Implementation and Evaluation of Ordo: a High Performance Data Processing System

by

Minas Melas

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday March 20, 2023 at 12:00 PM.

Student number:	5397324	
Project duration:	February 1, 2022 – February 15, 2023	
Thesis committee:	Prof. Dr. Jan Rellermeyer,	TU Delft, supervisor
	Dr. Lydia Chen,	TU Delft
	Dr. Asterios Katsifodimos,	TU Delft

This thesis is confidential and cannot be made public until March 20, 2023.

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

Data processing systems have become increasingly important in modern computing, as the volume and complexity of data that needs to be analyzed has grown dramatically. Multiple data processing systems have been and are being developed, that are scalable, resilient and performant.

However, despite the advances made in data processing technology, there are still challenges that need to be addressed in order to optimize the performance, energy efficiency as well as the practicality of these systems. One such challenge is the need to effectively manage the underlying system's resources, including the system's throughput and the amount of work that each operator has to do and to use optimal data-structures that would lead in faster task processing speeds.

To address this challenge, this thesis proposes the implementation of a high-performance data processing system that exposes the underlying system's metrics to the application level and applys an innovative way for operator communication, by utilizing an efficient thread-safe data structure. By providing underlying system's metrics to the application's scheduler, the scheduler can schedule the tasks optimally according to the current system's state and adjust the system's resources during runtime. This alleviates the developers from having to fine-tune the system beforehand and allows the system to tackle fluctuating input workload more efficiently.

This thesis will explore the design and implementation of such system, as well as its impact on the performance, energy-efficiency and resiliency of data processing applications. We provide performance measurements as well as a qualitative comparison of our system compared to other state-of-the art systems, proving our hypotheses.

Acknowledgements

Throughout the research, design and implementation of this Thesis I have received enormous support from several people.

I would like to start by expressing my sincere appreciation to Professor Dr. Jan Rellermeyer for his committed assistance, stimulating discussions and precious feedback throughout the whole process, not only regarding the Thesis, but also regarding other matters of my curriculum through my final steps before graduation. Furthermore, I would like to extend my heartfelt thanks to the members of the graduation committee, Dr. Lydia Chen and Dr. Asterios Katsifodimos.

Finally, I would like to express my deep gratitude to my parents, my sister and my friends for their unwavering support, love, and encouragement throughout my academic journey.

Contents

1 Introduction

Data processing has been increasing in popularity and demand over the years as more and more applications aiming at processing huge amounts of data are being developed. Data processing is required in multiple different areas [1] including astronomy, medicine, finance, recommendation systems and many more interdisciplinary scientific researches. In addition, all these areas have gained more and more attraction in the Big Data [2] era, where tons of information and data is produced and distributed across the internet. Some examples include machine learning models [3] that require considerable amount of hours to train, fintech [4] applications that require fast and resilient transactions and high-frequency trading companies [5] want to react to a change in the market as fast as possible. These applications, along with many more, are using some form of data processing. Thus, one can see how important the performance is in terms of throughput, energy-efficiency, resiliency and more for a data processing system.

Nowadays, multiple different data processing systems exist, each one being unique according to it's design. There are trade-offs of using one system to another based on their different characteristics. For example, three big open-source stream-processing platforms, Apache Flink [6], Apache Storm [7] and Apache Spark [8], can have big discrepancies in terms of performance, node failure and node recovery due to the differences in their architectures [9]. Some of the main characteristics include the language that the system has been developed, the scheduler [10] that is responsible for assigning tasks to operators as well as the utilization of the underlying system from the application. Out of these, we put more emphasis on the scheduler which is one of the most critical components, since it is the decisive factor of the efficiency as well as the resiliency of the system.

Most of the current data processing systems are using the Operating System's threaded-architecture [11], in which they assign each task to a thread which will run until the task is over. However, using the Operating System's threaded-architecture comes with the overhead of context switches, as well as it does not allow the application to make scheduling decisions, rather assigns the scheduling job to the Operating System's scheduler which is neither optimized nor designed for such applications. Another kind of scheduling, which has not been researched in data processing systems, is the event-driven scheduling architecture. The first steps on researching this architecture have been made by SEDA [12], but an implementation of the system has not been provided. To the best of our knowledge, research has not been conducted with regards to event-driven scheduling in the data processing field. Our system architecture proposes an event-driven scheduling architecture has the benefits of avoiding the context switches, since a thread-pool is initialized and a thread is assigned by the scheduler to perform a task. In addition, today's data processing systems do not expose the system's metrics to the applications. Our proposal is that exposing the system metrics to the applications, we allow the development of a scheduler that is closer to the application, allowing to take decisions regarding specifics such as the workload of each operator, the throughput of each operator and other metrics that are present in the application-layer that can guide the scheduler into making better scheduling decisions. In addition, most of today's data processing systems like Apache Spark and Apache Flink, require a lot of effort from the developer to fine-tune the system in order to achieve the desired performance [13]. However, by exposing the underlying system's metrics, the application can self-adapt during run-time and thus require much less configuration at start-time.

As aforementioned, in high performance data processing systems, the programming language as well as the utilization of the underlying system are very important. Most of today's data processing systems are built in Java (Hadoop) [14], [15] or Scala (Spark) [16], because Java and Scala are known for their versatility and ability to use multiple data science techniques [17]. In addition, they are flexible and portable languages with a broad user base, making them more attractive for developer that wish to create an application on top of a data processing system. However, both of these languages are not close to the machine, because they both use a JVM [14] which acts as a run-time engine that runs the applications. Using a JVM has a lot of advantages, like being able to run in any operating system (portability), writing applications easier because of not having to manage memory which is taken care of by the garbage collector (usability). However, the memory footprint of Java is higher than in lower level languages in which the developer can manage memory like C++ [18] and Rust [19]. The garbage collector by itself requires extra memory in order to trace the memory consumption of the Java program, making the programs that are written in lower level languages lighter and thus faster. In addition, the runtime of Java programs is also affected by the middle layers that exist between the application code and the byte code. JVM compiles the java programs into classes which are loaded by the Class Loader, verified by the ByteCode verifier and are fed as input to the JIT compiler which tries to output optimized execution code for the machine to run. On the other hand, compiled languages are producing the machine code at once making their execution faster. For this reason, we decided to write our data processing system in Rust, which is a very promising language that was developed in 2013 and has seen a lot of attraction the last years [20]. By using lower-level languages, allows programmers being closer to the hardware, by providing them libraries that will contain less lines of code and in the end will execute faster than libraries that are used in higher-level languages, more ownership and access to program's memory and more. Last but not least, the utilization of the underlying system becomes much more flexible in the application level by using lower-level languages. On top of that, most data processing systems do not provide APIs for lower-level languages, disabling developers to write their applications in these. However, the aforementioned speed and memory improvements take place in the application level as well.

1.1. Problem Statement

The primary goal of this thesis is the implementation and evaluation of a high-perfromance data processing system based on the event-based scheduling architecture. We propose that allowing the application to access system metrics results in the ability to build more efficient schedulers that target the application directly and avoid latency overheads of the generalized operating system scheduling mechanisms. In addition, we propose that exposing such metrics to the application, allows the system to reach an equilibrium state, where it adapts quicker to fluctuating workloads as well as reduces the need for excessive fine-tuning of parameters like other state-of-the art systems require. We identify three key points for evaluating our system and comparing it with other state-of-the art systems:

- **Raw Performance**
  - Runtime
  - Throughput
- **Efficiency**
  - Energy Efficiency
  - Per item latency
- **Resiliency**
  - Behavior on worker errors

1.2. Research Questions

Following our problem statement, we can formulate the main research question as :

RQ : *Can the underlying system's metrics be exposed in the application level in order to allow the building of efficient schedulers that can reach an equilibrium state and outperform other state-of-the art systems?*

We split this main research question into more precise sub-parts as follows:

RQ1 : *Can we improve the performance and practicality of a data processing system by exposing the underlying system's metrics to the application?*

RQ2 : *Can a scheduler that has access to the system metrics reach an equilibrium state ?*

RQ4 : *What is the most effective way of building an application on such system?*

RQ5 : *What are the limitations that such system imposes?*

RQ3 : *Does such system outperform other state-of-the art systems in , terms of performance, practicality, energy efficiency and resiliency ?*

1.3. Chapters Overview

In order to answer our research questions, we formed the structure of our thesis as follows. We begin with Chapter 2 where we summarize some background information and previous related work relevant to our research. This serves as a foundation for the reader to understand the context and significance of our research. Chapter 3 contains our complete system and application design. It begins with the final design implementation, which includes the system design, the scheduler design as well as the application. In Section 3.4 we provide an overview of the challenges we encountered and a description of the various iterations we conducted before arriving at the final implementation. Additionally, we include a performance comparison of the different iterations, providing valuable insights into the optimization process. In Chapter 4, we move on to the experimental phase of our research, where we aim to validate our proposals and answer the remaining research questions. We perform a series of experiments on our final system design and compare it with other state-of-the-art systems. The results of these experiments provide valuable insights into the performance and efficiency of our proposed system and help to answer the remaining research questions. Finally, in Chapter 5, we conclude our work by summarizing our research questions and the results we obtained for each of them. We also impose future work that can act as guidance for further research to be made in this field.

2

Background and Related Work

A lot of progress has been made the past years in the data processing systems world. In this chapter we begin by discussing some basic computer science concepts that are relevant to our research. We start with a short discussion on programming languages including their compilation and runtime characteristics in section 2.1. We continue with a description of memory allocation in Section 2.2, a brief discussion of Processes and Threads in Section 2.3 and some relevant Data Structures in Section 2.4. We continue with the introduction of the most common and important data processing systems components (Section 2.5). Subsequently, we explore the two programming models that we mainly focus on Section 2.6 and some existing architectures that are based on these models on Section 2.7. We then proceed to an overview of the state-of-the art data processing systems and how these evolved in the last years, as shown in Section 2.8, before we state our research's main goals in section 2.9.

2.1. Programming Languages

Programming languages act as the intermediary between humans and the computers. In the modern world, we have multiple different devices, services and applications with a variety of functions. As such, different programming languages are required for different purposes. Some languages are easier to understand and use, but offer less control over the hardware (most usually the interpreted languages). On the other hand, there are languages that are closer to the machine, which offer a lot of low level utilities to the programmers, with the cost of being harder to understand and easier to create bugs (compiled languages). There are also languages that combine the characteristics of the previous two, which first compile the programs and then interpret the compiled code. We decided to implement our system in a low-level language, Rust, in order to have as much flexibility as possible in both the system and application design.

2.1.1. Compiled Languages

Compiled languages tend to be faster and more efficient than interpreted languages. This is taking place because they are compiled directly into machine code that the processor can execute. In addition, these types of programming languages also give the developer more control and flexibility over the hardware, like memory management. We chose to develop our system in such a language (Rust), because of the performance advantages we can gain from it's versatility.

Compiled languages need to be manually compiled before executed. Every time a change is made the program has to be re-compiled. Some examples of compiled languages are Rust, C and C++.

2.1.2. Interpreted Languages

Interpreted languages have a computer program, the interpreter which runs through a program line by line and executes each program instruction. In this case, if a change is made in the program, it won't need to be manually compiled and "build", which can save all the compilation time that may take place in the compiled languages. In addition, interpreted programs can be modified while the program is running. Interpreted languages have a lower runtime performance than the compiled languages.

Examples of such programming languages include Python, Ruby and Javascript.

2.1.3. Compiled and Interpreted Languages

There are some languages that can be considered both compiled and interpreted languages. Such examples are Java, Scala. These languages's source code is first compiled into byte code which runs on the Java Virtual Machine (JVM), which is usually a software based interpreter [21]. The two data processing systems that we will compare our system with are written in this type of languages. Apache Spark [8] is written in Scala, while Apache Flink [6] is written in both Java and Scala.

2.2. Memory allocation

In order to implement a high-performance data processing system, it is important to know how memory works, since the system receives and processes a substantial amount of data over time. While developing our system, we tackled some challenges regarding to memory as described in Section 3.4, therefore in this section we analyze the way that modern operating systems allocate memory.

In modern operating systems the so called memory virtualization is taking place for protection and efficient memory sharing among different processes. Each process has a separate view of memory in the system. This view is called address space [22]. The address space of a process contains all the memory required for the program to run. Among other things like the code and some statically-initialized variables, the stack and the heap of the program lies in it's address space. The code is static so it lives in the top of the address space since it can't grow while the program runs. On the other hand, the stack and the heap can grow and shrink while the program runs, that's why we put them at the opposites ends of the address space. As we can see in Figure 2.1 the heap starts right after the code (at 1KB) and grows downward, while the stack starts at the end of the address space (16KB) and grows upward. In this example, the free address space that can be used by both the heap and the stack is 13KB and the current heap and stack sizes are 1KB each.

2.2.1. Stack

The stack is being used during the program's runtime. While the program runs, it stores the function call chain, the local variables, the parameteres and return values to the stack. When a function is called, a block is reserved on the top of the stack for all these local variables and bookkeeping of data. Whenever this function returns, the block becomes unused and can be used by the next function call. Each thread has it's own stack, which is sometimes called the thread stack.

2.2.2. Heap

The heap memory is set aside for dynamic memory allocation, managed by the user. Unlike the stack, the allocation and deallocation of blocks have no specific pattern in the heap. The user/ application is responsible for reserving and freeing space from the heap. There are many custom heap allocators available, or virtual machines like the JVM which contain their own internal memory allocator. In the case of the JVM, it initially requests some amount of heap memory from the operating system and it manages this memory while running with the use of garbage collectors. This is one core difference between Rust and Java/ Scala. In the former one, the developer is responsible for handling memory allocation, however for the latter cases, the JVM will take care of that. As aforementioned, in Section 3.4 we will see how JVM outperformed our first system implementation and how we overcame that by changing our system's memory allocation method.

2.3. Processes and Threads

Modern operating systems require the ability to provide concurrent operations even on a single CPU. For this reason, the concept of processes was created, which are abstractions of executing programs. Threads are very closely related to processes but with some key differences. Our system is heavily based on concurrency, therefore in this section we will analyze the different ways that concurrency can be achieved in modern operating systems.

2.3.1. Processes

Modern operating systems require the ability to provide concurrent operations even on a single CPU. A process is an executing program's abstraction that is leveraged by the operating system in order to turn the CPU into multiple virtual CPUs. A process is an instance of an executing program, such that it is

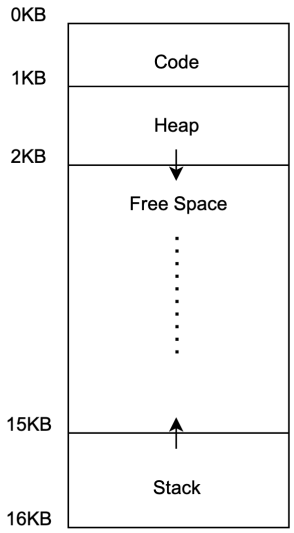


Figure 2.1: Address space example

consisted of the program counter, registers and variables of the program. Processes are very handful because they provide the ability to The CPU to switch between them, giving the illusion of parallelism. For example, some operations like disk accesses require a lot of time to complete. In that case, the operating system can decide to switch to another process in order to avoid the slow disk's processing time. However, switching to another process is not cost-free, as it requires a virtual address space switch.

2.3.2. Threads

A thread is a path of execution within a process. A thread consists of a program counter that keeps track of the next instruction to execute, some registers that save the current working variables and a stack, known as thread stack, which keeps the execution history. The main difference between threads and processes is that threads share the same address space. These attributes are required in order to allow the operating system to switch between threads without losing it's state. In standard operating systems, every process has a designated address space and a single thread of execution. However, having more than one threads within a process can yield a lot of benefits. Threads can have a positive impact in the performance if the operations performed within the process contain substantial computing and are I/O intensive. In addition, in the multi-core systems each thread can run on it's own core, achieving true parallelism. There are two types of threads, user level threads that are managed in user space and kernel level threads that are managed by the kernel. The aforementioned benefits were the reason that we chose to use threads in order to parallelize our system.

- **User level threads**

  User level threads are managed entirely by the user space. In that regard, the kernel does not know their existence and acts as if the process is single threaded. This indicates that the process has to store information about it's threads in order for the run-time system to be able to switch between them. This information is included in a thread table that consists of each thread's program counter, registers, the thread's state and more. User level threads have many benefits, such as enabling the applications to build their own customized scheduling algorithms. In addition, they scale better since the kernel requires a lot of space when there are a large number of threads.

- **Kernel level threads**

  Another type of threads are the kernel level threads. In this case, the kernel is responsible for managing and scheduling the threads. In order to do so, as in the case of the user level threads, the kernel needs to save the state of the threads. Thus, as in the user level threads case, the kernel holds a thread table that keeps each thread's program counter, registers, state and more.

- **Thread pools**

A thread pool is a collection of threads that are waiting for tasks to execute. Thread pools boost performance and decrease latency in execution because they remove the need of the creation and destruction of threads, along with the allocation and deallocation of the resources that these require. As we discuss in Chapter 3, thread pools play a fundamental role in the design of our scheduler.

2.4. Data Structures

Data structures are specific ways of organizing data such that it can be used efficiently, according to the application requirements. There are many data structures, some being simple and others being more complex. In this section we discuss about a very common data structure, the queue, as well as different derivations from it, which are used in many data processing systems, including the one we implement.

