Journal of Parallel and Distributed Computing 137 (2020) 205–223

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Joker: Elastic stream processing with organic adaptation

Basri Kahveci *, Buğra Gedik

Bilkent University, 06800, Ankara, Turkey

ARTICLE INFO

Article history:
Received 31 January 2018
Received in revised form 18 March 2019
Accepted 21 October 2019
Available online 16 December 2019

Keywords:
Stream processing
Elasticity
Parallelization

ABSTRACT

This paper addresses the problem of auto-parallelization of streaming applications. We propose an online parallelization optimization algorithm that adjusts the degree of pipeline and data parallelism in a joint manner. We define an operator development API and a flexible parallel execution model to form a basis for the optimization algorithm. The operator interface unifies the development of different types of operators and makes operator properties visible in order to enable safe optimizations. The parallel execution model splits a data flow graph into *regions*. A region contains the longest sequence of compatible operators that are amenable to *data parallelism* as a whole and can be further parallelized with *pipeline parallelism*. We also develop a stream processing run-time, named *Joker*, to scale the execution of streaming applications in a safe, transparent, dynamic, and automatic manner. This ability is called *organic adaptation*. Joker implements the runtime machinery to execute a data flow graph with any parallelization configuration and most importantly change this configuration at run-time with low cost in the presence of partitioned stateful operators, in a way that is transparent to the application developers. Joker continuously monitors the run-time performance, and runs the optimization algorithm to resolve bottlenecks and scale the application by adjusting the degree of pipeline and data parallelism. The experimental evaluation based on micro-benchmarks and real-world applications showcase that our solution accomplishes elasticity by finding an effective parallelization configuration.

1. Introduction

As the world is becoming more connected and instrumented, we have been witnessing an increasing interest in systems that can process continuous streams of data in near real time. Telecommunications, trading platforms, manufacturing systems, network monitoring systems, and health-care systems are just a few examples of the domains that can make use of stream processing systems to analyze high rates of data streams swiftly and extract actionable insights.

Stream processing is a computational paradigm that processes large volumes of continuous data streams in an on-the-fly manner. The high throughput processing requirement of streaming applications necessitates taking advantage of multi-processors and multiple machines. Streaming applications are generally represented as data flow graphs, where operators are generic data manipulators and streams connect operators to each other using FIFO semantics. The data flow graph representation allows developers to express their computations easily without handling the complexity of concurrent and distributed nature of streaming applications. Additionally, it captures a rich set of parallelization opportunities, such as pipeline, task, and data parallelism [18]. However, there are several challenges in taking advantage of parallelism to scale streaming applications, while keeping the simplicity of the data flow graph representation.

First, it is not a straightforward task to find a good degree of parallelism for a streaming application. The number of possible parallelization configurations is exponential in the size of the data flow graph [11]. Unlike in relational databases, where a small set of relational operators with well-studied semantics and cost models exist, streaming applications usually have operators with user-defined functions (UDFs), whose costs may depend on data stream rates and data distributions. Furthermore, streaming applications have a long-running nature with highly dynamic workloads. An efficient solution should continuously apply parallelization optimizations and adjust the mapping of logical application pieces to available computing resources to achieve elasticity.

Second, in order for streaming applications to flourish, we need toolkits of generic streaming operators. Since operator properties such as arity, selectivity, and state play critical roles in the application of parallelization methods, operator development APIs should achieve two things at once: (i) ease the development of generic, reusable operators, (ii) provide visibility into the

* Corresponding author.
E-mail addresses: basri.kahveci@bilkent.edu.tr (B. Kahveci), bgedik@cs.bilkent.edu.tr (B. Gedik).

https://doi.org/10.1016/j.jpdc.2019.10.012
0743-7315/

operator properties so that parallelization optimizations can be applied in a safe, effective, and profitable manner.

Third, an integrated middleware solution that performs automatic and transparent parallelization optimizations is needed. Having such a solution is a major challenge because of the discord between the workload and resource dynamicity of streaming applications, and the ability of the runtime systems to adapt. Additionally, applying parallelization optimizations becomes harder in the presence of stateful operators. An integrated solution should free the developers from the burden of hand-optimizing streaming applications by detecting and resolving bottlenecks automatically. It should also cooperate with user code to handle the heavy lifting in a transparent way, such as state migration and preservation of sequential execution semantics while doing performance optimizations.

We address the challenge of parallelizing streaming applications by defining a flexible parallel execution model and developing an effective online optimization algorithm that adjusts the parallelization configuration dynamically at run-time. First, we extend the *pipelined fission* parallel execution model, first introduced in [11], which splits a data flow graph into subgraphs called *regions*. A region contains the longest sequence of compatible operators that are amenable to data parallelism as a whole and can be further divided into *pipelines* to apply pipeline parallelism. Second, the parallelization optimization algorithm detects bottlenecks in the execution of streaming applications and resolves these bottlenecks by adjusting the degree of pipeline and data parallelism in a joint manner. We determine a set of metrics to characterize the performance of a parallelized execution. Using these metrics, the parallelization optimization algorithm identifies bottlenecks, decides on a series of parallelism changes to resolve the bottlenecks, and improves the throughput, all at run-time. While making parallelism changes, the algorithm takes operator properties and interactions of the parallelization optimization techniques into consideration. It also evaluates the profitability of parallelism change decisions. If a parallelism change does not improve execution performance, it is reverted and blacklisted. Last, profitability evaluations are adaptive to the dynamicity of run-time workloads.

We address the second challenge by introducing an operator development API that preserves the simplicity of the data flow graph representation, while at the same time making the relevant operator properties visible to the runtime for performing safe parallelization optimizations. The operator development API defines a unified interface for stateless, stateful, and partitioned stateful computations via a set of tuple processing, operator scheduling, and state management primitives. This API allows processing of tuples one-by-one or in batches, and simplifies the development of source, sink, and processing operators with multiple input/output ports. Consider a simple `Barrier` operator which is commonly used in streaming applications. It takes one tuple from each input port and combines them into a single output tuple. The operator implementation is required to buffer incoming tuples until it receives at least one tuple from each input port. More importantly, it needs to apply back pressure on its input ports in order to limit the memory usage of the internal buffer, especially when its input ports receive tuples at different rates. Our operator development API frees the developers from the burden of such complex tasks for implementing the `Barrier` operator.

Last, but not the least, we address the challenge of having an integrated solution by developing an elastic stream processing engine, named *Joker*. Joker is able to scale the execution of a streaming application in a safe, transparent, dynamic, and automatic manner. This ability is called *organic adaptation*. Based on its parallel execution model, Joker implements the runtime machinery to execute a given data flow graph with any parallelization configuration and most importantly change this configuration at run-time with low cost, in a way that is transparent to the operator implementations. Joker continuously monitors the run-time performance and feeds the measurements into the parallelization optimization algorithm. During parallelism changes, Joker migrates operator state transparently, thanks to the state management primitives present in the operator development API. It capitalizes on the consistent hashing algorithm [19] to minimize the amount of migrated operator state. Joker is developed as an open source[1] single-host runtime to demonstrate the usefulness of our contributions. However, the parallelization optimization algorithm, the operator development API, and the runtime techniques we introduce are extensible to distributed settings and applicable to any stream processing engine.

In summary, this paper makes the following contributions:

- It introduces a flexible parallel execution model for streaming applications, and to the best of our knowledge, the first online parallelization optimization algorithm that resolves bottlenecks of a streaming application by adjusting the degree of pipeline and data parallelism in a joint manner.
- It proposes an operator development API that greatly simplifies the development of stateless, stateful, and partitioned stateful operators, while at the same time making operator properties visible to the runtime in order to enable safe parallelization optimizations.
- It develops the accompanying runtime machinery to execute a streaming application in a parallelized manner, characterize its performance with a metric set, and carry out parallelism changes at run-time transparently to the user code, in the presence of stateful and partitioned stateful computations.
- It presents a detailed evaluation of the parallelization optimization algorithm to demonstrate its effectiveness.

