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, Aachen, Germany
ehsan.poor.mohammady@rwth-aachen.de,
{reelfs,stoffers,wehrle}@comsys.rwth-aachen.de

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

***Abstract*—Various Internet of Things and Industry 4.0 use cases such as city-wide monitoring, Smart Grid control, 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 derived from IoT-specific characteristics, i.e., ultra low latency due to physical interactions. Although various extensions have appeared to optimize such SPF-provided mechanisms, they still lack the modules, data models, and algorithms to properly handle *algorithm selection* in IoT deployments. The algorithm selection problem refers to selecting dynamically which internal logic a deployed streaming task should use in case of various alternatives. To the best of our knowledge, this work is the first solution that adds 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

While the Internet of Things and Industry 4.0 have become very prominent in public, they have also become a very active hot topic in research. In the consumer section, *things* are becoming smarter by adding diverse sensors for any purpose, while the same holds for industrial automation. This development and the general paradigm shift leads to enormous amounts of data. The term Big Data describes approaches for dealing with such huge amounts of data, but the storage schemes and the complex processing involved in Big Data technologies makes them 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 processing technique becomes the method of choice: processing data streams on the fly. Thus, rather new emerging Stream Processing Frameworks (SPFs) [3–5] are gaining significant attention as they fundamentally differ to classical Big Data solutions and promise being the silver bullet for low latency scenarios.

However, the service management solutions of current SPF implementations often fail in providing ultra low latency for IoT / edge-computing use cases, although the research community has identified this as a shortcoming and is actively working on approaches for adding the required capabilities [6, 7]. Different techniques can be applied in order to optimize stream processing latency, including task allocation, task replication, load balancing, load shedding, processing topology adaptations, and more [8].

Among the aforementioned techniques, *Algorithm Selection* is a very promising approach for reducing processing latency in certain scenarios. Algorithm selection assumes having a choice between different variants, flavors, algorithms, or implementations of functionally equivalent building blocks, which may have different behavior during execution. It deals with the problem of selecting which version of such a building block to use. We pave the way for more involved Algorithm Selection schemes via the following tightly-coupled main 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 on top.
- We developed a data modeling schema for characterizing algorithm performance with regard to *a)* device computational power and resources and *b)* stream-specific data input rate.
- We provided a stream-specific optimization problem formulation to Algorithm Selection, which allows for efficient reasoning solutions.
- We evaluated and showed the feasibility of our approach by implementing it as an extension of Apache Storm.

We give detailed background information and an overview of related work in Section II. Afterwards, in Section III, we present a generic model of SPFs, and discuss benefits of Algorithm Selection in SPF IoT settings. More importantly, we extend our generic SPF model with new modules and algorithms for enabling Algorithm Selection. In this section, we also introduce a required new method for characterizing algorithms in the SPF context. This paper finishes with an evaluation of our Proof of Concept implementation within Apache Storm in Section IV and a final conclusion.

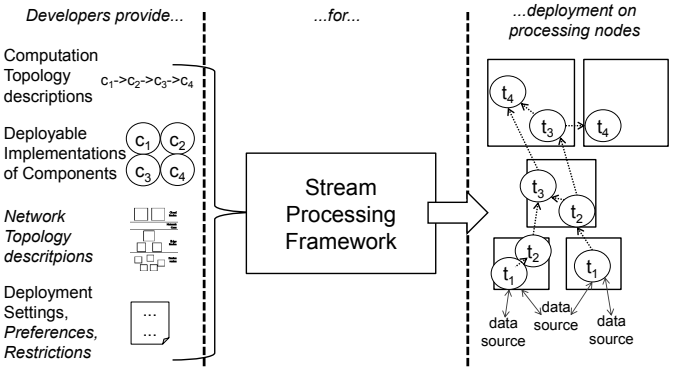


Fig. 1. Overview of SPF functionality. Given certain input from the developer, SPFs allow the easy deployment of stream-computation architectures.

