Dynamic Algorithm Selection for the Logic of Tasks in IoT Stream Processing Systems

Ehsan Poormohammady, Jens Helge Reelfs, Mirko Stoffers, Klaus Wehrle
Communication and Distributed Systems, RWTH Aachen University, Germany
ehsan.poor.mohammady@rwth-aachen.de, {reelfs,stoffers,wehrle}@comsys.rwth-aachen.de

Apostolos Papageorgiou
NEC Labs Europe, Germany
apostolos.papageorgiou@neclab.eu

***Abstract*—Various Internet of Things (IoT) and Industry 4.0 use cases, such as city-wide monitoring or machine control, require low-latency distributed processing of continuous data streams. This fact has boosted research on making Stream Processing Frameworks (SPFs) IoT-ready, meaning that their cloud and IoT service management mechanisms (e.g., task placement, load balancing, algorithm selection) need to consider new requirements, e.g., ultra low latency due to physical interactions. The algorithm selection problem refers to selecting dynamically which internal logic a deployed streaming task should use in case of various alternatives, but it is not sufficiently supported in current SPFs. To the best of our knowledge, this work is the first to add this capability to SPFs. Our solution is based on *i)* architectural extensions of typical SPF middleware, *ii)* a new schema for characterizing algorithmic performance in the targeted context, and *iii)* a streaming-specific optimization problem formulation. We implemented our solution as an extension to Apache Storm and demonstrate how it can reduce stream processing latency by up to a factor of 2.9 in the tested scenarios.**

I. Introduction

The increased popularity of IoT and Industry 4.0 leads to systems that handle enormous amounts of data, but Big Data technologies often miss an important factor: low latency [1]. *Low latency* is a critical requirement for industrial setups which depend on control loops for machines, e.g., for synchronizing power sources or other Smart Grid elements [2]. Due to the missing low-latency capabilities of classical Big Data (store first, process afterwards) approaches, a different technique becomes the method of choice: processing data streams on the fly. Thus, emerging Stream Processing Frameworks (SPFs) [3–5] are gaining attention.However, current SPF implementations often fail in providing ultra low latency, although the research community has identified this as a shortcoming[6, 7]. Different techniques can be applied to optimize stream processing latency, including task allocation, load balancing, load shedding, processing topology adaptations, and more [8].

Among the aforementioned techniques, *Algorithm Selection* is very promising for reducing latency in load-heavy scenarios. It assumes having a choice between different variants, algorithms or implementations of functionally equivalent building blocks, which have a different behavior in terms of required resources like CPU or RAM. Algorithm selection deals with finding the combination of such building blocks that leads to the lowest latency.After investigating related work (cf. II), we pave the way for Algorithm Selection schemes via the following tightly-coupled scientific contributions of this paper:

- We designed a generic SPF model based on state-of-the-art SPFs, and added architectural extensions required for Algorithm Selection (cf. III-A).
- We developed a model for characterizing algorithm performance with regard to *a)* device specific resources and *b)* stream-specific data input rates (cf. III-B).
- We provided a stream-specific local optimization problem formulation to Algorithm Selection allowing efficient selection decisions (cf. III-C).
- We evaluated the feasibility of our approach by implementing it as an extension to Apache Storm (cf. IV).

II. Background and related work

Stream Processing Frameworks (SPFs) are solutions that facilitate and manage the execution of distributed applications that consist of multiple processing steps which act upon data streams, i.e., continuously consume and produce data items (*tuples*). These processing steps use the output of previous steps and provide input to the next ones, so that they typically run sequentially or according to a DAG (Directed Acyclic Graph). Although different SPFs, e.g., Apache Storm [3], Samza [4], or Heron [5], use their own terminologies, developers always provide a sequence (*topology*) of steps (*components*) that are to be executed, along with the implementation of each component and required settings (e.g., desired number of instances for each step, desired number of used devices) to the SPF. The SPF then generates one or more instances of each component (*tasks*) and deploys them on hosting devices (*nodes*). Finally the data stream flows through the tasks.

