big data and cognitive computing

MDPI

Article

Evaluating Task-Level CPU Efficiency for Distributed Stream Processing Systems

Johannes Rank [1,*], Jonas Herget [1], Andreas Hein [2] and Helmut Krcmar [2]

[1] Wittges Lab, Technical University of Munich (TUM), Parkring 13, 85748 Garching, Germany
[2] Krcmar Lab, Technical University of Munich (TUM), Boltzmannstr. 3, 85748 Garching, Germany
* Correspondence: johannes.rank@tum.de; Tel.: +49-89-289-17645

Abstract: Big Data and primarily distributed stream processing systems (DSPSs) are growing in complexity and scale. As a result, effective performance management to ensure that these systems meet the required service level objectives (SLOs) is becoming increasingly difficult. A key factor to consider when evaluating the performance of a DSPS is CPU efficiency, which is the ratio of the workload processed by the system to the CPU resources invested. In this paper, we argue that developing new performance tools for creating DSPSs that can fulfill SLOs while using minimal resources is crucial. This is especially significant in edge computing situations where resources are limited and in large cloud deployments where conserving power and reducing computing expenses are essential. To address this challenge, we present a novel task-level approach for measuring CPU efficiency in DSPSs. Our approach supports various streaming frameworks, is adaptable, and comes with minimal overheads. This enables developers to understand the efficiency of different DSPSs at a granular level and provides insights that were not previously possible.

Keywords: CPU efficiency; big data; distributed stream processing; performance; task-level measurement; profiling; flink; spark

Citation: Rank, J.; Herget, J.; Hein, A.; Krcmar, H. Evaluating Task-Level CPU Efficiency for Distributed Stream Processing Systems. *Big Data Cogn. Comput.* **2023**, *7*, 49. https://doi.org/10.3390/bdcc7010049

Academic Editors: Ella Pereira, Rubem Pereira and Geyong Min

Received: 1 January 2023
Revised: 1 March 2023
Accepted: 7 March 2023
Published: 10 March 2023

1. Introduction

In recent years, the complexity of Big Data systems and applications has continued to increase due to growing data volumes, and the integration of more sophisticated artificial intelligence (AI) models [1]. This trend is expected to continue with the development of "the future internet", which is characterized by the increasing interconnectivity and intelligence of devices and systems [2]. Especially in the context of distributed stream processing systems (DSPSs), this growing complexity poses significant challenges for performance management. DSPSs have been used for many years, and their applications range from Internet of Things (IoT) predictive maintenance [3] to stock market analysis [4]. However, the increasing data volumes caused by the IoT combined with the integration of large AI models and the increasing demand for real-time analysis are placing additional demands on the CPU resources of stream processing systems, making it more challenging to identify and address performance issues.

While in the past, latency and throughput were the dominant performance metrics in the DSPS domain, CPU efficiency is also becoming increasingly important [5]. There is an increasing need to meet pre-defined performance objectives for the lowest possible resource input [6]. There are several drivers for this development:

- One reason is the growing popularity of the "edge computing" paradigm. Edge computing involves the processing of data at the edge of a network and closer to the actual data sources. This is an important concept because the volume of generated raw data, usually in the context of IoT, can be too large to be transmitted to a cloud data center and also because the latency requirements of the business case can be too strict [7]. However, a challenge of this paradigm is the limited CPU resources

of many edge devices. These devices are often constrained by factors such as size, power, and cost, which can limit the computational resources that are available for stream processing [8]. Therefore, ensuring CPU efficiency is critical because it allows these systems to make the most effective use of the limited CPU resources that are available [9].
- In addition, in the context of cloud computing, CPU efficiency becomes increasingly important [10]. As these systems often operate at scale to process large volumes of data and are based on a pay-per-use model, the operating cost can be significant. By ensuring that the CPU is used efficiently, it is possible to reduce the number of resources required to process a given workload and, in turn, achieve significant cost savings;
- Finally, improving CPU efficiency also results in reduced power consumption, which is a key concern nowadays because the production and use of electricity often cause the emission of greenhouse gases. Hence, optimizing CPU efficiency can help to mitigate the environmental impact of these systems [10].

To address these challenges, it is necessary to develop new approaches and techniques that deal with the performance complexity of future DSPS and provide more detailed insights into their CPU behavior. One way to achieve this is by measuring CPU consumption on the "streaming task" level. We refer to a streaming task as a conceptual and logical unit of work (LUW) within a streaming application. Hence, a streaming task may contain a series of operations; for example, the streaming task "read from Kafka" may contain a sequence of operations that include requesting the data and serializing the payload of the responses. A user should have the flexibility to customize the level of abstraction by specifying this LUW. As a streaming task definition is on a logical level, its physical representation may result in multiple physical operations that are related to this task. Hence, task-level measuring requires the measuring of each instantiation of this operation, which in the context of DSPSs requires the considering of all nodes of the cluster as well as the parallelization settings (thread-parallelism vs. process-parallelism). By tracking the CPU consumption of individual tasks, it is possible to identify bottlenecks and understand how changes to the system or application affect the internal performance behavior. This "white-box approach" is in contrast to measuring CPU consumption at the system or process level, which only provides a broad overview and may not reveal the specific causes of performance issues. In addition, task-level performance measurement allows for more fine-grained performance tuning. By understanding the specific tasks that are consuming the most CPU resources, it is possible to identify opportunities for optimization and make targeted changes to the system to improve performance. This is particularly important in the context of distributed stream processing systems, which may consist of complex and interconnected operations that can be difficult to optimize otherwise. In addition, to evaluate CPU efficiency, the CPU consumption must be set in relation to the number of processed events to obtain a full picture of how the DSPS behaves under varying workload conditions.

In this paper, we present an approach and implementation that allows for measuring of the CPU efficiency of streaming applications at the task level. This approach offers several advantages compared to previous performance approaches.

- First, it is independent of the actual stream processing engine (SPE) being used, does not require proprietary APIs, and even supports DSPSs that are based on different programming languages (e.g., Java/C). Hence, it offers a single approach for multiple systems;
- Second, the measurement toolchain does not significantly impact performance or introduce overheads, which is an issue associated with traditional profiling and tracing approaches. Therefore, our approach is even applicable to production environments;
- Finally, this approach can be adapted and integrated into any DSPS as long as it is based on a recent Linux kernel ($\geq$V3.18).

In this way, our approach allows developers and performance analysts to optimize the performance of their systems and ensure that they can meet the performance demands of

applications in an increasingly complex and data-driven world. In addition, we provide an extensive evaluation in which we assess the approach itself and demonstrate its potential by comparing the task-level CPU efficiency of three popular DSPS frameworks. It is worth noting that, while our toolchain is also capable of tracking network transmission and memory consumption at the process level, in this paper, our analysis focuses exclusively on task-level CPU analysis. While there are a multitude of other important performance factors, such as state management, scheduling, and reliability, they are beyond the scope of this paper. In summary, we make the following contributions:

1. We describe the conceptual basis of our approach and demonstrate how it can be integrated using the Yahoo Streaming Benchmark as an example. (YSB) [11];
2. We provide open-source tool support for this approach;
3. We evaluate the consistency of the yielded measurement results and the performance overheads of our approach and show that it can be used under high CPU load and in production environments without significantly distorting the results;
4. We extensively demonstrate the potential of this approach by analyzing the task-level CPU efficiency of the three popular open source SPEs Apache Spark Structured Streaming (Spark STR), Apache Spark Continuous Processing (Spark CP), and Apache Flink (Flink) in various experiments.

The remainder of this paper is structured as follows: Section 2 highlights related work in the area of performance evaluation and benchmarking of distributed stream processing systems. Section 3 describes our conceptual approach and the design of our measurement tool. Section 4 describes the testbed used for the evaluation, as well as our extensions to the YSB. Section 5 evaluates the quality of the approach itself by measuring its overheads and consistency. Section 6 provides an extensive performance analysis in which we perform task-level CPU measurements for three popular SPEs to demonstrate the potential of this new approach. Section 7 concludes this paper. Finally, Section 8 indicates the limitations of the work and presents possible future areas of research.

2. Related Work

Existing research in the performance management of DSPS is primarily research into benchmarking approaches that focuses on latency and maximum sustainable throughput. Chintapalli et al. [11] were among the first to provide a "click and run" application benchmark that presented these metrics and supported common open-source frameworks such as Apache Flink, Spark Streaming, and Storm. The YSB features an extract, transform, and load (ETL) processing pipeline, including setup automation scripts and an integrated result calculator. In their work, they benchmarked the throughput and latency of these SPEs by scaling the load from 50k to 170k events per second (e/s). Later, it was revealed that the link between Kafka and the SPE can become a bottleneck limiting the total possible throughput [12]. They did not support the measuring of CPU utilization. We enhanced their implementation and integrated our task-level measurement tool, which is discussed in more detail in Section 4. Several other approaches built upon the YSB, including Karakaya et al. [13] and Shahverdi et al. [14]. Both evaluated the performance of various frameworks based on the YSB. While Ref. [13] focused on scalability benchmarking by increasing the cluster sizes from 1 to 6 nodes, Ref. [14] extended the benchmark itself by incorporating additional frameworks such as Spark STR and HazelcastJet. Both evaluations included CPU utilization, in addition to latency and throughput; however, they did this on a global basis combined for all worker nodes. Hence, these approaches do not offer insights into the performance of individual operations.

Karimov et al. [15] benchmarked the latency and sustainable throughput of Spark Streaming, Flink, and Storm using micro-benchmarks of common streaming operations such as joins and aggregations. In this way, they could measure the performance of individual streaming tasks but only when running them in an isolated manner. The approach does not offer a technique for measuring individual operations. A novel aspect of their work was an accurate definition of latency measurement for stateful operations. They stated that

the latency of a stateful operation should be calculated based on the event time of the last event considered for the operation. We followed this recommendation and included this latency calculation in our YSB enhancement while also offering the original YSB latency in addition to ensuring that our results remain comparable with previous work.

In addition, Van Dongen et al. [16] focused extensively on fine-grained latency measurements. In contrast to [15], their setup measured the latency of individual streaming tasks. They achieved this by sending every intermediate result back to a single Kafka broker who served as the timekeeper. This way, they also achieved absolute global time in a distributed environment. While they did not use the YSB itself, their application was inspired by it and partially resembled the same pipeline. Their approach is not able to measure the CPU consumption of individual streaming tasks and induces much overhead due to additional network transmissions.

In [17], Van Dongen et al. presented the open stream processing benchmark (OSPbench). They applied different workload scenarios (stable load, periodic bursts, and overload situation during start-up) and analyzed their effect on different streaming pipelines while collecting CPU utilization and memory consumption. The work focused on determining optimal configuration settings and the maximum sustainable throughput. They extended their work in [18] to focus on cluster scalability for both horizontal and vertical scaling while looking at throughput and latency. This work is closest to our experiments presented in Section 6.2. We complement their results by looking at the task-level performance and more nuanced CPU metrics to explain the observed scalability effects. It should be noted that the OSPbench differs from the YSB implementation. In addition, the OSPbench focuses on containerized cloud deployments via Docker, while our implementation runs natively under Linux.

Kross and Krcmar [19] are closest to our measurement approach with respect to performance simulation and prediction. They presented an approach for automatically extracting stack traces, framework configuration settings, and other complementary performance metrics for Apache Spark. They provided a dedicated Java agent to extract information and transform the results into a performance model based on the Palladio Component Model [20]. In this way, they can simulate the model and make predictions on how Spark behaves under different conditions. Their approach can measure CPU consumption at the stage level and thus provide a more in-depth analysis compared to measurements at the process or host level. However, their approach was developed for batch and not stream processing. As the stages of a batch job are not identical to the tasks of a streaming application, the approach does not work for DSPS. Furthermore, the extraction approach is dedicated to the Spark framework and requires the framework's API to provide complementary metrics. It can not be used for different engines. Lastly, the use of Java profiling, which is required to extract the stack traces, causes overheads that may distort the overall measurement.

Finally, the approach in this paper builds upon our previous work [21], in which we sketched out our first idea for achieving task-level CPU efficiency analysis. In this paper, we present the following major extensions:

1. We extend the concept to support distributed systems;
2. We present the details of our technical implementation;
3. We evaluate the consistency of the measurement results and the performance overheads of the approach;
4. We integrate the prototype into an existing benchmark;
5. We perform extensive experiments to analyze the performance of three open-source streaming frameworks;
6. We provide our prototype open-source, including installation scripts that ease the setup.

In summary, most previous performance studies either focused on end-to-end latency and throughput or neglected CPU efficiency at the task level. Furthermore, many evaluations in the streaming domain were based on the YSB pipeline indicating that it is a well-studied candidate for demonstrating our task-level approach.

3. Task-Level Performance Measurement

In this section, we present our conceptual approach to measuring CPU efficiency at the task level. We first explain the enabling technologies on which our concept is based. Afterwards, we provide an overview of the components and their interplay. Finally, we elaborate on how task-level measurement works and how the scope of the analysis is defined.

3.1. Enabling Technologies

Profiling is a common technique used to measure the performance of individual tasks or operations in a system. The most common profiling approach is based on "stack trace sampling", in which stack traces of a process are collected at a specified interval, usually several times per second. While most programming languages offer dedicated profilers, another technology has gained popularity in recent years. The extended Berkeley Packet Filter (eBPF) is a Linux kernel technology that allows users to attach small programs, called "eBPF programs", to specific kernel functions to monitor and modify the behavior of the kernel. This can be useful for a wide range of purposes, including networking, security, and performance monitoring [22]. One advantage of using eBPF for profiling is that it achieves better performance compared to traditional profilers that operate in user space. This is because eBPF operates at the kernel level, allowing it to collect stack traces and other metrics more efficiently by avoiding the need for continuous system calls. In contrast, traditional profilers require these system calls, which results in continuous user/kernel space switches. Such switches are time-consuming and resource intensive. Another advantage of eBPF is that it is language-independent, meaning that it can be used to profile tasks implemented in different languages. In the context of DSPSs, this is useful because, while most frameworks are implemented in Java, some systems also run, for example, on C/C++. Hence, eBPF allows for a "single source of truth" for metrics across different languages. In addition to being able to sample stack traces, eBPF also supports the use of tracepoints, kprobes, and uprobes to measure additional metrics in kernel space. Tracepoints allow users to insert custom code at specific points in the kernel code, while kprobes and uprobes allow users to intercept function calls and returns at the kernel and user level, respectively. These features can be particularly useful in further enhancing the performance analysis. Our eBPF implementation can, for example, trace incoming and outgoing network transmissions to support CPU-efficiency calculations in situations where the number of processed events is unknown. It can check if the packages are received by or sent to an external IP address to omit local inter-process communication. Lastly, eBPF has been part of the Linux Kernel since release 3.18 and hence is an "on-board tool" that is officially maintained.