The rest of the paper is organized as follows. Section 2 elaborates the terminology we use in the paper. Section 3 explains Joker's operator development API, parallel execution model, and runtime abilities. Section 4 introduces Joker's parallelization optimization algorithm and Section 5 presents the experimental evaluation. Section 6 discusses related work and Section 7 concludes the paper.

2. Background

We model a streaming application as a directed acyclic graph (DAG) of operators. An operator is a logical unit that carries out a piece of the computation in a continuous manner. It receives tuples from its input ports, processes them, and emits output tuples via its output ports. Selectivity of an operator is the number of emitted output tuples for each input tuple. Operators with no input ports are called source operators and they introduce tuples to the execution. Operators with no output ports are called sink operators. Streams are the connections that transfer unbounded sequence of tuples from upstream operators to downstream operators with FIFO semantics.

Streaming operators may or may not have *state*, which is a piece of data maintained between firings of incoming tuples. Namely, streaming operators can be *stateless*, *stateful*, and *partitioned stateful*. Stateless operators do not maintain state across tuples. Stateful operators maintain arbitrary state which may depend on the entire history of the stream. Partitioned stateful operators maintain independent state across partitions which are determined based on a given *partitioning key attribute*.

In this paper, the DAG representation of a streaming computation is called a *data flow graph*. A data flow graph is realized

[1] Source code is available at https://github.com/metanet/joker.

for execution with a *parallelization configuration*, which contains the number of replicas for each operator, and mappings between operator replicas and available computation resources. An operator can be instantiated with multiple replicas in a parallelization configuration. Execution of a data flow graph can be parallelized by utilizing several parallelism techniques. In this study, we focus on three:

- **pipeline parallelism**: An upstream operator can execute concurrently with a downstream operator.
- **task parallelism**: If an operator is connected to multiple downstream operators, each downstream operator can perform a separate task on the data stream concurrently.
- **data parallelism**: If an operator is being executed with multiple replicas, each operator replica can process a different portion of the data stream concurrently. The input stream of the operator can be distributed among its replicas in a round-robin fashion or based on the value of a partitioning key attribute, which exists in all tuples of the input stream. Data parallelism is applicable to stateless and partitioned stateful operators. It is a particularly favorable parallelism technique because it is not limited by the number of operators in the data flow graph. However, a data parallel execution should preserve the sequential execution semantics, that is, it should produce the same result as if each operator has a single replica.

Elasticity is the ability of stream processing engines to adapt to varying workloads, operational changes, and availability of computing resources by applying various optimization techniques at run-time. We focus on two techniques in this study: *fusion* and *fission*. Fusion is a technique that trades communication cost against pipeline parallelism [18]. Cheap operators are fused into the same execution unit (i.e., thread) to eliminate communication overhead. The fission technique replicates an operator and splits the input stream over its replicas to achieve data parallelism [18].

The life cycle of a streaming application consists of three fundamental stages. First, operators of the streaming application are coded. We name this stage *design-time*. Then, the application is formed by configuring operators for execution and connecting them via streams. This stage is called *composition-time*. The last stage is *run-time*, in which the configured operators are instantiated and executed by a stream processing engine.

3. Solution overview

In this section, we give a brief overview of our solution. We first introduce the building blocks of our organic adaptation solution, which are the operator development API and the parallel execution model. Then, we describe Joker's runtime system and elaborate on how it realizes the organic adaptation solution.

3.1. Operator development API

We introduce an operator development API that supports the development of flexible, generic, and reusable operators. Such an API enables strong cooperation between the user-defined operator code and the runtime, and extends runtime's understanding of the operator behavior for performing parallelization optimizations in a safe manner.

First, the operator development API creates an isolation layer between operator implementations and the runtime. Once an operator is invoked, it receives a batch of tuples in an input object, processes them, and emits the resulting tuples with an output object. The input and output objects work as buffers. This approach enables the runtime to define boundaries of an operator invocation precisely. Thus, the runtime can perform execution and parallelization optimization tasks transparently to operators, such as applying back pressure, making parallelism changes, and migrating operator state. Additionally, this approach brings up the opportunity of batching optimizations.

Second, we follow a different approach than the prevalent event-driven model for operator invocations. SPL [16], S4 [22], Apache Samza,[2] Apache Storm,[3] Apache Flink,[4] Apache Heron,[5] Hazelcast Jet,[6] ChronoStream [25], and many other solutions employ the event-driven model for operator invocations. In the event-driven model, the operator code is invoked when a tuple arrives at one of its input ports. Since there is no guarantee for the ordering of tuple arrivals across different ports, a multi-input port operator should buffer tuples if they arrive out of order, limit memory usage, and apply back pressure on its own. Our operator development API contains an operator scheduling API which frees operator implementations from the burden of such complex tasks. Operators specify the required number of tuples for their input ports using a flexible specification scheme that supports conjunctive and disjunctive requirements. If a requirement is conjunctive, the operator is invoked when the required number of tuples are available at all of its input ports. If it is disjunctive, the operator is invoked when the required number of tuples are available at any of its input ports. In this model, the runtime becomes responsible for handling the complexity of satisfying the tuple requirements for multi-input port operator invocations. For instance, it is sufficient to declare just the number of input ports and tuple requirements to develop a *barrier* operator, which expects one tuple from each input port.

Third, the operator development API contains a schema specification API to unify the development of task-specific and generic operators. A task-specific operator defines all of its properties at design-time where a generic operator can provide these properties partially at design-time and extend them at composition-time when the data flow graph is formed for execution. These properties are operator state type (stateless, stateful, partitioned stateful), partitioning key attributes if the operator is partitioned stateful, input/output port counts, and input/output schemas. Moreover, operator properties determine the type and scope of various applicable parallelization optimization techniques.

Fourth, we enable stateful computations with a set of state management APIs. An operator can maintain its computation state using an in-memory state object, which is provided by the runtime during invocations. The runtime creates a single global state object for a stateful operator and independent state objects for different partitioning key instances of a partitioned stateful operator. When a stateful operator is invoked, its state object is passed to the operator. Partitioned stateful operators are invoked on a partitioning key instance basis. Joker runtime invokes a partitioned stateful operator for a particular partitioning key instance, along with the input tuples and the state object maintained for that partitioning key instance. Thus, Joker greatly simplifies the development of partitioned stateful computations. The state management API also forms the foundation of the runtime capabilities, such as load balancing and transparent operator migration. Albeit not implemented, the runtime can also optimize the storage of externalized operator state by compression or overflowing to disk.

Last, but not the least, our operator development API abstracts parallelization of streaming applications from the specification of these applications. For instance, Apache Storm, Apache Flink, and Apache Spark's programming interfaces contain methods for

[2] Apache Samza website: http://samza.apache.org retrieved in March 2019.
[3] Apache Storm website: http://storm.apache.org retrieved in March 2019.
[4] Apache Flink website: https://flink.apache.org retrieved in March 2019.
[5] Apache Heron website: https://github.com/apache/incubator-heron retrieved in March 2019.
[6] Hazelcast Jet website: https://jet.hazelcast.org retrieved in March 2019.

```
interface Operator {
  SchedulingStrategy initialize(InitializationContext context);
  void invoke(InvocationContext context);
  default void shutdown() {}
}
```

Fig. 1. Joker operator API.

developers to specify various parallelization hints, such as how many replicas to create for an operator, how to chain operators, etc. In contrast, our API frees the developers from such burdens. Streaming applications are specified in a way that is free from any parallelization hint, and Joker runtime parallelizes these applications obliviously to the developers.

Operators implement the `Operator` interface, shown in Fig. 1. Operators receive their properties and configurations, initialize their internal state, and return their scheduling requirements in the `initialize()` method. Dually, they perform clean up tasks in the optional `shutdown()` method. They implement their computation logic in the `invoke()` method.

Fig. 2 depicts a specific barrier operator implementation. It is a partitioned stateful operator with 2 input ports and 1 output port. As it is marked as partitioned stateful, it receives a state object for each invocation. `SumBarrierOperator` returns a scheduling requirement indicating that the operator must be invoked when there are exactly 10 tuples on both input ports for a partitioning key instance. This requirement enables processing of tuples in batches, as each invocation is done with 20 tuples in total. The operator takes a tuple from each input port and finds the maximum value of the `num` field in these tuples. Then, it calculates the summation of these values for all tuples in the batch, and adds it to the value stored in the given state object for each partitioning key instance. It also sends this value to the output port. The runtime creates a separate state object for each partitioning key instance. Port counts of this operator are defined at design-time. However, it is possible to define them at composition-time as well, when the data flow graph is formed for execution. Moreover, the partitioning key attribute is also defined at composition-time.