2.4.1. Queues

A queue is a collection of objects that are stored in a sequence. The end of the sequence is also referred to as the tail of the queue and the start of the sequence is known as the head of the queue. There are three common operations in queues, the addition of an object which adds the object in the tail of the queue, the peek of the queue which returns head of the queue and the removal of an object which removes the head from the queue. Queues follow the first in First in First Out methodology, which means that the data that is stored first will also be accessed first. A queue design can be seen in Figure 2.2.

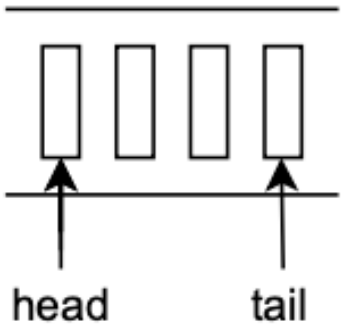


Figure 2.2: Design of a simple queue

2.4.2. Priority Queues

A priority queue [23] is a data structure that maintains a set of elements, each associated with a value called a key which is the element's priority. The priority queue can be either a max priority queue or a min priority queue. In the former case, the element with higher priority is dequeued before the element with lower priority, while in the latter case the opposite event takes place. If two elements have the same priority, they are dequeued according to their order in the queue. Our system is using this data structure in order to prioritize the tasks according to the metrics as we discuss in section 3.2.3.

2.4.3. Ring Buffers

A ring buffer, also known as circular buffer or circular queue is a fixed-size buffer which is connected end-to-end. A ring buffer consists of a queue and two pointers, one representing the head and one representing the tail. Whenever the head pointers gets to the end of the fixed-sized buffer, it wraps around to the first element of the array. Data is allowed to be overwritten in a ring buffer, so every time the head pointer wraps around to the first element, each addition is essentially overwriting the previously added elements. This is a very suitable data structure for our system, as we handle large amounts of data and we would like to re-use memory of already processed data. In addition, two different operators require access on the same data structure at the same time, attributing the ring buffer a very good choice because of the two different pointers it contains, the head and the tail. A

producer can continuously write on the memory targeted by the head pointer while the consumer can be reading from the memory targeted by the tail pointer at the same time.

2.5. Data Processing Systems Components

Data processing systems are emerging because of the need to process large amounts of data in many different sectors. There are multiple components that compose a data processing system, which also play a role in the performance and efficiency of the system. In this section we discuss some of the most common components that we pay more attention to due to their importance.

2.5.1. Processing

There are two main distinct models of processing data, batch processing and stream processing [24]. Under the batch processing model, some collection of data takes place over a period of time before it is fed into the system for processing. Stream processing refers to the process of data as soon as it enters the system, where the processing takes place in real time.

- **Batch processing**

  There are multiple applications that fit to the batch processing model because of their nature. Some examples include email systems that gather emails in batches before sending them all to their destinations. By doing this, the senders can delete or edit the emails after they have chosen to send them, if the batch has not been dispatched yet. Other examples include payroll and billing systems, processing of input and output in operating systems and more. The response is provided by the system after a job completion. Apache Spark is an example of a batch processing system.

- **Stream processing**

  Stream processing is targeted mainly by applications that are more interactive. This includes applications that require quick responses to events such as high-frequency trading companies, the stock market, social media, e-commerce transactions and others. In this case, the response is provided by the system immediately. Apache Flink and our system are stream processing systems.

2.5.2. Operators

In data processing systems there are multiple processes that are responsible for filtering the data that is being passed through the system, according to the requirement of the application. These processes, also known as operators, receive data as input, process it and produce an output. One operator's output can be fed to another operator's input, also known as downstream operator, with this process repeating itself until the data hits the last operator, which will produce the final output. The first operators in the hierarchy are called source operators and they are responsible for receiving input from different resources like the disk, the memory or a stream.

A very common example of such an operator is the map operation. Map takes a function which is called for each value in the input and produces an output which is sent to the output. Other operator examples are the ones that implement split, filter, flat and flatmap operations.

Anyone can design a data processing application by using these operators and provide the desirable input or functions to be called on each one of them, in a sequence that is related to the application's needs. For our system application, we had to create several operators that communicate with each other and process the data as discussed in Section 3.3.

2.5.3. Scheduler

The scheduler is one of the most important components in every data processing system, because it's decisions have impact on many vital system's metrics such as the throughput, resiliency, energy-efficiency. Efficient scheduling implementation is a difficult task to achieve in such systems, because of the non-deterministic nature of the problem that it has to solve. An optimal job scheduling is impacted by a lot of factors such as how much time a task will need to finish, the current state and the availability of the operator that the task is going to be assigned, the relative importance of the different tasks that it has to schedule and many other factors such as statistical outliers, noise and so forth. In this

regard, developing and tuning the scheduler for each different workload is infeasible, which is the reason why current schedulers use simple and generalized heuristics in order to make decisions [25]. The scheduler is one of our main contributions in this research. We propose that by exposing several system metrics to the scheduler, it can take decisions that will eventually allow the system to reach a peak equilibrium state and enhance it's performance and resiliency.

2.6. Programming Models

Data processing systems need to acquire, analyze and output huge amounts of data in short amount of time. Thus, in most cases, the systems dispatch multiple threads and assign a part of the workload to each one of them. Nowadays, most data processing systems use the operating system threaded architecture 2.6.1. Nevertheless, the event driven architecture 2.6.2 seems very promising for data processing systems.

2.6.1. Threaded Model

In this model, the data processing system relies on the operating system's threaded model. In that regard, the system creates the desired amount of threads, each one responsible for executing a specific task, leaving the rest of the work for the operating system. By doing so, the CPU concurrency is exploited, since each thread can run on a different CPU. However, the operating system is responsible for the thread scheduling, leaving no space for application-level scheduling optimizations. This model has some performance implications for high performance applications since context-switching is a big overhead. In addition, large amount of memory is required by the thread stacks, since every thread requires it's own stack used to store the thread's runtime data. There are more limitations for using threads in general, including the difficulty of writing thread-safe code, the dangers of deadlocks as well as the performance costs of synchronized code, depicted by Amdahl's law, however these concerns do not trouble the data processing systems which just assign an amount of work to each thread, without the need of synchronization.

2.6.2. Event Driven Model

Another existing model of parallelism is the event driven model. In this case, there is an event loop which handles incoming events. These events can be handled synchronously or asynchronously, depending on the system's capabilities and requirements. In the synchronous case, a time consuming event would cause a delay before the processing of the following events. In the asynchronous case, the events can be processed asynchronously and are handled by the event loop when their execution is finished. It is quite difficult to take advantage of real CPU concurrency in this model, however an implementation of a thread-pool that handles the events in parallel would also take advantage of the multiple cores. Nevertheless, in order for this implementation to be efficient, there should be low thread contention meaning little amount of synchronization.

2.7. System Architectures

There are many architectures that are based on the two programming models mentioned in Section 2.6. In this section, we discuss the architectures of Hadoop, which was used for the descendant state-of-the art data processing systems like Spark and Flink and we analyze the architecture of an event-based system that inspired our system's design due to it's novel architectural approach.

2.7.1. Hadoop Architecture

There are three types of nodes in a Hadoop cluster, each having different responsibilities and functions to execute.

First, there is a client node which acts as an interface between the hadoop cluster and the outside network. It is providing the data specifying how it is supposed to be processed and collects the results. Following a master-slave architecture, the second type is the master node, also called NameNode, which is responsible for coordinating and controlling the storage and parallel processing of data. This node is running Yarn resource manager in order to manage the application resources across all the worker nodes. It also runs the job tracker which is used for both job resource management and scheduling as well as monitoring of the jobs. Finally, there are the worker nodes, also called DataN-

odes, that are used to store and process data according to the master node's instructions. These nodes run Yarn's NodeManagers which are responsible for monitoring resources of the virtual machine and reporting results to the ResourceManager on the master node. This design can be seen in Figure 2.3.

2.7.2. Staged Event Based Architecture

A design implementation that levarages the multiple CPU cores as well as the event-driven model is SEDA [12]. SEDA is a design built for highly concurrent internet services, however we believe that the design proposed can provide a lot of improvements and novel research questions to the data processing system architectural designs. Our system is mainly based on this architecture, as discussed in Chapter 3.

SEDA consists of several application specific event-driven stages connected by distinct queues. Each stage is made up of an incoming event queue, a thread pool as well as an event-handler. In order to keep the stages efficient and available under large fluctuating workloads, SEDA makes use of dynamic resource controllers which dynamically adjust resource allocation and scheduling of the tasks dynamically. These resource controllers are present on every stage of the system. There are two types of resource controllers :

- **Thread pool controller** Thread pool controller, as shown in Figure 2.4, is responsible for altering the amount of threads executing withing the stage. The thread controller observes the input event queue and assigns a thread whenever the input queue is exceeding a certain threshold, limited by a maximum thread threshold per stage in order to not overallocate threads in a stage. Threads are removed whenever they remain idle for some time in the stage.

- **Batching controller** Batching controller, depicted in Figure 2.5, is responsible for adjusting the number of events that will be processed each time by the event handler. Due to cache locality and task aggregation, processing many events at once increases throughput [26]. Nevertheless, large batches lead to increased response times. The batch controller tries to make a trade-off between these two aspects, by attempting to find the lowest batching factor that leads to high throughput. In order to achieve that, it observes the output rate of events from the stage and adjusts the batch factor accordingly. First it tries to decrease the batching factor until the output rate decreases and if this decrement is little, then the batching factor is increased by a small amount.

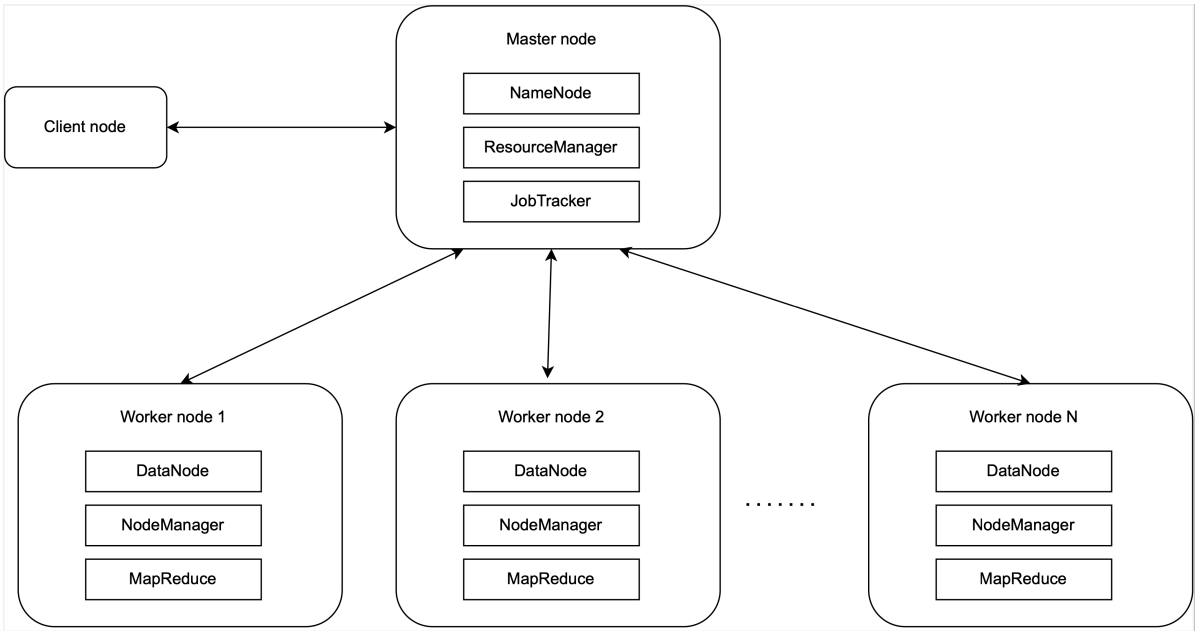


Figure 2.3: Hadoop's Architecture

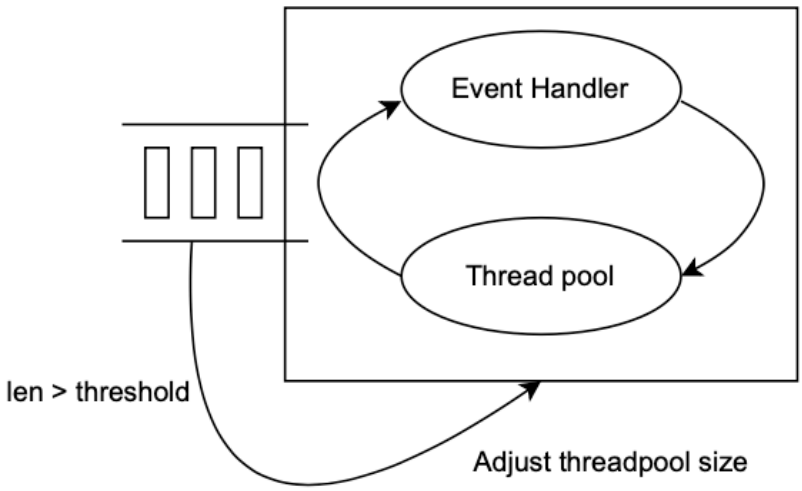


Figure 2.4: Stage's thread controller

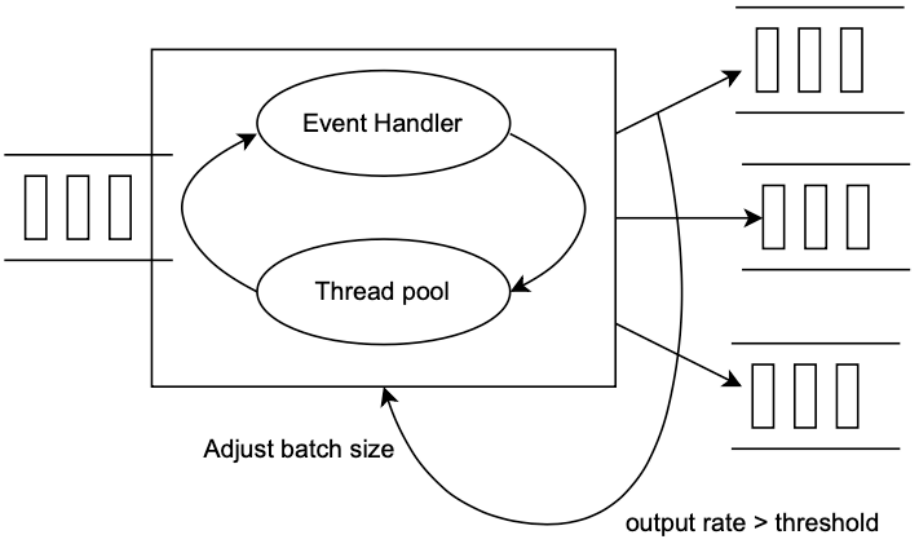


Figure 2.5: Stage's batching controller

2.8. Related Data Processing Systems Overview

There are multiple data processing systems that have been developed throughout the years. Each one has it's own design characteristics that distinguish it from the others. One has to choose the system that suits more into the functional and non-functional requirements of the application. Nevertheless, apart from the design choices, current state-of-the art data processing systems use the operating system threaded model, which poses limitations as discussed in Section 2.6.1.

2.8.1. Hadoop

In traditional data processing systems the entire data would be processed on a single machine having a single processor. However, this method became inefficient and infeasible as the data started growing both in size and in the speed it is produced. One of the first attempts to tackle this problem is Apache Hadoop, which was introduced in 2010. Hadoop is a fault-tolerant distributed data processing system that is used to store and process large amount of data. It partitions the data in many hosts and process it in parallel, according to the application computations. Hadoop has three main components. First, the Hadoop File System (HDFS) [27] is the file system storing the data in a distributed manner. Second, a framework responsible for resource management and job scheduling named YARN [28]. The third component is MapReduce [29], a system based on YARN, which is responsible for the parallel analysis

and transformation of the data. MapReduce programs read input data from disk, map a function across this data and reduce the results storing them on disk. All these components are shown in Figure 2.3

2.8.2. Spark

Another data processing system that was developed after the introduction of Hadoop is Apache Spark [8]. Spark was developed in response of the limitations of the MapReduce paradigm. Reading and storing the results on disk were problematic for iterative and interactive workloads, due to the low I/O performance of the disk. Spark's architecture is based in the resilient distributed dataset (RDD). As the name suggests, RDDs are fault-tolerant, immutable collections of objects of any type, distributed over multiple nodes. Each dataset RDD is partitioned on different nodes in the cluster so that it can be operated in parallel. One of the fundamental performance improvemenets of the RDDs is the ability of the user to opt to persist them in memory, so that they can be reused multiple times efficiently, reducing the latency of iterative and iteractive applications by several orders of magnitude compared to Hadoop's MapReduce [30] [31].

2.8.3. Flink

Both Hadoop and Spark are designed to process data in batches. Even if it is possible to opt for streaming in spark, it does not actually process data in a streaming fashion, rather it uses micro batches to emulate streaming. Flink [6] was developed in order to fill this gap and introduce a distributed process engine for stateful computations over data streams. Thus, Flink is capable of processing data in real time, which is not possible in Apache Spark's batch processing method. There have been many comparisons between these two data processing systems showing that there is no single framework for all data types and sizes. Spark is about 1.7x faster than Flink in large graph processing, while Flink is up to 1.5x faster than Spark for batch and small graph workloads [32]. As the authors suggest, this behavior is explained by the different design choices imposed by Spark and Flink. First of all, the memory management plays a crucial role in the performance of the system. The JVM's garbage collector can become a huge overhead as more and more objects are allocated in the JVM. In particular, the authors spotted that during the experiments Flink, instead of accumulating many objects in the heap, it stores them in an exclusive memory region in order to avoid memory issues, in contrast to Spark. Some other observations include the pipelined execution of Flink compared to the staged execution of Spark, the optimizations of Flink which are handled internally compared to Spark which has to be manually optimized to specific datasets and finally the parameter configuration which is very tedious in Spark.