[8] provides an overview of stream processing optimizations. These include approaches for optimal task deployment, load balancing, algorithm selection, and more. They can be diversely approached, leading to different benefits in different scenarios. For example, to load balance, [9] presents a method for minimizing the maximum and the variance of load by distributing the tuples based on capabilities and status of nodes. Similarly, the earlier work of [10] contributed a language that allows developers to specify rules about how load balancing shall be performed. Further, [11] and [12] have extended Apache Storm to enhance task placement mechanisms based on new runtime statistics. However, the optimization category of *algorithm selection* (cf. [8]) has neither been enabled for modern SPFs, nor sufficiently explored as an optimization problem.

978-3-901882-98-2

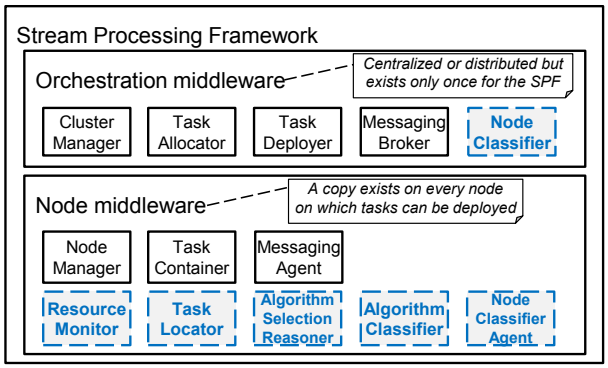
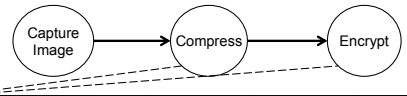


Fig. 1. Generic SPF middleware architecture. Our extensions for enabling algorithm selection are shown in blue, shaded, dashed boxes.



Component Name	Algorithm Name	Device Type	CPULoad PerInputRate *(in %, where x is the input rate in MB/sec)*	RAMLoad PerInputRate *(in %, where x is the input rate in MB/sec)*	CPULoad Threshold *(in %)*	RAMLoad Threshold *(in %)*	Basic Response Time *(milli-seconds)*
Compress	JPEG	Type1	8.45x	2.3x	90	90	240
Compress	JPEG2000	Type1	11.31x	3.1x	90	90	163
Compress	JPEG-XR	Type1	10.76x	2.4x	90	90	172
...	...	...	...	...	...	...	...
Encrypt	AES-128	Type1	5.2x	4.3x	90	90	62
Encrypt	Blowfish	Type1	7.3x	5.1x	90	90	44
...	...	...	...	...	...	...	...

Fig. 2. Algorithm characterization schema including examples. Different (alternative) algorithms exist for the Compress and Encrypt tasks. The entries are based on implementations which we benchmarked and used in our evaluation. These functions may depend more on our implementation, the system, and the used libraries, and not on the algorithms themselves.

III. IoT-aware Algorithm Selection in SPFs

A. Extending SPF architectures to support algorithm selection

In Figure 1, we show the modules that typically comprise an SPF middleware. The terminology is based on state-of-the-art open source SPFs[1]. The blue dashed boxes show the modules that we have added to enable efficient algorithm selection. *Typically*, there are two main parts: *i)* The orchestration middleware, being responsible for the deployment of the topology and for coordinating the messaging and the communication between different tasks running on different nodes. It usually runs as a single central instance. *ii)* The node middleware executing stream processing tasks residing on every processing node (i.e., server, computer, VM, or any device that hosts tasks).

Orchestration middleware: The *Cluster Manager* registers the devices that comprise the cluster and maintains information about their status and the tasks that are running on them. The *Task Allocator* retrieves information from the Cluster Manager and additional settings and determines which tasks shall run on which nodes. Based on that decision, the *Task Deployer* remotely deploys the tasks onto the devices. Finally, a *Messaging Broker* is usually contained, organizing the delivery of streaming data items (tuples) from task to task.

Node middleware: The *Node Manager* interacts with the Cluster Manager in order to attach the node to the cluster. It also interacts with the Task Deployer in order to receive the packaged executables of the tasks, which it is going to run inside the *Task Container*. Finally, a *Messaging Agent* is also provided as part of the SPF node middleware in order to send and receive data streaming items without the application developer having to care about how this is implemented.