3.2. Parallel execution model

The main goal of the optimization algorithm is to automatically discover the parallelization configuration that maximizes the throughput of a streaming application at run-time. To that end, we first formalize our parallel execution model based on the application of fusion and fission techniques [18]. We extend the parallel execution model developed by Gedik et al. in [11]. Their parallel execution model works on chain topologies only, where there is no branching, i.e., operators are organized into a series. On the other hand, our solution can parallelize topologies that contain branches. We partition the data flow graph into regions using the longest compatible operator sequence principle. A region contains operators for which data parallelism is applicable as a whole. Once we discover the regions, we split the sequence of operators in a region into pipelines. We run pipelines concurrently on separate threads to achieve pipeline parallelism. We further create multiple replicas of pipelines to achieve data parallelism. Each pipeline replica performs the same sequence of tasks in parallel on a different portion of the data stream.

Second, we define the rules and constraints to discover region boundaries on top of the operator development API. We form the regions starting from the source operators. As we proceed through operators of the data flow graph, we add them into the current region, as long as their state types and schemas are compatible with the current operators in the region. When we encounter an incompatible operator, we terminate the current region and start a new one. We define the compatibility rules below. Our reasoning is that streaming applications tend to reduce the volume of the data stream as the data flows through stages of the computation [3]. Hence, our approach is to form long regions and assign more processing resources for early stages as much as possible, in order to have more opportunities for scaling.

Our region formation rules are as follows:

- Source operators have their independent regions, called *source regions*. Source regions are excluded in the optimization process.
- Operators in a region can have at-most-one predecessor and successor. If we encounter a branching in the data flow graph, we

```
@OperatorSpec(type=PARTITIONED_STATEFUL, inputPortCount=2, outputPortCount=1)
public class SumBarrierOperator implements Operator {
  @Override
  public SchedulingStrategy initialize(InitializationContext context) {
    OperatorConfig config = context.getOperatorConfig();
    SchedulingStrategy strategy = new SchedulingStrategy();
    strategy.setPartitionKeyFieldName(config.get("partitionKeyFieldName"));
    strategy.setPortCount(2);
    strategy.setTupleAvailabilityMode(EXACT);
    strategy.setTupleCount(10);
    return strategy;
  }

  @Override
  public void invoke(InvocationContext context) {
    Object key = context.getPartitionKey();
    int batchSum = 0;
    for (int i = 0; i < 10; i++) {
      Tuple tuple0 = context.getInputTuple(0, i);
      Tuple tuple1 = context.getInputTuple(1, i);
      int max = Math.max(tuple0.getInt("num"), tuple1.getInt("num"));
      batchSum += max;
    }
    System.out.println("key:" + key + ",batchSum:" + batchSum + ",sum:" + sum);
    context.addOutputTuple(Tuple.of("key", key, "batchSum", sum));
    StateStore stateStore = context.getStateStore();
    int sum = stateStore.getOrDefault("sum", 0);
    stateStore.set("sum", (sum + batchSum));
  }
}
```

Fig. 2. `SumBarrierOperator` implementation.

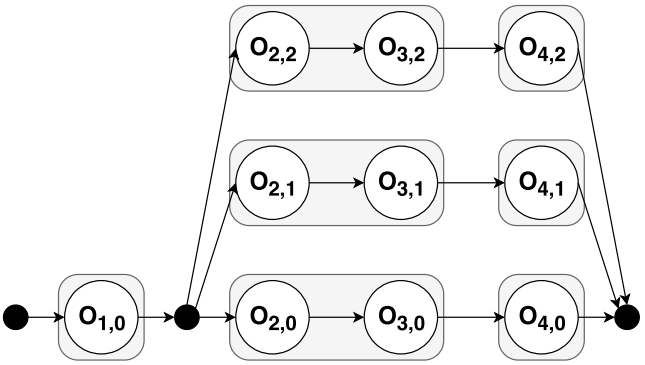


Fig. 3. A sample parallelization configuration.

terminate the current region and start new regions. This implies that each region has a single entry and exit point. If an operator has multiple downstream connections, each downstream operator starts its own region, and output tuples of the operator are forwarded to all downstream regions.

• A data parallel region starts with a stateless or partitioned stateful operator, and can contain only these two operator types. We preserve the safety of the computation for data parallel regions. A data parallel region should always produce the same result with its sequential counterpart, no matter which parallelization configuration it runs with. Tuples should leave the region in the same order they enter the region. Earlier studies either ignore order preservation [15,20,23,26], or use a variety of splitters and mergers to achieve perfect order [12]. We optimize for the general scenario and guarantee the sequential execution semantics only for tuples having the same partitioning key instance, i.e., ordering is not preserved across tuples belonging to different partitions. For this reason, we require data parallel regions to contain at least one partitioned stateful operator. Joker runtime distributes the incoming data stream using a hashing-based scheme. It directs tuples that have the same partitioning key instance to the same replica of the region. Thus, we eliminate the need for explicit mergers and reduce the parallelization overhead. Our experience with real-world stream processing applications has shown that ordering across partitions is uncommon. For the scenarios where it is needed, explicit operators could be added into the data flow graph to achieve the desired ordering.

• Partitioning key attributes of the first partitioned stateful operator of a data parallel region become partitioning key attributes of the region. Subsequent partitioned stateful operators must be compatible with the region, i.e., their partitioning key attributes must contain partitioning key attributes of the region. Similarly, input schemas of stateless operators must contain partitioning key attributes of the region. With these rules in place, once a tuple enters a data parallel region with a partitioning key instance, the same key is used for all invocations of the tuple along the way, and the ordering is preserved for the tuples having the same key. Moreover, when the degree of data parallelism is changed for a region, some of the keys are moved between region replicas, and all tuples belonging to those keys and waiting in the input queues of the operators are moved between region replicas without breaking tuple ordering.

• After we form all source regions and data parallel regions, we combine rest of the operators with respect to *the at-most-one predecessor and successor rule*. These regions can contain stateless and stateful regions together, but not partitioned stateful operators, and they are amenable to only pipeline parallelism. In other words, they can have multiple pipelines with a single replica.

Fig. 3 displays a parallelization configuration of a data flow graph with 4 operators: o_1, o_2, o_3, and o_4. The data flow graph has 2 regions: (i) the left-hand region is the source region with a single pipeline and a single replica, (ii) the right-hand region has 3 operators that are parallelized by 2 pipelines and 3 replicas, where the first pipeline consists of o_2 and o_3, and the second pipeline consists of o_4. Operator replicas are depicted as circles and pipelines are depicted as rectangles surrounding the operators. The black dots represent region boundaries and the arrows represent the direction of the tuple streams. In this figure, output tuples emitted by the source operator are distributed among the three replicas of the second region.

3.3. Joker runtime

Joker is implemented with Java 8[7] as a single-host stream processing runtime. It utilizes multi-threaded execution to realize the parallelism techniques described in Section 3.2. In detail, it assigns pipeline replicas to Java threads. For instance, Joker runs the parallelization configuration given in Fig. 3 with 7 threads. During execution, a Joker thread traverses operators of its pipeline replica and comes back to the first operator for the next traversal. As operators can have different scheduling requirements, Joker places tuple queues between operators. Tuple queues between operators of the same pipeline are implemented as single-threaded queues, whereas the queues between operators of different pipelines are implemented as concurrent lock-free queues. Joker employs back pressure by bounding the size of the tuple queues.

One of our major contributions is Joker runtime's ability to auto-scale a streaming application in the presence of stateful operators. The first major challenge of the auto-scaling ability is preserving safety, which is solved by the operator development API and region formation rules, as described in Sections 3.1 and 3.2.