2.8.4. Phoenix++

Phoenix++ [33] is a shared-memory Map-Reduce framework, which was implemented as an improvement of the Phoenix [34] framework.

The authors suggest that Phoenix had a couple of limitations, such as being ineliciotus to adapt to a wide range of MapReduce applications because of it's poor task overhead, as well as for it's performance issues due to the data structure choice used to store the intermediate key-value pairs and due to an inefficient combiner's implementation. A combiner acts as an optional localized reducer and processed data from the Mapper before passing it to the Reducer. In particular, Phoenix is using a static MapReduce pipeline where intermediate key-value pairs are stored in a hash-table of predefined size, a design that is limiting the performance in workloads with large number of keys. In addition, the combiner execution takes place at the end of every map stage, reducing the total overhead of invoking the combiner but costing a lot in terms of memory allocation and increasing the possibility of cache misses.

The authors propose that by running the combiner more frequently, performance will improve because of reduced memory allocation pressure. Therefore, they propose Phoenix++, a framework that is modular and allows users to adapt performance critical elements of the pipeline, according to the workload they are working with. This is achieved by exposing the key-value grouping and storage and by allowing the users to disable the final key-value sort and by allowing them to replace the default memory allocator.

2.9. Main goals

The main goal of our research is to address the gap in current state-of-the-art systems by enabling developers to build effective schedulers that are based on the system's state. We believe that by utilizing metrics such as the operator's throughput and the amount of work each operator has to process at a certain point in time, we can make intelligent scheduling decisions that will not only enhance the system's performance, but also improve its resiliency. By allowing the system to self-adapt according to the state of its processors, we propose that it can reach an equilibrium with an optimized performance and resiliency.

In addition to this, we believe that the utilization of thread pools can complement such a system. By allowing the scheduler to dynamically adjust the size of the thread pool based on the needs of the applications, we believe that we can further enhance the system's performance and resiliency.

Furthermore, we plan to implement processes that communicate via shared memory using shared-memory thread-safe ring buffers. This, to the best of our knowledge, has not been done in similar work in the field of data processing systems, and we believe that it will provide additional benefits in terms of performance and resiliency. By allowing processes to efficiently communicate and share data using shared memory, with a design that minimizes the need for locking mechanisms, we believe that we can significantly improve the overall capabilities of the system.

3
Design

In this chapter we analyze all the design choices and the implementation of our work. We start by discussing the system design in Section 3.1 which includes the data structures used as well as the higher level architecture of the system. We continue in Section 3.2 where we examine two different scheduler implementations, a simple round robin scheduler and a more advanced priority scheduler. We then continue with the application we designed on top of our system in Section 3.3. We finalize the chapter in Section 3.5 where we briefly summarize the chapter and note the most important takeaways from our work.

3.1. System Design

This section contains an overview of the design of the system we implement. The code of our system, including the tests can be found at [35]. We start by explain the data structures, the different operators and how these communicate with each other in order to process the tasks. We then analyze some of the challenges that we faced and how we overcome them.

As we have discussed, the main proposition of our research is the implementation and evaluation of a high-performance data processing system which exposes decisive metrics to the application level, in order to allow for the design of more efficient schedulers, which are able to use the system's metrics in order to take decisions and reach a peak equilibrium state. To our knowledge, similar systems that expose metrics such as the operator's throughput, the work that is queued for the operator and others, have neither been developed nor researched. We believe that such systems can result in the creation of schedulers that will allow the system to adapt and react to changes during run-time, making stream-processing more efficient and performant. By taking into account the run-time metrics, we propose that a scheduler, after a start-up phase will reach an (peak) equilibrium until it stops. In addition, our system requires a very low amount of fine-tuning parameters, which can be cumbersome and require a lot of work in other systems like Apache Spark and Apache Flink.

In order to achieve the aforementioned results, we require an efficient data structure that multiple threads can use in order to communicate with each other. We opted for shared memory thread-safe ring buffers as we believe is the most efficient way to achieve the fastest communication between processes when multiple threads are involved. Similar work has been done in close domains like data acquisition [36], but to our knowledge, no research has been concluded for data-processing systems implementing multi-threaded ring buffers as a communication method. In particular, each operator of our system contains one input ring buffer and one output ring buffer. The goal is for it to read from the input ring buffer as many items as the scheduler suggests, whenever the scheduler decides it's time to activate this operator. It then processes these elements and writes the results to it's output ring buffer, which is the input ring buffer of the downstream operator. By doing so, we create a graph of processes that communicate with each other over this shared memory as shown in Figure 3.2.

3.1.1. Shared Thread-Safe Ring Buffers

For our system to be efficient we had to implement a data structure that would work and scale under heavy workloads. For that we implemented a shared thread-safe ring buffer which is the main compo-

nent of our system and is being used by all the operators for the sending and retrieval of it's tasks. The main reasoning behind this choice was to design a data structure that can be accessed by different threads at the same time without the need of locking mechanisms, which would work as a communication mean between the processes. This design allows us to efficiently control the resources (threads) of each process, while matching the requirement of our system to expose some important metrics such as the workload of each process to the scheduler.

Applications in Apache Spark and Apache Flink, on the other hand, are parallelized into multiple tasks that are distributed and concurrently executed in a cluster. However, this design does not allow for dynamic parallelization and the burden of optimizing the parallelization is assigned to the developer and is dependent on different factors such as the application and the workload.

In order to achieve this, we implement a ring buffer of a fixed capacity. For achieving parallelism, we have to split the ring buffer in a way that multiple threads can simultaneously read and write data on it. If a thread could only read or write to one slot of the ring buffer, the system would not scale with the number of threads increasing. Thus, we introduce the notion of slices. The slices are subqueues that are logical partitions of the ring buffer.

There are two different types of slices, the read slices and the write slices. With the help of queue pointers and atomic operations, multiple variable sized read, write or both slices can be requested simultaneously. In the case of a request of a read slice of size n, a slice of size $max(n, readable\_amount)$ will be returned, where $readable\_amount$ is the total amount of items that have not been yet read in the ring buffer. From this point, the requester is the sole owner of this read slice and can start reading items from it. In the case of a write slice request of size n, the request will either succeed or fail depending on the state of the ring buffer. If there is enough space in the ring buffer for the reservation request, a writable slice of size n will be returned, otherwise the option $None$ will be returned. Again, the requester will be the sole owner of this writable slice and it can start writing it. In both read and write slices, after having completed the desired work, the requester should free the slice. By freeing the slice, again using atomic operations, the queue pointers will be updated into their correct positions so that the next requester has the correct view of the ring buffer's state.

As aforementioned, for the implementation of the read and write slices, we used queue pointers. The queue pointers are the following : $head$, $shadow\_head$, $tail$, $shadow\_tail$. In simple queue implementations, only $head$ and $tail$ pointers are enough in order to keep track of the state of the queue. However, in our implementation we require the $shadow\_head$ and $shadow\_tail$ pointers for keeping track the state of the slices as well. In particular, the space between $head$ and $shadow\_head$ refers to the space that has already been requested and has not been freed yet. Whenever a requester successfully receives a read slice of size n, the $shadow\_head$ pointer will progress n steps. The $head$ pointer proceeds whenever a read slice is being freed. Our system respects the order of the requests, so if a read slice is being freed that does not point to the same place as the $head$ pointer, the $head$ pointer will not proceed. However, when all the slices that are before this one are freed, the space for this slice will also be freed in an accumulated manner. The steps that the $head$ pointer proceeds is equal to the accumulated size of the continuous read slices that have been freed. Similarly, the area between $tail$ and $shadow\_tail$ relates to the space that has been reserved for writing by all the write slices. Requesting and freeing write slices will proceed the $shadow\_tail$ and $tail$ pointers in similar fashion to the $shadow\_head$ and $head$ pointers, by taking into account the order of requests. An example of requesting and freeing a read slice from the ring-buffer can be seen in Figure 3.1.

3.1.2. Operators (Processes)

Operators (also named processes) contain the logic of the operations we want to perform on the data. Each operator is a struct that takes some input, processes it and produces some output. Operators require a way to communicate in order to send the data to the downstream operators. As aforementioned, the most efficient way we think is to achieve this is by the use of the shared thread-safe ring buffers. Each operator is composed by distinct ring buffers, some that are responsible for gathering the input data and some that are responsible for gathering the output data. Consequently, an operator's output ring buffers are the same as it's downstream operator's input ring buffers. In this way, operators can request read slices from their input ring buffers and process the data contained on it according to the operation they are performing, before freeing the read slice. Consequently, they can request for writable slices from the downstream operator and write the results on it. After having finished writing the results, they can also free the write slices in which point they have completed their job. In our

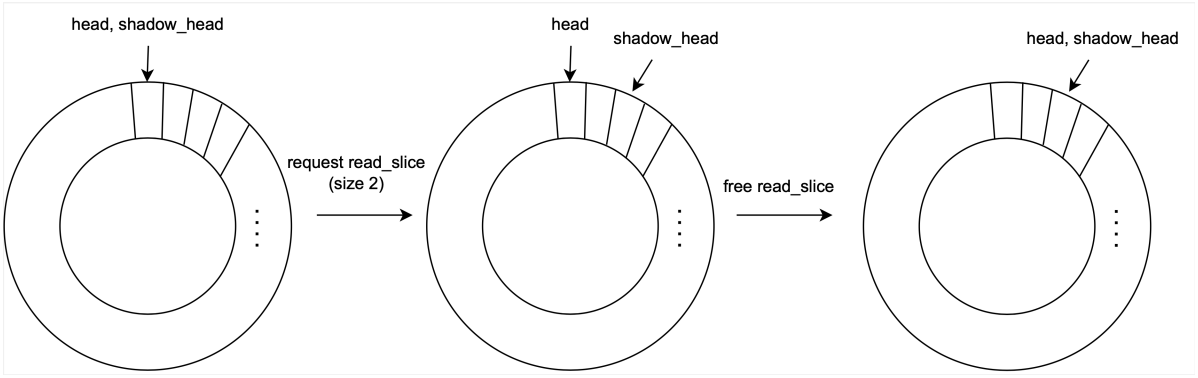


Figure 3.1: Design of a simple shared-memory thread-safe ring buffer

experimental application, we used one input ring buffer and one output ring buffer for every process, therefore we will be mentioning this design for the rest of the thesis. In Figure 3.2 we can see an example of an application consisted of the source and sink operators as well as four other operators. In this example, we can see one input and one output ring buffer for each operator (which is also the minimum amount required for the communication between the processes). It is illustrated how a ring buffer is used both by the upstream and the downstream operator, being the output queue for the former and the input queue for the latter.

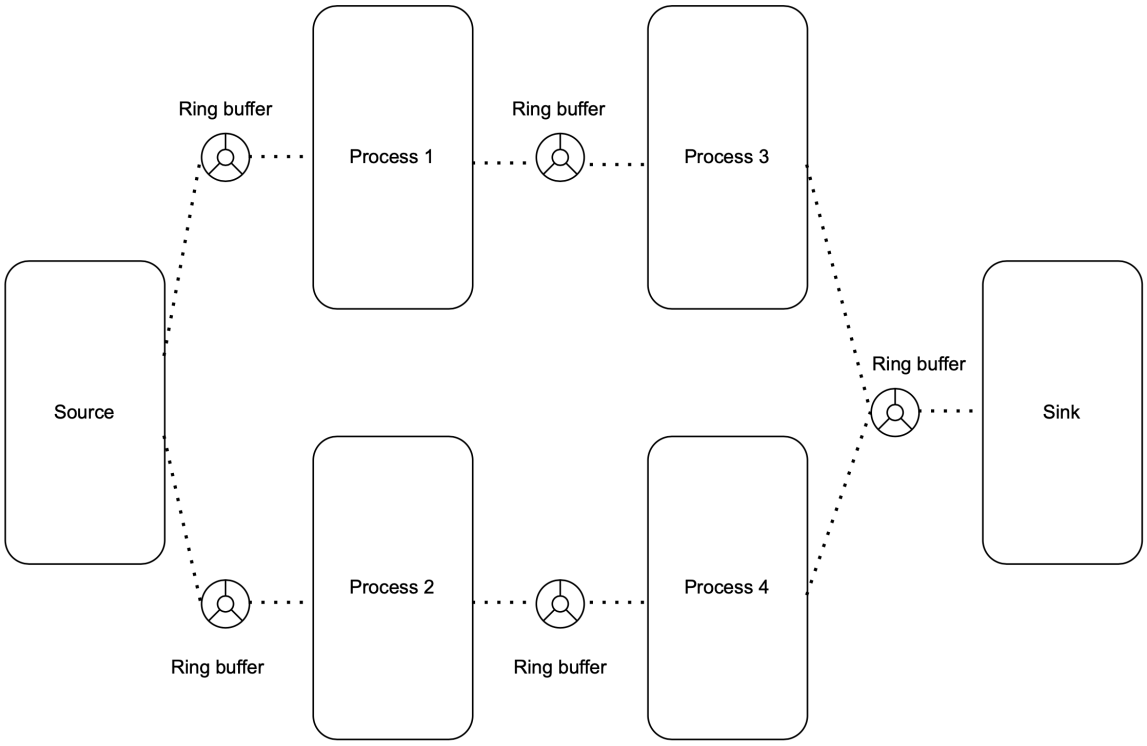


Figure 3.2: Example of an application design consisted of six processes

3.1.3. Tests

In order to check the stability and correctness of our system, we implemented several tests. First we created a test for testing the writable slices only, in which we spawn five threads, each reserving a writable slice of 250 size. It then increments a counter that is shared among all the threads and appends the result to it's write slice. Finally, after all the threads have finished their job we are requesting a read slice of the accumulated write slice sizes and check if the correct numbers are inside it.

Another test we implemented is a producer-consumer scheme. In this scenario, the producers are adding items in the ring buffer while the consumers are constantly reading these items. For this case, we are not testing the slices, we rather add only one item per time in the ring buffer we just wanted to let the program run for long time so we implemented a handler to exit and print some metrics concerning the performance of the system. We create another similar test case for checking if the correct values and ensuring that there are no duplicate values are inside the ring buffers.

In the next scenario, we enhance the previous test case for using slices. Now producers and consumers reserve a slice of a specific size. After a specific amount of items have been read, the metrics like the producers time, the consumers time and the total time are printed. We also create another very similar test case which is used for testing if the correct values are inside the ring buffers in the end and ensuring that there are no duplicate values.

Finally, we implement the most generic test case which would be closer to a real-world application scenario. In this test case we do not use the producer-consumer scheme, so there is no notification between the producers and the consumers with the use of semaphores. Here, we spawn a certain amount of producers and consumers threads which run repeatedly, until their required amount of work is finished. Thus, the producers are consistently requesting write slices, fill them up and commit them while the consumers repeatedly request read slices, read them and commit them. By doing this, apart from the performance measurements of the core data structure, we are also testing if the ring buffer data structure does not return slices that are already in use and that the system works as expected.

3.2. Scheduler

The Scheduler is one of the main components of our system. The scheduler is responsible for assigning work to each operator and by efficiently doing so reaching a peak equilibrium state. The main goal of our scheduler is to utilize the system's metrics such as the recent selectivity (throughput) of each processor as well as the amount of work that is queued to each processor. We propose that by the utilization of these (and other) metrics, we can create a system that can reach a peak equilibrium by only configuring a small amount of parameters beforehand. Such parameters include the desirable size of each process's queue, the total number of threads and the desired input rate of the sink process's input queue. We expect such a scheduler to reach a peak equilibrium state and to outperform current state of the art schedulers in terms of run-time. We also expect it to adapt efficiently and fast to data input fluctuation (e.g. vastly changing input that would require more processing on other operators than before) and to disruptions such as unexpected changes to the thread pool size.

Current scheduler implementations do not rely on such metrics, as they have not been designed in a way that would allow them to do so. Scheduler implementations like Apache Spark's split the jobs into stages which utilize all the underlying resources until the job is done. Therefore, a process will be activated irregardless of the amount of work it has to do, the recent throughput that it achieves and other performance relevant metrics.

We started by implementing a simple scheduler that schedules the jobs in a round robin fashion. Later, we constructed a process priority scheduler which takes into account the different processes' priorities, which takes into account the metrics that we are interested in, in order to make a scheduling decision.

3.2.1. Thread pool

As aforementioned, having the threads in user-space provides a lot of benefits, including the removal of context switches, the removal of space stored by the kernel's thread stacks as well as the allowance of implementing an application level scheduler. In all our scheduler implementations we are using a thread pool that spawns a constant number of threads. Each thread lives forever and is constantly getting the next process to be executed, which removes the need of context switches. The priority that the processes will be picked is decided upon the scheduler implementation.

3.2.2. Round Robin Scheduler

Our first approach of a scheduler implementation is a round robin scheduler. In this case, the operators/ processes are stored in a simple array. Upon the start of the scheduler, the thread pool will spawn a predefined specific amount of threads. Each thread will enter an infinite loop, in which it repeatedly picks the processes one by one from the array in a circular manner, since after the last process the

thread will start picking processes starting from the first position of the array again. The amount of work that each operator will undertake on each activation is determined by a pre-defined static variable that we manually tune.