Overall, the use of eBPF for profiling offers several advantages over traditional profilers and other performance approaches. However, while stack trace sampling based on eBPF can show performance hot spots in the code, it does not retrieve the actual CPU cycles consumed. This, however, is an important metric for calculating CPU efficiency. A DSPS may run on hosts that have processors with different clock-rates. For this reason, we also query performance information via Performance Monitoring Counters (PMC). CPUs provide a number of programmable registers to count PMCs such as CPU cycles, branch misses, or cache misses at the process level. We combine these results with our stack-trace sampling, to calculate the CPU consumption at the task level. PMCs provide high accuracy and incur almost no overheads [22].

3.2. Conceptual Approach

In this section, we provide an overview of the major components of our toolchain and the steps (0–8) of how they interact with each other. As depicted in Figure 1, measurements on a host are collected via the *profiler* component. As the DSPS usually runs on a cluster of machines, the *profiler* component needs to be deployed on all hosts that belong to the DSPS. The measuring process is triggered by the central *profilingCoordinator* component (1).

The *profilingCoordinator* may run on any host as long as public-key authentication with the other cluster nodes is available. It retrieves all required information about the cluster from the *cluster.txt* file. This file, as well as the *application_profile*, has to be configured by the user in advance (0).

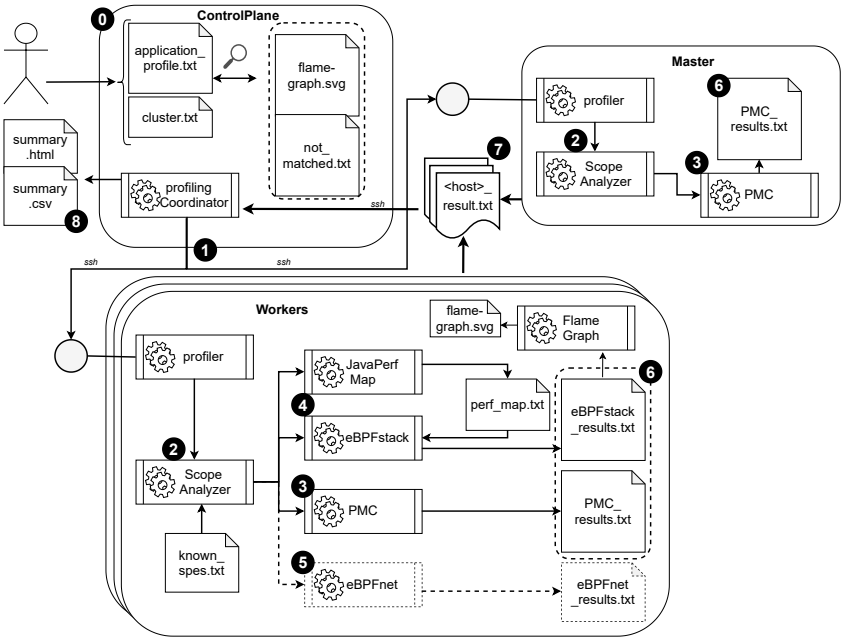

Figure 1. Conceptual architecture of the measurement tool.

The *profiler* component uses three major subcomponents, the *scopeAnalyzer*, the *eBPFstack*, and *eBPFnet*. The *scopeAnalyzer* is responsible for detecting all process IDs (PIDs) that belong to the DSPS (2). Furthermore, it is responsible for distinguishing between managing processes and workload processes. A *workload process* executes the streaming application (in Spark, this process is named Executor, in Flink, TaskManager), while the managing processes perform various supporting framework tasks (for example, in Spark, this is the Driver, while, in Flink, it is the JobManager).

The *scopeAnalyzer* identifies the relevant processes via a list of known process names, which can be configured in the *known_spes.txt* file. If an emerging streaming framework is not covered, the file can be adjusted. Furthermore, the *ScopeAnalyzer* provides the option of identifying relevant processes by tracing exec() during the startup of an engine. The *scopeAnalyzer* then passes the identified PIDs to the *profiler* component. For each PID, the *profiler* calls the *PMC* component that measures a configurable list of performance events such as CPU cycles or branch misses.

The *profiler* only starts the *eBPFstack* component via which the stack trace sampling is performed (4) for workload processes. This samples stack traces at 99 Hz using eBPF, which we found to achieve consistent performance results while causing only little overheads. If the target DSPS is based on Java, the *JavaPerfMap* also starts. This component is required for the symbol resolution of class and method names in the stack trace. Optionally, the profiler can also start network tracing with the *eBPFnet* component (5). *PMC*, *eBPFstack*, and *eBPFnet* generate an individual performance result file for every PID measured (6). After the measurement phase has been completed, the *profiler* performs a pre-aggregation in which the individual performance results of all processes are combined into a single *<host>_result.txt* file (7). During this step, the calculation of the task-level CPU consumption also takes place using the *application_profile.txt* file. To find potential keywords and the right level of abstraction, the *eBPFstack* uses the *FlameGraph* component to provide an interactive Flamegraph representation of the sampled stack traces. The Flamegraph supports filter and drill-down functionality to ease the analysis. The next section explains more about the definition of the *application_profile*. Finally, the *profilingCoordinator* fetches the results

and calculates a *summary.csv* file that contains performance results for the whole DSPS. In addition, an HTML representation of the results is generated that can be automatically written to a web server for automatic result updates (8).

3.3. Stack Trace Analysis

In this section, we present how the "profiler" calculates the task-level measurements and how the "application_profile" affects the abstraction level of the performance analysis.

A "streaming application" is composed of at least one or a finite number of "logical operations" that specify the processing steps as defined in the source code. As shown in Figure 2, these logical operations form a directed acyclic graph, where each node represents a "logical operation", and the edges represent the flow of data. A "streaming task" in our approach is a single, or a sequence of "logical operations" (subgraph), and is defined by the user in the "application_profile". The "application_profile" itself is a list of "task_names", which refer to class or method names. Each "task_name" contained in the "application_profile", for example, Filter, Project, or Kafka, defines a corresponding "streaming task", for which the performance results (e.g., CPU cycles) are aggregated by the toolchain. In other words, the "application_profile" defines the scope and abstraction level of the performance analysis. This allows the user to start the performance evaluation with a broad scope for initial analysis and allows for drilling down further by changing the specification in the "application_profile".

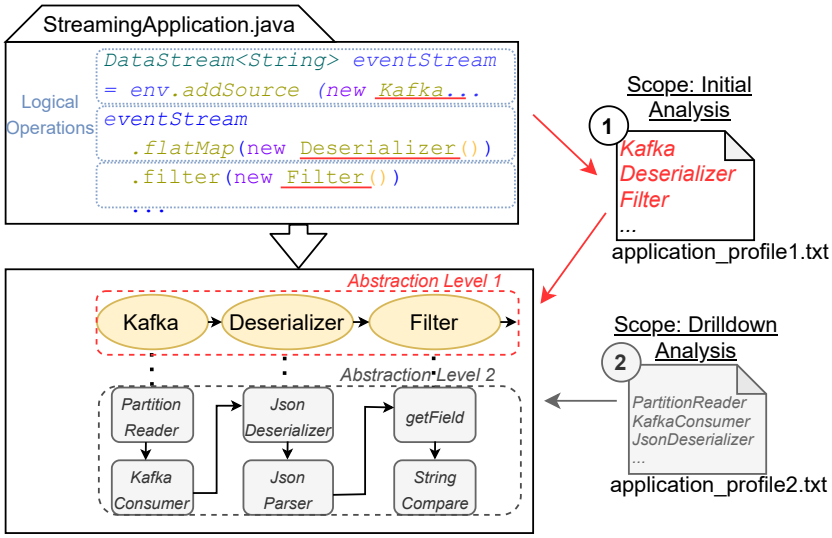

Figure 2. Scope definition.

The tool automatically determines which code paths belong to a "streaming task". As an example, as depicted in Figure 2 during the Scope "Initial Analysis", all stack traces that captured the "Partition Reader" or "Kafka Consumer" need to be assigned to the streaming task "Kafka". Therefore, the profiler component has to analyze each stack trace that is contained in the "eBPFstack_results.txt" via stack trace walking. Algorithm 1 shows the main procedure for how this is accomplished. Each streaming task is initialized with the value 0, which describes the number of samples each task profiled (2–3). Subsequently, each stack trace is transformed into the individual operations of the call hierarchy (7). We iterate over the resulting list starting with the latest operation, and check whether a matching task exists. If so, we map the samples of this trace to the associated streaming task (10–16). If no related streaming task exists, we assign the samples to the dummy task "unmatched". In this case, we also print the trace to a dedicated file for manual inspection (17–20). This way, the developer can refine the "application_profile" for further analysis. When all traces are finally processed (21), the percentage of each streaming task on the total CPU utilization is determined. However, this percentual utilization is not sufficiently

accurate because the clock-rate of a CPU can be different on the individual "hosts" of the distributed cluster. Therefore, we include the results provided by the PMC. The PMC provides us a "pmc_result_file" that contains performance statistics on the process level. It includes elements such as pid_cpu_time, pid_cycles, pid_instructions, pid_branch_misses, pid_cache_misses, and more. Developers can easily extend these metrics by querying additional events. For this concept, we only need the element "pid_cycles". These are the total_cycles of the process consumed during execution. The tool calculates the CPU cycles of each "streaming task" by multiplying the percentual share with the "pid_cycles" (22–24).

Algorithm 1: Stack trace analysis to identify the code paths associated with each streaming task and aggregate the number of CPU cycles consumed for each task

```
1  task-level analysis (stacktr, profile, pmc);
   Input  : profile contains the content of the application_profile as a list of strings,
            stacktr<trace, samples> contains a map of stack traces and the number of
            samples that were collected for each trace, and pmc contains the PMC
            results
   Output : result ← map containing the streaming tasks as key and the number of
            consumed CPU-cycles as value
2  tasks ← <profile.getElements(), 0>
3  totalsamples ← 0
4  foreach trace in stacktr do
5      samples ← stacktr.getValue(trace)
6      totalsamples ← +samples
7      operations ← trace.split()
8      assigned ← false
9      foreach operation in operations do
10         if tasks.contains(operation) then
11             task ← tasks.getTask(operation);
12             tasks.put(task, tasks.getValue(task) + samples);
13             assigned ← true
14             break
15         end
16     end
17     if !assigned then
18         tasks.put('unmatched', tasks.getValue('unmatched') + samples);
19         printf(trace) → ./umatched_stacktraces.txt
20     end
21 end
22 foreach task in tasks do
23     results.put(task, tasks.getValue(task)/totalsamples ∗ pmc.get(pid_cycles))
24 end
```

4. Testing Approach

In this chapter, we present the setup of the extended YSB and the integration of our concept. First, we start with a general introduction to the YSB and explain our extensions to the original implementation. Next, we present how we integrated the profiling approach into the extended benchmark. Finally, we present our testing approach and the environment used for the experiments.

4.1. YSB Extensions

The business scenario of the YSB [11] is an advertisement clickstream. Each event represents a user viewing an advertisement that belongs to one campaign. The streaming applications join these events to their respective *campaign_id* via a Redis key-value store

and count in 10-s windows, for each campaign, how many events were received. These results are finally written back to Redis. As depicted in Figure 3, the original pipeline was composed of seven streaming tasks (without the Enrich task). The *Kafka* task is responsible for pulling events from the Kafka broker and deserializing the byte representation into JSON strings. During the *Deserialize* task, the JSON strings are transformed into Java objects. Afterwards, the *Filter* removes all events that do not have the *event_type* property "view," while the *Projection* removes unnecessary fields such as "user_id" or "page_id." During the *Join*, the *campaign_id* is obtained from Redis. Afterwards, a *keyBy()* or *groupBy()* (not depicted) ensures that all events with the same *campaign_id* are further processed by the same worker. This is important for the *Window* task, which is instantiated for each campaign to count its number of advertisements on a 10-s basis. Finally, the result of each window gets written to Redis as part of the (*Sink*) task.

General Changes: The main reason why we chose the YSB is that it is a well-studied benchmark and features a full application pipeline. However, there are some known limitations that we had to address to make the results more representative. Kafka, as the central message broker, is prone becoming a bottleneck because it is confronted with a high ingestion rate by the producers, as well as a high pull rate by the consumers [15]. Kafka's performance is mainly dependent on its transaction log. Since persistence is of no interest for our performance evaluation, we equipped each broker with a 250 GB RAM disk. In this way, we achieved ingestion rates above 3 m e/s, for our four-broker setup, albeit without any reading consumers. In addition, we introduced the new *Enrich* task to the YSB pipeline. Its main purpose is to perform an additional latency calculation called *pre-window latency* that covers the time from event generation until the stateful processing starts (*Windowing+Sink* task). In this way, we can calculate the processing time of the windowing itself by subtracting the pre-window latency from the total latency. Furthermore, the pre-window latency shows how time-consuming the stateless part of the application is compared to the stateful part. In addition, it indicates problems with the source pipeline (Kafka or load generation). If the pre-window latency is high, while the processing time of the *Window+Sink* is low, it could indicate that the incoming events arrive with a high latency, e.g., because Kafka was overloaded. Hence, the Enrich task not only provides additional performance insights but also serves as a quality indicator for the validity of our measurements. *Redis* can become a bottleneck as well if lots of windows are being used [12]. In the YSB, every *campaign_id* translates to one window at least every 10 s. However, we only used 16k campaigns at the most, which translates to 1600 write operations per second, which is still low even for a single Redis instance.