The second challenge is employing the auto-scaling ability in a transparent manner without interfering with the user code. We address this challenge by developing a key–value store based state management API for stateful and partitioned stateful operators. Joker runtime creates independent state objects for different partitioning key instances and guarantees that a partitioned stateful operator processes tuples of the same partitioning key instance using the same state object.

The third major challenge of auto-scaling is to perform parallelism changes dynamically at run-time, while the streaming application is running. In this regard, Joker runtime can split a pipeline into smaller pipelines, merge subsequent pipelines of a region into a longer pipeline, and change the number of replicas of a data parallel region. When the number of replicas is changed, partitioning key instances are re-distributed across the region replicas and the associated state, including the tuples in the operator queues and operator-specific state objects stored in key–value stores, is transparently migrated. Since Joker is implemented as a single-host runtime, operator state objects and queued tuples are moved between operator replicas during migrations. Therefore, partitioning key re-distribution does not involve serialization/deserialization cost. Nevertheless, we minimize the amount of migrated state using the consistent hashing algorithm [19].

4. Organic adaptation

The essence of our solution is its ability to infer the workload and resource dynamics of a streaming application at runtime, and to scale the application automatically by improving

[7] Java 8 website: http://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html retrieved in March 2019.

its parallelization configuration. We liken this behavior to biological organisms' capability to adapt to their environment for survival. Consequently, we name our solution as *organic adaptation*, and the parallelization optimization algorithm as *adaptation algorithm*.

When Joker runtime starts executing a streaming application, it creates a single pipeline and a single replica for each region. Then, it runs a three-phase scheme to incrementally improve the parallelization configuration of the application as follows:

- **Profiling**: In the first phase, a lightweight profiler collects several metrics to be used in the adaptation phase.
- **Adaptation**: The second phase starts by identifying bottleneck pipelines. After that, the adaptation algorithm investigates possible parallelism changes to resolve the identified bottlenecks. Then, Joker runtime applies the changes that are estimated to provide the largest improvement to the execution performance.
- **Evaluation and Control**: In the final phase, the adaptation algorithm evaluates the performance of the execution with the resulted parallelization configuration, and takes one of two possible actions. If the new configuration improves performance, the algorithm persists the applied changes. Otherwise, it reverts them and goes back to the second phase to look for other possible changes. The organic adaptation solution runs until it notices that the execution is stabilized, and no more parallelism changes occur afterwards.

4.1. Profiling

Joker runtime collects three types of profiling metrics to feed the adaptation algorithm:

- It collects CPU utilization ratios of pipeline replica threads to discover bottlenecks. It divides the amount of CPU time a pipeline replica thread utilizes by wall clock time to calculate its CPU utilization ratio. Threads' CPU times are fetched via *JMX* interfaces. Our reasoning is that a heavily utilized thread turns into a bottleneck. If we assign more computing resources to a bottleneck pipeline, we would be able to improve performance.
- For each pipeline replica thread, Joker runtime estimates how its CPU time is distributed among its operators. We define a metric, *operator cost*, to be the portion of CPU time a pipeline replica thread spends on an operator. For instance, if a pipeline contains 2 operators, *o1*, *o2*, and their costs are estimated as 0.4 and 0.5, it means that *o1* consumes 40% of its pipeline replica threads' CPU times. Note that these operator costs do not add up to 1. This is because execution of a pipeline involves CPU consuming tasks as well, such as moving tuples across operators and pipelines, partitioning streams, and managing state objects. We name the portion of the CPU time a pipeline replica thread spends for such tasks as the *pipeline overhead*. We estimate the pipeline overhead by $1 - \sum_{i=1}^{n} cost_i$ where n is the number of operators in the pipeline and $cost_i$ is the operator cost. In the example above, the pipeline overhead is $1 - (0.4 + 0.5) = 0.1$.
- Throughput is measured as the number of tuples processed for a time period. We use region throughput values as our evaluation metric. Our solution tries to optimize the throughput of the source regions by adjusting the degree of pipeline and data parallelism of each region in the data flow graph.

Since Joker runtime executes each pipeline replica with a separate thread, the profiler calculates pipeline metrics by averaging metrics of each pipeline replica. The profiler runs each second. In each run, it collects CPU utilization ratios, operator costs, and region throughput values. We run the profiler for 30 s and calculate the exponential average of the metric values collected in each run.

Algorithm 1: Adapt(T[], C[])

```
Input: T[]: New throughput values of regions
Input: C[]: New CPU util. ratios of regions
Result: Multiple regions can be adapted.
Data: Regions: Regions topologically sorted
Data: A: Currently adapting regions
if A.Length = 0 then
    foreach R ∈ Regions do
        UpdateMetrics(R, T[R], C[R])
    foreach R ∈ Regions do
        if ResolveIfBottleneck(R) then A += R
else
    N ← GetNonResolvedRegion(A, T, C)
    if N = null then
        foreach R ∈ A do
            Finalize(R, T[R], C[R])
        A ← []
    else
        Revert(N)
        if !ResolveIfBottleneck(N) then
            foreach R ∈ A \ N do
                Revert(R)
            A ← []
```

Our profiling mechanism is a lightweight solution. It relies on Java *volatile* keyword semantics[8] for metric collection. As described in Section 3.3, a pipeline replica thread traverses its operators and comes back to the first operator. For some of the traversals, it stores the new metric values and the currently running operator into a volatile field. Dually, the profiler reads this field from its own thread. We employ a sampling mechanism to estimate operator costs. The profiler counts how many times it encounters an operator in this volatile field. If it makes N reads in the last second and encounters a particular operator C times, then the operator cost is estimated as C/N.

4.2. Adaptation algorithm

Recall that a region is the longest sequence of operators for which data parallelism is applicable as a whole. Regions can be further divided into pipelines to utilize pipeline parallelism, and multiple replicas can be created for pipelines to utilize data parallelism. The adaptation phase aims to resolve bottlenecks and increase the throughput by changing the degree of pipeline and data parallelism incrementally.

Algorithm 1 presents the `Adapt()` procedure, which is the main entry point of the adaptation algorithm. It works on regions. It maintains internal state for pipeline and replica configurations, metric values, and adaptation process of each region.

The `Adapt()` procedure is invoked with the metrics collected by the profiler. When there is no ongoing adaptation, it first calls the `UpdateMetrics()` procedure to update the metrics of each region. Then, it iterates over the regions to resolve bottlenecks. A region is considered as a bottleneck if it contains at least one bottleneck pipeline. Bottleneck regions are resolved in the `ResolveIfBottleneck()` procedure and are collected into a variable. The `Adapt()` procedure takes no action if no bottleneck or no candidate parallelism change is found.

Our adaptation algorithm is inherently a greedy solution. It improves the throughput of a streaming application by applying parallelism changes incrementally at each adaptation period. In each period, the adaptation algorithm changes the degree of parallelism for all bottlenecks at once. Another approach would

[8] Java 8 language specification: http://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html retrieved in March 2019.

Algorithm 2: ResolveIfBottleneck(R)

```
Input: R: Region to resolve bottleneck pipelines if present
Result: If bottleneck pipelines are detected and could be resolved
        with a non-blacklisted parallelism change, the new
        parallelization configuration is applied.
Output: true if a parallelism change is applied, false otherwise
B ← FindBottleneckPipelines(R)
if B = null then return false

CFG ← SplitBottleneckPipelines(R, B)
if CFG = null then CFG ← ExpandRegion(R, B)

if CFG = null then
    return false
R.A ← B                                  ▷ Set adapting pipelines
R.CFGbase ← R.CFG                        ▷ Save current par. config
R.CFG ← CFG                              ▷ Set new par. config
ApplyParallelizationConfiguration(R)
return true
```

Algorithm 3: SplitBottleneckPipelines(R, B)

```
Input: R: Region of the bottleneck pipelines
Input: B: Bottleneck pipelines
Output: New config if pipelines are splitable and splits are
        profitable, null otherwise
Threshold: UT: Min utility value to make a split
A ← []
foreach P ∈ B do
    if R.Pipelines[P].Length = 1 then return null

    C ← R.C[P]
    L ← P.Length
    S = argmin_{k∈1..(L−1)} ‖C[1..k] − C[k + 1..L]‖
    ▷ If distr. of operator costs is not good enough...
    if GetUtility(C, S) < UT then return null

    CFG ← R.CFG + S                      ▷ Create split pipeline config
    if CFG ∈ R.Blacklist[P] then return null

    A += CFG
return A
```

Algorithm 4: ExpandRegion(R, B)

```
Input: R: Region of the bottleneck pipelines
Input: B: Bottleneck pipelines
Output: New config if region is expandible, null otherwise
if R.type != partitioned stateful then return null

CFG ← R.CFG
CFG.ReplicaCount++                       ▷ Expand the region
foreach P ∈ B do
    if CFG ∈ R.Blacklist[P] then return null
return CFG
```

be resolving each bottleneck individually in separate adaptation periods. Although being a simpler solution, this approach does not work for a scenario where an upstream region is connected to two parallel downstream regions that are both bottleneck. In this scenario, if only one region is resolved at a time, it would not be profitable since the throughput is still bounded by the other bottleneck region. Therefore, all bottlenecks found in the same adaptation period are resolved together.