The round robin scheduler does not take into account any metrics that would benefit the system in terms of either performance or availability. For example, in the case that an operator does not have any items to process in it's input queue, CPU cycles would be wasted by constantly checking the activation amount of this process. In addition, even if a process has a low amount of items in it's input queue, the whole system would benefit more by activating another process with more items and activating this process later when it also has gathered more items in it's input queue. Another devastating effect that the round robing scheduler can have that can, in the worst case, block the whole system is the case where the downstream operator's input queue is full. In the good scenario, the current thread will spin-lock until another thread picks the downstream operator and process some of it's input elements. However, in the worst case scenario, all the threads can get stuck at the current thread, waiting ineffectively for another thread to activate the downstream operator. We started tackling this problem by introducing static weights. Each process has different processing times and workloads, so it makes sense if every operator processes a different amount of items in each activation. This greatly improved the performance and the availability of the system. However, in a real world scenario where the loads can change dramatically, the same problems will appear. For tackling this, we need a scheduler that takes decisions based on dynamic metrics which change throughout the system's lifetime.

Our first implementation of the round robin scheduler was using the same fixed pre-determined variable for both read and write slices. Therefore, each operator is activated with this variable each time. However, by doing so, we are wasting memory space and CPU cycles. This happens because in some applications like the one that we are building, the size of the output a process will produce might not be known beforehand. For example, the split string process's output is correlated to the input text. Therefore, it is not possible to reserve the exact sized slice amount required and update it while splitting the lines. In order to tackle this problem, we introduced $selectivity$. This variable is initialized to a default value (for every process) and it keeps track of the average output size of each process dynamically in run-time. Then, for each process that the output size is unknown (so every process except for the split string and filter processes), we are reserving a write slice of $selectivity$ size. If the slice gets full, we commit it and we are reserving another slice of the same size, until the required work is completed.

3.2.3. Process Priority Scheduler

The second scheduler that we implemented is the process priority scheduler. Here, the processes are stored inside a shared priority queue that prioritizes them based on the following equation :

$$readable\_amount_i * (target_i + i)$$

As we have previously mentioned, $readable\_amount_i$ is the total amount that can and has not yet been read from the input queue of the i^{th} process. In the special case of the source operator (where i is equal to 0), the $readable\_amount$ refers to the items that have yet to be read from the output source that provides them, since the source operator does not have an input queue. The equation introduces another new variable, $target_i$, which refers to how many items the i^{th} process requires in order for the last process to have a certain percentage of it's queue full. In order to calculate $target$ for each process, we firstly set the $last\_queue\_limit$, an integer that is correlated to how many items we want to have in the last queue. Then, every $500\ ms$, the $target$ of each process is calculated, by starting from the last process and calculating how many items are missing in order to reach the $last\_queue\_limit$. Recursively, we calculate every process's $target$ by determining how many items are missing from the downstream process's input queue. Initially, for the first $500\ ms$, we use a pre-defined static target that we manually tune. In that way, every $500\ ms$, the query plan changes which also affects the scheduling decisions for the next $500\ ms$. We chose the value of $500\ ms$ since it produced the best results for our application. By choosing low times for this metric, the accuracy of the target values will be higher since the target values will be updated more frequently and inaccurate values will also be used for less amount of time. However, more frequent updates lead to more time that the shared priority queue will be locked. On the other hand, by choosing high times, the lock times will be lower however the accuracy of the targets will also be lower.

We can see that this equation first prioritizes the processes that have simultaneously many items to read and their target is high (which means that their downstream processes also require a lot of items on their input queues in order to fulfill the $last\_queue\_limit$). Then, next in priority are the processes that either do not have many items to read or they have a low target, followed by the processes with low amount of items and low target. This prioritizing function is also convenient because if a process does not have any items to read, or the downstream process does not require any items, the priority of it will be 0.

We found that addition of the process's index in the equation is important because in the case that processes have the same priorities, we want to prioritize the last operators in order to avoid blocking the system. This blocking can take place in the situation where all operators have items to read, but also every operator has a target of 0, meaning that it does not require any more items on it's input queue. In that case, all operators would have a priority of 0 and if we do not prioritize the last operator before the next to the last and so on, the system could block since the input queues could get full. However, with the addition of the operator's index, we are ensuring that in that case the latest operators have higher priority.

In the process priority scheduler, the reserving of write slices works in the same way as the latest implementation of the round robin scheduler. Thererefore, $selectivity$, which keeps track of the average write slice sizes for each operator, is being used in order to approximate the write slice size that will be required for the split string and filter processes (since these are the two processes with unknown output sizes).

3.3. Application Design

In order to test our system, we did build an application that has a real-world scenario that we can benchmark in all the systems we want to compare, including our system, Spark and Flink. We built a log parser that parses a big text file and counts all the words that start with a specific character (e.g. all the hashtags contained in the file). We started with a simple design and did multiple iterations in order to improve it while comparing it with Spark and Flink. In the current section, we will describe the design of the final output, however we will also describe the challenges that we faced building the application and the iterations we did before reaching the final result, as described in section 3.4.

3.3.1. Processes

In order to achieve the desired results we build four distinct processes. We followed this approach, because in the end we want to expose each process's metrics to the scheduler, which will take decisions on to which process it will assign work next and how much work it will assign to it. Therefore, by specifying multiple processes, each having a single particular job to do, we allow the scheduler to have more possibilities for it's task assignment and is able to make a decision from a wider pool of choices. The processes we created for our application are the following:

A source process responsible for reading the text file and writing every line to the downstream process. The next process is processing these lines, splits them into words and writes them to it's downstream process, which is filtering all the words starting with the character a and writes them to it's downstream process, which is also the sink process. The sink process is increasing a counter for every word that come into it's input queue.

- **File Reader**

  File reader is the source process of our application. As a source process, it contains only an output ring buffer. File reader is in charge of reading in parallel a text file and writing each line to it's output queue. For achieving this, we experimented with two different ways of reading in parallel. In one, we are splitting the text file into smaller text files and provide an array of read buffers to the process. In the second way, we are providing the single big text file along with an integer $partitions$ which specifies how many partitions we want. Consequently, the process creates an array containing $partitions$ amount of read buffers. The array containing the read buffers is shared among the different threads that are activating the process. By doing so, each thread can get and remove a unique read buffer pointing to a distinctive place in the text file, read from it and return it to the shared array. The initialization of the read buffer array is taking place at the constructor of the struct. A thread activates this process by providing how many bytes it

wants to read from the read buffer as a parameter. Then, the process removes a read buffer from the shared array, reads from it as many bytes as specified by the caller and writes these bytes line by line to the output ring buffer, so that every entry in the output ring buffer is the bytes corresponding to a line. A more detailed explanation on why we chose to provide bytes instead of strings lies in Section 3.4.

- **Split String**

  Split string is the second process of the data processing system application we built. It contains an input ring buffer, which is the same as the output ring buffer of the file reader, as well as an output ring buffer which is the same as the input ring buffer of the next process. This process is responsible for splitting the string into words. As aforementioned, the input ring buffer contains the lines of the text file in bytes. After multiple iterations that we discuss in Section 3.4, we ended up with providing a tuple containing an array of the bytes of the line as the first entry and a two dimensional array as the second entry. The two dimensional array contains the index of the start of the word at the first entry and the index of the end of the word as the second entry. In this way, the next process can simply check every word by iterating the start and end indexes in the line provided at each entry of it's ring buffer. The thread that activates this process specifies how many entries from the input ring buffer it wants to process.

- **Filter**

  Filter process contains one input queue which is the same as the split string's output queue and one output queue that is the same as the sink's input queue. This process is responsible for filtering out the words that start with a specific character and clones these words to it's output queue. The words in the output queue are stored in the same format as in it's input queue, so each entry is a tuple with an array of bytes as the first entry and a two dimensional array containing the indexes of the start and the end of the word respectively. Like all other processes, the caller should specify the number of the input ring buffer's entries that it wants to process at each call.

- **Output**

  This is the sink process and it contains the simplest logic of all the processes. It's job is to count the words that are being forwarded to it's queue.

  We can see that every process has one and only parameter that is set by the caller, what we call $batch\_size$. This parameter is very important as it is responsible for how many CPU cycles a process will occupy, as well as how full the ring buffers of the processes will be. This is the parameter that the scheduler is making use of in order to utilize the $target$ of each process and eventually achieve high throughput and availability.

  In Figure 3.3 we can see the layout of the processes and how their ring buffers are shared between them. The first ring buffer, which lays between the file reader and the split string processes, contains an array of bytes for each line of the source file. So, according to the ASCII table, the first entry of bytes would correspond to a line starting with the characters "#Ae" and the second entry corresponds to a line starting with the characters "CIV". Then, split string process will split the bytes as words and put them in it's output ring buffer in the format we have already mentioned. In this case the ring buffer contains one of the words which is "#Ae" (the first entry) as well as one other word from the same line. It also contains a word from the next line, which start from index 6 and ends at index 11. Finally, the last ring buffer contains the filtered words, so for this example the words that start with a hashtag (#). The first entry of the final ring buffer contains the word "#Ae". The sink process will take each entry of this ring buffer and write it to the terminal. As mentioned in the System Design 3.1, multiple threads can request read slices and write slices from the ring buffers in order to achieve parallelism. Each of these slices will be reserved and freed until the thread finishes it's job, so it is free to use it without any locking mechanisms.

3.4. Challenges

There were a lot of challenges we faced both in the system design and the application design. We start by briefly describing the challenges we faced and how we resolved them in 3.4.1. We then proceed

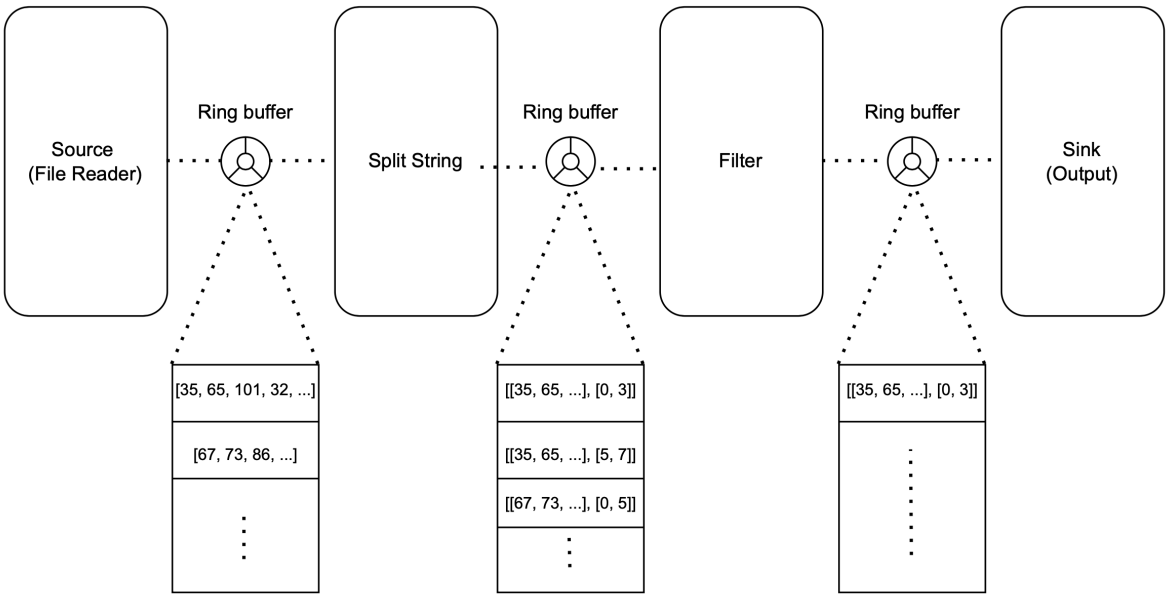


Figure 3.3: Hashtag Parser application design

with presenting more analysis and some experiments comparing the runtime of our system during the different main iterations we did before reaching our final optimized application and system in 3.4.2.

3.4.1. Summary of challenges

- **Atomic Operations**

  The concurrent nature of the design presented a number of challenges, particularly with regards to the atomic operations that needed to be performed. One of the most significant challenges was ensuring that the order of these atomic operations was carefully considered, as a change in the order could result in bugs or unexpected behavior during certain runs.

  In particular, we chose to design a system that allows the order to be strictly preserved. Even if this is not a requirement for the application that we test, and even if this is a more complex design than just keeping a bitset that marks the slices that have been committed, we decided to implement this feature because it is a common need in streaming big data applications. Some examples include processing time-series data since the order of the elements is crucial to preserve time-dependent calculations. Another example is event processing or financial transactions, where events are processed in the events in which are occurred and the dependencies between them should be conserved. Apache Flink respects the order by default and Apache Spark has operations that can preserve the order of the data as well.

  An example of this challenge was observed when committing write slices. In order to fully understand this scenario, it is important to first explain the concept of a ”semi-commit.” A ”semi-commit” occurs when a write slice that is after the next expected write slice to be committed is committed. In this case, the slice is marked as committed, but is not actually committed until the previous write slices have also been committed in the correct order. Therefore, this system design allows for applications to be built on top that can preserve the order of the incoming events.

  The scenario in question occurs when a write slice is being committed. At this point, the system checks how many write slices after this slice have been semi-committed, in order to commit them all at once. However, before the system can actually commit these slices, another thread may commit the next expected slice instead of semi-committing it. This can happen because the slice that is being committed is first changing a pointer that points to the next expected slice to be committed via atomic operations, before actually committing all the slices. This would violate the order rule for committing and could also cause the system to become stuck in the event that the slices before the wrongly committed slice have not been actually committed.

To address this challenge, we engaged in a rigorous process of testing and experimentation to produce a stable data structure that, according to our tests, is free of bugs. This involved carefully analyzing the order of atomic operations and making necessary adjustments to ensure the correct order was being followed.

- **Variable Slice Size**

  Another significant challenge we encountered during the development of our application was the issue of variable slice sizes. Initially, we had implemented the system with static slice sizes, meaning that a thread could only request a specific, pre-determined size for write and read slices. This approach had several advantages, such as ease of committing slices since all slices had the same number and the $read$ and $shadow\_read$ pointers, as well as the $write$ and $shadow\_write$ pointers, moved at a similar pace during each slice commit. It also provided a convenient baseline for our experiments, as we knew exactly what slice size a process would require from the next process when input workloads were known.

  However, despite its conveniences, this approach was highly limiting for general applications, and also imposed limitations on performance. For instance, it made it impossible to implement a scheduler, as we wanted to implement it, since it was heavily based on requesting different amounts of slices according to the amount of work to be done, and take into account the system's throughput. Additionally, in the case of static slice sizes, some processes may always be reserving more space than necessary, resulting in wasted space.

  To overcome this challenge, we had to redesign our approach and implement a system where the slices could be of variable sizes. This required a significant amount of work and experimentation to determine the optimal way to handle variable slice sizes and how to incorporate this into our scheduler. We had to consider how to handle the different read and write pointers, how to manage the committing of slices, and how to ensure that the system was utilizing resources efficiently. We also had to take into account the impact of variable slice sizes on the overall performance of the system.

  This approach allows for more flexibility, performance and stability for unknown and varying workloads, as the system is able to adapt and make more efficient use of resources. It also provides a more general solution that could be applied to a wider range of applications. Overall, this was a significant challenge that required a great deal of effort to overcome, but ultimately resulted in a more robust and versatile system.

- **Memory Allocation**

  As we delved deeper into the design and development of our application, it became increasingly apparent that there were a number of hurdles and obstacles that we would need to overcome in order to achieve our desired outcome. One of the most significant challenges that presented itself had to do with the management and allocation of memory.

  In our initial implementation, we had utilized strings in each ring buffer of every process. This involved reading from a file and then storing each line as a string in the split string process, which would then divide the line strings into individual words and store them in the filter process. The filter process would then filter out any words that did not begin with a desired character, and store the remaining words again as strings in the output process.

  However, as we began to implement the same approach using Spark and Flink, we quickly realized that the performance of these frameworks was significantly better than our own program. Through extensive performance measurements and observations, we were able to identify that a significant amount of runtime was being spent on memory management during string allocations and freeing of memory.

  Our program was utilizing malloc and free calls each time a string was created or replaced, whereas Spark and Flink were taking advantage of the Java Virtual Machine's memory allocator. The JVM's memory allocator is slab-alike [37], and allows for the fast creation of objects as well as efficient management of freeing up space when objects are no longer needed. Additionally, the JVM utilizes string pools [38], which save a significant amount of memory when similar strings are used.

In light of these findings, we knew that we needed to take a different approach to tackling the memory allocation issue. Initially, we experimented with the use of smartstrings [39], a library that stores strings on the stack instead of the heap for strings that are 23 bytes or less. This change resulted in a noticeable improvement in performance and a reduction in time spent on heap allocations. However, we knew that we needed to explore other options in order to achieve optimal performance.

We ultimately decided to handle tuples of arrays of bytes and two pointers depicting the start and end of words. By handling the data in this manner, we were able to utilize the heap again, but without any restrictions on the size of the strings. This also allowed us to manage the frequency of calls to the underlying operating system's memory allocator by controlling the number of bytes stored in each vector. Furthermore, we reduced the number of times malloc was called, as multiple words could be expressed by multiple pointers in the same byte array.

We stored the bytes of each line in a vector of bytes, and wrapped each data structure's input vector of bytes under a shared pointer. This way, whenever a vector was cloned to the downstream ring buffer, its pointer was also cloned. This allowed for the allocated space to be freed from the heap once all pointers were destructed. Through this extensive process of trial and error, we were able to overcome the challenges of memory allocation and ultimately achieve optimal performance for our application.