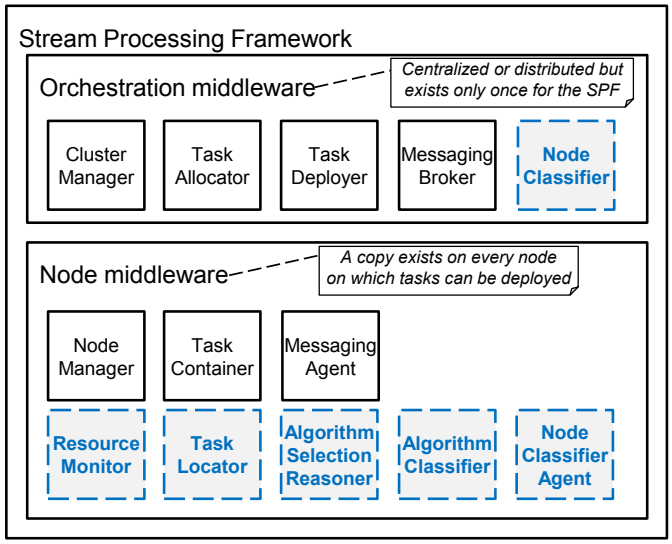


Fig. 2. Generic SPF middleware architecture. Our extensions for enabling algorithm selection are shown in blue, shaded, dashed boxes.

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 (also called *tuples*). These processing steps use the output of previous steps while they provide input to the next ones, so that the steps typically run sequentially or according to a DAG (Directed Acyclic Graph). A high-level overview of SPF operation is shown in Figure 1. However, note that different SPFs, e.g., Apache Storm [3], Samza [4], or Heron [5], use their own terminologies and may differ in detail. Developers provide the sequence (*topology*) of steps (*components*) that is 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 framework then generates one or more instances of each component (*tasks*) and deploys them on the hosting devices (*nodes*). Afterwards, the data stream flows through the tasks, e.g., as shown by the arrows on the right side of Figure 1. Together with our extensions, we will provide more information about the SPF internal structure and operation in Section III.

Although SPFs appeared rather recently, research on (non SPF-supported) stream processing exists since at least two decades. E.g., [8] provides a good overview of stream processing optimization categories. These include approaches for optimal task deployment, load balancing, algorithm selection, and more. They can be diversely approached, where all solutions highlight their benefit in different scenarios. For example, to load balance, [9] presents a method for minimizing both, the maximum and the variance of load implemented 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 captured by specialized loggers and network monitors. However, the optimization category of *algorithm selection* (cf. [8]) has neither been enabled for modern SPFs, nor sufficiently explored as an optimization problem. To the best of our knowledge, no current SPF architecture provides support for applying and optimizing Algorithm Selection, while the problem has not been addressed in an SPF context on a theoretical level either.

III. IoT-aware Algorithm Selection in SPFs

Apart from the obvious benefit of selecting faster algorithms whenever the system can afford it, the main way in which algorithm selection can enhance stream processing latency is by preventing overloading of the constraint IoT devices on which stream processing tasks are running. This kind of overloading can also lead to system failures and cause some tasks to stop working, thus leading to the breaking of entire processing chains (running topologies). For example, two algorithms / task implementations for face detection might have a significantly different trade-off in CPU and RAM resource usage, e.g. the faster implementation might require more memory. This is not an issue as long as the IoT gateway devices on which the face detection tasks run are not overloaded, but it might be a better idea to switch to the usually slower algorithm and avoid a too high RAM or CPU load if many tasks are running on the device. This is due to overloaded devices generally leading to higher latencies than when just using a slower algorithm not causing an overload. We further explain in Section III-B by giving a more detailed example. However, realizing an algorithm selection solution as an SPF mechanism based on these principles firstly requires SPF architectural extensions, algorithm characterization models, and optimization solutions, which are described in the rest of this section.

A. Extending SPF architectures to support algorithm selection

In Figure 2, we show the modules that typically comprise an SPF middleware. We use our own terminology in order to reflect a generic SPF architecture, which is based on our

analysis and comparison of 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, SPF middleware consists of 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. It resides on every processing node (i.e., server, computer, VM, or any device that hosts tasks). The following components are placed on the two middleware sides:

1) Orchestration middleware side: 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 configuration files or settings, and runs an algorithm to determine which tasks shall run on which nodes. Based on that decision, the *Task Deployer* will remotely deploy the tasks onto the devices. Finally, SPF orchestration middleware solutions usually contain a *Messaging Broker*, which organizes (and sometimes mediates) the delivery of streaming data items (tuples) from task to task.