Algorithm 2 describes the `ResolveIfBottleneck()` procedure, which resolves bottleneck pipelines of a single region. It starts with finding the bottleneck pipelines and returns immediately if there are none. The `FindBottleneckPipelines()` procedure returns pipelines that have CPU utilization ratios greater than a configurable threshold, called the *CPU utilization bottleneck threshold*. Increasing the degree of pipeline parallelism is a cheaper operation compared to increasing the degree of data parallelism since the latter involves iteration over partitioning key instances to re-distribute operator state and queued tuples. For this reason, the `ResolveIfBottleneck()` procedure first tries to increase the degree of pipeline parallelism by splitting bottleneck pipelines into smaller pipelines. If the `SplitBottleneckPipelines()` procedure does not offer any pipeline parallelism change, then the `ExpandRegion()` procedure in Algorithm 4 is called to increase the replica count, if the region is a data parallel region. If a candidate parallelism change is found by one of these calls, the `ResolveIfBottleneck()` procedure realizes the new parallelization configuration on the region. Note that the adaptation algorithm can increase both the degree of pipeline and data parallelism in separate adaptation periods, depending on the bottlenecks in the region and the metrics provided by the profiler. It can decide to perform a pipeline split at one adaptation period, and then increase the number of replicas in future adaptation periods, or vice versa.

The main use of operator cost estimation is to make pipeline split decisions. The `SplitBottleneckPipelines()` procedure in Algorithm 3 defines a parameter, *split utility*, which predicts the ratio of increase in throughput when a pipeline is split at a particular operator. Consider the scenario where a 3-operator pipeline has the following operator costs: 0.2, 0.1, 0.4, and pipeline overhead: 0.3. If we split this pipeline at the second operator, then the first pipeline runs the first operator, and the second pipeline runs the second and third operators. Since the total operator cost of the second pipeline is greater than the first pipeline, we speculate that the second pipeline would bound the new throughput value. Then, the split utility value of the second operator is calculated as $1/(0.3 + max(0.2, 0.1 + 0.4))$. The divider in the equation is the summation of pipeline overhead of the bottleneck pipeline and total operator cost of the new pipeline. The overall calculation estimates the ratio of new throughput value to the current throughput value as if the new pipeline consists only the operators in the equation. Similarly, the split utility value of the third operator is calculated as $1/(0.3 + max(0.2 + 0.1, 0.4))$. Since splitting the pipeline at the third operator results in a higher split utility value, we choose the third operator as a pipeline split candidate. At this point, we check whether if the split utility value of the candidate operator is greater than a threshold, called *split utility threshold*. A split can non-profitable when a pipeline has a highly unbalanced operator cost distribution. The `SplitBottleneckPipelines()` procedure returns no parallelism change if it cannot find a candidate pipeline split whose split utility is greater than this threshold.

4.3. Evaluation and control

The `Adapt()` procedure proceeds to the evaluation phase in the next period after parallelism changes. In this phase, the profiler measures the throughput of the adapting regions. There can be multiple bottleneck regions at a time if their processing costs are close to each other. Moreover, some of the bottleneck regions can be closer to the source compared to the other bottleneck regions. In this case, it is sufficient the check new throughput values of the regions that are closest to the source, because their throughput values represent their downstream, i.e., the other bottleneck regions, as well. For this reason, `GetNonResolvedRegion()` call in the `Adapt()` procedure checks the throughput of the adapting regions that are closest to the source regions. If

Algorithm 5: Finalize(R, Tn, Cn)

```
Input: R: Region
Input: Tn[]: New throughput values
Input: Cn[]: New CPU util. values
Result: New region config is persisted.
foreach P ∈ R.A do
    R.Blacklist[P] ← []                    ▷ Clear blacklist
R.T[R.A] ← Tn[R.A]                         ▷ Update throughputs
R.C[R.A] ← Cn[R.A]                         ▷ Update CPU utils
R.CFGbase ← null
R.A ← null                                 ▷ Reset bottleneck pipelines
```

Algorithm 6: Revert(R)

```
Input: R: Region
Result: Reverts and blacklists the parallelism change
foreach P ∈ R.A do
    R.Blacklist[P] += P.CFG
R.CFG ← R.CFGbase
R.CFGbase ← null
ApplyParallelizationConfiguration(R)
```

Algorithm 7: UpdateMetrics(R, T, C)

```
Input: R: Region to update metrics
Input: T: New throughput values
Input: C: New CPU util. values
Result: Metrics of the region's pipelines are updated. If load
        change is detected for a pipeline, its blacklisted changes
        and non-resolvable flag are reset.
foreach P ∈ R.Pipelines do
    if IsLoadChanged(R, P, T[P], C[P]) then
        R.Blacklist[P] ← []
    R.T[P] ← T[P]
    R.C[P] ← C[P]
```

the ratio of increase in throughput is greater than a threshold, called *throughput increase threshold*, then the ongoing adaptation process is considered to be successful.

The `Adapt()` procedure calls `Finalize()` to complete a successful adaptation process. The `Finalize()` procedure, as shown in Algorithm 5, persists parallelism changes, updates metrics, and clears blacklisted parallelism changes. In case of a failed adaptation, the `Revert()` procedure in Algorithm 6 reverts and blacklists the applied parallelism changes. After blacklisting, the `Adapt()` procedure looks for other parallelism changes to continue the current adaptation process. If it could not find another parallelism change, it reverts and blacklists all parallelism changes applied in the current adaptation process.

The adaptation algorithm can accommodate changes in the workload as well. Workload changes may invalidate past decisions of the algorithm. Consequently, the algorithm forgets past decisions when it detects a workload change. The `UpdateMetrics()` procedure in Algorithm 7 calls `IsLoadChanged()` to compare the new throughput and CPU utilization ratio values with the current metric values of the region. If the ratio of difference is greater than a threshold, called the *workload change threshold*, the pipeline's blacklist is reset.

Additionally, the adaptation algorithm tolerates noises and transient fluctuations in the workload. During an ongoing adaptation process, it only updates metrics of the regions that are being adapted. The reason is that the ongoing adaptation process could fail, and blacklists of the other regions could be reset redundantly. Stability of the adaptation algorithm is preserved by preventing such situations. The profiler also contributes to the stability of the adaptation algorithm. After a parallelism change is applied, the profiler first waits for some time to allow the execution to stabilize.

4.4. Discussion of the adaptation algorithm parameters

The adaptation algorithm has four parameters: *CPU utilization bottleneck threshold*, *split utility threshold*, *throughput increase threshold*, and *workload change threshold*. We run experiments to investigate these parameters in Section 5.2.