3.4.2. Iterations

In this section we present the main different approaches we followed in order to adapt the application while discovering bottlenecks and limitations throughout our evaluation and comparison with Spark and Flink.

Expectations, Workload and Variables

Our expectation is for our system to be faster than Spark and Flink for any number of threads, in any sized workload that is neither memory nor CPU bound.

We began our system design by experimenting with a weight-based approach, where we assigned different weights to various processes within the system and meticulously fine-tuned these weights to achieve optimal results. It is important to note here that these weights are a property of the prototype implementation and not inherent to the system design. As we will see in the upcoming paragraphs, the final design does not require any weight fine-tuning, as it is able to determine and adjust the size of the asked slices automatically.

In all the following experiments, the weights we chose are 1.2 for $file\_read$ process, 1.2 for the $split\_string$ process, 42 for the $apply\_regex$ process and 42 for the $output\_result$ process. Each weight is multiplied in the activation function of each process by a pre-set variable $WRITE\_SLICE\_SIZE$, which is set to 10.000 for the first implementation and 300 for the rest implementations. The slices requested are fixed and everytime, we request the same slice for each process. Since the workload for these experiments is the same line repeated, we know exactly the size that the $split\_string$ process will require from the next process, therefore we could get an optimal baseline for our experiments, after fine-tuning the weights as discussed. The $WRITE\_SLICE\_SIZE$ variable is not needed at all in the final implementation since we are dynamically changing the slice sizes according to the selectivities of the processes.

In order to calculate the performance and compare the following implementations and adaptions, we used a program of 11GB size, which contained the exact same line repeatedly, because this allows the split string process to request the exact slice size amount (optimal) from it's output queue, since we know beforehand how many words are in each line. Our benchmark application reads the entire file in parallel and calculates how many words start with $'\#'$. We applied the same application in Apache Spark and Flink in order to compare the runtimes. Between every run, we cleared the caches.

First implementation

Our first system and application design approach consisted of a variable write slice size implementation. In this design, each process may request an arbitrary sized slice (either from it's input or

from it's output ring buffer). For the first implementation, we decided to use the same size for both input and output slices. In this approach, we used a round-robin scheduler in which each thread is activating a process in a round robin fashion with the specified slice size $WRITE\_SLICE\_SIZE$ (which is pre-defined) as input. Consequently, the process is reserving a slice with this size from it's input queue, processes the data and produces the output, writing it to another slice of the same size it reserves from it's output queue. Since every process is responsible for different operations, it is expected that every process should use a different slice size. Thus, we implemented weights, where each process is multiplying it's input and output slice size with this pre-defined weight value. Therefore for this system and application design implementation, in order to find the best configuration for a specific workload, someone should fine-tune this weight parameter as well as the slice size that will be multiplied with the weight.

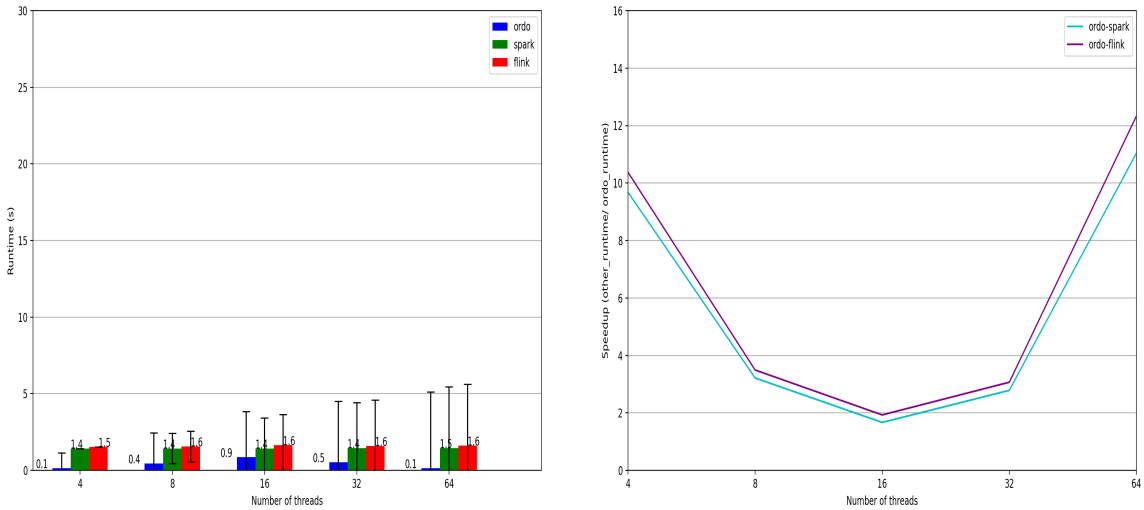
We evaluated this implementation's runtime against Spark's and Flink's runtime for the same application under the same workload. For that, we utilized the DAS-6 cluster [40], which is a high-performance computing environment composed of 20 nodes equipped with dual 16-core 2.8 GHz CPUs and 128GB of memory. At first, we used a smaller workload (12 MB) where our program met our expectations as it performed better in terms of runtime for completing the same task compared to Apache Spark and Apache Flink, as we can see in Figure 3.4. Nevertheless, we can see in Figure 3.4a that even though the average runtime of our system is significantly lower than the other systems (e.g. for 64 threads it is 93.3% lower against spark and 93.7% against Flink, the standard deviations are pretty high, so we cannot draw clear conclusions about the systems' performances. This was expected, since the workload is very small and the runtimes are less than two seconds. From the Figure 3.4b depicting the speedup ($other\_runtime/ordo\_runtime$ we can also see that our system outperforms the other systems by at least twice as much in almost all cases.

However, in order to experiment more and to draw stronger research conclusions, we started increasing the workload to 500MB as shown in Figure 3.5. In this case, we can already see from both Figures 3.5a and 3.5b that our system has started being outperformed by Spark and Flink for the lower number of threads. In particular, the speedup between Ordo and Spark for 4 threads is 0.32 and 0.41 between Ordo and Flink, indicating that Spark is moderately more than three times faster and Flink is slightly more than 2.4 faster than Ordo for 4 threads for this 500MB workload. The runtime improves when we increase the amount of threads, but again the standard deviation is pretty high and we cannot draw dependable conclusions. In any case, we expect our system to outperform Spark and Flink for lower amount of threads too. It is worth mentioning that we experience an unexpected behavior by Spark for 16, 32 and 64 threads, where the runtime is increased significantly. We noticed that at first, a single task starts by itself and it takes long time to finish in this experiment, however we did not investigate this behavior further, since it is out of scope and it did not repeat in the following experiments.

After these observations, we increased the workload substantially, reaching 11 GB. The results of this workload are shown in Figure 3.6a. We can see that our system's (Ordo) runtimes are significantly worse than Spark and Flink. The gap between our system and the state-of-the art systems is shrinking as the threads increase, from 817.5% increase for 4 threads between Ordo and Spark and 527.1% between Ordo and Flink for 4 threads to 21.5% and 56.8% increased runtime for 64 threads. These results did not meet our expectations, therefore we tried to find the bottlenecks to our system. We also plot the speedup in Figure 3.6b. We can see how the speedup improves and almost reaches the value 1 which indicates that the systems have same performance, however the implementation does not meet our requirements and requires investigation to find the bottleneck. In all the rest of the experiments in this section, we will continue using the bigger workload of 11 GB.

First adaptation

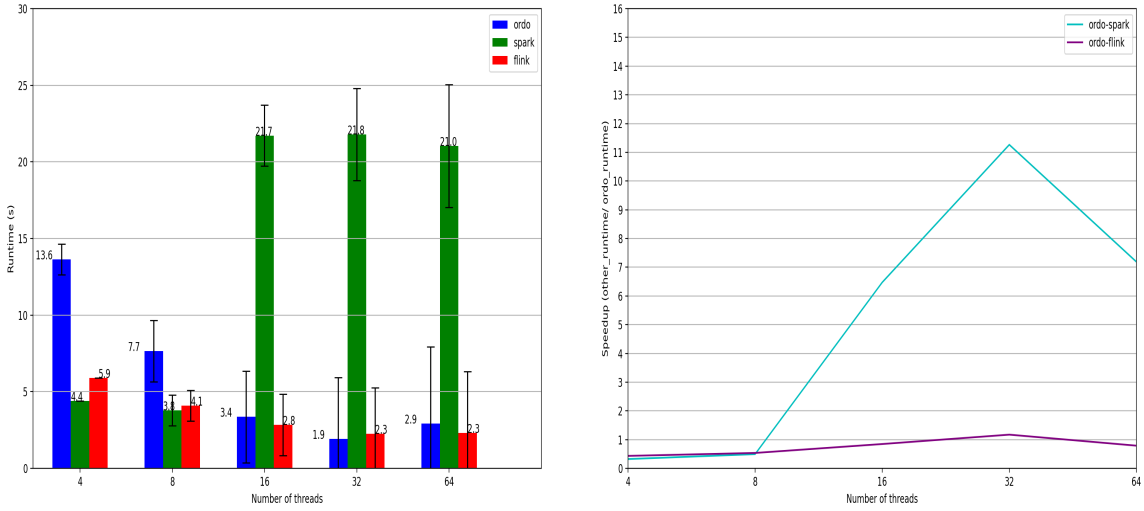
After carefully monitoring the performance of our system while running our application with the mentioned workload, we found that the reason our system took so long to perform the same task as Spark and Flink lied in the memory allocation. As described in 3.4.1, our system relied too heavily on the operating system's allocator since malloc was called too often (everytime a string was created), in contrast to Spark and Flink applications where the JVM was taking care of the



(a) a. Runtime plot of our first implementation compared to Spark and Flink (12MB workload)

(b) b. Speedup plot of our first implementation compared to Spark and Flink (12MB workload)

Figure 3.4: Runtime and speedup plots of our first implementation compared to Spark and Flink (12MB workload)

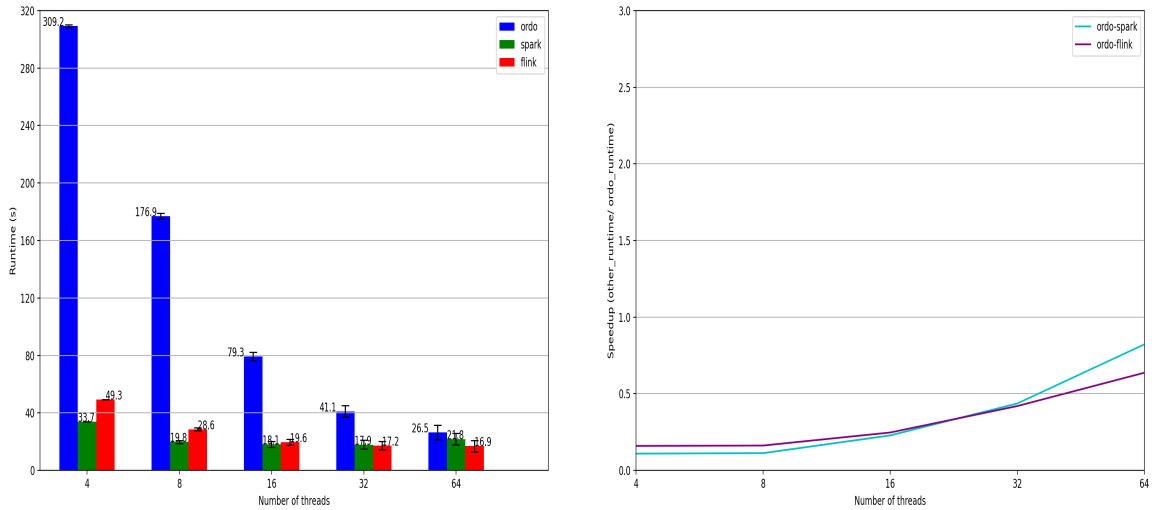


(a) a. Runtime plot of our first implementation compared to Spark and Flink (500MB workload)

(b) b. Speedup plot of our first implementation compared to Spark and Flink (500MB workload)

Figure 3.5: Runtime and speedup plots of our first implementation compared to Spark and Flink (500MB workload)

memory allocation and string pools are used. Our first approach to tackle it was to implement smart strings instead of normal strings. As discussed, by doing so we would allocate every string less than a certain amount of bytes to memory instead of the heap, minimizing the number of malloc calls. Our workload is beneficial for this implementation, since all the strings are less than 23 bytes, therefore all the strings were allocated to the stack instead of the heap. As shown in Figures 3.7a and 3.7b showing the runtimes and speedup respectively. We can see the that ordo is performing better for any amount of threads compared to Spark and Flink, with a drastically improvement compared to our first implementation. In particular, the runtime for 64 threads is

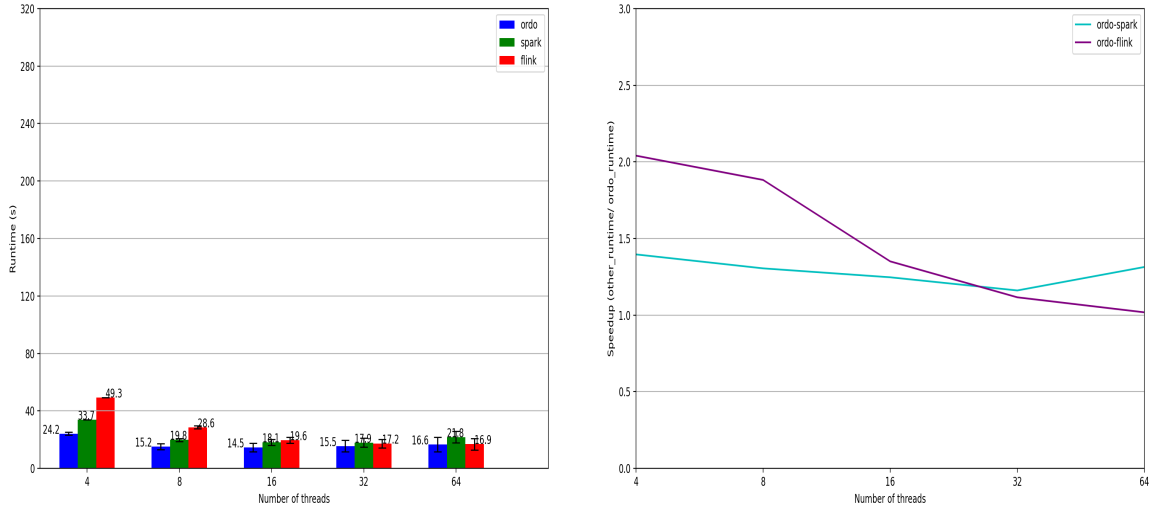


(a) a. Runtime plot of our first implementation compared to Spark and Flink (11 GB workload)

(b) b. Speedup plot of our first implementation compared to Spark and Flink (11 GB workload)

Figure 3.6: Runtime and speedup plots of our first implementation compared to Spark and Flink (11 GB workload)

reduced to an average of 16.6 seconds, an improvement of 37.3%, while the performance has improved vastly for the lower amount of threads as well, as for 4 threads, the runtime reduced to 24.2 seconds from the initial 309.2 seconds, a decrease of 92.2%. Also, we can see from the speedup plot that our system performs better in any number of threads, being slightly more than twice as fast for 4 threads and slightly less than twice as fast for 8 threads compared to Flink. Furthermore, Ordo is 1.4 times as fast compared to spark for 4 and 8 threads. For higher number of threads the performance is almost similar to Flink but still 1.4 times faster than Spark. For the higher number of threads (16-64) we also observed a higher standard deviation, therefore we cannot yet draw final conclusions on which system is performing strictly better.



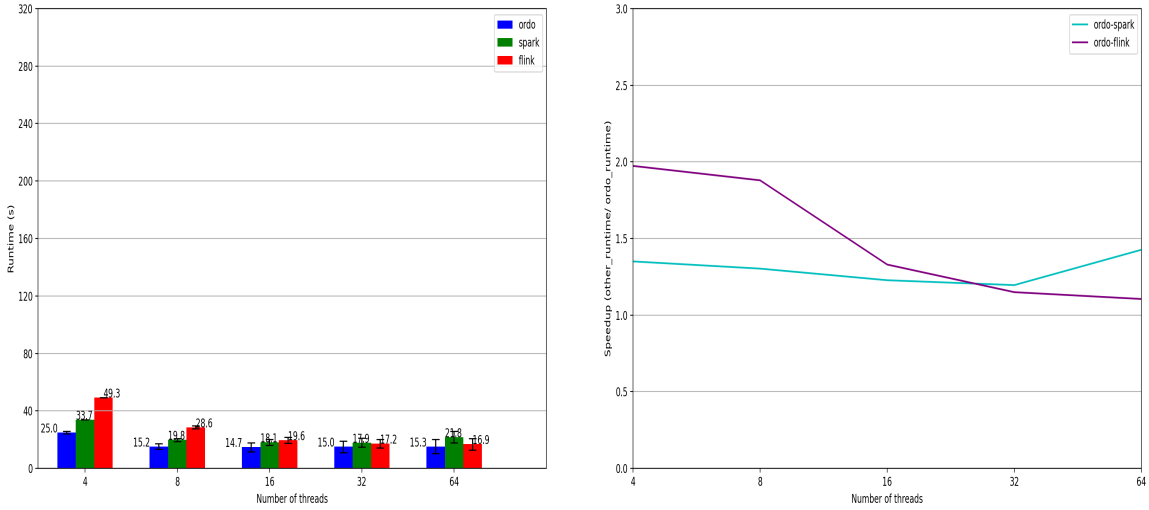
(a) a. Runtime plot of smart strings implementation compared to Spark and Flink

(b) b. Speedup plot of smart strings implementation compared to Spark and Flink

Figure 3.7: Runtime and speedup plots of smart strings implementation compared to Spark and Flink

Second adaptation

Our third iteration consisted of transitioning from smart strings to bytes. The reason we did not settle for smart strings was the limitation they impose. Smart strings provide the benefit of storing the strings in the stack instead of the heap only if the string is less than 23 bytes. In addition, being able to decide how many bytes will be processed gives more flexibility, since we do not rely on the size of the strings while reading them. We can see the results of the third implementation in Figures 3.8a 3.8b where for the same configuration, workload and application we have very similar results as our first adaptation.