2) Node middleware side: 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 adding algorithm selection support, we propose the following SPF extensions:

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 reason why the reasoner resides in the node middleware and not in the orchestration middleware is that our optimization is working locally, i.e., it only needs to know about other tasks running on the same node. In contrast to our approach, a global optimization would make sense if we wanted to minimize node load while adhering to end-to-end latency constraints. However, our goal is formulated differently, i.e. as *minimizing end-to-end latency while adhering to node load constraints*. 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, in case this is desired. Finally, the *Node Classifier Agent* cooperates with the Node Classifier for performing and communicating the node classification.

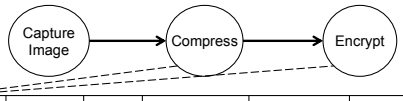
B. Characterization of algorithms for SPF algorithm selection

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. The typically used and well-known metrics of asymptotic computational complexity and memory requirements are unsuited for implementation-specific real-world application.

Simplified, executing a task requires resources (computation steps and memory) and depends on the input size. As stream processing tasks usually act upon individual inputs (often pre-known fixed-size tuples), we can further derive latency as the timespan until the computation on one tuple has finished. This implies the computational power of the hosting device being the next parameter also influencing the actual latency. Due to having constrained devices in our setup, the question is rather what load the algorithms put on the hosting device and how much memory they consume. As explained before, this depends on the tuple input rate, implementation of algorithms, and the hosting node—becoming even more complex as the latter two may differ in distinct combinations.

Based on the above observations, we developed the schema of Figure 3. The data model assumes that there *might* be a set of algorithms that can be employed to implement the same task. This is quite typical for certain categories of algorithms, e.g., compression (as we see in our examples and our evaluation), while it must be also noted that the approach is not limited to algorithms with the strict sense of the term. "Different algorithms" might simply refer to different implementations of the same functionality, whatever this functionality is. Different implementations of the same functionality can certainly quite often have different characteristics with regard to CPU-intensity, RAM-intensity, and other parameters. For each combination of a *Component*, an *Algorithm*, and a *Device Type* (i.e., an attribute

[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.



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. 3. Algorithm characterization schema including examples. Different (alternative) algorithms exist for the Compress and Encrypt tasks. The entries are based on implementations which we benchmarked on our test environment and used in our evaluation. The computed functions may depend more on our implementation, our system, and the used libraries, and not on the algorithms themselves. Some further details, e.g., about the device types, are provided in the evaluation section.

according to a distinct combination of CPU and RAM), we define the following metrics, each obtainable via benchmarking:

- **CPU Load Per Input Rate:** The CPU load (in %) caused by the execution of 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)[2], the processor characteristics, CPU queue characteristics, and various other factors. This exponential increase is explained based mainly on queueing theory in [13], in which it is also explained that various 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%. Since "algorithm" in our context means "any task", while modern compilers and processors are very complex, the easiest way to find this threshold is (automatic) testing.
- **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 3 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 depending on the input rate at the range of interest.

[2]For example, an algorithm which takes two seconds to execute but 99% of that time is used for an I/O operation, is most likely not significantly affected by CPU load. Contrary, operations of a CPU-bound algorithm that find the CPU at 90% utilization will have a 90% probability to be kept waiting.

Independent from the exact schema that we used, the new features that are contributed here lie in *i)* expressing CPU and RAM usage as a function of tuple input rates, *ii)* combining algorithmic efficiency measures with user-defined IoT device categorization, *iii)* formulating algorithmic efficiency in a way that fits the algorithm selection optimization problem, and *iv)* using thresholds based on the previously described observation of how latencies per tuple relate to CPU / RAM load.

C. Algorithm selection optimization problem for SPFs

For realizing algorithm selection, we propose using an optimization problem statement. Although we found no concrete formulations of this problem, some works (e.g., [8]) imply doing so. Therefore, we contribute an optimization problem for algorithm selection that complies with *i)* the technical landscape of SPFs, *ii)* the conceived algorithm characterization schema, and *iii)* assumptions related to SPF technology.

For achieving our low latency goal, we minimize the amount of time that the input tuple stays in the topology until it is completely processed, which we define as Lat. For example, in a rolled out sequential topology with d_z tasks deployed on z devices (D_1, ... D_z) as shown in Figure 4, 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 provide the basis optimization problem according to this example for simplicity. Even for more complex topologies, the basic elements and the problem remain the same, but might require a slightly different formulation. With 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.