4.5. Discussion of the organic adaptation solution

We further discuss two aspects of the organic adaptation solution:

- **Single host runtime**: We implement the organic adaptation solution on top of a single host runtime. However, our solution has no limitation that prevents it to be extended to a distributed runtime. The parallel execution model, state management API, runtime capabilities, and the adaptation algorithm have general applicability and can be implemented in a distributed setting.
- **Settling time**: The adaptation algorithm is model-free. It gradually increments the degree of pipeline and data parallelism. In cases where the needed degree of parallelism is high, the algorithm can have a long settling time. We can address this problem by extending the algorithm as follows: rather than splitting bottleneck pipelines into two pipelines, the adaptation algorithm can create more than two pipelines. Similarly, when the algorithm decides to increment the degree of data parallelism, it can create two or more new replicas at once. Thus, the algorithm can calibrate the trade-off between shortening the settling time and fine-tuning the degree of parallelism.

5. Experiments

In this section, we evaluate the effectiveness of our solution based on five kinds of experiments. First, we use micro-benchmarks to evaluate the adaptation algorithm with varying topologies and application sizes. Second, we investigate the sensitivity of the adaptation algorithm to different values of the adaptation parameters. Third, we study the overhead of the adaptation algorithm on the execution. Fourth, we use three real-world applications to compare the throughput achieved by our adaptation algorithm to that of no optimization. Last, we compare our algorithm with a model based solution [11]. All experiments except the last one are repeated three times and average values are reported.

We performed our experiments on a host with 2 Intel Xeon processors. Each processor has 6 cores with 2-way SMT, exposing 24 hardware threads. We run Joker on Java HotSpot VM (TM)[9] with 4 GB heap size.

5.1. Micro-benchmarks

For the micro-benchmarks, we use an operator that has two variations: stateless and partitioned stateful. This operator performs integer multiplications for each input tuple. When it is partitioned stateful, it also counts how many times an integer value is multiplied and persists this value using the KV store API provided by Joker runtime. We adjust the number of multiplications to simulate the varying amount of work being performed by the operator. At the one end of the spectrum, where there is little work per tuple, it is more difficult to achieve scalability because

[9] Java HotSpot VM website: https://www.oracle.com/technetwork/articles/javase/index-jsp-136373.html retrieved in March 2019.

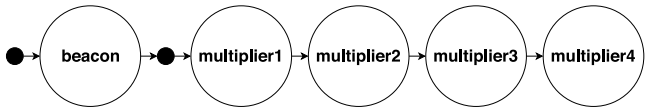


Fig. 4. A data flow graph that consists of a source region and a data parallel region with 4 operators.

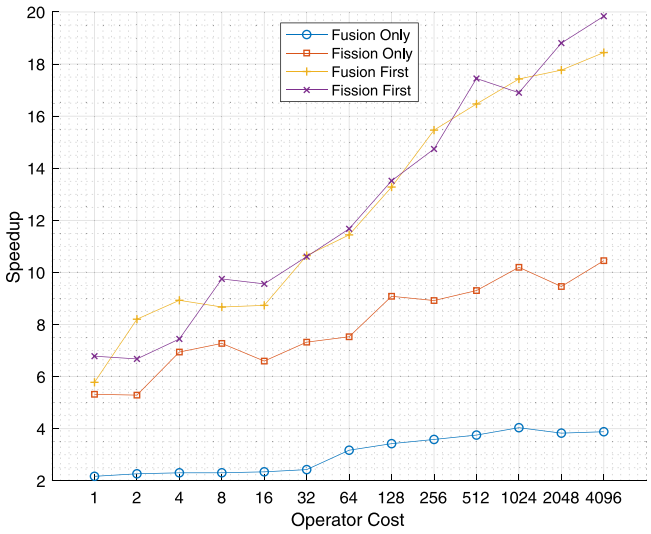


Fig. 5. Speedup of the 4-operator region.

the overhead of parallelization becomes significant compared to the actual work. Achieving scalability becomes easier as the overhead of parallelization becomes insignificant compared to the cost of tuple processing.

In all micro-benchmarks, the *CPU utilization bottleneck threshold* is 0.8, the *split utility threshold* is 0.2, and the *throughput increase threshold* is 0.1.

We evaluate our solution with data flow graphs that contain varying costs and numbers of the multiplication operator. We start with a simple data flow graph that contains a single data-parallel region of 4 multiplication operators, as shown in Fig. 4.

Fig. 5 plots the speedup achieved by the adaptation algorithm compared to the no-parallelization case, as a function of per-tuple processing cost. It presents the results for 4 different combinations of possible adaptation actions. *Fusion Only* means that the adaptation algorithm solely optimizes the fusion technique, hence adjusts the degree of pipeline parallelism. Similarly, only the fission technique is optimized by *Fission Only* in order to adjust the degree of data parallelism. *Fusion First* is the approach presented in Algorithm 2, as it first applies a pipeline parallelism change, and a data parallelism change afterwards to resolve a bottleneck. Last, *Fission First* reverses this order.

The *Fusion Only* results in Fig. 5 show that increasing the degree of pipeline parallelism does not improve the throughput significantly when the overall computation is cheap. When per-tuple work is less than 64 multiplications, the adaptation algorithm achieves 2x speedup with two pipelines, where the first pipeline runs the first and second operators, and the second pipeline runs the third and fourth operators. When per-tuple work becomes more costly, the adaptation algorithm achieves 4x speedup as it runs each operator in a separate pipeline. It is important to note that solely increasing the degree of pipeline parallelism cannot go over 4x speedup since there can be at most 4 pipelines. In this context, the main advantage of data

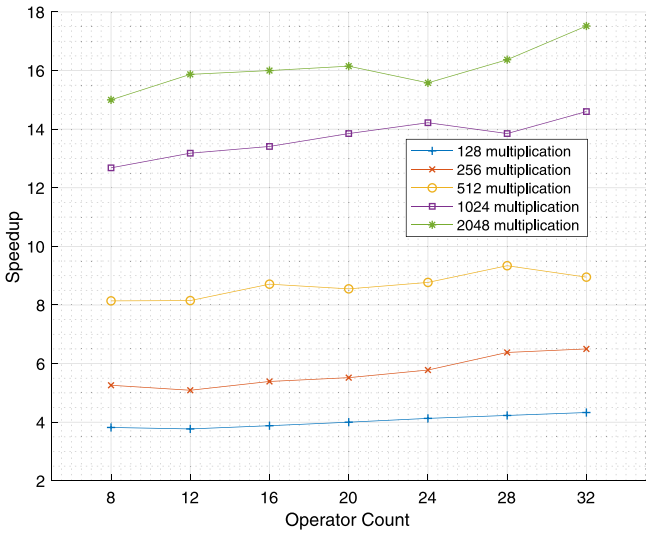


Fig. 6. Speedup of the data parallel region for varying operator costs and operator counts.

parallelism is that it is not bounded by the number of operators. The *Fission Only* results in Fig. 5 support this claim. Even for low-cost operators, increasing the degree of data parallelism achieves at least 5x speedup. As the application becomes more costly, the speedup goes up to 10x.

Fig. 5 reveals that the adaptation algorithm achieves higher speedups with the joint optimization of the fusion and fission techniques. For all operator costs, the joint optimization results are higher than individual application of the optimization techniques. As operators become more costly, the adaptation algorithm performs better with the joint optimization, for instance, 20x speedup is achieved for high operator costs. However, changing the order of application of optimization techniques does not create a significant difference in the results. Since changing the degree of pipeline parallelism does not involve operator state migrations, it is a cheaper operation compared to changing the degree of data parallelism. Therefore, we conclude that the adaptation algorithm applies parallelism changes in an effective order.

We repeat this experiment by increasing the cost and number of operators together. Fig. 6 shows that our solution is also able to scale long data flow graphs. The algorithm presents significant speedups as the data flow graph becomes longer.

Our parallel execution model divides a data flow graph into regions and the adaptation algorithm parallelizes each region separately. It could happen that scaling up a region can eventually cause another region to turn into a bottleneck, or multiple regions can become bottlenecks at the same time, for instance, if they are in parallel branches of a data flow graph. Therefore, we evaluate our solution with data flow graphs that contain multiple regions. Figs. 8 and 9 demonstrate the results of the multi-region experiments. In Fig. 7, we have four different topologies. In the *Tree* topology, as shown in Fig. 7a, a data parallel region is connected to two downstream data parallel regions, each of which consists of a single multiplication operator. Another topology, *Reverse Tree* as shown in Fig. 7b, connects two data parallel regions into a single downstream data parallel region. The last two settings are composed of a chain of two data parallel regions, which have different partitioning key definitions. In Fig. 7c, *Chain of 2 operators* topology places a single multiplication operator in each region, while *Chain of 4 operators* topology in Fig. 7d places two operators in each region. In this experiment, the adaptation algorithm works as it is described in Algorithm 2.