For implementing algorithm selection support, we propose:

Orchestration middleware extensions: The new *Node Classifier* cooperates with the Cluster Manager to map each of the nodes of the cluster to one of the device types that will be used for characterizing the behavior of algorithms. This module is added because different device types have different processing characteristics and each algorithm has to be described by different resource utilization functions for the different device types it might run on.

Node middleware extensions: The *Resource Monitor* monitors the status of consumed and available resources on each node. This new module is required because the resources on a node are shared among multiple tasks and probably other applications which are running on the node, and thus the amount of the available resources might fluctuate frequently. The *Task Locator* talks to the Node Manager and Task Container for obtaining a local view of the streaming application topology running on this node. This particularly includes algorithm selection possibilities, i.e., the alternative implementations of components. The *Algorithm Selection Reasoner* is the core module of our extension, which runs an optimization algorithm for selecting the current algorithms of choice in each of the tasks to minimize latency (cf. subsection III-C). The *Algorithm Classifier* characterizes the available algorithms in a way that fits the SPF algorithm selection problem, and feeds the reasoner with its output. The Algorithm Classifier could also reside in the orchestration middleware. However, running it in the node middleware allows for updating the algorithm characteristics based on local monitoring. Finally, the optional *Node Classifier Agent* cooperates with the Node Classifier for performing and communicating the node classification.

B. Characterization of algorithms for SPF algorithm selection

One of the major obstacles towards setting up a solution with dynamic algorithm selection capabilities is the absence of useful metrics that characterize the algorithms. Therefore, we developed a schema with multiple, experimentally measurable metrics. The schema assumes that there is a set of algorithms that can be employed to implement the same functionally equivalent task. This is quite typical for certain categories of algorithms, e.g., compression or encryption (as shown in Figure 2) . This approach is not limited to algorithms in a strict sense. “Different algorithms” might simply refer to different implementations of the same functionality.Different implementations of the same functionality can certainly often differ with regard to CPU-intensity, RAM-intensity, and other parameters.

[1]For example, our *Task Allocator* corresponds with Nimbus’s scheduler in Storm and the YARN ResourceManager in Samza, our *Node Manager* would be the Supervisor in Storm and the YARN Node Manager in Samza, while our *Messaging Broker* would be ZeroMQ and/or RabbitMQ/Kafka in Storm.

For each combination of a *Component*, an *Algorithm*, and a *Device Type*, we define the following metrics, each obtainable via benchmarking:

- **CPU Load Per Input Rate:** The CPU load (in %) caused by executing this algorithm on this device type, expressed as a function $f(x)$, where x is the input rate in MB/s.
- **RAM Load Per Input Rate:** As above, but for the RAM.
- **CPU Load Threshold:** The threshold of CPU load (in %) above which the execution time (i.e., *latency per tuple*) of the algorithm on this device type is expected to increase rapidly. Up to that point the execution time either increases slowly or it remains stable, depending on the nature of the algorithm (IO-intensity vs. CPU-intensity), processor characteristics, CPU queue characteristics, and more. This exponential increase is explained based on queueing theory in [13], while further (delay-causing) bottlenecks (e.g., for moving processes etc.) appear when "CPU saturation" is reached. As stated there, "*100% CPU utilization is not twice as bad as 50% CPU utilization, it is much worse than that*". Therefore, various works (cf. [13, 14]) use *overload thresholds*, usually between 75%-95%.
- **RAM Load Threshold:** As above, but for the RAM.
- **Basic Response Time:** The expected latency per tuple of the algorithm on this device if no other tasks run on it.

Figure 2 shows example instances of the above schema based on a simple example topology, in which a task captures images and sends them to a next task which compresses them, while a third task encrypts the data. The table shows three alternatively applicable algorithms for the Compress task and two for the Encrypt task. We benchmarked each of the choices and modeled their characteristics as a linear function of the input rate.

