

[50] Thomas, P.; Kirschnick, J.; Hennig, L.; Ai, R.; Schmeier, S.; Hemsen, H.; Xu, F.; Uszkoreit, H. “Streaming text analytics for real-time event recognition”. In: Proceedings of the International Conference Recent Advances in Natural Language Processing, RANLP 2017, 2017, pp. 750–757.

[51] Vavilapalli, V. K.; Murthy, A. C.; Douglas, C.; Agarwal, S.; Konar, M.; Evans, R.; Graves, T.; Lowe, J.; Shah, H.; Seth, S.; Saha, B.; Curino, C.; O’Malley, O.; Radia, S.; Reed, B.; Baldeschwieler, E. “Apache hadoop yarn: Yet another resource negotiator”. In: Proceedings of the 4th Annual Symposium on Cloud Computing, 2013.

[52] Venkataraman, S.; Panda, A.; Ousterhout, K.; Armbrust, M.; Ghodsi, A.; Franklin, M. J.; Recht, B.; Stoica, I. “Drizzle: Fast and adaptable stream processing at scale”. In: Proceedings of the 26th Symposium on Operating Systems Principles, 2017, pp. 374–389.

[53] 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. 34–6, 2022, pp. 6759.

[54] Weyns, D. “Software Engineering of Self-adaptive Systems”. Cham: Springer International Publishing, 2019, pp. 399–443.

[55] Wu, S.; Hu, D.; Ibrahim, S.; Jin, H.; Xiao, J.; Chen, F.; Liu, H. “When fpga-accelerator meets stream data processing in the edge”. In: 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), 2019, pp. 1818–1829.

[56] Zaharia, M.; Das, T.; Li, H.; Hunter, T.; Shenker, S.; Stoica, I. “Discretized streams: Fault-tolerant streaming computation at scale”. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, 2013, pp. 423–438.

[57] Zhang, Q.; Song, Y.; Routray, R. R.; Shi, W. “Adaptive block and batch sizing for batched stream processing system”. In: 2016 IEEE International Conference on Autonomic Computing (ICAC), 2016, pp. 35–44.

[58] Zhang, S.; He, B.; Dahlmeier, D.; Zhou, A. C.; Heinze, T. “Revisiting the design of data stream processing systems on multi-core processors”. In: 2017 IEEE 33rd International Conference on Data Engineering (ICDE), 2017, pp. 659–670.

[59] Zhang, Y.; Mueller, F. “Gstream: A general-purpose data streaming framework on gpu clusters”. In: 2011 International Conference on Parallel Processing, 2011, pp. 245–254.

[60] Zhou, S.; Xie, M.; Jin, Y.; Miao, F.; Ding, C. “An end-to-end multi-task object detection using embedded gpu in autonomous driving”. In: 2021 22nd International Symposium on Quality Electronic Design (ISQED), 2021, pp. 122–128.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Pesquisa e Pós-Graduação
Av. Ipiranga, 6681 – Prédio 1 – Térreo
Porto Alegre – RS – Brasil
Fone: (51) 3320-3513
E-mail: propesq@pucrs.br
Site: www.pucrs.br