A Dynamic Resource Demand Analysis Approach for Stream Processing Systems

Johannes Rank, Andreas Hein, Helmut Krcmar
Technical University of Munich
85748 Garching, Germany
{johannes.rank, andreas.hein, helmut.krcmar}@tum.de

Abstract

Systems that provide real-time business insights based on live data, so-called Stream Processing Systems (SPS), have received much attention in recent years. In many areas such as stock markets or surveillance, it is essential to process data immediately and react accordingly. As the processing of real-time data is at the heart of SPS, their performance in terms of latency, throughput, and resource utilization constitutes a crucial role. Traditional performance and benchmarking approaches for SPS usually focus on the throughput and latency, trying to answer the question of which engine processes the incoming events fastest. However, neglecting the corresponding resource utilization provides only a limited and sometimes even misleading view on their actual performance. Depending on the use-case, an engine that achieves faster processing results at the cost of higher memory utilization is not always best suited, which can be shown based on the example of IoT edge computing devices with limited resources. For this reason, we developed a dynamic performance approach to analyze the resource demands of an SPS. The approach yields fine-grained performance metrics based on the individual processing steps of the SPS and without requiring any knowledge of the actual source code. More-over it takes the whole system (engine and streaming application) into account. Since, we do not rely on code instrumentation or language-specific profiling techniques but instead, use the dynamic tracing capabilities of the Linux kernel, we can support a broad range of different SPSs. We evaluate our approach by inspecting the CPU performance of Apache Flink while performing the Yahoo streaming benchmark.

1 Introduction

While there is much work in analyzing the performance based on throughput and latency, there is currently a lack of proper resource-based evaluations such as CPU or memory. We argue that evaluating and benchmarking performance for different SPSs without taking resource utilization into account can lead to distorted or even misleading results. As an example, the stream processing engine (SPE) Apache Flink has a pipelining architecture that usually requires a fair number of buffers to allow stable processing. These buffers have to be held in memory increasing its utilization. Spark's Resilient Distributed Datasets (RDDs), on the other hand, are kept in memory by default, but can also be configured to be held on disk in order to save memory utilization [4]For use-cases such as stream processing on edged-computing devices, an SPE that requires less memory at the cost of a slightly increased latency could offer the better cost-performance ratio. This illustrative example demonstrates why resource considerations for the whole SPS (SPE and streaming application) are important to provide a complete performance picture. To tackle this problem, we introduce a novel resource-centered performance evaluation approach that yields fine-grained resource metrics. In this work, we exclusively focus on the CPU consumption, but other resource considerations will be integrated into future work. The approach further provides the following capabilities:

- *In-depth metrics analysis*: Resource metrics are collected for the individual processing tasks.
- *Full scope*: The whole system comprised of SPE and application is monitored.
- *Dynamically applicable*: The approach can be applied to running SPSs without a restart.
- *Production safe*: The tools and programs used as part of this approach are non-disruptive and can also be applied in production scenarios.
- *SPE independent*: The approach support a broad range of SPS.
- *No domain knowledge required*: No knowledge about the SPE or source-code is required.
- *Stable metrics*: The approach is resistant to temporary performance degradations caused by one-time effects or system errors.

2 Related Work

The Yahoo streaming benchmark (YSB) [2] has received much attention from both practitioners and researchers. The benchmark features a full data pipeline

in which advertisement events are first consumed from Kafka, joined with campaigns obtained from a redis database, grouped by campaigns via a windowing task and finally stored back to redis. The benchmark in its original form only focuses on latency and throughput. Van Dongen/Van den Poel [5] extended the YSB to provided more accurate latency measurements. In addition, they monitored the CPU utilization on the container level. While this is a first step towards providing a more extensive view on the performance behavior, the monitoring is still conducted on a high level providing no insights which streaming tasks consume the most CPU. Furthermore, CPU utilization, as reported by *top*, is generally considered to be an inaccurate metric since stall-cycles are interpreted as "busy" despite the fact that the application is waiting.

3 Approach

What makes a uniformly applicable performance approach possible in the first place is the fact that all SPSs are based on a common generic architecture. They usually consume data from a *known source* (e.g. Twitter events) via a *socket connection*. In addition SPSs are usually comprised of a programming API that is represented by the streaming application and a streaming engine (SPE) that is responsible for management tasks such as orchestration or fault tolerance [3]. To provide a complete view on the resource consumption, our resource analysis needs to take all three components (Workload, application and SPE) into account. Figure 1 illustrates our approach.