C. Algorithm selection optimization problem for SPFs

Although we found no prior formulations of algorithm selection as an optimization problem, some works (e.g., [8]) imply it. We contribute a formulation that complies with our schema of section III-B. For achieving our low latency goal, we minimize the time that a tuple stays in the topology until it is completely processed, which we define as Lat. For example, in a sequential topology with d_z tasks deployed on z devices (D_1, ... D_z) as shown in Figure 3, Lat sums up to the amount of time from the moment that the input tuple enters the topology at link L_1 until the moment task t_{d_z} finishes its operation on it. We formulate the problem according to this example for simplicity. In Equation (1) we define Lat, where $Lat(L_i)$ is the latency over link L_i and $L(D_i)$ is amount of time taken by the tasks which are located on device D_i.

$$Lat = \sum_{i=1}^{z} Lat(L_i) + Lat(D_i) \tag{1}$$

Since our algorithms are typically different implementations of the same functionality, the algorithm selection does not affect input or output sizes. It could affect data quality in some cases, but we focus on latency, since it is our primary goal. Data quality -or any other metric- could be taken into account by just defining a different, weighted, objective function. Based on this

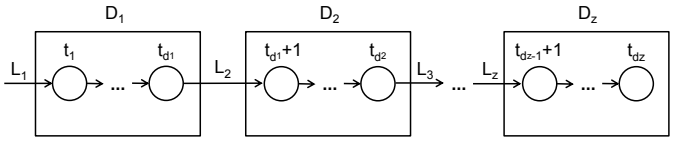


Fig. 3. Visualization of deployed sequential topology and involved elements. Each hosting device D_i executes a set of subsequent tasks t_j. The data flows in a well defined fixed topology through these devices.

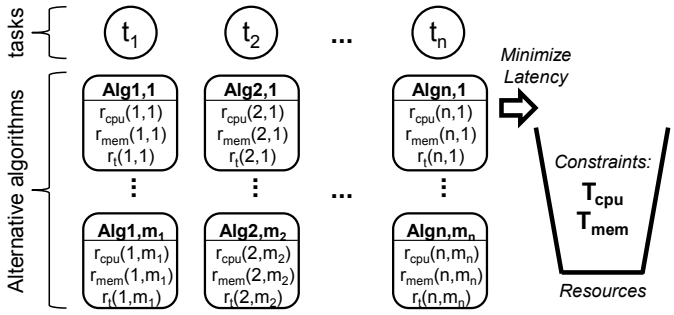


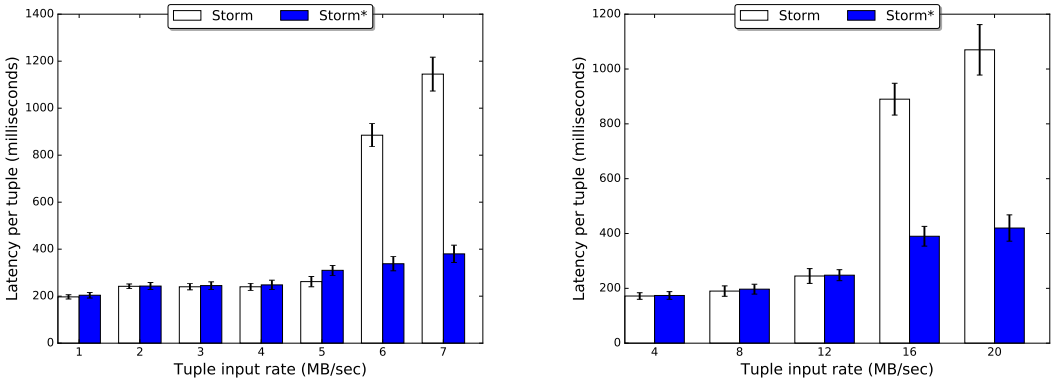
Fig. 4. SPF Algorithm Selection reduced to a Knapsack Optimization Problem. Each task t_j deployed on a single node may have various algorithms with different resource usage characteristics r_{cpu}, r_{mem} and a base latency r_t. The combinatorial choice is to be minimized with regards to latency while adhering to device resource contraints T_{cpu} and T_{mem}.

definition, the problem is reduced to minimizing each $Lat(D_i)$ separately, since the communication overhead $Lat(L_i)$ remains stable. Consequently, the optimization of each $Lat(D_i)$ can be modeled as an MMKP (Multi-choice Multi-dimensional Knapsack Problem) with the setting that is presented in Figure 4. There, $r_{cpu}(i,j)$, $r_{mem}(i,j)$, and $r_t(i,j)$ correspond with the *CPULoadPerInputRate*, the *RAMLoadPerInputRate*, and the *BasicResponseTime*, respectively, of the j-th algorithm of the i-th task. Further, m_i equivalent algorithms exist for the i-th task, while T_{cpu} and T_{mem} are the *CPUThreshold* and the *RAMThreshold* parameters (assuming here that they are the same for all algorithms, which is typical for the same device).