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

At this point, another important assumption about functionally equivalent algorithms of stream processing tasks comes into play. They are usually different implementations of either the same or of a very similar functionality. Big differences with regard to the input or the output of the algorithms would probably cause incompatibilities when switching the used algorithm. Therefore, we assume that the algorithm selection does not significantly affect the input or output sizes. Thus, only algorithms obeying this assumption are subject to our dynamic algorithm selection. Based on this definition, the problem is reduced to minimizing each $Lat(D_i)$ separately, since the communication overhead $Lat(L_i)$ remains stable. Further, for simplicity, we will assume in this problem formulation that all tasks belong to the same topology (i.e., that we handle topologies independently of each other), although the problem

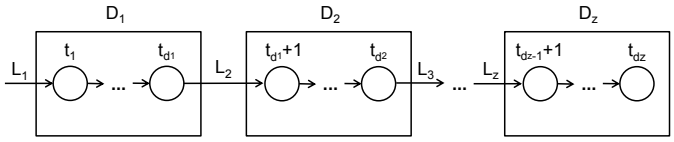


Fig. 4. Visualization of deployed sequential topology and involved elements. Each hosting device D_i executes a set of subsequent tasks t_j. The data flows in a well defined fixed topology through these devices.

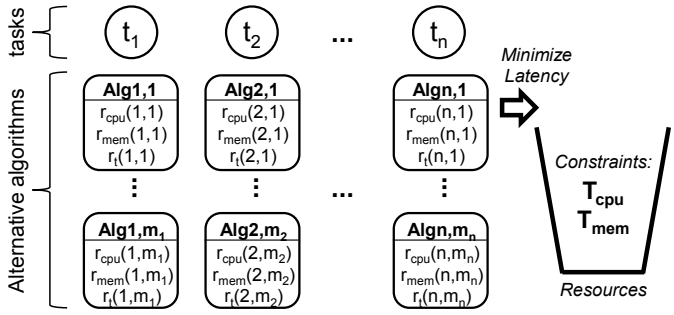


Fig. 5. SPF Algorithm Selection reduced to a Knapsack Optimization Problem. Each task t_j deployed on a single node may have various algorithms with different resource usage characteristics r_{cpu}, r_{mem} and a base latency r_t. The combinatorial choice is to be minimized with regards to latency while adhering to device resource contraints T_{cpu} and T_{mem}.

can be extended to jointly optimize algorithm selection of multiple topologies.

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 5. 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 for simplicity 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:

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

The MMKP is NP-hard and for larger systems it would be difficult to find the optimal solution fast. However, since we reduced the optimization problem to the local scope of one device, there is usually only 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]. .

IV. Evaluation

The presented SPF extensions have been implemented as extensions of Apache Storm and used for this evaluation. As a proof of concept, we demonstrate the latency benefits of algorithm selection for a concrete topology.

A. Setup

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

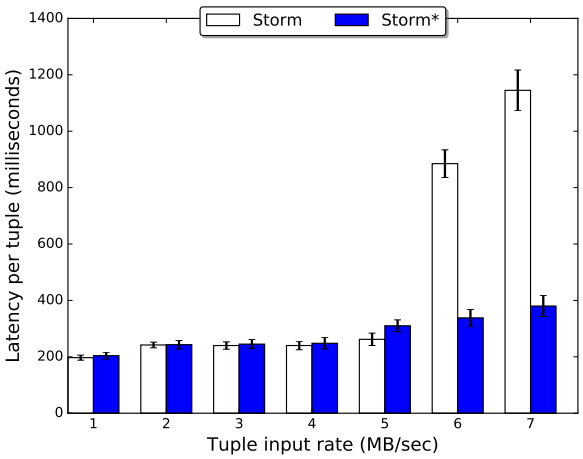
Scenarios and variables: An implementation of the example *topology of Figure 3* has been used for the measurements, containing three alternative algorithms for the Compress task and two alternative algorithms for the Encrypt task, with the exact characteristics shown in Figure 3, which were obtained through test measurements. We performed the experiments for two different device types: Type1 and Type2 were simulated with VMs and reflect IoT GW capabilities. They both run at 3.10 GHz with 2GB RAM, but Type1 has 1 core, while Type2 has 2 cores. Note that Figure 3 shows the numbers only for Type 1, while a similar pattern (with lower CPU/RAM usage and response times) appears for Type 2. For each device type, we varied the tuple input rate. Although we have also performed experiments for different values of the *number of task instances* of each task (i.e., Compress and Encrypt), the results provide no additional insights, so in the selected results that we will discuss, we have fixed this variable at 6 instances for each task.