During our Spark STR implementation, we identified that the original YSB featured a cache for Flink, while Spark had no cache. Consequently, after a short warm-up period, Flink was able to serve every join from the cache, while Spark still had to query Redis. While Spark does not support a true dynamic cache implementation, we considered it an inequality because the benchmark would artificially favor Flink over Spark. Therefore, we decided to make the implementation more even and equip Spark with a static cache that is filled once during startup. Since there are no changes during runtime, Flink and Spark will have a 100% hit-ratio after a short startup period making them equal from a performance perspective while simultaneously relieving the workload for Redis. Whether a real connection with Redis would result in a more interesting analysis is open to debate. However, as Spark STR does not support a dynamic cache, implementing no cache for both frameworks would be the only alternative.

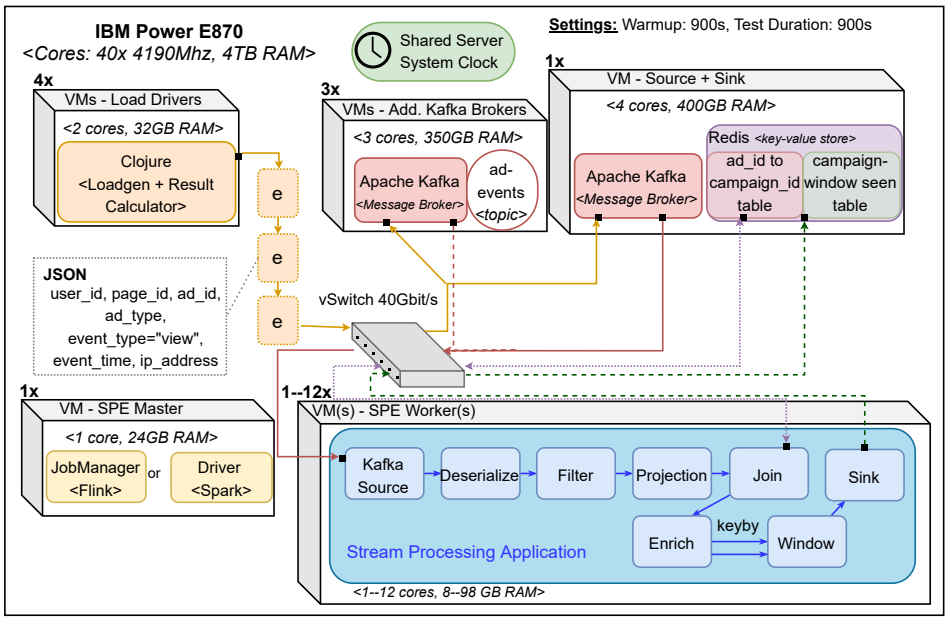

Figure 3. Benchmark and testbed.

We also noticed that the *load generation* is resource-intensive. Shahverdi et al. [14] deployed 10 generators for a peak load of 150k e/s. Therefore, we multi-threaded the load-generation so that the program uses as many threads as there are (virtual) cores available. Furthermore, the original generator wasted processing time on dynamically calculating unimportant fields, such as *user_id* or *ip_adress*. We coded these values statically since they have no effect on processing and get removed by the *Projection* task anyway.

Versions: Originally, the YSB supported the frameworks Storm (v0.9.7), Flink (v1.1.3), and Spark Streaming (1.6.2). Flink matured significantly since 2016 and has introduced many new features. For example, the original YSB did not implement native Windowing but used a custom implementation instead. We reworked the Flink implementation to a newer version (v1.14.3) and supported native Windowing based on *event_time* and *watermarks*. Although still widely used, we did not include Storm and Spark Streaming. Instead, we provide an implementation of the more modern Spark STR (v3.2.0) that solves many of the problems of the original Spark Streaming API, getting closer to the idea of a true streaming engine. Most importantly, Spark STR no longer requires a configured micro-batch interval but instead determines its micro-batch duration dynamically based on processing time.

In addition, we also provide an implementation of Spark STR's experimental *Continuous Processing Mode*. Spark CP performs true record-at-a-time processing. To the best of our knowledge, we are the first to provide such an implementation for the YSB pipeline. However, it should be noted that Spark CP comes with several limitations. Most importantly, it does not support aggregations (*windowing* task) or re-partitioning to different nodes (*groupby()* or *keyby()*). For this reason, we needed to change the application semantic as follows. For every event, we calculate, based on the *campaign_id* and the *event_time*, to which logical window (*time_bucket*) the event belongs. As displayed in Figure 4, we use Redis as a state store and a HashMap as a cache, which is periodically flushed to Redis. While this workaround provides the same semantic as a true windowing operation, there is no such solution for the missing *groupby()* operation. Consequently, multiple workers process the same *campaign_id* for the same 10 s *time_bucket*. Hence, for each logical window, we obtain a number of partial windows that is equal to the number of worker nodes. To resolve these partial windows, we need to aggregate them, which is carried out by the result calculator. This different processing logic affects the performance in the following way. First, depending on the number of workers, Spark CP requires significantly more windowing operations, which puts more pressure on the *Sink* task. Second, for each new partial window, we have an additional read operation to Redis, which again is an overhead.

Lastly, there is no *groupby()* operation and consequently no event redistribution between the workers, which is a significant communications savings. The remainder of the Spark CP implementation is identical to Spark STR. CP mode is activated by calling the "continuous" processing trigger. The framework itself handles all the changes that are required. This makes it especially interesting to compare STR with CP at the task level because it induces fundamental efficiency changes, as we see for *Kafka* in Section 6.1.

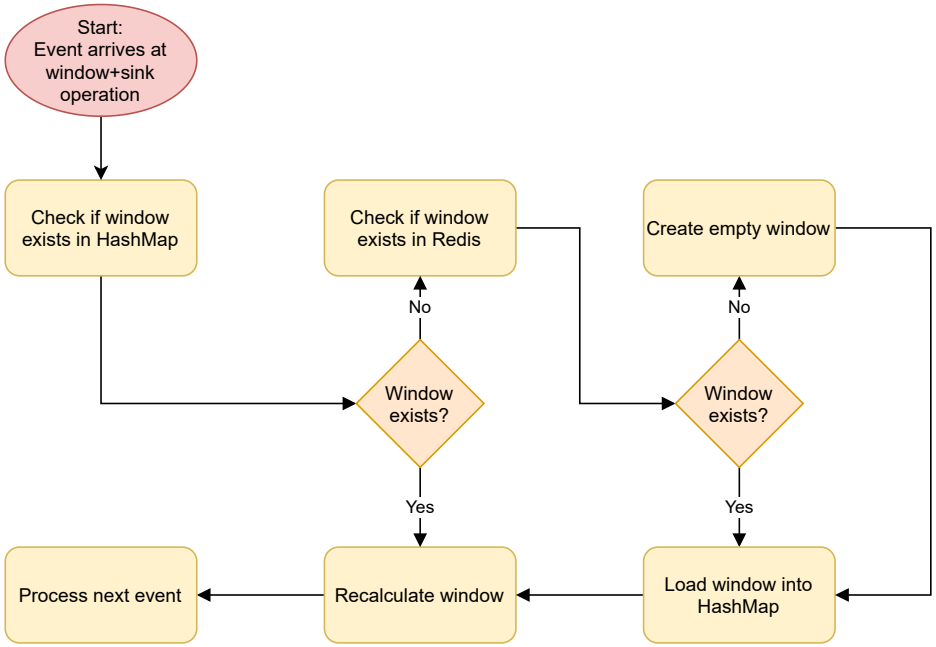

Figure 4. Spark CP windowing implementation.

4.2. Testbed

Our testbed configuration is displayed in Figure 3. All VMs run Ubuntu 20.04 LTS and are deployed on an IBM Power E870 Server with 40 cores (4190 Mhz) and 4 TB RAM. Due to the single server deployment, all VMs are connected via a virtual switch that is provided by the hypervisor. This setup ensures that we do not run into bandwidth saturation and that no unpredictable latency effects are caused by the network, which is a common issue in performance research [23]. Similarly to [16], we aim to minimize time skew between our VMs to achieve good latency measurements. However, we take a much simpler approach. As all VMs are located on the same server, they share the same hardware-clock, which is read during boot, minimizing any skew as long as the cluster is restarted regularly. We use four VMs as Kafka brokers and equip each one with a 250 GB ram disk to ensure fast transaction speed. Four VMs are used as load generators. Up to 12 physical cores distributed on up to 12 VMs are used as workers. On IBM Power8, the SMT setting can be dynamically changed between ST, SMT2, SMT4, and SMT8, which translates to 1–8 vCPUs per physical core. We set the SMT setting of all Workers to 4 since this resulted in the best overall throughput and latency.

5. Approach Quality

While eBPF profiling has vast performance advantages over traditional profiling, it is still possible that the results are distorted. For this reason, we first check how much overhead is caused by eBPF profiling. In addition, the timed sampling of the profiler may result in variances. To evaluate the quality of the performance data, we need to assess the consistency of the results.

5.1. Profiling Overheads

Our goal is to measure the overheads caused by eBPF profiling and classical profiling and compare their results with the non-profiled baseline. We use a two-node cluster with one core per VM and focus on a high-load scenario. Since profiling is a CPU-intensive activity, we need to test it under high CPU utilization to see how it affects the performance of the SUT. To find the highest stable throughput, we increase the load level in steps of 10k events per second (e/s) and select the highest load level that does not cause instability (latency within 15 s). For Spark STR and CP, we determined a sustainable load of 240k e/s and for Flink 300k e/s. We use eBPF profiling as our main method and the Java profiler "JPROFILER" [24] as an alternative method for comparison. We set the sampling rate of JPROFILER to 1 ms and do not apply any profiling filters. Except for these changes, we used the default settings. There may be other settings that could optimize JPROFILER's analysis and reduce its overheads.

Table 1 shows the latency of all engines with and without profiling. The results indicate that eBPF profiling has much lower overheads than traditional Java profiling. For Flink and Spark CP, eBPF profiling increased the average latency by 3.1 ms (+3.9%) and 68.6 ms (+2.4%), respectively, while Java profiling caused unsustainable processing and increased the average latency by ~75 s and ~103 s, respectively. For Spark STR, eBPF profiling increased the average latency by 1.8 s (+18.7%), which is higher than for the other two engines, but still lower than Java profiling which increased the average latency by 5.7 s (+37%). However, this increase in latency may be partly due to other factors such as micro-batch size and CPU utilization. We observed that both eBPF and Java profiled applications had lower CPU utilization than non-profiled applications (2.6% and 5.1% less, respectively), which suggests higher efficiency. Spark STR achieves higher efficiency by increasing the micro-batch size, which also increases the latency. Therefore, we cannot attribute all the latency increase to eBPF profiling alone. In summary, our experiment demonstrates that eBPF profiling has superior performance over traditional Java profiling even under high-load scenarios. Thus, our approach provides a valuable addition to existing profiling methods.

Table 1. Performance Overhead of BPF and JPROFILER.

		AVG_LAT	AVG_PRE_WNDW	Processed Events	CPU-Util
Flink	No Profiling	80.1	1.2	134,892,334	89.0%
	BPF Profiling	83.3	1.0	134,896,023	90.8%
	Diff-BPF	3.1	−0.2	3690	1.8%
	JPROFILER	75,978.8	75,276.7	102,057,612	76.9%
	Diff-JPROF	75,898.7	75,275.5	−32,834,722	−0.1
Spark STR	No Profiling	9661.8	6100.7	107,898,950	80.4%
	BPF Profiling	11,472.6	7497.4	107,941,936	77.8%
	Diff-BPF	1810.8	1396.7	42,986	−2.6%
	JPROFILER	15,364.9	9935.9	108,028,000	75.3%
	Diff-JPROF	5703.1	3835.2	129,050	−5.1%
Spark CP	No Profiling	2784.1	5.5	107,908,636	91.8%
	BPF Profiling	2852.7	8.3	107,791,544	92.0%
	Diff-BPF	68.6	2.9	−117,092	0.2%
	JPROFILER	106,657.0	97,112.4	80,136,000	92.0%
	Diff-JPROF	103,872.9	97,106.9	−27,772,636	0.2%

As displayed in Table 1, the latency for all engines increases when using profiling. For Flink, it increases by 3.9%, for Spark CP 2.4%, and for Spark STR 18.7%. For Spark STR,

this influence seems quite high; however, it could also be caused by other effects. For this reason, we also took the process's CPU util. into account. Looking at the utilization, we see that the profiled application took 2.6% less CPU and therefore had higher efficiency. Spark STR achieves this typically by setting a larger micro-batch size, which in turn increases latency. Hence, the increased latency can at least partially be explained by this. Considering that this reflects a worst-case scenario, the overheads caused by the profiling are acceptable.

5.2. Profiling Consistency

Next, we check if the yielded results remain consistent. In this experiment, we measure on a single-node with one core to prevent variances due to uneven load balancing. For each SPE and load level, we perform five measurement runs and plot the standard deviation. As displayed in Figure 5, the standard deviation of almost all tasks is below 3%. Only with Spark did we observe high deviations for the *Waiting* task when using low load levels (20k and 40k).

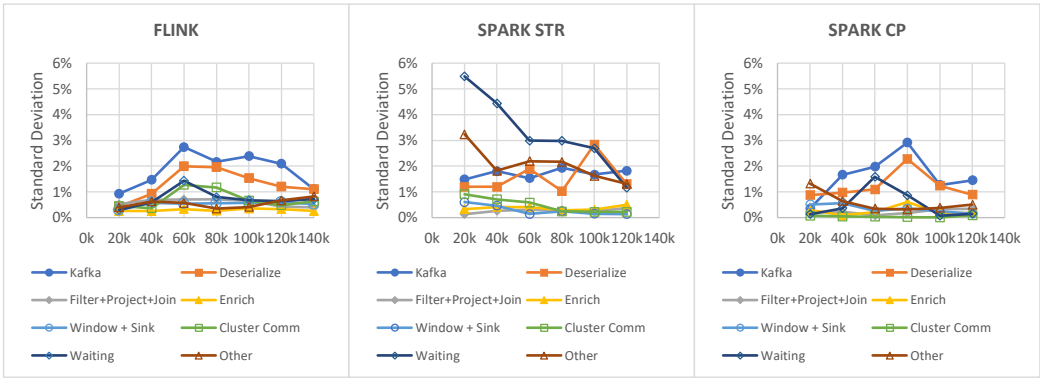

Figure 5. Profiling consistency.