With these definitions, and with $s_{ij} \in \{0,1\}$ denoting if the j-th algorithm of the i-th task has been selected (for each i must exist exactly one value of j for which $s_{ij} = 1$), the MMKP problem formulation becomes:

$$\begin{aligned} \text{Min} \quad & \sum_{i=1}^{n}\sum_{j=1}^{m_i} r_t(i,j) \times s_{ij} \\ \text{subject to} \quad & \sum_{i=1}^{n}\sum_{j=1}^{m_i} r_{cpu}(i,j) \times s_{ij} \leq T_{cpu} \\ \text{and to} \quad & \sum_{i=1}^{n}\sum_{j=1}^{m_i} r_{mem}(i,j) \times s_{ij} \leq T_{mem} \end{aligned} \tag{2}$$

MMKP is NP-hard and for larger systems it would be difficult to solve fast. However, since we reduced the problem to the local scope of one device, we expect a limited number of tasks having alternative algorithms so that even exhaustive search may be performed fast enough. Moreover, heuristics for finding near-optimal solutions can be adapted for our problem and used in order to solve it in polynomial time. Many such heuristics for the MMKP problem are explored in [15].



(a) Avg. processing latency for Device Type1 (one CPU core) (b) Avg. processing latency for Device Type2 (two CPU cores)

Fig. 5. Average processing latency per tuple for the two device types with varying tuple input rates for vanilla Apache Storm and our adapted version Storm* (ticks denote the 90% confidence). While we observe Storm starting to break at a certain data input rate, i.e., the latency rockets up due to the underlying greedy algorithm choice, our solution still results in comparably very low latencies due to the application of algorithm selection.

IV. Evaluation

Our Storm-based solution including algorithm selection (denoted as *Storm**) is compared with a *greedy* approach deployed with the vanilla release of Storm. The latter statically deploys the fastest available algorithm for each task.

Setup: An implementation of the *topology of Figure 2* was used, with three alternative algorithms for the Compress task and two for the Encrypt task, and the observed characteristics of Figure 2, obtained via test measurements. We used two device types: Type 1 and Type 2 were simulated with VMs and reflect IoT GW capabilities. They both run at 3.10 GHz with 2GB RAM with Type 1 having 1 CPU core, while Type 2 having 2 cores. Figure 2 shows the numbers only for Type 1, but a similar pattern (with lower CPU usage) appears for Type 2. We varied the tuple input rate. Although we performed experiments for different *numbers of instances* of each task (i.e., Compress and Encrypt), the results provided no additional insights. Thus, in the selected results that we will discuss, we have fixed this variable to 6 instances for each task. The main metric was the average *Latency per tuple* (as described in Section III). For each value of the input rate we conducted 10 repetitions and each repetition continued until the "capture image" tasks had captured and forwarded 5000 images.