(a) a. Runtime plot of our second adaptation compared to Spark and Flink

(b) b. Speedup plot of our second adaptation compared to Spark and Flink

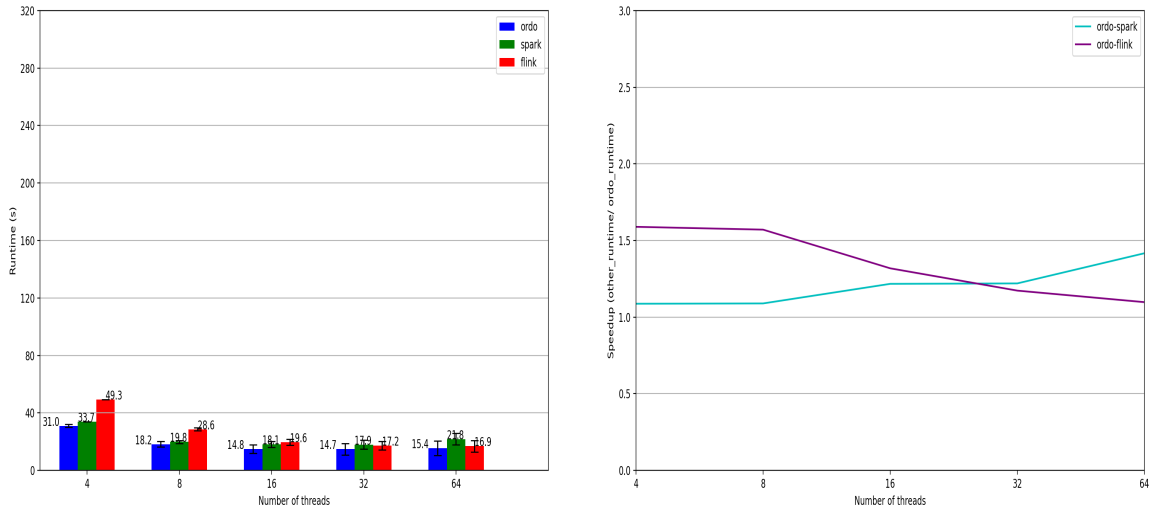
Figure 3.8: Runtime and speedup plots of our second adaptation compared to Spark and Flink

Final design

Finally, our last iteration was to implement the selectivities for all the benefits we have discussed. In this implementation, the weights have disappeared entirely, since the scheduler is able to take the optimal decisions based on the throughput of the system. As shown in Figures 3.9a and 3.9b the average runtime for every thread amount is very similar to the other implementations, while there was no need to fine-tune the weights and to have the optimal decisions, as we had in the previous runs where e.g. we knew exactly how many splits we will have in each line. We can also see that the speedup is pretty much the same. We can see that the runtime in some cases it is slightly worse than our previous implementations (e.g. for 32 threads), but this is expected since our previous implementations were the baseline for our experiments, since they were optimized in any way possible (we knew exactly the amount of splitted words we have for each line and we fine-tuned the weights). This is our final implementation, which we also used for the experiments we conducted in the section 4.

3.5. Discussion

In this section, we analyzed the design decisions we took and the adaptations we had to do while implementing and evaluating our system. The main characteristic of our system is the exposure of decisive metrics to the application level. This allows for the creation of efficient schedulers that can adapt and react to changes during runtime which we believe will lead to a system more efficient, performant and resilient than the current state-of-the art systems that do not take into account these metrics in their scheduling decisions. The system has been designed to require minimal fine-tuning of parameters, a common challenge in systems such as Apache Spark and Apache Flink.



(a) a. Runtime plot of our final implementation compared to Spark and Flink

(b) b. Speedup plot of our final implementation compared to Spark and Flink

Figure 3.9: Runtime and speedup plots of our final implementation compared to Spark and Flink

One of the challenges faced during this research was the need for a data structure that would work and scale under heavy workloads. The shared thread-safe ring buffer was chosen for its efficiency in achieving fast communication between processes when multiple threads are involved. The implementation of slices as logical partitions of the ring buffer allowed for parallelism, with variable sized read and write slices able to be requested simultaneously. To the best of our knowledge, the use of a thread-safe ring buffer in this context has not been studied before, making this an original contribution to the field. Our system has been tested and shows promising results, which are shown in section 4.

Overall, the design we present offers an innovative solution for improving the efficiency and performance of data processing systems. The exposure of system's metrics to the application level as well as the use of a shared, thread-safe ring buffer for communication between processes is a novel contribution to the field that shows promising results. Further research could focus on improving the scalability of the system and testing it in a range of different scenarios.

4

Experiments

In this section, we aim to compare the performance of our custom data processing system against the widely-used Spark and Flink systems in terms of runtime, throughput, resiliency, and energy consumption. To do this, we have developed the application we discussed in section 3.3 that we will run on all three systems, using the DAS-6 cluster sites [40] as our testing environment environment. Through careful measurement and analysis, we hope to answer our research questions and gain insights into the relative strengths and weaknesses of each system.

4.1. Specifications

To ensure the reliability and robustness of our results, we conducted all the non-energy related experiments in the DAS-6 cluster [40], which is a high-performance computing environment composed of multiple nodes equipped with dual 16-core 2.8 GHz CPUs and 128GB of memory. The energy related experiments were ran on Astron cluster, which is a hybrid cluster that meets specific research needs (one of them being energy-efficient computing), being part of the DAS-6 cluster sites. The ASTRON nodes we used were equipped also with dual 16-core 2.8 GHz CPUs with 258GB memory. These nodes allowed us to run our experiments under realistic conditions and accurately measure the performance and energy of each system.

The input workload that we used for all the following experiments was 20GB of text data generated by a script that parses the Gutenberg library. We repeatedly copied the result until the file reached 20GB. Finally, the application here parses and counts all the words that start with the letter 'a' in the input data.

In order to eliminate the influence of caching and other external factors, we ran all experiments 10 times, with the caches cleared between each run. We plotted the standard deviations in each graph to provide a visual representation of the variation between the runs. These precautions allowed us to thoroughly evaluate the performance of our system and compare it to the performance of the existing Spark and Flink systems with confidence.

4.2. Runtime

We expect our system to outperform current state of the art systems in terms of total runtime required to complete the application. The results, shown in Figure 4.1, demonstrate the clear advantage of our system compared to Flink and Spark. We found that the application for our system is I/O bound, meaning that it becomes limited by the input/output performance of the system. For our system, this limitation becomes obvious at 16 threads. In contrast, we observed that the applications for Flink and Spark were limited at 64 threads and onwards. This means that our system is able to scale more efficiently and utilize more resources effectively than Flink and Spark.

Additionally, our system outperformed both Flink and Spark by significant margins. For 4 threads, our system outperformed Spark by 53% and Flink by 40%. For 8 threads, our system outperformed both by almost 54%. These results demonstrate the superiority of our system in terms of total runtime required to complete the application.

We also evaluated the runtime for additional configurations for Apache Spark in order to test the resiliency of our system, as described in section 4.5, however the configuration we have chosen in Figure 4.1 seems to be the optimal one amongst all the experiments with regards to the runtime performance for Spark. Overall, our results show that our system, Ordo, is able to outperform the current state of the art systems in terms of total runtime required to complete the application.

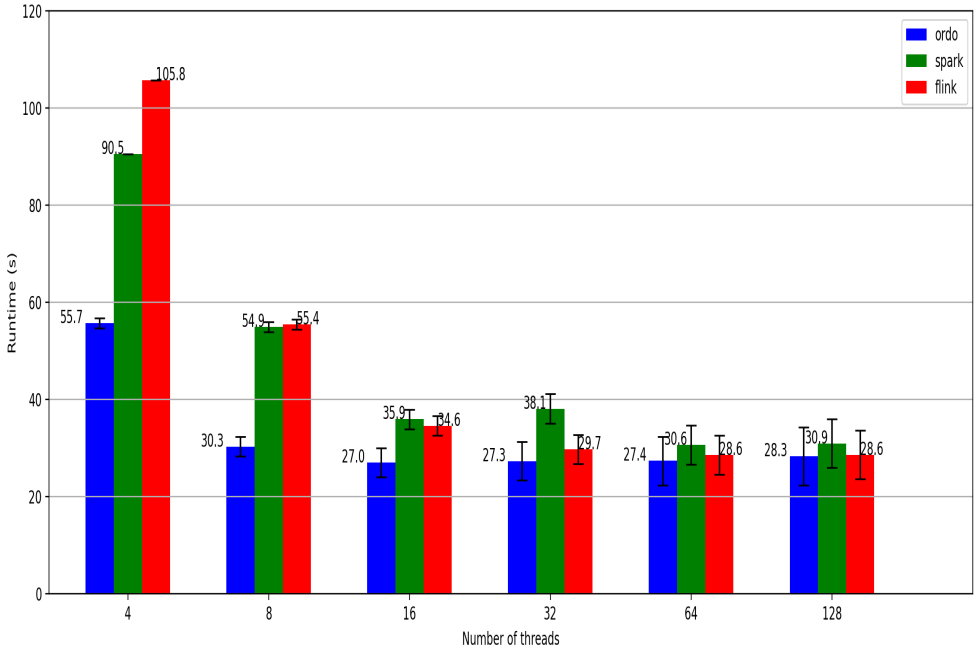


Figure 4.1: Runtime comparison with increasing number of threads

4.3. Throughput

The start-up time for our system can vary depending on various factors such as the input data and the processing power of the system. In this particular experiment, we observed that the system required 1.5 seconds to reach its peak equilibrium, which it maintained until the end.

As shown in Figure 4.2, the system reaches the equilibrium at 1750MB/s and maintains this level of performance until the end of the experiment. This indicates that our system is able to effectively process the input data and maintain a high level of performance even when faced with a high volume of data. We can see that in the start-up phase, the system reaches a throughput of 2000MB/ s, however this higher throughput could take place because initially there is less contention and therefore the slices do not require to wait for previous slices to be committed before they are also committed, resulting in faster throughput for these starting ms.

Towards the end of the experiment, we can see a decline in the throughput of the system, which is due to the diminishing input data. Overall, our experimental results indicate that our system is able to effectively reach a peak equilibrium and maintain high performance levels.

4.4. Per item latency

The results of our experiments indicate that our system is capable of achieving an evenly distributed per-item latency when the equilibrium is reached. As expected, we observed some deviations in the per-item-latency until the equilibrium was reached. This is likely due to processes being activated more frequently than desired until the system has stabilized and the processes' selectivities and priorities accurately reflect the current state of the system.

The per-item-latency after random sampling is shown in Figure 4.3a. The figure depicts the his-

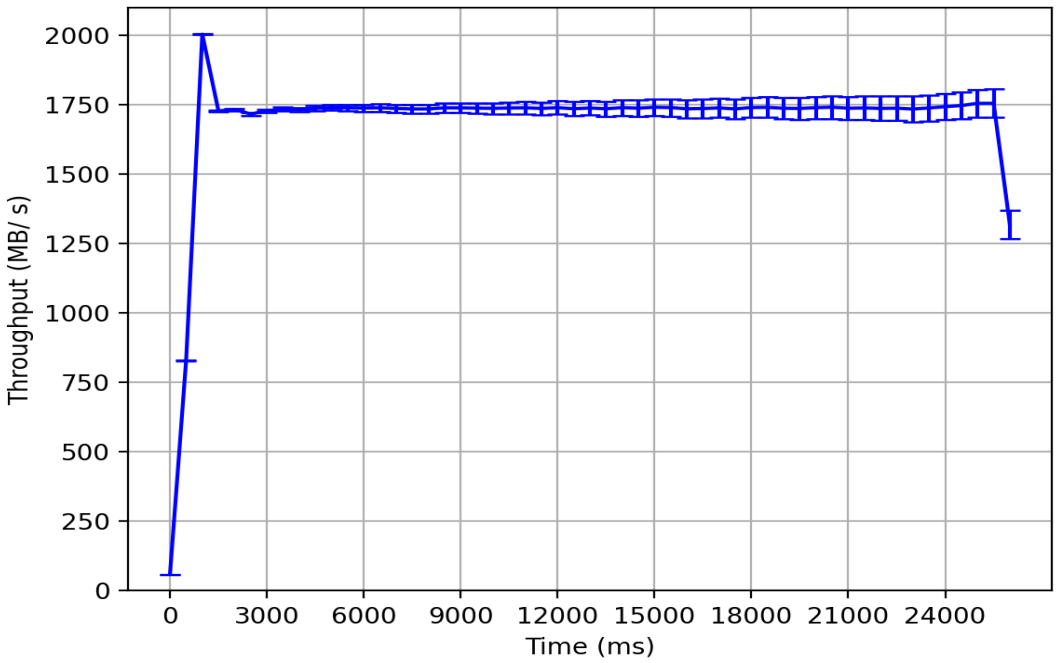
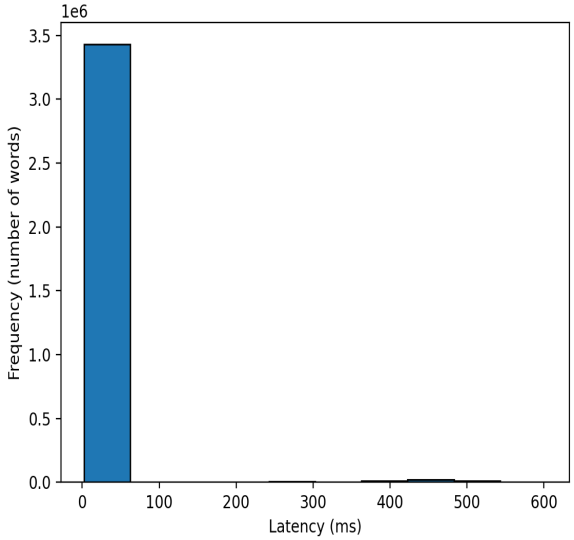
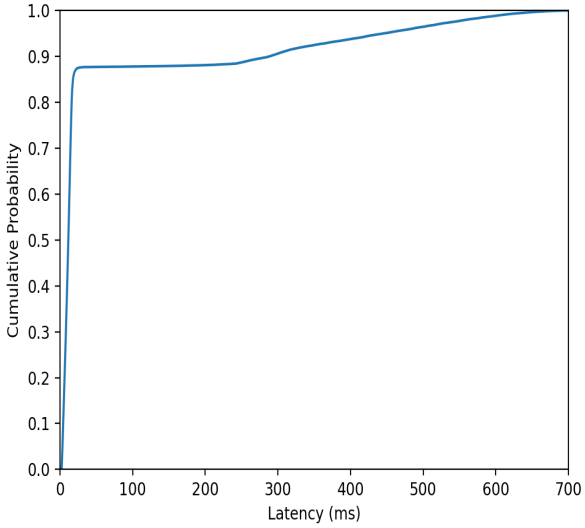


Figure 4.2: Throughput of ordo over time for 32 threads



(a) a. Histogram of the average per item latency



(b) b. Cumulative distribution function plot for the average per item latency

Figure 4.3: Histogram (a) and CDF plot(b) for the average per item latency of Ordo (random sampling every 100 items, run for 32 threads)

togram of the average per-item-latency after random sampling every 100 items. It clearly demonstrates the even distribution of latencies for the majority of items, with almost 3.5 million words being processed within 0-80 milliseconds, as indicated by the first bar. However, we also observe some very low height bars in the range of 350-650 milliseconds, each concerning around 100,000 words. These deviations are likely the result of processes being activated more frequently than desired until the system has stabilized.

We can also see, in Figure 4.3b, the cumulative distribution function plot of the average per item latencies for our system (by random sampling every 100 items randomly). Our system processes the 87% of the items in less than 24ms. As the histogram plot, we can see the outliers that are processed in higher latencies, up to 650ms.

Overall, these results are consistent with our expectations and demonstrate the effectiveness of our system in achieving an evenly distributed per-item latency. While there may be some temporary deviations until the equilibrium is reached, these are likely to disappear once the system has stabilized.

4.5. Resiliency

In order to evaluate the resiliency of our system, we conducted an experiment to evaluate the throughput in a situation where we systematically reduced the number of threads by 60% every 4 seconds, in order to simulate a case where the system experiences a significant drop in resources. In order to avoid being left with no threads after a lot of decrements, we set the minimum number of final threads to 2 so that the system can produce a final result in the end. It is logically expected that this reduction in threads would result in a drastic reduction in throughput and an increase in run-time.

Consequently, We implemented a strategy to enhance the system's ability to efficiently handle sudden resource drops. In our implementation, each thread checks the size of the thread pool and adjusts it accordingly if necessary. This allows the system to dynamically respond to changes in the number of available threads, and we expect that it will help mitigate the negative effects of the thread reduction in our system's performance.