Metrics and experiment details: The main metric was the average *Latency per tuple* (as described in Section III). In each experiment (i.e., 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. This usually took a few minutes to complete. With a configured frequency of running our algorithm selection reasoning every 5 seconds (configurable), the system had lots of possibilities to actually switch between executed algorithms many times. However, due to the input rate being stable throughout an entire run, Storm* did not change its decisions often in a single run. An additional metric that we evaluate is the average *CPU load* of the hosting devices, because it has a huge impact on the latency results.

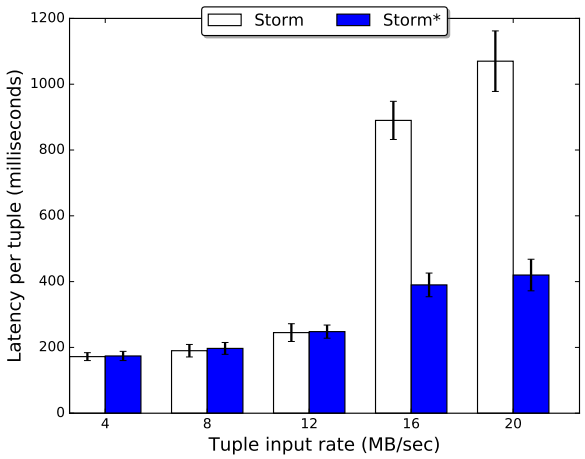
B. Results and discussion

Based on the results, which are presented in Figures 6 and 7, we can make the following main observations:

- **Storm* and Storm perform similarly in cases of lower utilization**: For lower tuple input rates, i.e., up to 5 MB/s

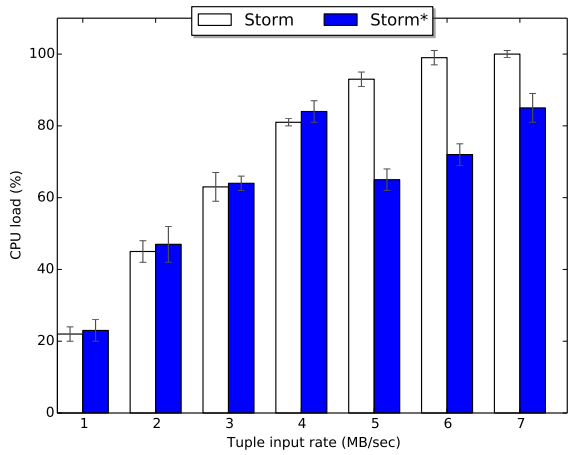


(a) Avg. processing latency for Device Type1 (one CPU core)

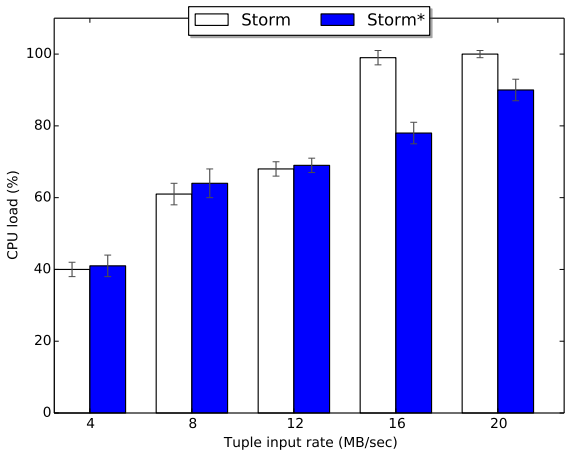


(b) Avg. processing latency for Device Type2 (two CPU cores)

Fig. 6. 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.



(a) Avg. CPU load for Device Type1 (one CPU core)



(b) Avg. CPU load for Device Type2 (two CPU cores)

Fig. 7. Average CPU load 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 getting very close to 100% utilization for higher tuple input rates, our solution keeps CPU utilization at comparably very low levels, which contributes hugely to achieving lower processing latency per tuple.