To check if these variances do indeed result indeed from profiling randomness, we took the deviations of the average latency (LAT), average pre-window latency (WNDW), processed events (Events), and CPU utilization (normalized to 100% = util./vCPUs) into account. As shown in Table 2, each measurement run itself suffers from deviations (independent of the profiling). This is especially evident when looking at the CPU utilization for Flink and Spark CP. Based on this finding, we conclude that variations in the profiling results can be partially explained by actual differences in the processing. Overall, except for the Spark STR 20k and 40k run, the profiling yields quite stable results and is adequate for our performance evaluation approach.

Table 2. Standard deviations per load level.

	FLINK				SPARK_STR				SPARK_CP			
LOAD	LAT	WNDW	EVENTS	CPU	LAT	WNDW	EVENTS	CPU	LAT	WNDW	EVENTS	CPU
20,000	20.4	0.5	1399	0.1%	176.8	10.8	1673	0.3%	908.6	0.1	1089	0.1%
40,000	7.8	0.0	3744	0.8%	69.9	34.8	5021	0.1%	777.4	1.2	2887	0.8%
60,000	9.9	0.0	9773	3.0%	119.7	30.8	5170	0.2%	454.9	2.3	410	3.0%
80,000	4.5	0.7	8106	3.5%	238.3	56.4	15,485	0.1%	124.3	0.1	8485	3.5%
100,000	2.8	0.1	11,150	3.2%	496.4	128.0	6583	0.2%	237.1	4.5	13,165	3.2%
120,000	6.6	0.6	21,167	4.2%	404.4	212.0	13,161	0.2%	539.0	48.3	5111	4.2%
140,000	3.4	0.8	14,594	4.5%	-	-	-	-	-	-	-	-

5.3. Advantages over Micro-Benchmarking

To the best of our knowledge, our approach is the first work that can be integrated into any DSPS for measuring CPU efficiency at the task level. Nevertheless, there are other approaches that are able to benchmark the CPU usage of individual tasks. The most common baseline approach for this is micro-benchmarking, which, however, has some limitations.

- First, our approach is not limited to testing a single isolated task but can be applied to complete application pipelines, providing a more comprehensive picture of system

performance. This is particularly important in the context of distributed stream processing, where interactions between different tasks and components can have a significant impact on overall performance;
- Second, our approach is designed to be applied to already existing, potentially productive applications. This means that it can be used for monitoring and analysis purposes without the need for a dedicated performance testing environment;
- Third, the approach enables a more holistic picture of system performance by incorporating other context metrics. These include metrics such as IPC value or cache misses, which can provide valuable insights for root cause analysis. The approach manages to embed the measured performance in an overall context to better understand the performance of the streaming application;
- Fourth, our approach requires less effort than micro-benchmarking, as only a single run is required to analyze all tasks of the application. In contrast, micro-benchmarking requires individual measurement runs and customized application pipelines for each operation;
- Finally, our approach allows wide customization via the application profile. This way, developers can start at a higher level of abstraction and gradually refine the scope to gain more granular performance insights.

6. Evaluating Task-Level CPU Efficiency

In this chapter, we demonstrate the potential of task-level CPU measurement by performing three experiments. We do not claim to obtain optimal results for each SPE, as this would require extensive fine-tuning at the application and configuration level. Our experiments consider the following five factors:

1. Load level: Configured as e/s and per node (10k steps; 10k–150k);
2. SPEs: Flink, Spark STR, and Spark CP;
3. Nodes: Number of worker nodes (1, 2, 4, 6, 12);
4. CPUs: Number of cores per node (1, 2, 4, 6, 12);
5. State: Number range of *campaign_id* (100, 1k, 2k, 4k, 8k, 16k).

Given our requirements of using five repetitions per configuration, the experiments required for a full factorial design in accordance with [25] would be:

$$n = 5\prod_{i=1}^{k} n_i = (5) \times (15) \times (3) \times (5) \times (5) \times (6) = 33{,}750 \quad (1)$$

Since one measurement takes at least 900 s, such a design is not practical. For this reason, we choose to use a fractional factorial design. We split our analysis into three experiments and select a subset of factors for each. First, we look at load scalability to answer the question of how increasing load affects the performance behavior of every task in the YSB pipeline. Next, we compare scale-up and scale-out configurations and measure how this affects the CPU efficiency of each task. Finally, we look at state scalability and measure how the performance of each task is affected when increasing the state.

6.1. Load Scalability

For this experiment, we use two worker nodes, each equipped with one CPU core, and use the default number of campaigns (100). Hence, the factors for this experiment are (1) load and (2) SPE. We scale the load in 20k steps from 20k e/s to 300k e/s as long as the throughput is sustainable. The results displayed in Figure 6 show the average CPU cycles that are required both in total, and to process a single event (total cycles divided by the number of processed events) for the first four tasks of the YSB pipeline. Since these values represent the resource consumption of one node, we normalized the applied load relative to one node (10k–150k e/s).

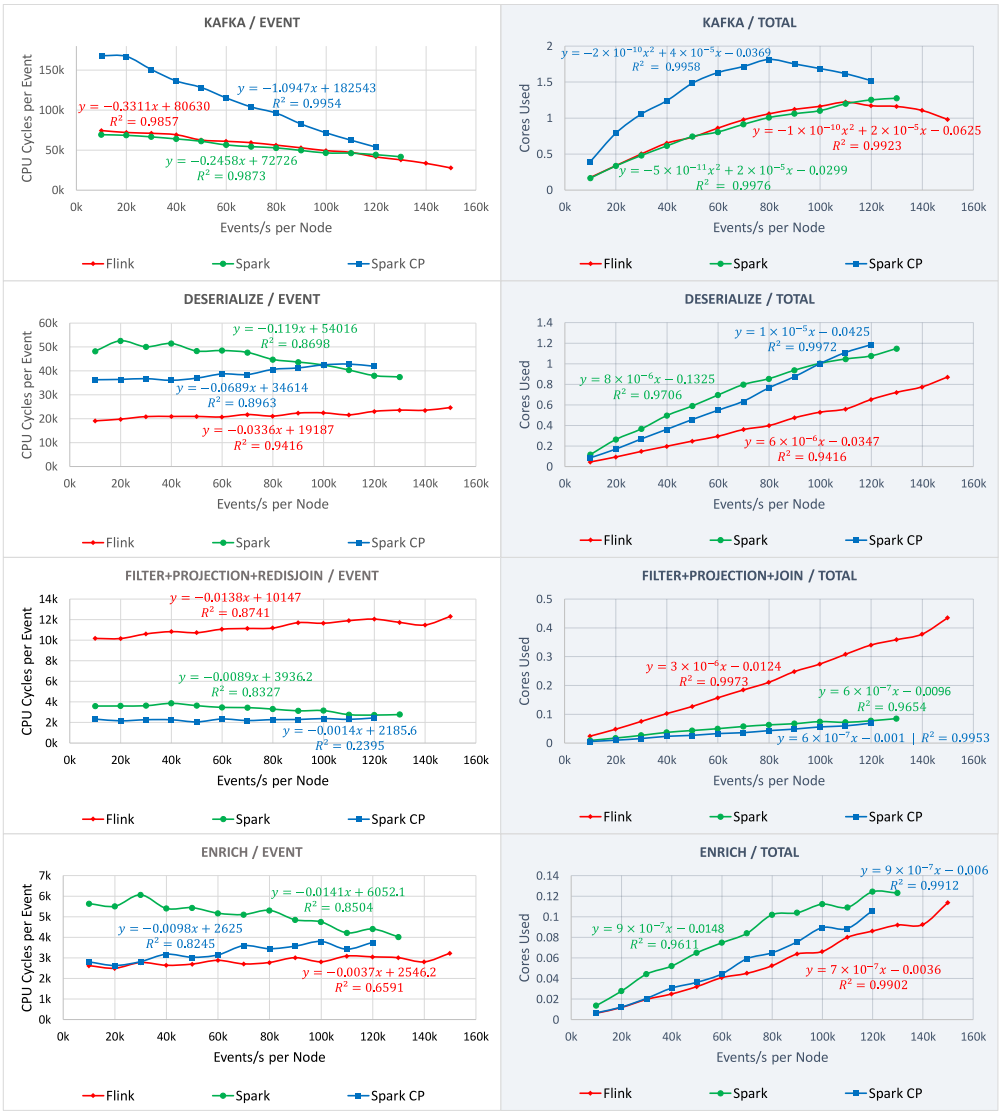

Figure 6. CPU consumption per task when scaling the load-part1.

Kafka: For all three engines, the *Kafka* task requires the most CPU resources. This task fetches the individual records from the Kafka broker, checks the offset, and splits the data by partition. For all SPEs, we observe that the CPU cycles per event fall linearly with increased load, meaning that the efficiency increases (superlinear scalability). While Spark STR and Flink require a similar amount of CPU resources, Spark CP starts with a considerably higher consumption. Spark's CP mode uses the *KafkaContinousPartitionReader* internally, whereas Spark STR uses the *KafkaMicroBatchReader* that fetches a higher number of records in a single request and hence provides better resource efficiency. Interestingly, despite Flink being a record-at-a-time processing engine, it appears to use a micro-batch approach in its Kafka consumption similar to Spark STR, which results in nearly identical processing efficiencies, presumably at the cost of higher latency. However, looking at the latency in Figure 7, it is apparent that Flink's latency is very low (<104 ms), achieving excellent latency results and efficient processing at the same time. It should be noted that, with version 1.14, Flink has introduced a new *KafkaSource* package which supersedes the old *KafkaConsumer* implementation, hence these results are only valid for newer versions of Flink. For Spark CP, it should be checked if a micro-batch approach such as Flink is possible without significantly increasing the latency because its current processing costs are quite high.

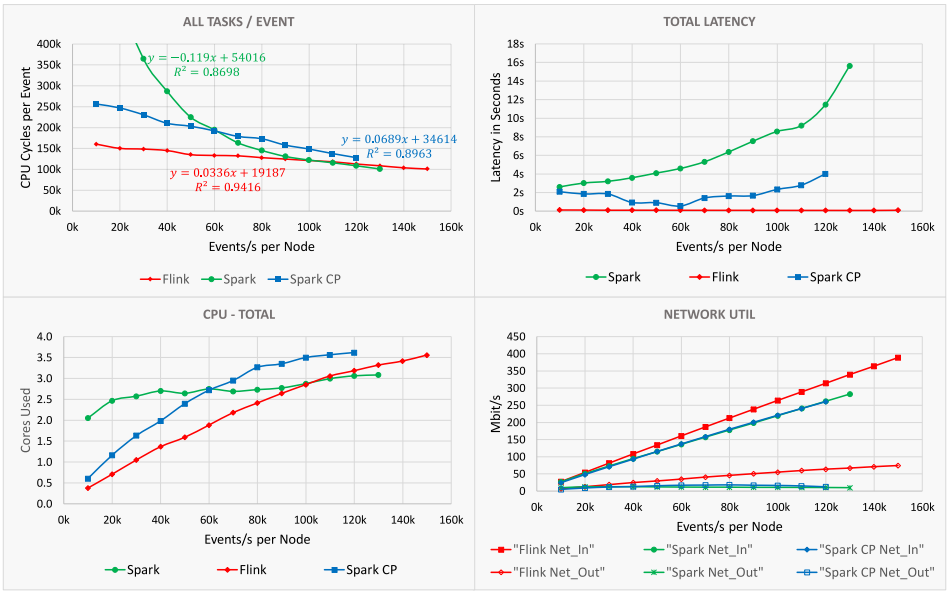

Figure 7. Total CPU and network consumption when scaling the load.

Deserialize: Deserialization is a very common task for stream processing systems [26]. The *Deserialize* task of the YSB parses the received JSON strings of Kafka into Java objects. Such parsing is known to be highly CPU bound [27] and also, in the YSB, deserialization accounts for the second highest CPU consumption. For Flink and Spark CP, the processing effort per event remains quite constant. Since every event is treated individually, no efficiency gain takes place when increasing the load. For Spark STR, we observe a linear decrease in CPU cycles per event (super linear scalability), which is caused by the larger batch sizes that can be processed more efficiently. However, Flink is about three times as effective in this regard and likewise Spark CP is twice as efficient and only gets outperformed at about 100k e/s (high load).

Filter+Projection+Join: For better comparability among the frameworks, we grouped the tasks *Filter*, *Projection* and *Join* together. While *Filter* and *Projection* are typical map operations, the *Join* is implemented as a RichFlatMap function that uses a cache to store the *ad_id* to *campaign_id* mapping. Due to our long warm-up period of 900 s, the cache of all SPEs results in a hit ratio of 100% during the measurement, which makes actual communication with Redis obsolete. It can be seen that Spark STR and CP require fewer CPU resources than Flink. This is largely due to Spark's Dataset API, which treats each event as an appended row to the internal table structure. For this reason, Spark's SQL execution engines benefit from these classical SQL operations. Interestingly, the record-at-a-time processing mode of Spark CP works even more efficiently than the micro-batch approach of Spark STR, independent of the load level. In addition, we see quite different scaling behaviors. For Flink, the number of cycles required to process a single event increases slightly, whereas, for Spark CP, it remains constant. For Spark STR, the efficiency increases with higher load levels but is not able to outperform CP.

Enrich: The Enrich task is again a common map operation that adds an additional latency field to each incoming event. For the calculation, the event time is subtracted from the current system time. For Flink, we observe that, due to the *record-at-a-time* processing, the CPU consumption per event remains quite constant as each event is treated individually. The same effect might also be expected for Spark CP, but instead it is observed that the number of CPU cycles increases slightly. Spark STR's micro-batch approach starts with worse efficiency in low load situations and improves with higher load, as is the case for the *Deserialize* task. Overall, Flink provides the highest efficiency followed by Spark CP, while Spark STR does not outperform the others at any load level.

Windowing + Sink: Figure 8, depicts the second part of the YSB pipeline, which starts with the combined *Windowing+Sink* task. *Windowing* is a core task in stream processing

applications [28] and can either be performed incrementally or in full. Flink's YSB window is based on an incremental aggregation function, which evaluates every event as it arrives. In contrast, Spark STR uses a full evaluation function due to its micro-bench approach. A full evaluation first collects events and then waits for a trigger to process them all at once. The advantage of an incremental window is that the state remains small, whereas the full evaluation requires less computation. Every ten seconds, a *window* is instantiated for each *campaign_id* (jumping) and counts the sum of sighted events. After 10-s, the aggregated results of each window are written to Redis (*Sink*). As already explained in Section 3.1, Spark CP does not support stateful operations. Hence, CP uses Redis as its state store, meaning, for every event, CP reads its corresponding *Window+campaign_id* combination either from the sink or an internal HashMap. For this reason, we combined both tasks to be able to compare them with the other SPEs.