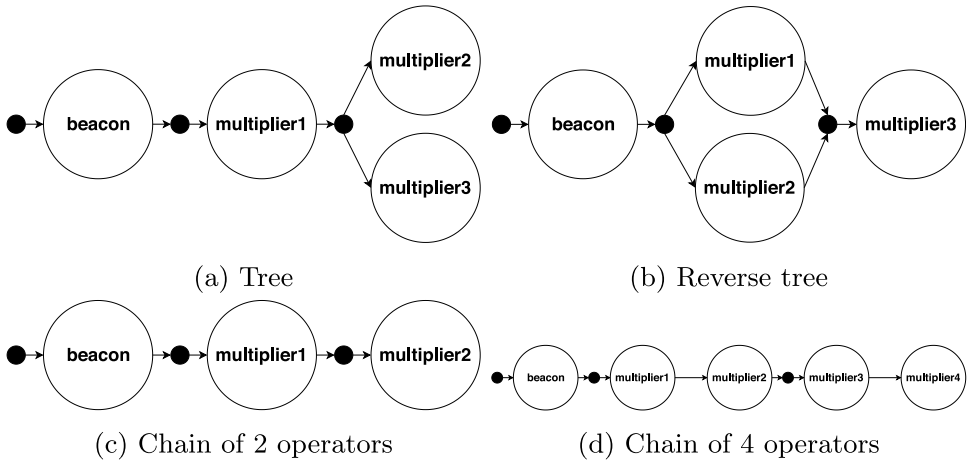


Fig. 7. Data flow graphs with multiple regions.

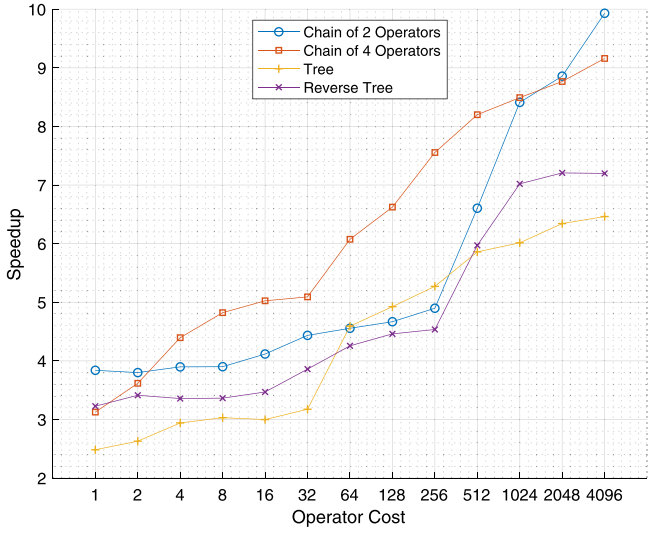


Fig. 8. Speedup of the multiple-region flow.

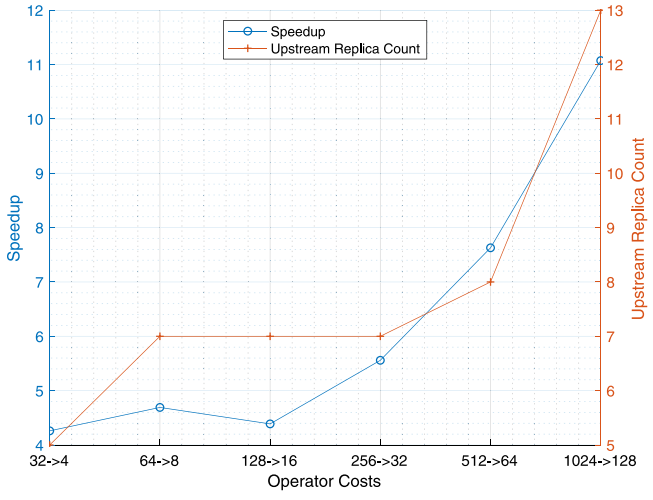


Fig. 9. Speedup of the flow of a costly and a cheap region.

Fig. 8 reveals that our solution resolves bottlenecks that span multiple regions. In the *Tree* and *Reverse Tree* topologies, the adaptation algorithm works incrementally and distributes CPU resources among the three regions. For instance, in the *Tree* topology, the upstream region becomes bottleneck initially. When the algorithm resolves this bottleneck by adding more replicas, the downstream regions receive more tuples and eventually turn into bottlenecks. Then, the algorithm resolves bottlenecks of the downstream regions together. The algorithm works similarly for the *Reverse Tree* topology. Differently than the *Tree* topologies, which contain 3 data parallel regions, the adaptation algorithm achieves a more balanced CPU resource distribution and higher speedups for the *chain* topologies, since they contain less number of operators and each operator receives a higher degree of data parallelism.

Fig. 9 slightly differs from previous experiments. It contains two data parallel regions with operators that have different multiplication counts. Namely, the upstream region is 8 times more costly than the downstream region. For instance, in the first iteration, the upstream operator and the downstream operator perform 32 and 4 multiplications per tuple, respectively. In all iterations, the adaptation algorithm creates 4 replicas for the downstream region. We report the number of replicas of the upstream region in Fig. 9. As the upstream region becomes more costly, it gets more replicas and achieves linear speedups. In the last iteration, our solution assigns 13 replicas to the upstream region and achieves 12x speedup.

In the last experiment of this section, we are interested in how our solution resolves bottlenecks in the presence of operators with different cost and selectivity values. In particular, we build a data parallel region of 4 operators with the following multiplication counts: 1024, 256, 1024, and 2048. The multiplication operator has selectivity of 1, i.e., it produces an output tuple for each input tuple. After the first operator, we place an operator that duplicates each input tuple to 4 output tuples. After the subsequent multiplication operators, we place operators that randomly filters input tuples with selectivity values 0.25 and 0.5.

Joker starts running this data flow graph with the parallelization configuration shown in Fig. 10a. In this configuration, the data parallel region is executed by a single pipeline replica thread, which manages to process 80 ktps (thousand tuples per second) and becomes a bottleneck. The adaptation algorithm attempts to solve this bottleneck by splitting the pipeline into two pipelines: *[m1, s1, m2]* and *[s2, m3, s3, m4]*. The reason of this split is that, the duplication and filter operators have negligible costs compared to the multiplication operators, and the multiplication operators receive similar costs because of the selectivity values. Therefore, the adaptation algorithm distributes the multiplication operators equally among pipelines. However, the new parallelization configuration results in only 4% increase in the throughput. Therefore, it is reverted and a new pipeline replica is created for the