(12 MB/s) for Device Type 1 (2), no significant differences can be observed as shown in Figure 6. 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.
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 6) or because of little overhead due to the additional monitoring and control functions of Storm*.

- **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%.

- **Storm* achieves low latency by avoiding CPU bottlenecks**: As shown in Figure 7, the CPU load for lower input rates was comparable for the two approaches, as the available resources were always sufficient. The slightly higher CPU load of Storm* in some cases with lower input rates might have happened again either due to chance or due to the monitoring and MKKP-solving software. However, as the input rate increases, Storm* switches to alternative algorithms which require less CPU load and have (normally) higher response times compared to the fastest available alternative. 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. This is the main reason of the latency results of Figure 6, as well.

V. Conclusion

Due to their nature, Stream Processing Frameworks are becoming more and more the solution of choice in low latency IoT environments as usual Big Data (store first, process afterwards) approaches are lacking 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 describe our solution in a new generic model of todays SPFs which we extended by new components for this purpose, *ii)* we base our selection technique on a new method of modeling practical algorithm characteristics in terms of stream processing on a resource-wise variety of devices, *iii)* we handle the problem mathematically, we formulate algorithm selection as an optimization problem—an instance of the Multi-choice Multi-dimensional Knapsack Problem: *locally minimize the latency of subsequent processing tasks*, while adhering to device resource constraints.

We show the potential of dynamic algorithm selection by evaluating a simple real-world scenario within our implementation in Apache Storm. In comparison to vanilla Apache Storm, we show latency reductions of up to a factor of 2.9. With that, we pave the way to significantly reduce latency by applying algorithm selection to SPFs. However, our approach may be extended into several directions, e.g., with regard to reducing the offline overhead of the introduced modules, finding optimal time intervals for running the algorithm selection process, and investigating the joint optimization of algorithm selection with other stream processing optimizations such as task placement.

References

[1] C.L. Philip Chen and Chun-Yang Zhang. Data-intensive Applications, Challenges, Techniques and Technologies: A Survey on Big Data. *Information Sciences*, 275:314–347, 2014.

[2] Murat Kuzlu, Manisa Pipattanasomporn, and Saifur Rahman. Communication network requirements for major smart grid applications in HAN, NAN and WAN. *Computer Networks*, 67:74–88, 2014.

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

[4] Apache Software Foundation. Apache Samza project. http://samza.apache.org/ (last visited in 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 Int. Conference on Management of Data*, SIGMOD '15, pages 239–250. ACM, 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 Int. Conference on Management of Data*, SIGMOD '13, pages 725–736. ACM, 2013.

[7] Guangxiang Du and Indranil Gupta. New Techniques to Curtail the Tail Latency in Stream Processing Systems. In *4th Workshop on Distributed Cloud Computing*, DCC '16, pages 7:1–7:6. ACM, 2016.

[8] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A Catalog of Stream Processing Optimizations. *ACM Computing Surveys*, 46(4):1–34, March 2014.

[9] Niko Pollner, Christian Steudtner, and Klaus Meyer-Wegener. Operator Fission for Load Balancing in Distributed Heterogeneous Data Stream Processing Systems. In *9th ACM Int. Conference on Distributed Event-Based Systems*, DEBS '15, pages 332–335. ACM, 2015.

[10] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *11th Int. Conference on Compiler Construction*, CC '02, pages 179–196. Springer, 2002.

[11] Jielong Xu, Zhenhua Chen, Jian Tang, and Sen Su. T-Storm: Traffic-Aware Online Scheduling in Storm. In *IEEE 34th Int. Conference on Distributed Computing Systems*, ICDCS '14, pages 535–544. IEEE, 2014.

[12] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive Online Scheduling in Storm. In *7th ACM Int. Conference on Distributed Event-based Systems*, DEBS '13, pages 207–218. ACM, 2013.

[13] Charles Hooper. Faulty Quotes 6 CPU Utilization, February 2010. https://hoopercharles.wordpress.com/2010/02/05/faulty-quotes-6-cpu-utilization/ (last visited in 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 *39th Euromicro Conference on Software Engineering and Advanced Applications*, SEAA '13, pages 357–364. IEEE, 2013.

[15] Bing Han, Jimmy Leblet, and Gwendal Simon. Hard Multidimensional Multiple Choice Knapsack Problems, an Empirical Study. *Computers and Operations Research*, 37(1):172–181, January 2010.