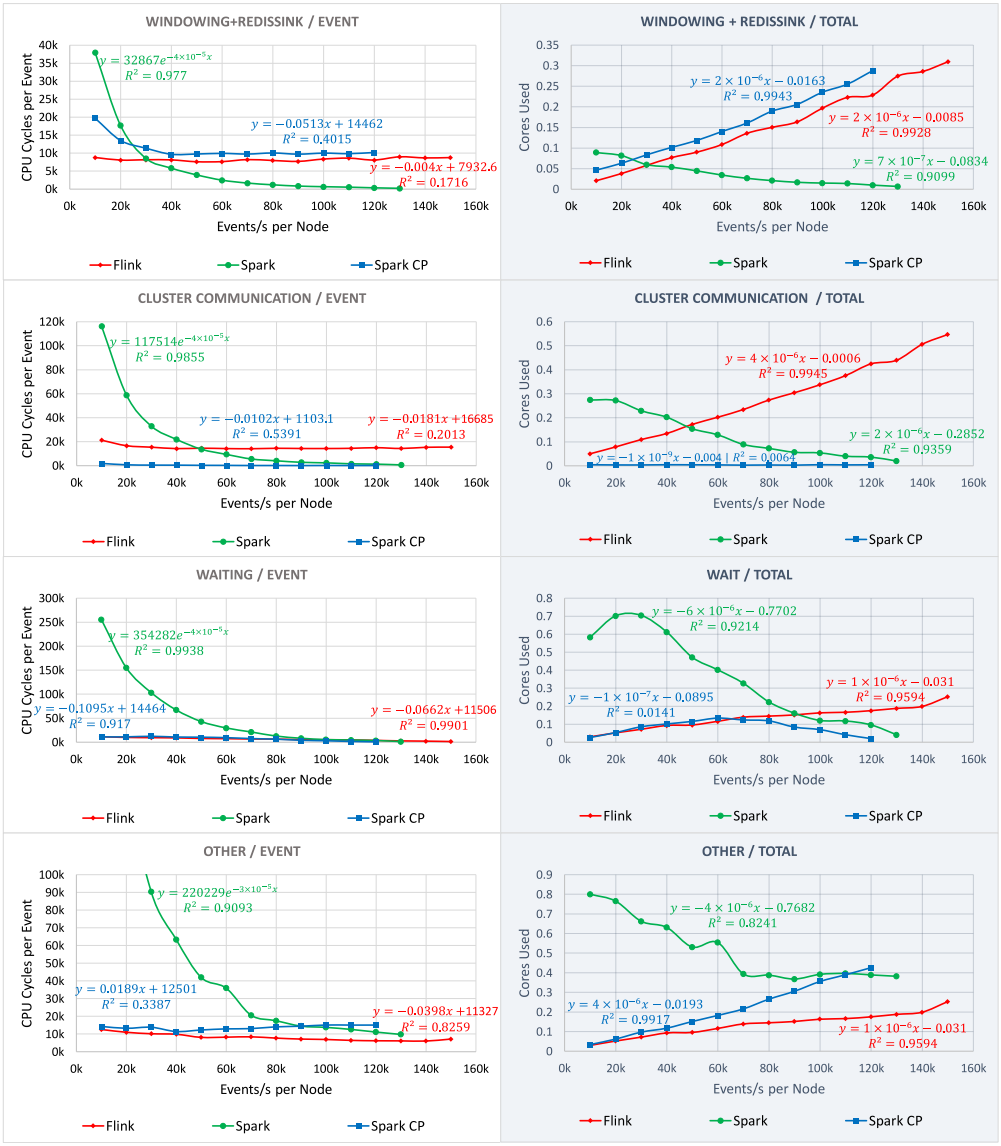

Figure 8. CPU consumption per task when scaling the load-part2.

For Flink and Spark STR, we can split up the task into its *Sink* and *Windowing*, as displayed in Figure 9:

- *Windowing*: For Flink, the CPU costs remain constant, while, for Spark STR, it decreases. We explain this as follows. Despite the increased load, the number of windows is only affected by the range of *campaign_ids* (default: 100), which remains the same

throughout the different load levels. Nevertheless, Flink has to process every event individually and hence does not benefit from any efficiency gains. Spark STR, on the other hand, increases its efficiency through its use of micro-batches, which leads to fewer window calculations (full evaluation function);

- *Sink*: The costs for both SPEs, Flink, and Spark STR decrease. This is again due to the number of windows that remain constant (1× window for each *campaign_id* every 10 s). These windows are in turn directly transformed into write operations. Hence, the actual number of events received by the *Sink* task is not determined by the load, but by the windows, which is always 10 e/s. In summary, the selectivity of the *Windowing* task also increases with increasing load levels [27], resulting in higher CPU efficiency for both frameworks.

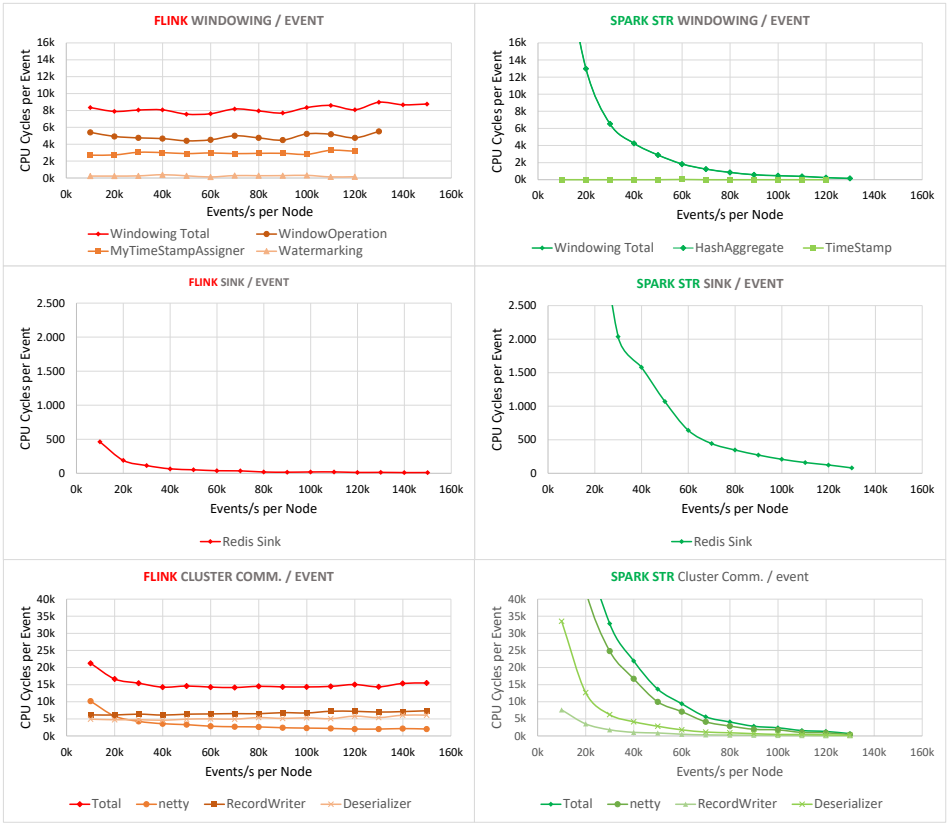

Figure 9. Drilldown: window, sink, and cluster communication.

As displayed in Figure 8, Flink's *Sink* efficiency gains do not affect the combined *Windowing+Sink* task since the write operations make up less than 1% of the windowing costs. This is no surprise given that, for a duration of 10 s, a load level of 150k e/s will result in 1.5 m e/s to be processed by the windowing task, while only 100 events are written to Redis (for Spark, this depends on the micro-batch size). Overall, Flink starts to be outperformed by Spark STR at about 40k e/s. (mid load), although Flink's *Sink* task is more efficient. For this reason, we will check in Section 5.3 if this remains true when increasing the range of *campaign_ids*, which will also increase the number of write operations. Our assumption is that Flink will outperform Spark STR in high- and mid-load situations. Spark CP is almost able to keep up with Flink and scales similarly to it. It is outperformed by STR at about 30k e/s.

ClusterCommunication: While the first generation of SPSs was single-node systems that only allowed a scale-up approach [29,30], modern systems are designed to distribute their work among several workers. This, however, comes with increased overheads. The task *Cluster Communication* covers all data communication between the different workers, which happens, e.g., during data repartitioning based on a *keyby()* or *groupby()* operation.

For example, in the context of the YSB, we need to ensure that all events with the same *campaign_id* are processed by the same node before it gets aggregated by the *Window* task. Flink's *Cluster Communication* CPU costs remain constant on a per-event basis. We found this surprising because Flink utilizes an intricate control flow mechanism that improves CPU efficiency in high-load situations via buffering. For example, during data transmission via the *netty* framework, it uses bigger payloads that decrease the protocol overheads on a per-event basis [31]. Hence, we expected that *Cluster Communication* in general should decrease and indeed, looking at the CPU cycles of netty (Figure 9), we can see that the efficiency considerably increases. However, we also accounted for serialization (RecordWriter) and deserialization (RecordDeserializer) in the Cluster Communication task. Both operations increase on a per-event basis and even the performance gains of netty. Overall, the communication costs of Flink are higher than those of Spark STR/CP. For Spark STR, the CPU costs of *Cluster Communication* fall with increased load and it outperforms Flink at about 50k e/s. Spark's network communication is also based on the *netty* framework with similar scalability to Flink. However, in contrast to Flink, the (de-)serialization's costs also decrease. This is not only due to the micro-batch approach, but also because Spark requires less network traffic overall, as shown in Figure 7.

Wait: The *Wait* task includes *pthread_cond_timedwaits* and *epolls* that typically occur when the execution engine is blocked until a condition is fulfilled. For example, for Spark STR, e.g., this primarily happens as long as the SPE is waiting for the previous micro-batch job to complete. Spark STR always attempts to process batches as soon as possible. In contrast to Spark Streaming, there is no fixed batch size or batch duration. For all engines, the waiting decreases with increased load levels. We noticed that, for Spark, the initial waiting effort is extremely high. This behavior is also reflected when looking at Spark's total CPU consumption. Figure 7 confirms that Spark already consumes 50% of the total CPU when only processing 10k e/s, which is similar to the observations in [14,15].

Other: All stack traces that were not assigned to one of the previous tasks remain in the *Other* task. Examples include GC, Safepoints, and CodeGeneration (Spark), among others. For Spark STR, the CPU consumption of the *Other* task is significant during low-load scenarios but drops exponentially with higher workloads. Again, this shows that Spark STR has high base overheads. On a per-event basis, the costs for Spark CP remain constant, while, for Flink, they are slightly decreasing.

Total: Looking at the overall statistics, as shown in Figure 7, we see that, for the entire YSB pipeline in the 10k to 100k range, Flink provides both the best CPU efficiency and the best latency. After 100k e/s, Spark STR outperforms Flink with regard to CPU efficiency, although this comes at the cost of higher latency. During low-load situations, Spark STR has considerably higher CPU costs than the other two frameworks, which was already observed by [15]. Spark CP offers a good trade-off between latency and CPU costs as well, and it outperforms Spark STR in both areas in the load range of 10k to 60k e/s. Looking at the network utilization, we observe that Spark STR and Spark CP require almost identical network resources even though Spark CP uses more window operations and Redis connections. However, this effect seems to be marginal at only 100 campaigns. On the other hand, Flink requires significantly more network resources, which is also visible in the cluster communication costs. This could indicate that Spark STR provides better scale-out capabilities than Flink, which we will analyze in Section 5.2.

CPU Stalling: Thus far, we have looked at the total number of cycles spent on a task. However, due to the PMC metrics, we can split these cycles up further into **stalled** and **retired** cycles. Stalling occurs whenever the SPE runs on the CPU (not idle) and has work to perform, but the processor is not able to make any progress, e.g., due to cache misses or bad branch prediction [32]. Hence, the distinction between "stalled" and "retired" cycles allows further consideration of whether efficiency differences are due to an SPE actually having more work to do, e.g., less efficient code semantics (retired cycles), or whether the resource environment (CPU architecture, memory subsystem, ...) is not well aligned to the SPE and would, for example, profit from bigger caches. Looking at the retired cycles

as displayed in Figure 10, we see that, during low loads, Flink actually has less work to perform than the other SPEs. At 120k e/s, the required number of retired cycles for Flink and Spark STR is equal; however, at this point, Flink is stalled more frequently than Spark STR, which explains why Spark provides better efficiency at this stage. Interestingly, Spark's CP mode requires fewer retired cycles and has less stalling compared to STR during low load. In summary, the efficiency of all SPEs increases with higher load, as shown by the increasing IPC value that profits from dropping *page faults* and *branch misses*.

Figure 10. CPU stalling behavior when increasing load.

6.2. Scale-Up vs. Scale-Out

Scalability is the system's ability to increase its processing capability by spreading the workload across additional resources such as cores (scale-up) or nodes (scale-out) [33]. Both approaches have advantages and disadvantages. Usually, scale-out approaches induce overheads due to increased communication effort, while scale-up is considered to be less cost-efficient [34]. However, especially when using a JVM-based SPE, the performance of a scale-up approach may also degrade due to increased state management and, thus, e.g., more frequent GC cycles [35]. For modern SPEs that are designed to run in a distributed manner and allow high degrees of parallelization, scalability is of particular importance. Mc Sherry et al. [36] showed that many performance evaluations of Big Data systems were scaled to such an extent that the actual performance gain was consumed by the parallelization overheads that the scaling induced. In addition, Van Dongen et al. [18] analyzed the throughput scalability of four SPEs by scaling the system in both directions,

horizontally and vertically. In this experiment, we do a similar evaluation but again focus on the individual streaming tasks and how these scale with regard to CPU efficiency.