We can see the results of our experiments in Figure 4.4. The red plot shows the sudden drops in throughput and the long run-time that occurred when the number of threads is reduced without implementing our new strategy. In contrast, the blue plot shows that in the new strategy, where each thread is adjusted to resize the thread pool if it sees that the size is not as expected, the system is able to maintain a high level of throughput despite the reduction in threads. We can also observe that the runtime of the restoring experiment is around 27 seconds, which is the optimal runtime we observed for 32 threads in section 4.2. This demonstrates the effectiveness of our strategy in countering the negative effects of sudden resource drops on system performance, and our expectations are met.

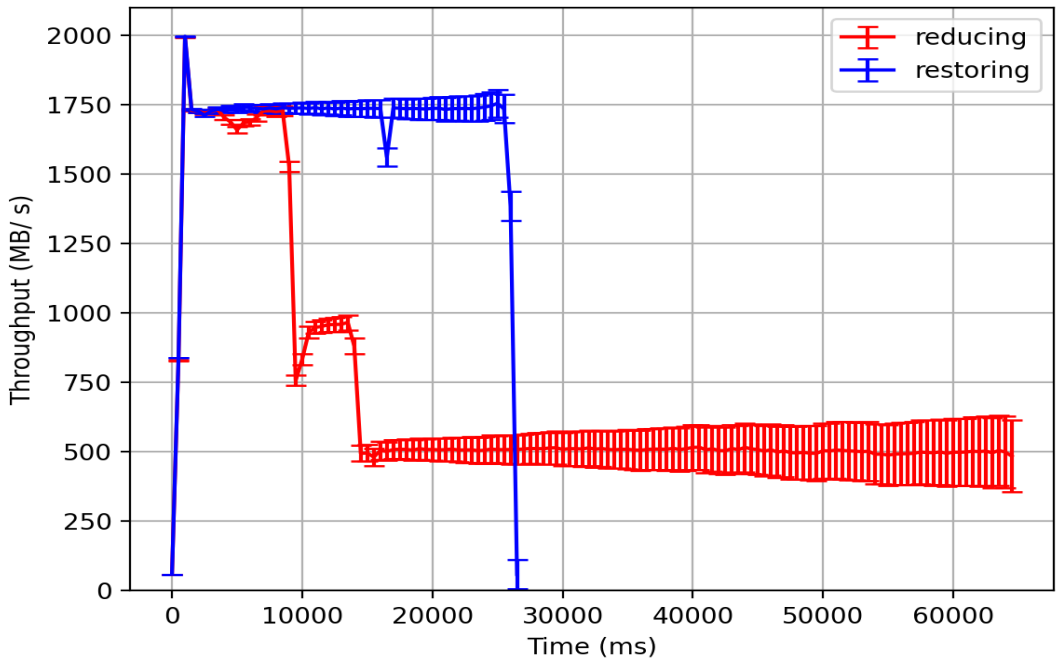


Figure 4.4: Throughput of ordo over time for a reducing and a restorative implementation

We also tested the resiliency of one of systems that are well-known for their high fault-tolerance, Apache Spark, for comparison. In order to do so, we first tried to implement a similar drop in resources as our system's experiment. It is worth noting that Apache Spark does not have the ability to revive executor's threads. However, it can revive Executor and Worker nodes. Therefore, we created several experiments that required the revival of Executors and Workers by Apache Spark, as shown in Figure 4.5 and 4.6.

The DAS-6 cluster is consisted of 20 nodes, each having 64 cores and 128 GB of memory, so for our first experiments, in Figure 4.5, we first started a configuration with 64 executors, each having 2 GB memory and 1 core. Similarly to our system's experiment, we first tried reducing the number of executors every 4 seconds by 60%. However, with this implementation, we could not get any results because the system was too busy spinning up the new executors and it did not progress in the tasks completion at all. Thus, we adapted the experiment to reduce the executor processes every 10 seconds by 60%. As we can see, for the configuration of 64 executors, the difference between the normal implementation (where no executor processes were terminated) and the revive implementation (where we did kill 60% of executor processes every 10 seconds), is large. The normal implementation required 40.8 seconds to complete while the implementation where executors were terminated throughout required more than double time to complete (85.8 seconds). We tested different configurations as well, such as for 32 executors (with 4GB of memory and 2 cores each) and 16 executors (with 8GB memory each and 4 cores each). We can see for the former case, the runtime increased by 5.6% and in the latter case by 3.75%. This was expected, since the lower amount of the total executors, the less amount was terminated, due to our 60% ratio. These experiments show how more efficient and performant is the revival of threads instead of processes (executors) in systems where constant processing is required.

Finally, since Apache Spark provides the ability to revive Workers, we also performed some experiments where, instead of executors, the Workers were terminated unexpectedly only once and not periodically like the executor experiments. In each of the experiments, we used 1 executor for each worker. The results are shown in Figure 4.6. We first a configuration with 2 workers and 64 cores per executor. However, the runtime was much lower than the optimal runtime we have observed for spark, indicating a resource contention, so we tried a different configuration with less executors. We can see that for a normal run with 2 workers and 32 executors each, the average runtime is 31.9 seconds. However, when we terminated one of the workers, 15 seconds after the start of the experiment, we can see that the runtime increased to 37.4 with an increase of 17.2%. We also performed an experiment terminating all workers (2 in this case) which resulted in a runtime of 34.1 with an increase of 6.8% compared to the normal run. Similar behavior is shown in the experiment with 4 workers (16 cores each), where the runtime increased by 30% with 1 worker failure and by 12.8% by 4 worker failures. We can clearly see an increase in the standard deviation while we increase the worker failures. This is reasonable, because in each run the worker might have a quite different amount of tasks in a different stage. The time to start-up the worker might also play a crucial role in this increase. Another interesting fact is that for the case where we terminated all the workers appeared to require less runtime than the case where we terminated one worker. We suspect the following reasons for this:

- **Data locality** When a worker fails, the data that was stored on that worker needs to be shuffled to another worker in order to continue processing. This can be time-consuming and can lead to longer runtimes. However, when all workers fail, the same worker could continue working on the same job after it's revival, so there is no data locality to consider.
- **Task scheduling** When a worker fails, the remaining tasks may need to be re-scheduled on the remaining workers, which can also add overhead and increase the runtime of the job. If all workers fail, the tasks may be scheduled on the same worker after revival.
- **Job recovery** When a worker fails, Spark's fault-tolerance mechanisms may kick in to recover the lost tasks and data. This can also add overhead and increase the runtime of the job. If all workers fail, there is no need to recover any lost tasks or data.

In conclusion, we can see the advantage of our system in terms of resiliency compared to Spark. It is evident that recovering threads is much more light and efficient than recovering processes. By enabling thread recovery, we gain the advantage of more stability, data locality and less task scheduling, since each process can continue processing with the same throughput, which can greatly affect the performance of the system.

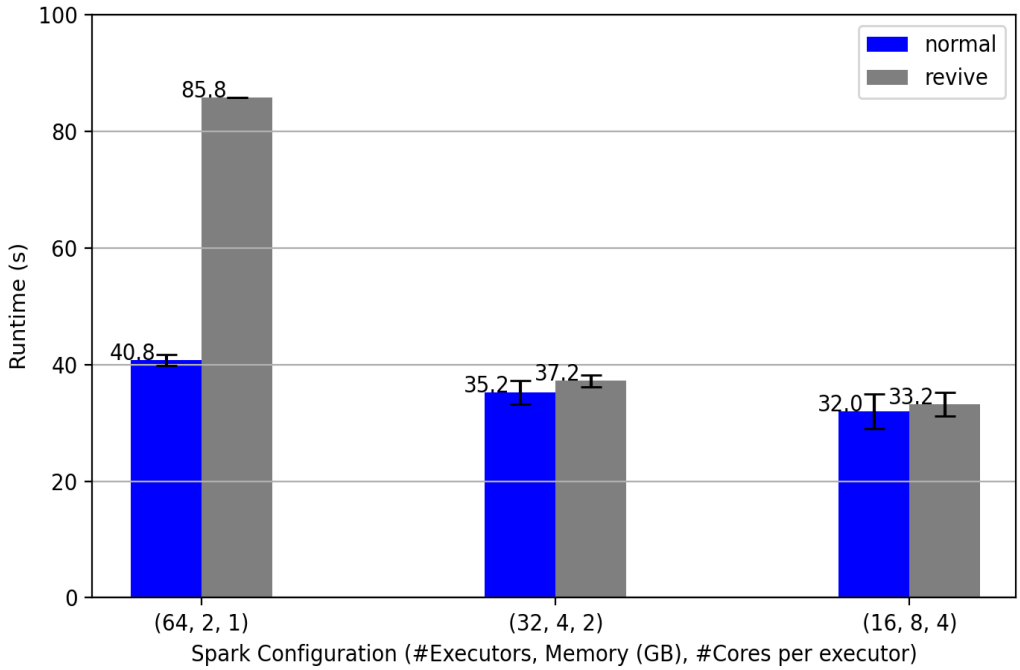


Figure 4.5: Runtime of spark over time for normal vs executor-disruptive runs

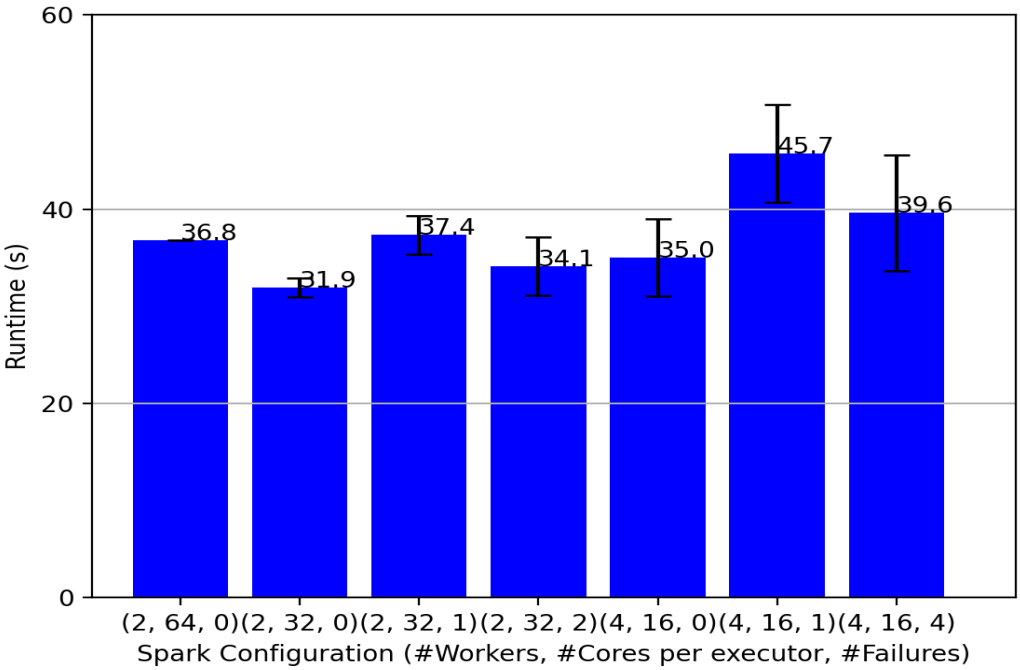


Figure 4.6: Runtime of spark over time for normal vs worker-disruptive runs

4.6. Energy Efficiency

In order to evaluate the energy efficiency of our system, we performed experiments in the ASTRON nodes. In particular, we used the Power Measurement Toolkit (pmt) software library which provides power consumption measurements on various hardware [41]. Power Measurement Toolkit provides the ability to measure the Joules and Watts consumption of any part of the applications desired. We used Intel RAPL, one of the most accurate tools for monitoring the energy consumption of the hardware [42]. We decided to measure the power consumption of the whole runtime for Ordo and compare it with the power consumed by Spark and Flink over the same workload we used in the rest of the experiments (20GB of text data).

For these experiments our expectations are that our system will be significantly more efficient in energy consumption than Spark and Flink for a couple of reasons. Firstly, as we previously showed, our system has greater runtime for any number of threads compared to both Spark and Flink. This is a reason for our system to have less Joules consumption. However, just the runtime is not enough to draw conclusions about the energy consumption, since it could require substantially more hardware resources than the other systems and therefore, even having less runtime, consume more joules than the other systems. Nevertheless, some of the reasons of the significant difference between Ordo and Spark and Flink can be because the latter systems :

- Run on a JVM which acquires hardware resources beforehand and runs operations like garbage collection
- Come with the overhead associated with distributing and coordinating the data processing tasks across multiple nodes in a cluster
- Contain built-in resource managers that allocate resources to data processing tasks based on various criteria such as CPU usage, memory usage, and network I/O and more

On the other hand, our system is more light-weight as it decides where to optimally schedule the tasks based on the workload and does not require any extra jobs like garbage collection. It also does not run on a JVM relying on garbage colletors and it does not have overhead associated with tasks like distributing and coordinating the data procesing tasks across multiple nodes (however this might be implemented in the future). We also expect all systems to improve (reduce) in Joules consumption up to 32 threads and then remain to close levels with possible deterioration due to the fact that the runtime showed similar behaviors because of the I/O-boundness of the problem.

In Figure 4.7 we can see the pure Joules consumed by the systems. Ordo performs substantially better in all thread amounts. We can see for 4 threads Ordo spends 4593.8 Joules while Spark requires 28371.9 and Flink 30652.6 Joules, with an improvement of 83.80% and 85.01% respectively. For 16 threads, which is the optimal joules consumption for each system, we observe an improvement of 79.7% and 80.2% against Spark and Flink respectively. After 16 threads, the power consumption increases slightly for all the systems probably because of the I/O boundness.

Figure 4.8 shows the Watts consumption of the systems. We can see that the Watts consumption is close for all the systems, with Spark achieving the best Watt consumption from all systems, followed by Flink and then Ordo. The standard deviation of the Watts is also higher, especially for the runs after 16 threads, showing that we cannot draw clear conclusions of which system is significantly better than the other. However, as Watts depict the Joules consumed per second, we can say that Ordo consumes more Joules per second because it completes the job execution faster than Spark and Flink. To be more precise, for 16 threads where the systems seem to have the optimal Watts consumption, Ordo consumed 136.5 Watts, Spark 109.6 Watts and Flink 112.4 Watts, with Ordo consuming 24.5% and 21.4% more Joules/s compared to Spark and Flink respectively.

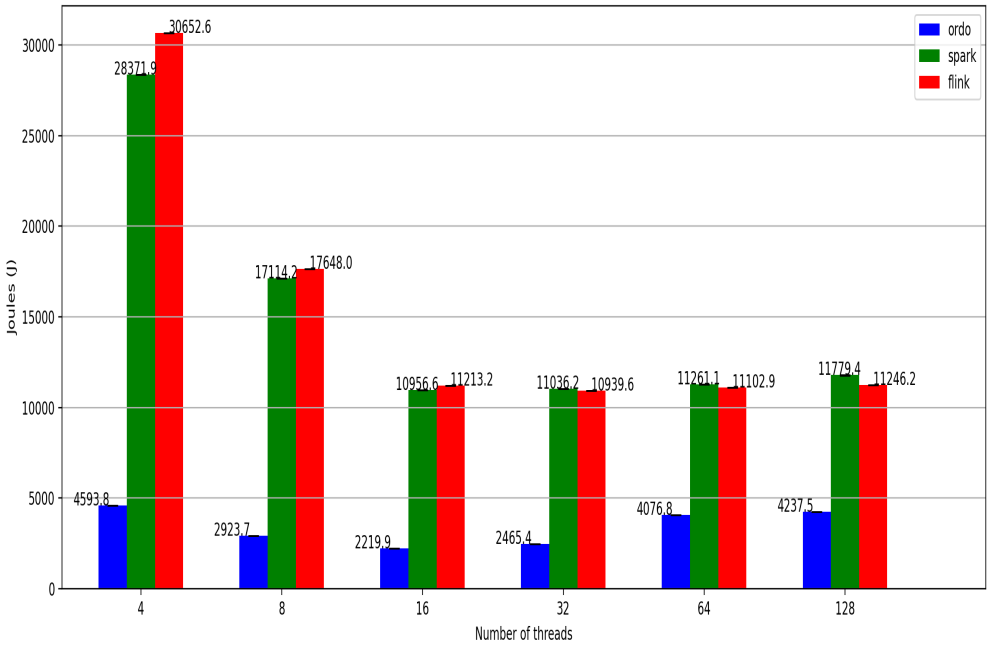


Figure 4.7: Joules consumption over the period of the job completion for Ordo, Spark and Flink

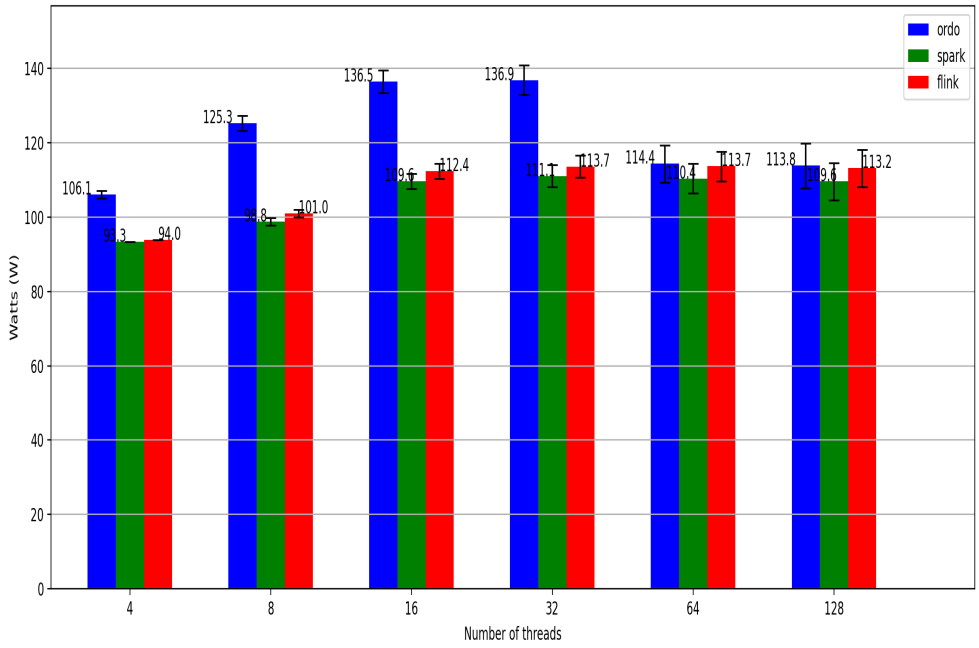


Figure 4.8: Watts consumption over the period of the job completion for Ordo, Spark and Flink

5

Conclusions and Future Work

5.1. Conclusions

The primary goal of this research is to answer the following main research question that we proposed in Section 1.2 "*Can the underlying system's metrics be exposed in the application level in order to allow the building of efficient schedulers?*". In order to answer this question, we designed, implemented through multiple rounds of refinement, both a system and an application on top of it in order to experiment and compare it with other state-of-the art systems. We divided the above question to sub-questions in order to be able to give more detailed answers as follows:

1. *Can we improve the performance and practicality of a data processing system by exposing the underlying system's metrics to the application?*

   As aforementioned in Section 3, Ordo's architecture exposes the underlying system's metrics such as the throughput of each process and the workload of each process to the application, allowing the scheduler to make optimal decisions dynamically during run-time. We started by implementing the system with a simple round-robin scheduler and continued with implementing a process-priority scheduler which takes into account the underlying system metrics. By doing so, we saw a significant improvement in all the aforementioned metrics in our analysis in Section 3.4. In this section, we saw that our final design, which consists of a process priority scheduler that takes into account the underlying system's metrics, we can achieve the same performance as the optimal baseline which required static optimal pre-configuration, which is not feasible in real-world streaming application scenarios since we used a workload that we knew what the optimal weights were, proving that our system can improve the performance and practicality of the data processing system by exposing the underlying system metrics to the application level.