Phase 1 The actual process tree of an SPS depends on the engine itself (e.g. Spark/Flink) as well as its configuration [1]. In order to identify what processes need to be traced, the first step in our approach is to obtain all process IDs (PIDs) that are part of the SPS. We developed the *ScopeAnalyzer.sh* as a simple Linux shell script that identifies the target PIDs based on process names. For emerging SPEs that are not yet covered by the tool, the *ScopeAnalyzer* provides an option to trace and identify new processes that were created via *exec()* during the startup of the engine.

Phase 2.1 In order to yield accurate performance metrics, we need to trace the workload that was actually consumed by the application. Here we make use of the two assumptions "known sources" and "socket connection", as described in section 3. We implemented a *bpftrace* program called *workload_trace.bt*. It uses the *kretprobe:sock_recvmsg* to count the bytes consumed by a defined set of PIDs and prints the sum together with its process name. The result is filtered for our known source e.g. *Kafka Fetcher* in case of the YSB. *bpftrace* is a high level language for eBPF, that was added to the Linux kernel in release 3.15 and allows to process events in kernel space. Hence, eBPF is very efficient and supported by most Linux systems.

Phase 2.2 During observation, we access performance monitoring counters (PMC) to obtain detailed CPU metrics for the target PIDs. Here, we make use of the *perf stat* command to count the cycles and instructions used by the different processes. Access to PMC information costs practically no performance.

Phase 2.3 For the identified PID(s) of the streaming application the CPU is sampled for stack traces. Each stack itself provides information which code path was executed (e.g. redis join), while the number of samples gives an approximation how often these code paths were executed and hence how much CPU was demanded. This way we get detailed insights into the inner workings of the application and the performance of its processing tasks without requiring any knowledge about the actual source code.

Phase 3 The *stack_count.raw* contains one entry for every single stack trace that was sampled during the observation. The *stack_fold.bt* script aggregates this information to show only distinct code paths and the number of their occurrences. In addition, we generate a FlameGraph. FlameGraphs were invented by Brendan Gregg and provide an easy way to visualize stack traces and to show hot code paths to the user. Both scripts are available via his git repository[1].

Phase 4 During the result generation, two reports are created. The absolute demand provides CPU metrics for the whole SPS grouped by SPE and application. In addition the *Instructions per byte* are calculated based on the workload measured in step 2.1.

4 Experiment

We applied our approach while running the YSB for Apache Flink in a single node configuration. The purpose of this experiment is to provide an illustrive example how the approach works and how detailed the metrics are. We configured the benchmark to run with 2k and 4k events/second. The testing environment was a 12 vCPU machine based on a Intel Xeon E-2620. Phase 1 yielded three PIDs that correspond to the SPE (*client.cli* + *cluster*) as well as the application (*taskmanager*). The workload collected during phase 2.1 was similar to the values reported by Flink's monitoring dashboard. The load for running 2k event/s was about 1.78 GB/h. Phase 2.2 showed that the CPU demand of the SPE and application scaled well with the 4k workload increase. As depicted in Table 1 the cluster process itself did not consume additional CPU, which is expected since no changes to the cluster were performed. The instructions required by the streaming application increased by 105%. The CPU profiling in phase 2.3 revealed that the streaming application spent only 75.1% of its CPU-time in a Java thread. The remaining time was spent for other tasks such as waiting for *glibc*. Overall the most CPU was required for consuming events from Kafka (44.6%), followed by the Redis join (24.6%), while the filter operation required so little CPU that it did only appear in the 4k