We compare six different cluster configurations with varying numbers of nodes and CPU cores assigned. However, in total, we always use the same number of resources. All configurations use 96 GB RAM and 12 physical hardware cores (smt4 = 48 virtual cores) in total, but we distribute them across up to 12 nodes. For every configuration, '**N**' represents the number of nodes, whereas '**C**' C denotes the number of cores per node, and we allocate 8 GB of RAM per core. For example, 'SCALE_**N**4_**C**3' translates to four nodes with three physical cores and 24 GB RAM per worker. The 'Core Used' axis represents vCPUs as seen by the operating system. We configured a fixed load of 600k e/s. Unfortunately, the Spark CP implementation was unsustainable using more than two nodes. This is why we only included the configurations for one and two nodes. The factors for this experiment are (1) SPE, (2) cores, and (3) nodes.

Kafka, Deserialize, Filter+Projection+Join, Enrich: As shown in Figure 11, Flink's CPU efficiency is almost not affected by looking at these tasks. The only exception is Kafka which benefits from using single-node or 2-nodes. Spark STR shows a "bell-shaped" efficiency curve, meaning it benefits from either a few- or many-node cluster. This effect was also confirmed by Awan et al. [35] when evaluating Spark as a Big Data platform (batch-processing). Interestingly, the IPC values, as displayed in Figure 12, show an almost exact inverted "bell-shape" curve (U-shape), so the efficiency decrease for mixed scaling approaches can at least partially be explained by the worse IPC efficiency. Furthermore, all tasks have a small efficiency outlier at four nodes for both Spark STR and Flink. A similar effect was observed in [18], which measured the worst scalability factor for four workers in their *aggregation pipeline*. Spark CP's CPU costs increased in a similar way to Spark STR when switching to a two-node cluster. However, with regard to Kafka, the costs increased by about 260%. This indicates why increasing the cluster size further no longer yielded sustainable results.

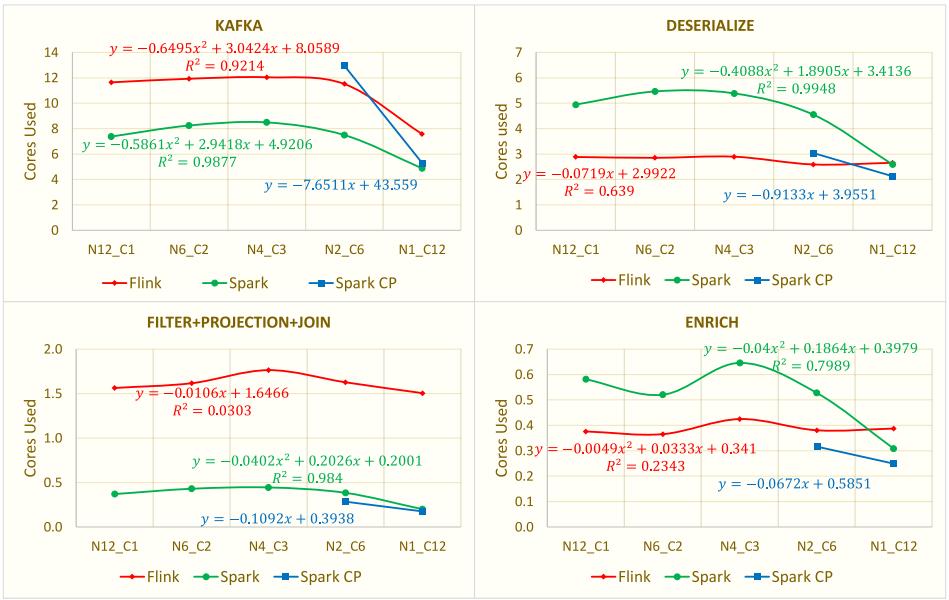

Figure 11. *Cont.*

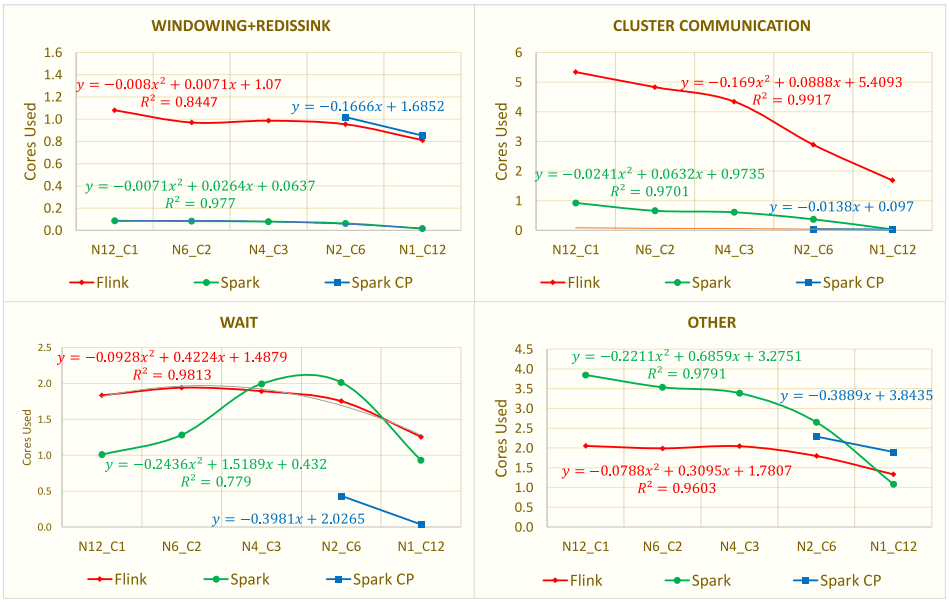

Figure 11. CPU consumption per task and cluster configuration.

Figure 12. CPU stalling behavior for different cluster configurations.

Windowing+Sink: This task benefits from a single-node deployment as well, but is otherwise hardly affected by the number of scale-out nodes. For Flink, we observe a small cost increase when switching from 6 to 12 nodes, while this has no impact on Spark STR. Overall, the *Windowing+Sink* task scales well with additional worker nodes.

Cluster Communication: As expected, the *Cluster Communication* task is heavily influenced by the number of nodes. For Flink, we observe that the communication costs increase

linearly with each worker. This is due to the fact that every *taskmanager* of Flink requires a logical connection with every other *taskmanager*, hence the number of communication channels increases. The level of thread-parallelism on the host itself, however, does not impact the performance because, as long as every thread is assigned to the same JVM, all communication is multiplexed over one channel [31]. In our experiment, we achieved a high linear correlation between communication costs and the number of nodes used ($R^2 = 0.9957$). For Spark STR, the communication costs for the scale-out approach increase as well. However, compared with Flink, these costs are much lower. This indicates that Spark STR has better scale-out capabilities compared to Flink, which can also be observed when looking at the total network utilization, as displayed in Figure 13.

Wait+Other: For all engines, the cycles spent on *Wait* and *Other* increase when scaling from one to two nodes. Adding even more nodes hardly affect the performance of Flink, while, for Spark STR, the *Wait* costs decrease. In contrast, the CPU demand for the *Other* task increases with higher node counts. This is mostly due to increased GC cycles (+1129% when scaling from 1 to twelve nodes). Interestingly, this observation seems to be in contradiction with other performance evaluations, such as [18]. Distributing a given state to several nodes should usually reduce total state management, which is also true if the state size remains constant. However, in the context of our YSB implementation, increasing the number of workers also increases the global state, e.g., due to the custom cache implementation (redundant caches on every worker). While this effect is measurable, its impact on the overall latency and CPU efficiency is quite low. Still, it explains why the GC cost increased.

Total: With regard to CPU efficiency, a single scale-up node achieves the best CPU efficiency for all SPEs (Figure 13). This is no surprise since a single-node requires no communication effort with other workers. However, we could also observe that adding even more worker nodes does not necessarily decrease the efficiency further. This is especially evident when looking at Flink for which only the *Cluster Communication* task increases linearly with the number of nodes, whereas the other tasks remain unaffected. For Spark STR, we observed a bell-shaped efficiency for most tasks meaning that it either favors many small or few powerful nodes. Despite the fact that a single-node achieved the highest efficiency, looking at the latency, we see a significant increase when choosing a single-node configuration over a multi-node cluster. For this reason, a scale-out approach is advisable when using Spark STR. This is also true for Spark CP, although we did not test more than two nodes. Flink, on the other hand, achieves solid performance results in scale-out and scale-up scenarios. With regard to network utilization, the communication costs for Flink increased significantly when switching to a scale-out approach. For Spark, on the other hand, the number of nodes has almost no effect on the network, indicating that Spark could be better suitable for very large scale-out approaches.

Stalling: The stalling behavior of Flink and Spark is almost identical. As shown in Figure 12, the stalling of both frontend- and backend cycles increases when switching from single- to multi-node. Running clusters bigger than four nodes does not affect the stalling behavior any further. Flink's IPC decreases when switching to a two-node cluster but is otherwise unaffected by the addition of any further nodes. Spark STR, however, depicts a small U-shape that partially explains the bell-shaped efficiency curve that we observed in Figure 12. Finally, the number of retired cycles has its peak with four and six nodes. This means that the four- and six-node configurations required the most work.

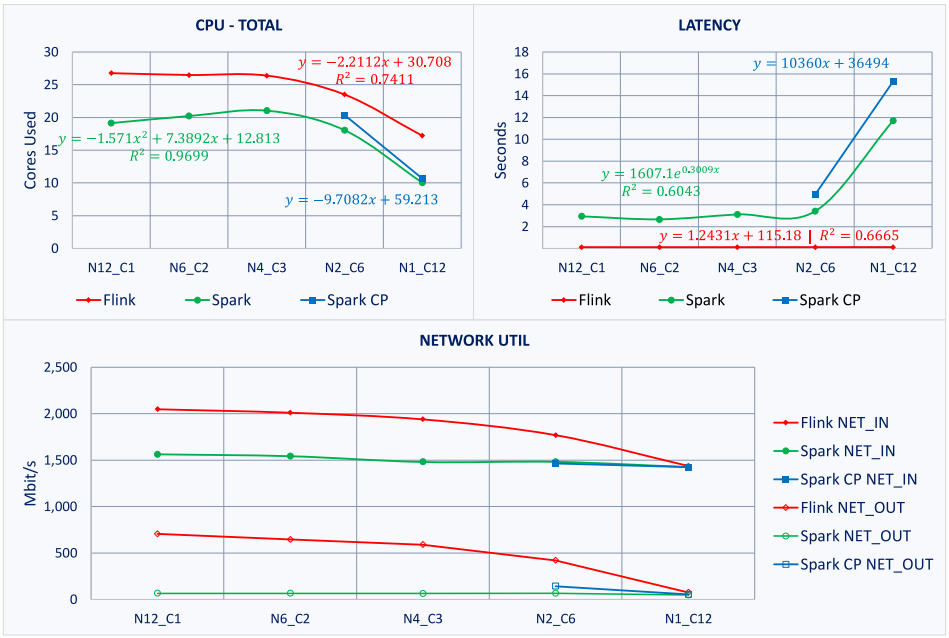

Figure 13. CPU consumption per task for different cluster configurations' summary.

6.3. State Scalability

Stream processing systems require state to store intermediate calculation results and buffering incoming events. An application's streaming task composition largely affects the state size. Among the most common stateful streaming tasks are *Joins* and *Windows* [37], which are also used in the context of the YSB. In addition, the workload characteristic may also influence the state size, e.g., the YSB's state is mostly affected by the number range of *campaign_ids*, as this directly determines the number of windows. State management can considerably degrade the performance of an SPE and result in increased latency or recovery time [38]. This is, e.g., caused due to state checkpointing, state migration, or, depending on the state backend, increased GC or I/O costs [39]. In this experiment, we explore how state scalability affects the individual streaming tasks of the YSB pipeline. For this reason, we apply a fixed load of 120k e/s to a 2-node cluster, with 1 core per node, and scale the *campaign_id* range from 1k to 16k. The factors for this experiment are hence (1) SPE and (2) campaign_ids.

Kafka, Deserialize, and Enrich: With regard to Flink and Spark CP, these tasks are hardly affected by the number of *campaign_ids*, as shown in Figure 14. For Spark STR, on the other hand, we observe a considerable efficiency gain the higher the number of campaigns gets. This seems to be counter-intuitive, but can be explained when looking at the latency in Figure 15, which increases considerably with a higher state. We conclude that Spark STR dynamically increases its batch size to keep up with the processing. This in turn results in efficiency gains that we already observed during the load scalability experiment in Section 6.1.

Filter+Projection+Join: This task shows consistent efficiency for Spark STR and Spark CP, as well as a small cost increase for Flink. Flink's performance degradation is caused by the *Join* task. We measured a 16.3% CPU increase when scaling from 1k to 16k *campaign_ids*. The reason why this effect is so small is due to the cache implementation. A real join with Redis would have a bigger impact on the actual performance.