2. *Can a scheduler that has access to the system metrics reach an equilibrium state ?*

   After having built the system with the desired characteristics, we experimented in order to check if our assumptions where correct. In section 4, we plotted the throughput of the average of 10 runs, including the standard deviations and the results were clear that our system reached a peak equilibrium state, which continued until the end of processing. We observed a higher peak for a very short time in the start of the application, which we believe is because of the faster slice commitments and the lower contention during the start-up phase.

3. *What is the most effective way of building an application on such system?*

   The system's scheduler is one of the main components and is responsible for the benefits we have discussed, such as the increased performance, resiliency etc, when it takes into account the underlying system's metrics. As we saw in 3.4, we tackled a lot of challenges building the application to achieve substantial greater results than our first implementation. Therefore, all the most effective way of building an application on such system is to being able to tackle the challenges that we faced, such as being able to measure the performance of the application and find the bottlenecks, as well as the requirement of using good memory allocation and data

structures within the application (within the operators) and finally building an efficient scheduler that takes into account the desired underlying system metrics.

4. *What are the limitations that such system imposes?*

   While our system is very effective and performant, as we saw in 3.4 it also imposes a lot of challenges that might be a limitation factor in some cases. The application developer trying to build an application on top of such system must be experienced in finding bottlenecks and developing low-level applications with optimal data structures.

5. *Does such system outperform other state-of-the art systems in , terms of performance, practicality, energy efficiency and resiliency ?*

   In section 4 we performed an extensive research and comparison of our system compared to stae-of-the art systems such as Apache Spark and Apache flink. We showed that our system outperforms both the other systems for any number of threads. For 4 threads, our system outperformed Spark by 53% and Flink by 40%, while for 8 threads it outperformed them by almost 54%. We noticed that the problem is I/O bound, however our system still reaches this boundnness faster than the other systems, indicating that it requires less threads for the same amount of work than the other systems. Furthermore, we compared the resiliency of our system against Spark, which is well-known for it's resiliency. We concluded that recovering threads (as Ordo does) is significantly lighter and more efficient than recovering processes, as Spark does. While our system had very similar runtime to our optimal runtime, after thread failures, Spark had significantly slower runtimes (almost double runtime for 64 executors, 32 GB memory and 1 core per executor). We also did Worker resiliency experiments in Spark, where we also observed a drop in performance when a Worker died (e.g. runtime increased by 30% with 1 worker failure for a configuration of 4 workers, 16 cores each).

   Finally, we conducted energy experiments to compare all three systems. The energy experiments provided great and very promising results for our system compared to the other two systems. In particular, for 16 threads, Ordo had 79.7% and 80.2% less Joules consumption against Spark and Flink respectively. As for the Watts consumption, we observed that Ordo had a slightly worse Watts consumption than the other two systems, however this can be appointed to the fact that Ordo consumes more Joules per seconds because it completes the same jobs in faster time than Spark and Flink (Ordo consumes 24.5% and 21.4% more J/s compared to Spark and Flink respectively).

By answering all the sub-questions, we have also answered our main research question : Can the underlying system's metrics be exposed in the application level in order to allow the building of efficient schedulers that can reach an equilibrium state and outperform other state-of-the art systems?. We showed that it is possible to expose the underlying system's metrics to the application level in an efficient way. We managed to build an efficient scheduler that reached an equilibrium state and outperformed other state-of-the art systems in many aspects, such as runtime performance, energy-efficiency and resiliency.

5.2. Future Work

The present work has focused on the design, implementation and validation of a high performance data processing system that is consisted of a scheduler which, with the help of the exposed underlying system's metrics, can reach an equilibrium throughput state. It proves that such system implementation is possible and it's capabilities are in-line with our expectations. Nevertheless, we have some suggestions that can be taken into consideration for future work in this field, some of them being arose by the challenges that we faced during the implementation of the system, while others being recommendations that we believe will be valuable to validate in the future.

To begin with, we built and tested our system implementation for a single machine. However, we believe that this can be extended in a more scalable version in the future, were more machines can contribute towards the same goal. In such system, each node will still contain a thread pool, however there can be more than one nodes that communicate with each other and distribute the workload, in order to have a more scalable system.

Another improvement that we can see is scaling the thread-pool of the system according to the workload, a suggestion that was also proposed in SEDA [12]. By doing so, the system will not be wasting resources (threads) by containing them inside the thread pool but keeping them inactive. This will also be very useful in our previous suggestion to make the system more scalable, since there will be multiple nodes and more thread pools to manage, so the total benefit will be more significant. This, of course, is another benefit of exposing the system's metrics to the application, since we can adjust the thread pools according to the throughput of the processes.

As already mentioned, two of our system's aims is to increase the efficiency and practicality. As discussed, we minimized the need to fine-tune parameters, which is a very common occurrence in such systems. An we see in this topic is to make the size of each ring-buffer variable, since each process has totally different responsibilities and workloads. On top of that, it is also possible to have a dynamic memory allocator that every some time intervals, dynamically changes the memory of each ring-buffer according to the process's average workload over some time period. This shows yet another benefit of exposing the system metrics and it can greatly save more system resources.

Another improvement we can see that we believe would improve the performance of the system would be to implement a mechanism to exit a process. Right now, if a process is activated, it will have to complete it's job and there will be no way of returning to the scheduler without completing it. However, we believe that this is imposing a limitation to the system, since there can be factors that would deem the return to the scheduler much more efficient for the system's performance. As an example of such limitation, right now a thread (inside a process) will repeatedly try to reserve a slice from the next process. However, if all threads enter the same process and try to request a slice from the next one, but the next process does not have enough slices to supply the requesters, the system will stuck. In order to avoid this, a mechanism can be introduced where a process tries to reserve a slice for a specific amount of times, before failing and returning to the scheduler, lowering it's priority as well such that the scheduler will prefer to pick another process before continuing.

Lastly, as we analyzed in Section 3, we decided to implement a system that takes care of the ordering of the incoming events. This is a more complex design and we could imagine our system being configurable, where the developer can suggest via a parameter if order preservation is required by the application. If so, the committing of slices could be handled as we have currently designed it, otherwise the system could fallback to a much easier implementation where the slice committed status is shown in a bitset.

Apart from the aforementioned improvements that can be applied directly to the contributions of this research, we suggest some other future works that can be made towards the validation and experimentation of such systems. One of the contributions would be to compare this system against other low-level data processing frameworks like Phoenix++ [34]. In addition, more experiments can be conducted, e.g. to evaluate huge fluctuations in the input stream, to which we believe our system would adapt to effectively.

Bibliography

[1] C.L. Philip Chen and Chun-Yang Zhang. “Data-intensive applications, challenges, techniques and technologies: A survey on Big Data”. In: *Information Sciences* 275 (2014), pp. 314–347. ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2014.01.015. URL: https://www.sciencedirect.com/science/article/pii/S0020025514000346.

[2] S. Kudyba, ed. *Big Data, Mining, and Analytics: Components of Strategic Decision Making (1st ed.)* Atlantis Press, Paris, 2014. DOI: 10.1201/b16666. URL: https://doi-org.tudelft.idm.oclc.org/10.1201/b16666.

[3] Lina Zhou et al. “Machine learning on big data: Opportunities and challenges”. In: *Neurocomputing* 237 (2017), pp. 350–361. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2017.01.026. URL: https://www.sciencedirect.com/science/article/pii/S0925231217300577.

[4] Sanjiv R Das. “The future of fintech”. In: *Financial Management* 48.4 (2019), pp. 981–1007.

[5] Bin Fang and Peng Zhang. “Big Data in Finance”. In: *Big Data Concepts, Theories, and Applications*. Ed. by Shui Yu and Song Guo. Cham: Springer International Publishing, 2016, pp. 391–412. ISBN: 978-3-319-27763-9. DOI: 10.1007/978-3-319-27763-9_11. URL: https://doi.org/10.1007/978-3-319-27763-9_11.

[6] Paris Carbone et al. “Apache flink: Stream and batch processing in a single engine”. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).

[7] Ankit Toshniwal et al. “Storm@ twitter”. In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 2014, pp. 147–156.

[8] Salman Salloum et al. “Big data analytics on Apache Spark”. In: *International Journal of Data Science and Analytics* 1.3 (2016), pp. 145–164.

[9] Martin Andreoni Lopez, Antonio Gonzalez Pastana Lobato, and Otto Carlos M. B. Duarte. “A Performance Comparison of Open-Source Stream Processing Platforms”. In: *2016 IEEE Global Communications Conference (GLOBECOM)*. 2016, pp. 1–6. DOI: 10.1109/GLOCOM.2016.7841533.

[10] Albert Reuther et al. “Scheduler technologies in support of high performance data analysis”. In: *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 2016, pp. 1–6. DOI: 10.1109/HPEC.2016.7761604.

[11] Andrew Tanenbaum. *Modern operating systems*. Pearson Education, Inc., 2009.

[12] Matt Welsh, David Culler, and Eric Brewer. “SEDA: An Architecture for Well-Conditioned, Scalable Internet Services”. In: *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*. SOSP ’01. Banff, Alberta, Canada: Association for Computing Machinery, 2001, pp. 230–243. ISBN: 1581133898. DOI: 10.1145/502034.502057. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/502034.502057.

[13] Chen Lin et al. “Adaptive Code Learning for Spark Configuration Tuning”. In: *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 2022, pp. 1995–2007. DOI: 10.1109/ICDE53745.2022.00195.

[14] Robert F Stärk, Joachim Schmid, and Egon Börger. *Java and the Java virtual machine: definition, verification, validation*. Springer Science & Business Media, 2012.

[15] Apache Software Foundation. *Hadoop*. Version 0.20.2. Feb. 19, 2010. URL: https://hadoop.apache.org.

[16] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in scala*. Artima Inc, 2008.

[17] James G. Shanahan and Laing Dai. “Large Scale Distributed Data Science Using Apache Spark”. In: *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’15. Sydney, NSW, Australia: Association for Computing Machinery, 2015, pp. 2323–2324. ISBN: 9781450336642. DOI: 10.1145/2783258.2789993. URL: https://doi.org/10.1145/2783258.2789993.

[18] Scott Meyers. *Effective modern C++: 42 specific ways to improve your use of C++ 11 and C++ 14*. ” O’Reilly Media, Inc.”, 2014.

[19] Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[20] Stephen Cass. “Top Programming Languages 2022”. In: (2022). URL: https://spectrum.ieee.org/top-programming-languages-2022.

[21] Howard Austerlitz. “CHAPTER 13 - Computer Programming Languages”. In: *Data Acquisition Techniques Using PCs (Second Edition)*. Ed. by Howard Austerlitz. Second Edition. San Diego: Academic Press, 2003, pp. 326–360. ISBN: 978-0-12-068377-2. DOI: https://doi.org/10.1016/B978-012068377-2/50013-9. URL: https://www.sciencedirect.com/science/article/pii/B9780120683772500139.

[22] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. *Operating systems: Three easy pieces*. Arpaci-Dusseau Books LLC Boston, 2018.

[23] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2022.

[24] Yunquan Zhang et al. “Parallel Processing Systems for Big Data: A Survey”. In: *Proceedings of the IEEE* 104.11 (2016), pp. 2114–2136. DOI: 10.1109/JPROC.2016.2591592.

[25] Hongzi Mao et al. “Learning Scheduling Algorithms for Data Processing Clusters”. In: *Proceedings of the ACM Special Interest Group on Data Communication*. SIGCOMM ’19. Beijing, China: Association for Computing Machinery, 2019, pp. 270–288. ISBN: 9781450359566. DOI: 10.1145/3341302.3342080. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/3341302.3342080.

[26] James R. Larus and Michael Parkes. “Using Cohort Scheduling to Enhance Server Performance (Extended Abstract)”. In: *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*. LCTES ’01. Snow Bird, Utah, USA: Association for Computing Machinery, 2001, pp. 182–187. ISBN: 1581134258. DOI: 10.1145/384197.384222. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/384197.384222.

[27] Konstantin Shvachko et al. “The Hadoop Distributed File System”. In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. 2010, pp. 1–10. DOI: 10.1109/MSST.2010.5496972.

[28] Vinod Kumar Vavilapalli et al. “Apache Hadoop YARN: Yet Another Resource Negotiator”. In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC ’13. Santa Clara, California: Association for Computing Machinery, 2013. ISBN: 9781450324281. DOI: 10.1145/2523616.2523633. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/2523616.2523633.

[29] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/1327452.1327492.

[30] Matei Zaharia et al. “Spark: Cluster Computing with Working Sets”. In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud’10. Boston, MA: USENIX Association, 2010, p. 10.

[31] Reynold Xin et al. “Shark: SQL and Rich Analytics at Scale”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Nov. 2012). DOI: 10.1145/2463676.2465288.

[32] Ovidiu Marcu et al. “Spark Versus Flink: Understanding Performance in Big Data Analytics Frameworks”. In: Sept. 2016, pp. 433–442. DOI: 10.1109/CLUSTER.2016.22.

[33] Justin Talbot, Richard M. Yoo, and Christos Kozyrakis. “Phoenix++: Modular MapReduce for Shared-Memory Systems”. In: *Proceedings of the Second International Workshop on MapReduce and Its Applications*. MapReduce ’11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 9–16. ISBN: 9781450307000. DOI: 10.1145/1996092.1996095. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/1996092.1996095.

[34] Colby Ranger et al. “Evaluating MapReduce for Multi-core and Multiprocessor Systems”. In: Mar. 2007, pp. 13–24. DOI: 10.1109/HPCA.2007.346181.

[35] Dr. J. S. Rellermeyer Minas Melas. URL: https://github.com/mmelas/ordo/tree/implement-target/src/log_parser.

[36] Rolando Inglés, Mariusz Orlikowski, and Andrzej Napieralski. “A C++ shared-memory ring-buffer framework for large-scale data acquisition systems”. In: *2017 MIXDES - 24th International Conference ”Mixed Design of Integrated Circuits and Systems*. 2017, pp. 161–166. DOI: 10.23919/MIXDES.2017.8005175.

[37] Jeff Bonwick. “The Slab Allocator: An Object-Caching Kernel Memory Allocator”. In: *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*. USTC’94. Boston, Massachusetts: USENIX Association, 1994, p. 6.

[38] Oracle documentation authors. “Annotation Type StringPool”. In: (). URL: https://docs.oracle.com/javacard/3.0.5/api/javacardx/annotations/StringPool.html.

[39] Rust documentation authors. “Crate smartstring”. In: (). URL: https://docs.rs/smartstring/latest/smartstring/.

[40] Henri Bal et al. “A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term”. In: *IEEE Computer* 49.5 (May 2016), pp. 54–63.

[41] Stefano Corda, Bram Veenboer, and Emma Tolley. *PMT: Power Measurement Toolkit*. 2022. DOI: 10.48550/ARXIV.2210.03724. URL: https://arxiv.org/abs/2210.03724.

[42] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. “A Validation of DRAM RAPL Power Measurements”. In: *Proceedings of the Second International Symposium on Memory Systems*. MEMSYS ’16. Alexandria, VA, USA: Association for Computing Machinery, 2016, pp. 455–470. ISBN: 9781450343053. DOI: 10.1145/2989081.2989088. URL: https://doi.org/10.1145/2989081.2989088.