[1] `https://github.com/brendangregg/FlameGraph`

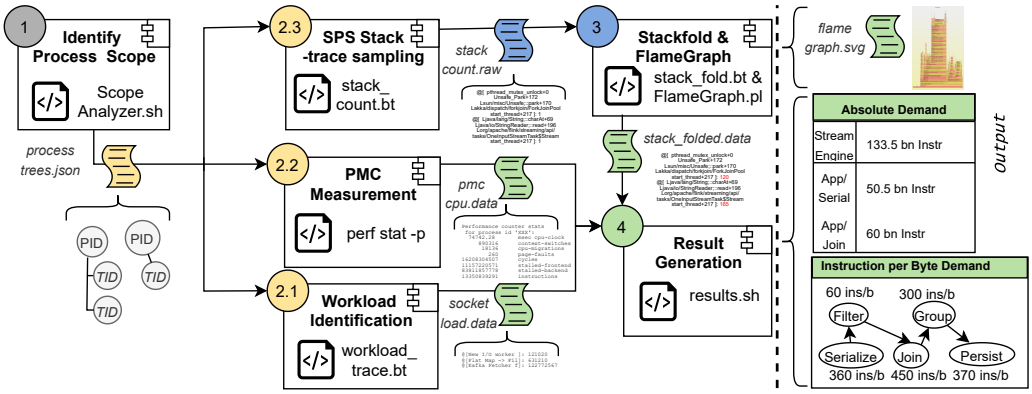

Figure 1: Performance Approach Tool Chain

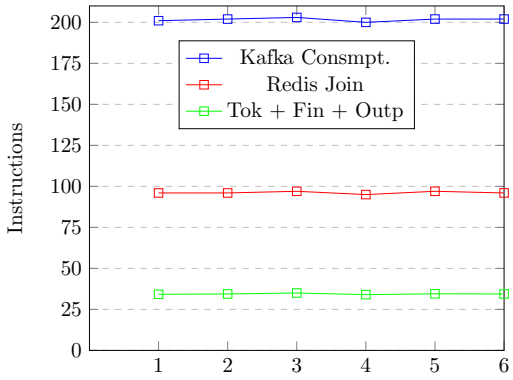

Figure 2: Relative CPU instructions for processing 1 Byte

test (0.02%). The overhead caused by the sampling at 999 Hz for 1h was neglectable during the monitoring run itself. Only after the sampling was completed a full core was consumed to write the results to disk. Hence, the metric collection itself was not affected, but a production environment would need spare resources. For calculating the CPU demands per byte we combine the sampled stack traces with the consumed instructions and put them into relation to the traced workload. As shown in Figure 2, this allows us to calculate the CPU instructions for processing a single byte from Kafka grouped by tasks (Tokenizer + Finalizer + Output summarized). Overall the approach yielded a very detailed performance picture of the application in which only the *filter* was omitted due to its neglectable impact on the CPU.

	2k Instr	2k IPC	4k Instr	4k IPC
Application	810 bil	0.73	1661 bil	0.75
SPE/Cluster	13.5 bil	0.29	12.5 bil	0.29
SPE/Client	1.0 bil	0.24	1.9 bil	0.23

Table 1: Avg CPU Instructions per hour

5 Conclusion

Streaming analytics has increased the need for resource focused performance evaluations. It becomes increasingly important to process data streams not only fast but also efficiently in terms of CPU utilization. In this paper we proposed an approach to dynamically inspect the CPU demands of an SPS at a fine-grained level. This way we can provide accurate and reliable performance accounts for the different components of an SPS (engine vs. application) as well as the individual processing tasks of the streaming application. We accomplish this without requiring any knowledge about the source code and without relying on performance tools that are limited to a certain SPE. This allows us to perform CPU performance evaluations in a uniform way. We think that using our approach has great potential when benchmarking SPEs, to provide an additional view. With this approach we can not only show how efficient an SPS works while achieving a certain throughput/latency result, but also how well individual processing tasks perform between the SPEs. This way benchmark results become more transparent to the user. For this reason, we plan to integrate our approach into the YSB and to add add memory measurements as an additional resource consideration.

References

[1] G. Hesse and M. Lorenz. “Conceptual survey on data stream processing systems”. In: *IEEE 21st Intern. Conference on Parallel and Distributed Systems (ICPADS)*. 2015, pp. 797–802.

[2] S. Chintapalli et al. “Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming”. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2016, pp. 1789–1792.

[3] S. Kamburugamuve and G. Fox. “Survey of distributed stream processing”. In: *Bloomington: Indiana University* (2016).

[4] O. Marcu et al. “Spark Versus Flink: understanding performance in Big Data analytics frameworks”. In: *CLUSTER 2016*. 2016, pp. 433–442.

[5] G. Van Dongen and D. E. Van den Poel. “Evaluation of Stream Processing Frameworks”. In: *IEEE Transactions on Parallel and Distributed Systems* (2020).