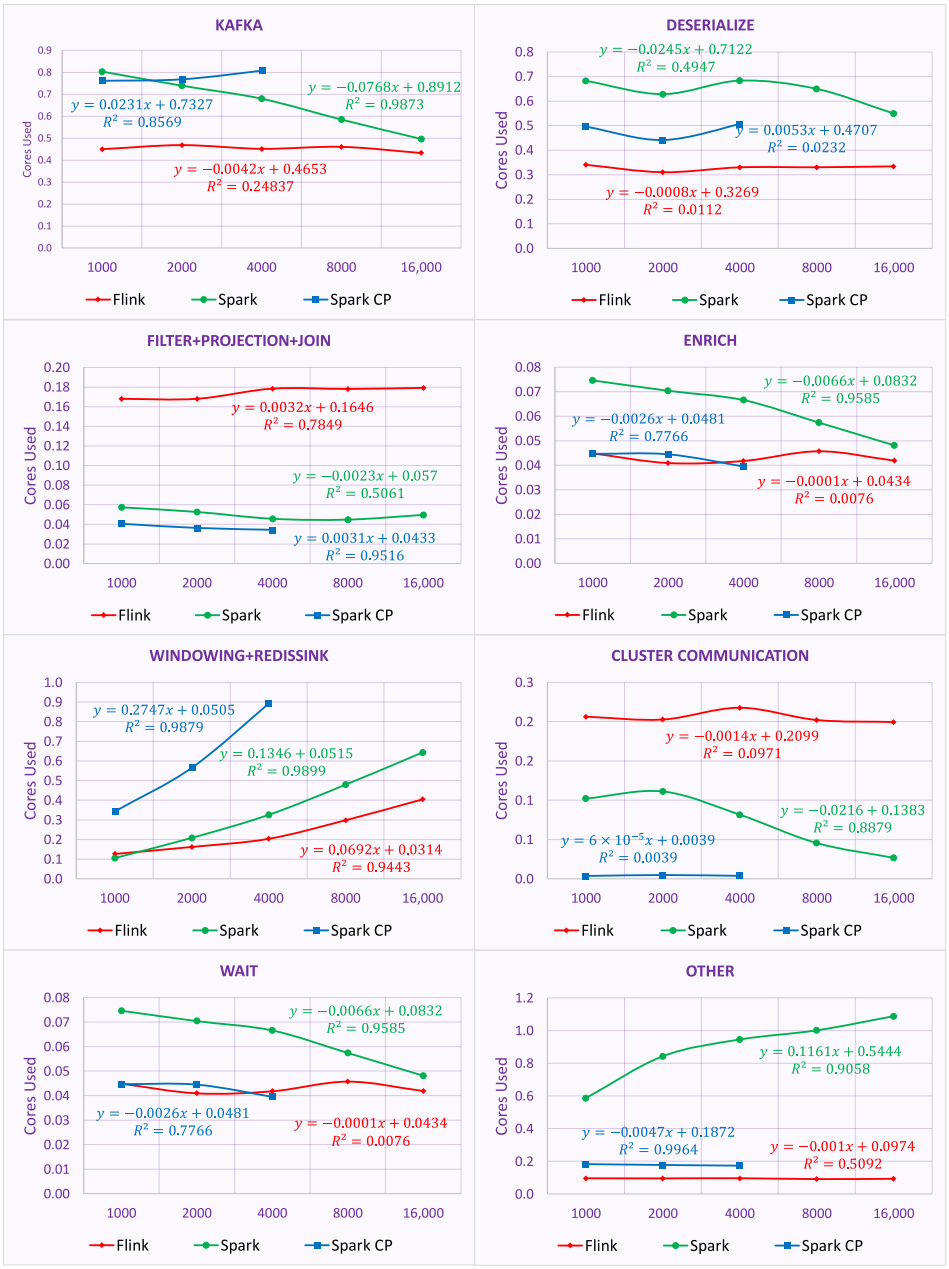

Figure 14. CPU consumption per task when scaling campaigns 1k–16k.

Windowing+Sink: As shown in Figure 14, for all SPEs, the *Windowing and Sink* task increases linearly with the number of *campaign_ids*. Due to the *groupby/keyby* operation, every campaign results in one window and at least one write operation to Redis. Hence, more campaigns mean more windows and consequently more writes. For Spark CP, the cost increase is even bigger, due to our workaround implementation as described in Section 3.1.

Cluster Communication and Wait: With regard to Flink, the CPU efficiency of both tasks is not affected by the number of campaigns. Although the *Window* task's *groupby/keyby* re-partitions the events based on the *campaign_ids*, not the number of campaigns, the ratio affects the performance. The load balancing is performed in a round-robin manner; with only two nodes available, half of the events are always sent to the first worker and the other half to the second one as long as the range of *campaign_ids* is even. Only for extremely uneven cases, e.g., *campaings* = 3, would we expect to see a difference. Due to this, Spark STR's *Cluster Communication* should not be affected as well. However, we observe again efficiency increases that are caused by the general efficiency gains due to the larger micro-batches (at the cost of higher latency).

Other: For Flink and Spark CP, the *Other* task remains stable. For Spark STR, we observe a linear incline of cycle costs the more campaigns that are configured. This is mainly due to the increased number of GC cycles.

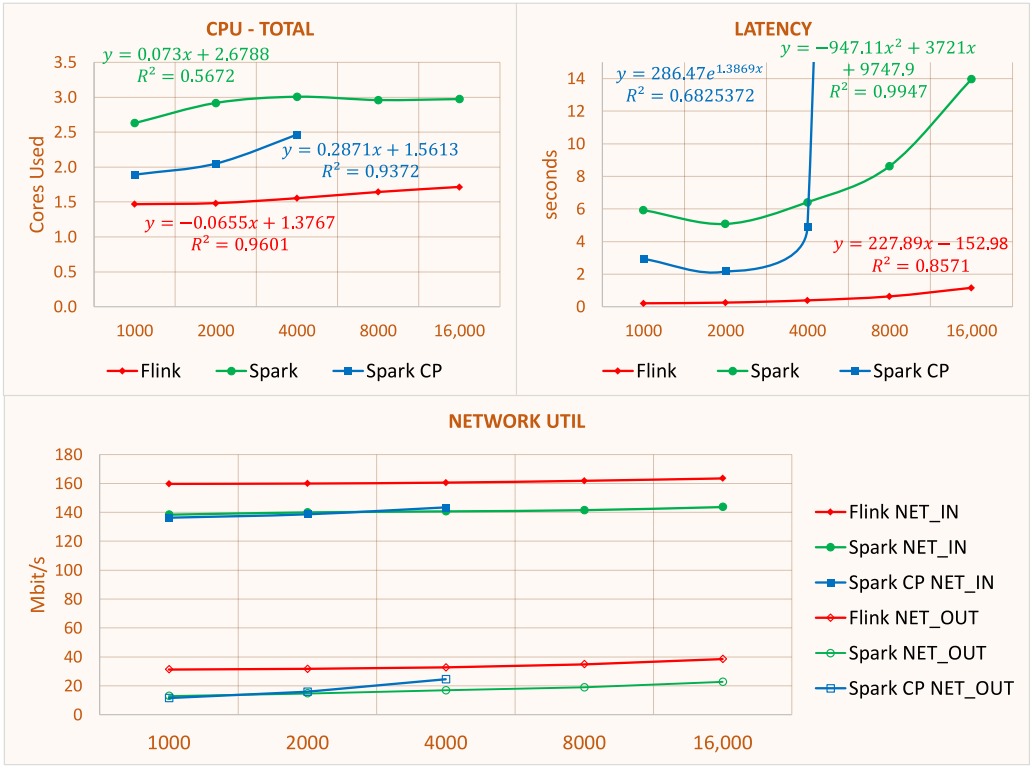

Figure 15. CPU consumption when scaling campaigns 1k–16k—summary.

Total: For all SPEs, the efficiency decreases mainly due to the cost increase of the *Windowing+Redissink* task. Flink's CPU utilization and latency both increase with the number of *campaign_ids*, whereas, for Spark STR, it appears that the total CPU costs cap at about 3.0 cores. However, looking at the latency, we see that the latency jumps up to 14 s. Hence, the CPU savings are bought at the cost of larger micro-batch sizes. Spark CP was not able to reliably scale beyond 4000 *campaign_ids*, probably due to the steep incline of the *Windowing+Sink* task.

Stalling: As displayed in Figure 16, both the number of retired and stalled cycles increase with the number of *campaign_ids*. The IPC of all SPEs drops, partially due to an increase in context switches and branch misses.

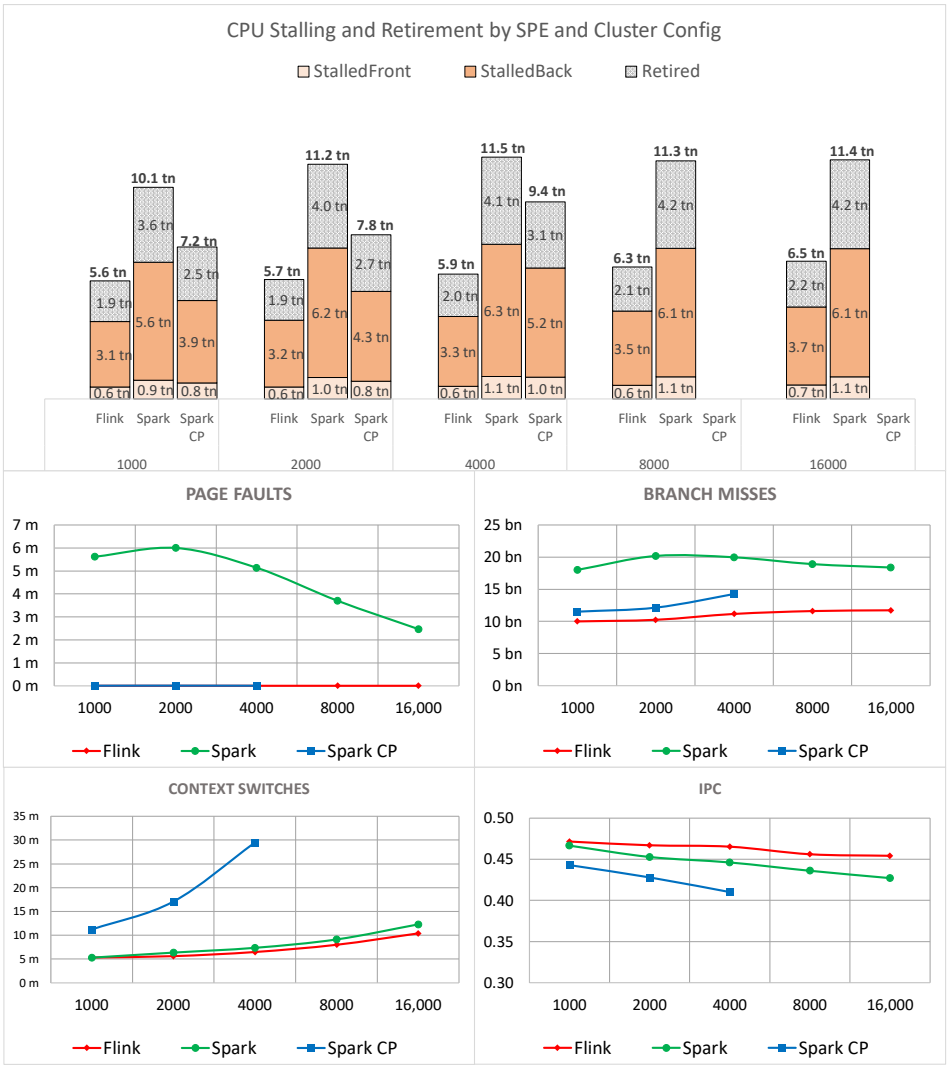

Figure 16. CPU stalling behavior when scaling campaigns from 1k to 16k.

7. Conclusions

CPU efficiency in the context of data stream processing is the ratio of the events processed by an SPE to the CPU resources invested. Thus far, performance research in the area of DSPSs has focused on latency and throughput. However, we argue that, for many business cases, it is becoming increasingly important to find the SPE that is merely capable of meeting the SLOs for latency and throughput with minimal resource consumption, whether due to resource constraints in edge computing scenarios or to save power and compute costs in large cloud deployments. In this paper, we have presented an approach for measuring the CPU efficiency of DSPSs at the task level. We have presented the concept from a conceptual as well as a technical perspective and evaluated the quality of our implementation in terms of consistency and overheads. This has shown that, despite the efficiency of eBPFs, there are still measurable performance overheads in scenarios with high base load. However, the overheads were found to be acceptable and did not skew the results. In contrast, the traditional approach, which was based on a classic Java profiler, caused significant overheads that even resulted in unsustainable processing for Flink and Spark CP. In terms of consistency, our approach showed quite stable results.

To demonstrate the potential of this new measurement approach, we focused particularly on measuring and analyzing the task-level CPU efficiency of three different SPEs under the influence of load-, cluster-, and state scalability. In this way, we could provide a detailed picture of the individual tasks of the YSB, which was not previously possible. The main findings of the experiments are summarized in Table 3.

Table 3. Performance findings summary.

Experiment	Analysis	FLINK	SPARK STR	SPARK CP
Load Scalability (low->high)	Total	- CPU efficiency: Highest ->low+mid High ->high - Latency: Lowest ->all - Network util.: Highest ->all	- CPU efficiency: Poor ->low+mid Highest ->high - Latency: High ->all - Network util.: Low ->all	- CPU efficiency: Poor ->high load High ->low+mid - Latency: Low ->all - Network util.: Low ->all
	Task-Level	- Highest CPU efficiency: Deserialize, Enrich, Sink, Framework	- Highest CPU efficiency: ClusterComm., Windowing	- Highest CPU efficiency: Filter+Projection+Join - Kafka: High CPU demand ->improvement required
Scale-up vs. Scale-out (single->12)	Total	- CPU efficiency: Highest ->single-node - Latency: Not affected	- CPU efficiency: Lowest ->single node - Latency: Poor ->single-node ->scale-out recommended	- CPU efficiency: Highest ->single-node - Latency: Poor ->single node ->scale-out recommended
	Task-Level	- ClusterComm.: Significant CPU efficiency decrease - Kafka: Highest efficiency for single-node; Stable for multi-node - Remaining tasks: Stable	- ClusterComm.: Minor linear efficiency increase - Windowing+Sink: Stable - Remaining tasks: Highest efficiency for single-node; Increasing efficiency for large cluster	- All tasks: Highest efficiency for single-node
State Scalability (1k ->16k)	Total	- CPU efficiency: Slightly decreases - Latency: Slightly increases - Network util.: Slightly increases	- CPU efficiency: Slightly decreses until 4k; Afterwards constant at cost of higher latency - Latency: Rapidly increases after 4k - Network util.: Slightly increases	- CPU efficiency: Decreases - Latency: Slightly increases until 4k; Unsustainable afterwards -Network util.: Slightly increases
	Task-Level	- Windowing+Sink: CPU efficiency decreases - Remaining tasks: Stable	- Windowing+Sink: CPU efficiency decreases	- Windowing+Sink: CPU efficiency decreases

Load Scalability: With regard to load scalability, we observed that, for the YSB application, Spark STR consumes the most resources in low and mid-load scenarios. In these cases, Spark STR is not only considerably less CPU efficient than Spark CP and Flink, but it still has a higher latency. Hence, if the application supports the continuous processing mode, developers should test Spark CP. In the example of our YSB implementation, it improved CPU efficiency, and latency at low and medium workloads, while STR outperformed CP in terms of CPU efficiency in high-load situations. However, as demonstrated for the *Filter+Projection+Join* task, CP even has the potential to outperform STR at all load levels, so it depends on the application's task composition. Spark STR's efficiency outperforms Flink at about 110k e/s with regard to CPU efficiency; however, it does so with considerably increased latency costs. Our task-level analysis showed that Flink has efficiency advantages over Spark STR and CP for the tasks *Deserialize*, *Enrich*, and *Sink* and, in general, has lower framework overheads (*Other and Wait*, for example, less GC). Spark STR/CP, on the other hand, was able to outperform Flink with regard to *Cluster Communication*, *Windowing* and *Filter+Projection+Join*. Finally, Flink and Spark STR achieved the same efficiency with regard to the *Kafka* task, whereas Spark CP required considerably more cycles. Hence, a future area of research might be to improve CP's Kafka connector.