Results and discussion: The first main observation is that *Storm* and Storm perform similarly in cases of lower utilization*: For lower tuple input rates, i.e., up to 5 MB/s (12 MB/s) for Device Type 1 (2), no significant differences can be observed as shown in Figure 5. In those cases, Storm and Storm* used the same algorithms, namely the faster ones, i.e., JPEG-2000 for compression, and Blowfish for encryption. Storm decides on this combination due to its greedy approach of statically using the fastest algorithm, whereas Storm* chose this combination because it was the fastest option that was not violating its constraints (CPU and memory). Storm was sometimes slightly faster on average, which is either because of differences within the margins of statistical error (cf. overlapping 90%-confidence in Figure 5) or because of little overhead due to the additional monitoring and control functions of Storm*. The second main observation is that *Storm* processes data faster in cases of high utilization*: As promised, for higher input rates, i.e., 6 or 7 MB/s (16 or 20 MB/s) for Device Type 1 (2), Storm* processes tuples more quickly. That is due to Storm* immediately switching to slower—but less resource-demanding—algorithms, namely JPEG and AES-128 in this case. This combination shares the available resources without leading to an over-utilization. In contrast to this, the Storm approach breaks, i.e., leads to significantly higher latencies, due to over-utilization. We were able to confirm this by closer looking into the load figures during our experiments revealing that Storm increases CPU / RAM load up to 95-100 %, while Storm* always keeps it well below 90%. The main reason for these results is that -by selecting the fastest algorithm- Storm often causes the CPU of the device to get overloaded. For Type 2 devices, which are more powerful than Type 1 devices, this overload occurs for higher input rates, but it still occurs. Even with more CPUs or any type of more powerful devices, the same behavior would be observed by simply increasing the input rates.

V. Conclusion

SPFs are becoming more popular for low-latency IoT environments as usual Big Data approaches (store first, process afterwards) lack real-time capabilities. To the best of our knowledge, our approach is the first empowering SPFs to apply dynamic algorithm selection, based on the following main contributions: *i)* we design a new generic model of modern SPFs which we extend by new components for our purpose, *ii)* we base our selection technique on a new method of expressing algorithm features in terms of stream processing-specific parameters, *iii)* we formulate algorithm selection as an instance of the Multi-choice Multi-dimensional Knapsack Problem, i.e., *locally minimize the latency of subsequent processing tasks*, while adhering to device resource constraints. In comparison to vanilla Apache Storm, we show latency reductions of up to a factor of 2.9 in higher-load scenarios.

References

[1] C.L. Philip Chen and Chun-Yang Zhang. Data-intensive Applications, Challenges, Techniques and Technologies: A Survey on Big Data. *Information Sciences*, 2014.

[2] Murat Kuzlu, Manisa Pipattanasomporn, and Saifur Rahman. Communication network requirements for major smart grid applications in HAN, NAN and WAN. *Computer Networks*, 2014.

[3] Apache Software Foundation. Apache Storm project. https://storm.apache.org (visited June 2017).

[4] Apache Software Foundation. Apache Samza project. https://samza.apache.org (visited June 2017).

[5] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter Heron: Stream Processing at Scale. In *ACM SIGMOD*, 2015.

[6] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management. In *ACM SIGMOD*, 2013.

[7] Guangxiang Du and Indranil Gupta. New Techniques to Curtail the Tail Latency in Stream Processing Systems. In *ACM Workshop on Distributed Cloud Computing*, 2016.

[8] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A Catalog of Stream Processing Optimizations. *ACM Computing Surveys*, 2014.

[9] Niko Pollner, Christian Steudtner, and Klaus Meyer-Wegener. Operator Fission for Load Balancing in Distributed Heterogeneous Data Stream Processing Systems. In *ACM DEBS*, 2015.

[10] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *11th Int. Conference on Compiler Construction*. Springer, 2002.

[11] Jielong Xu, Zhenhua Chen, Jian Tang, and Sen Su. T-Storm: Traffic-Aware Online Scheduling in Storm. In *IEEE Int. Conference on Distributed Computing Systems*, 2014.

[12] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive Online Scheduling in Storm. In *ACM DEBS*, 2013.

[13] Charles Hooper. Faulty Quotes 6 CPU Utilization, February 2010. https://hoopercharles.wordpress.com/2010/02/05/faulty-quotes-6-cpu-utilization/ (visited June 2017).

[14] Fahimeh Farahnakian, Pasi Liljeberg, and Juha Plosila. LiRCUP: Linear Regression Based CPU Usage Prediction Algorithm for Live Migration of Virtual Machines in Data Centers. In *IEEE SEAA*, 2013.

[15] Bing Han, Jimmy Leblet, and Gwendal Simon. Hard Multidimensional Multiple Choice Knapsack Problems, an Empirical Study. *Computers and Operations Research*, 2010.