Scale-up vs. Scale-out: In terms of CPU efficiency for the YSB, all three SPEs benefit significantly from a single scale-up node over a scale-out approach (>36% less CPU). However, with regard to latency, Spark STR and CP achieve considerably better results on a multi-node cluster, whereas, for Flink, the latency is unaffected by the number of nodes. This indicates that Flink would be the better choice when choosing a single-node scale-up approach. In addition, we observed that Flink's and Spark STR's *Cluster Communication* costs increase linearly with the number of nodes. However, for Spark STR, this rate of increase is lower than for Flink. It would be interesting to evaluate how this comparison behaves when scaling beyond our twelve-node setup. Overall, it seems that Spark STR is more suitable for a very-large scale-out approach.

State Scalability: Increasing the number of *campaign_ids* resulted in increased state due to a higher number of *windows* and larger *join* caches. For Flink, both CPU utilization and latency increased when scaling the state. Unfortunately, our Spark CP application was not able to scale beyond 4k campaigns, probably due to our experimental implementation. For Spark STR, the total CPU utilization increased as well but capped after 4k campaigns. However, at the task level, we observed that the CPU consumption of the *Windowing+Redissink* task increases linearly with the number of campaigns. These additional costs are compensated by general efficiency increases due to larger micro-batch sizes but at the expense of higher latency.

8. Limitations and Future Research

In this work, we mainly used Flink and Spark with their default settings. While we ensured optimal parallelization settings in terms of throughput and latency, we did not aim to achieve optimal resource efficiency, e.g., by tuning GC. The goal of this work was to show how task-level CPU analysis can be achieved and to demonstrate its potential. There is room for further improvement for all SPEs, but this would require specific analysis because many settings that would improve one task could worsen another. In addition, our experiments were conducted for small to medium streaming clusters (up to 12 nodes/48 vCPUs and a peak load of 600k e/s). While we did not evaluate large clusters with hundreds of nodes, our results are representative of most real-world implementations of similar sizes. As a future research direction, we intend to use our task-level results to parametrize performance models to achieve accurate latency and energy predictions. Kross et al. [40] have shown that, in controlled environments, Spark Streaming's response time for a load scale-up scenario can be predicted with high accuracy. However, as shown in this work, many factors besides load affect CPU efficiency and thus latency. This makes it difficult to achieve accurate predictions for real-world scenarios and often requires dedicated performance models for individual use cases. We assume that task-based performance analysis could allow for better parameterization so that better models could be created that could handle multiple influencing factors simultaneously. Furthermore, since CPU efficiency is related to energy savings, we would like to make predictions in this regard as well. Energy consumption is becoming increasingly important in the context of stream processing systems, and predicting the consumption at an early stage would provide valuable insights.

Author Contributions: Conceptualization, J.R., A.H. and H.K.; methodology, J.R. and A.H.; software, J.R and J.H.; validation, J.R. and J.H.; formal analysis, J.R.; investigation, J.R. and J.H.; resources, H.K.; data curation, J.R. and J.H.; writing—original draft preparation, J.R.; writing—review and editing, J.R. and A.H.; visualization, J.R. and J.H.; supervision, H.K.; project administration, A.H. and H.K.; funding acquisition, H.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Please find the source code of this work at https://github.com/rankj/YSB-task-level (accessed on 31 December 2022). The data of our experiments are also publicly available at https://doi.org/10.21227/ar4v-hk87.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AI	Artificial Intelligence
DSPS	Distributed Stream Processing System
eBPF	Extended Berkley Package Filter
IoT	Internet of Things
LUW	Logical Unit of Work
PID	Process ID
PMC	Performance Monitoring Counters
YSB	Yahoo Streaming Benchmark
SLO	Service Level Objective
Spark CP	Spark Continuous Processing
Spark STR	Spark Structured Streaming
SPE	Stream Processing Engine

References

1. Jung, J.J. Special Issue Editorial: Big Data for Mobile Services. *Mob. Netw. Appl.* **2018**, *23*, 1080–1081. [CrossRef]
2. Tan, L.; Wang, N.M. Future internet: The Internet of Things. In Proceedings of the 2010 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE), Chengdu, China, 20–22 August 2010; Volume 5, pp. V5–376–V5–380.
3. Apiletti, D.; Barberis, C.; Cerquitelli, T.; Macii, A.; Macii, E.; Poncino, M.; Ventura, F. iSTEP, an integrated Self-Tuning Engine for Predictive maintenance in Industry 4.0. In Proceedings of the 2018 IEEE International Conference on Big Data and Cloud Computing, Yonago, Japan, 12–13 July 2018; pp. 924–931.
4. Umadevi, K.; Gaonka, A.; Kulkarni, R.; Kannan, R.J. Analysis of Stock Market using Streaming data Framework. In Proceedings of the 2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI), Bangalore, India, 19–22 September 2018; pp. 1388–1390. [CrossRef]
5. Akram, S.; Bilas, A. A Sleep-based Communication Mechanism to Save Processor Utilization in Distributed Streaming Systems. In Proceedings of the Second Workshop on Computer Architecture and Operating SYSTEM Co-Design, Heraklion, Greece, 24–26 May 2011.
6. Brunnert, A.; Vögele, C.; Danciu, A.; Pfaff, M.; Mayer, M.; Krcmar, H. Performance management work. *Wirtschaftsinformatik* **2014**, *56*, 197–199. [CrossRef]
7. Kim, T.; Yoo, S.; Kim, Y. Edge/Fog Computing Technologies for IoT Infrastructure. *Sensors* **2021**, *21*, 3001. [CrossRef] [PubMed]
8. Xhafa, F.; Kilic, B.; Krause, P. Evaluation of IoT stream processing at edge computing layer for semantic data enrichment. *Future Gener. Comput. Syst.* **2020**, *105*, 730–736. [CrossRef]
9. Dhakal, A.; Kulkarni, S.G.; Ramakrishnan, K.K. Machine Learning at the Edge: Efficient Utilization of Limited CPU/GPU Resources by Multiplexing. In Proceedings of the 2020 IEEE 28th International Conference on Network Protocols (ICNP), Madrid, Spain, 13–16 October 2020; pp. 1–6. [CrossRef]
10. Abdallah, H.B.; Sanni, A.A.; Thummar, K.; Halabi, T. Online Energy-efficient Resource Allocation in Cloud Computing Data Centers. In Proceedings of the 2021 24th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN), Paris, France, 1 March 2021; pp. 92–99.
11. Chintapalli, S.; Dagit, D.; Evans, B.; Farivar, R.; Graves, T.; Holderbaugh, M.; Liu, Z.; Nusbaum, K.; Patil, K.; Peng, B.J.; et al. Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. In Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Chicago, IL, USA, 23–27 May 2016; pp. 1789–1792. [CrossRef]
12. Grier, J. Extending the Yahoo! Streaming Benchmark. Available online: https://www.ververica.com/blog/extending-the-yahoo-streaming-benchmark (accessed on 8 December 2022).
13. Karakaya, Z.; Yazici, A.; Alayyoub, M. A Comparison of Stream Processing Frameworks. In Proceedings of the 2017 International Conference on Computer and Applications (ICCA), Doha, United Arab Emirates, 6–7 September 2017; pp. 1–12. [CrossRef]
14. Shahverdi, E.; Awad, A.; Sakr, S. Big Stream Processing Systems: An Experimental Evaluation. In Proceedings of the 2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW), Macao, Macao, 8–12 April 2019; pp. 53–60. [CrossRef]
15. Karimov, J.; Rabl, T.; Katsifodimos, A.; Samarev, R.; Heiskanen, H.; Markl, V. Benchmarking Distributed Stream Data Processing Systems. In Proceedings of the 2018 IEEE 34th International Conference on Data Engineering (ICDE), Paris, France, 16–19 April 2018; pp. 1507–1518. [CrossRef]
16. van Dongen, G.; Steurtewagen, B.; Van den Poel, D. Latency Measurement of Fine-Grained Operations in Benchmarking Distributed Stream Processing Frameworks. In Proceedings of the 2018 IEEE International Congress on Big Data (BigData Congress), San Francisco, CA, USA, 2–7 July 2018; pp. 247–250. [CrossRef]
17. Van Dongen, G.; Van den Poel, D.E. Evaluation of Stream Processing Frameworks. *IEEE Trans. Parallel Distrib. Syst.* **2020**, *31*, 1845–1858. [CrossRef]

18. Van Dongen, G.; Van Den Poel, D. Influencing Factors in the Scalability of Distributed Stream Processing Jobs. *IEEE Access* **2021**, *9*, 109413–109431. [CrossRef]
19. Kroß, J.; Krcmar, H. PerTract: Model Extraction and Specification of Big Data Systems for Performance Prediction by the Example of Apache Spark and Hadoop. *Big Data Cogn. Comput.* **2019**, *3*, 47. [CrossRef]
20. Reussner, R.H.; Becker, S.; Happe, J.; Heinrich, R.; Koziolek, A. *Modeling and Simulating Software Architectures: The Palladio Approach*; MIT Press: Cambridge, MA, USA, 2016.
21. Rank, J.; Hein, A.; Krcmar, H. A Dynamic Resource Demand Analysis Approach for Stream Processing Systems. In Proceedings of the Symposium on Software Performance, Leipzig, Germany, 5–6 November 2020.
22. Gregg, B. *BPF Performance Tools*; Addison-Wesley Professional: Boston, MA, USA, 2019.
23. Souza, P.R.R.D.; Matteussi, K.J.; Veith, A.D.S.; Zanchetta, B.F.; Leithardt, V.R.Q.; Murciego, A.L.; Freitas, E.P.D.; Anjos, J.C.S.D.; Geyer, C.F.R. Boosting Big Data Streaming Applications in Clouds With BurstFlow. *IEEE Access* **2020**, *8*, 219124–219136. [CrossRef]
24. EJ-Technologies. Java Profiler-JProfiler. Available online: https://www.ej-technologies.com/products/jprofiler/overview.html (accessed on 30 December 2022).
25. Jain, R. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*; Wiley Professional Computing; Wiley: Hoboken, NJ, USA, 1991.
26. Nabi, Z.; Bouillet, E.; Bainbridge, A.; Thomas, C. Of Streams and Storms A Direct Comparison of IBM InfoSphere Streams and Apache Storm in a Real World Use Case. IBM White Paper 2014. Available online: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=c82f170fbc837291d94dc0a18f0223d182144339 (accessed on 6 December 2022).
27. Shukla, A.; Chaturvedi, S.; Simmhan, Y. Riotbench: An iot benchmark for distributed stream processing systems. *Concurr. Comput. Pract. Exp.* **2017**, *29*, e4257. [CrossRef]
28. Hesse, G.; Matthies, C.; Perscheid, M.; Uflacker, M.; Plattner, H. ESPBench: The Enterprise Stream Processing Benchmark. In Proceedings of the ACM/SPEC International Conference on Performance Engineering, Virtual, 19–23 April 2021; pp. 201–212. [CrossRef]
29. Abadi, D.J.; Carney, D.; Çetintemel, U.; Cherniack, M.; Convey, C.; Lee, S.; Stonebraker, M.; Tatbul, N.; Zdonik, S. Aurora: A new model and architecture for data stream management. *VLDB J.* **2003**, *12*, 120–139. [CrossRef]
30. Abadi, D.J.; Ahmad, Y.; Balazinska, M.; Cetintemel, U.; Cherniack, M.; Hwang, J.H.; Lindner, W.; Maskey, A.; Rasin, A.; Ryvkina, E.; et al. The design of the borealis stream processing engine. In Proceedings of the Cidr, Asilomar, CA, USA, 4–7 January 2005; Volume 5, pp. 277–289.
31. Kruber, N. *A Deep-Dive into Flink's Network Stack*. Available online: https://flink.apache.org/2019/06/05/flink-network-stack.html (accessed on 7 September 2022).
32. Chatzopoulos, G.; Dragojević, A.; Guerraoui, R. ESTIMA: Extrapolating Scalability of in-Memory Applications. In Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Barcelona, Spain, 12–16 March 2016. [CrossRef]
33. Hill, M.D. What is Scalability? *SIGARCH Comput. Archit. News* **1990**, *18*, 18–21. [CrossRef]
34. Hwang, K.; Shi, Y.; Bai, X. Scale-Out vs. Scale-Up Techniques for Cloud Performance and Productivity. In Proceedings of the 2014 IEEE 6th International Conference on Cloud Computing Technology and Science, Singapore, 15–18 December 2014; pp. 763–768. [CrossRef]
35. Awan, A.J.; Brorsson, M.; Vlassov, V.; Ayguade, E. How Data Volume Affects Spark Based Data Analytics on a Scale-up Server. In *Proceedings of the Big Data Benchmarks, Performance Optimization, and Emerging Hardware*; Zhan, J., Han, R., Zicari, R.V., Eds.; Springer International Publishing: Basel, Swizterland, 2016; pp. 81–92.
36. McSherry, F.; Isard, M.; Murray, D.G. Scalability! However, at what COST? In Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS XV), Kartause Ittingen, Switzerland, 18–20 May 2015; USENIX Association: Kartause Ittingen, Switzerland, 2015.
37. De Matteis, T.; Mencagli, G. Parallel patterns for window-based stateful operators on data streams: An algorithmic skeleton approach. *Int. J. Parallel Program.* **2017**, *45*, 382–401. [CrossRef]
38. To, Q.C.; Soto, J.; Markl, V. A survey of state management in big data processing systems. *VLDB J.* **2018**, *27*, 847–872. [CrossRef]
39. Del Monte, B.; Zeuch, S.; Rabl, T.; Markl, V. Rhino: Efficient Management of Very Large Distributed State for Stream Processing Engines. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, Portland, OR, USA, 4–19 June 2020; SIGMOD '20; Association for Computing Machinery: New York, NY, USA, 2020; pp. 2471–2486. [CrossRef]
40. Kroß, J.; Krcmar, H. Modeling and simulating Apache Spark streaming applications. *Softw.-Trends* **2016**, *36*, 1–3.