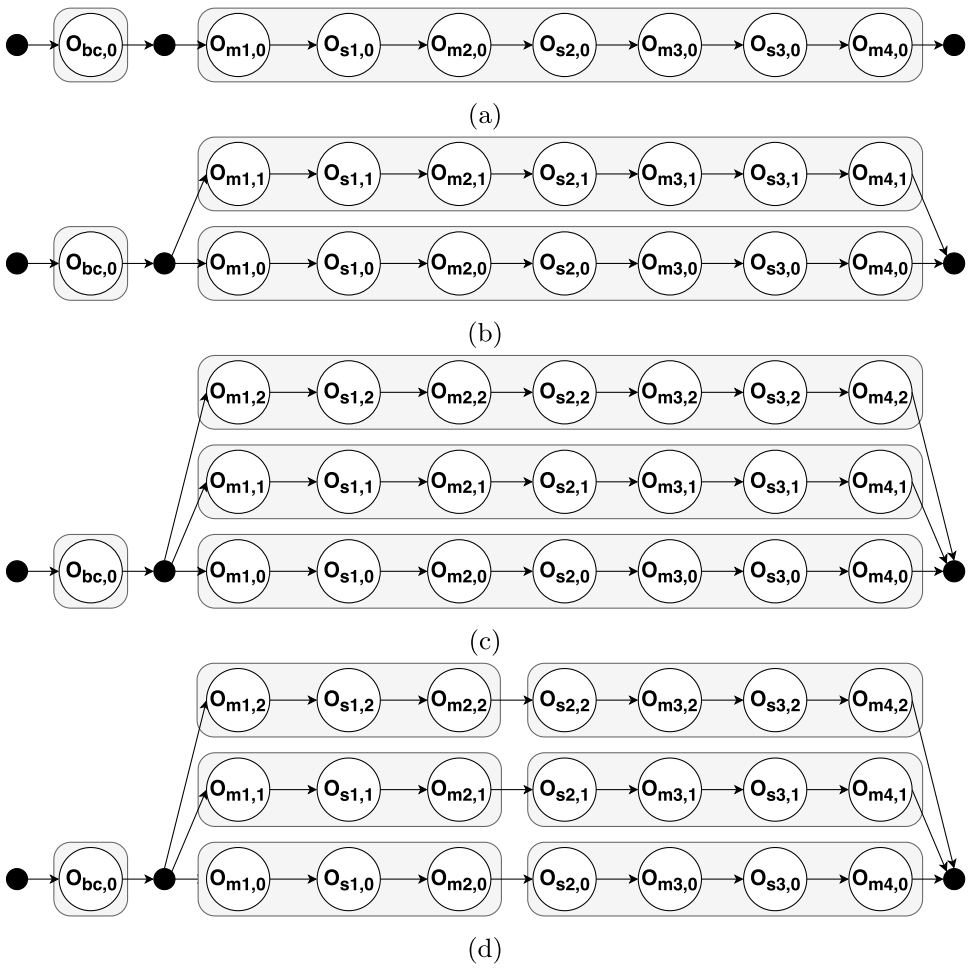


Fig. 10. Adaptation steps of the selectivity experiment.

region in the second attempt. Fig. 10b shows the resulting parallelization configuration. The new parallelization configuration is persisted as it improves the region throughput by 67%. In the next adaptation period, the region is still marked as bottleneck and passes over the same procedure. In this period, increasing the degree of pipeline parallelism increases the throughput 9%. Hence, the bottleneck is resolved by creating a new pipeline replica again. We observe that the throughput increases 46% with the resulting parallelization configuration shown in Fig. 10c. Even with 2 replicas, the region is still marked as bottleneck. This time, splitting the region pipeline leads to 14% improvement in the throughput, and the algorithm reaches to the parallelization configuration shown in Fig. 10d. In this configuration, the region processes 230 ktps with 2 pipelines and 3 replicas. After this point, the input stream is saturated, i.e., pipeline replica thread of the source operator fully utilizes its CPU, and there is no bottleneck anymore. Thus, the adaptation process terminates eventually.

5.2. Investigation of the adaptation algorithm parameters

In this section, we study the behavior of the adaptation algorithm for different values of the adaptation parameters: *CPU utilization bottleneck threshold*, *split utility threshold*, and *throughput increase threshold*.

The *CPU utilization bottleneck threshold* parameter impacts how our solution reacts to varying degrees of the workload received by the runtime. When the average CPU utilization ratio of replica threads of a pipeline exceeds this threshold, the adaptation algorithm marks the pipeline as bottleneck and attempts a parallelism change. If the threshold is too high, the algorithm will take no action even if the throughput could be improved with a higher degree of parallelism. As a result, the achieved speedup will suffer. If the threshold is too low, it will cause the algorithm to overreact and disrupt the execution unnecessarily with non-profitable parallelism changes.

In order to investigate the *CPU utilization bottleneck threshold* parameter, we run a simple data flow graph that contains a data parallel region of a single multiplication operator, which performs 256 multiplications for each input tuple. We start the execution with the input throughput of 40 ktps. Then, we repeatedly double the input throughput until it reaches to 640 ktps.

We use 3 values for the *CPU utilization bottleneck threshold* parameter: 0.5, 0.8 and 0.95. Fig. 11 demonstrates how the adaptation algorithm reacts to increasing amounts of input load. *Y*-axis on the left hand side shows the input throughput and the throughput handled by the runtime. *Y*-axis on the right hand side shows replica count of the region. The runtime is able to handle the load using a single operator replica until the input throughput increases to 640 ktps. However, when the threshold is 0.5, the second replica is created once the input throughput becomes 160 ktps and 320 ktps. Similarly, when the threshold is 0.8, the second replica is created once the input throughput reaches to 320 ktps. Since the input throughput is already handled by a single operator replica, these parallelism changes are evaluated as non-profitable and reverted. This behavior shows that although the adaptation algorithm can overreact to input loads when the threshold is too

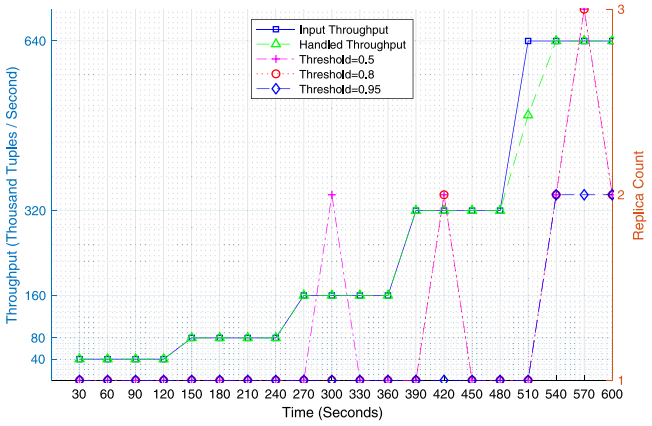


Fig. 11. Adjusting the *CPU utilization bottleneck threshold*.

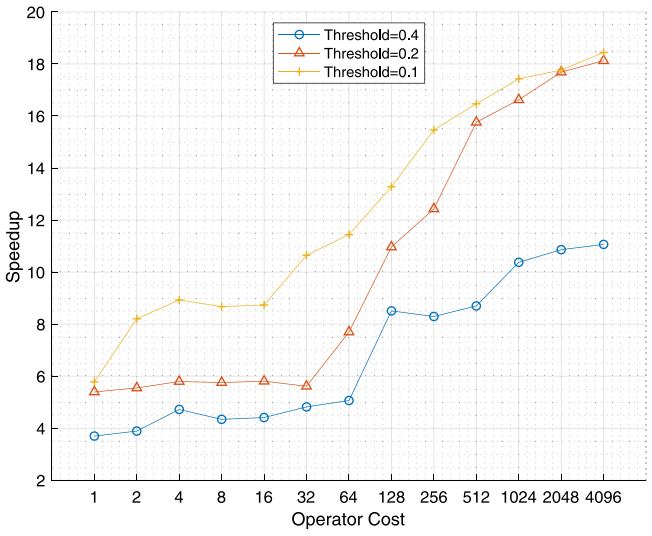


Fig. 12. Adjusting the *Throughput increase threshold*.

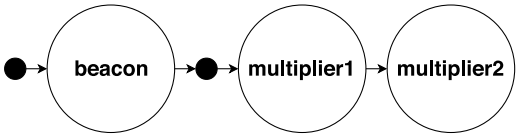


Fig. 13. A data flow graph that consists of a source region and a data parallel region with 2 operators.

low, it does not lead to overshoot. When the input throughput increases to 640 ktps, the runtime is able to handle only 500 ktps with a single operator replica. In this case, adding a new operator replica helps the runtime to fully handle the input throughput. After the adaptation algorithm settles on 2 replicas, when the threshold is lower than 0.95, it still considers the region as a bottleneck and tries out a non-profitable configuration with a third replica.

The *split utility threshold* determines to what degree pipeline parallelism is preferred to resolve a bottleneck. If a small value is set, the adaptation algorithm could try a pipeline split even if the bottleneck pipeline has a very unbalanced operator cost distribution. In such a case, a pipeline split would not be profitable and reverted by the adaptation algorithm. This behavior leads to longer settling time. On the other hand, if a large value is set, a profitable pipeline parallelism opportunity could be missed even though operator costs are balanced.

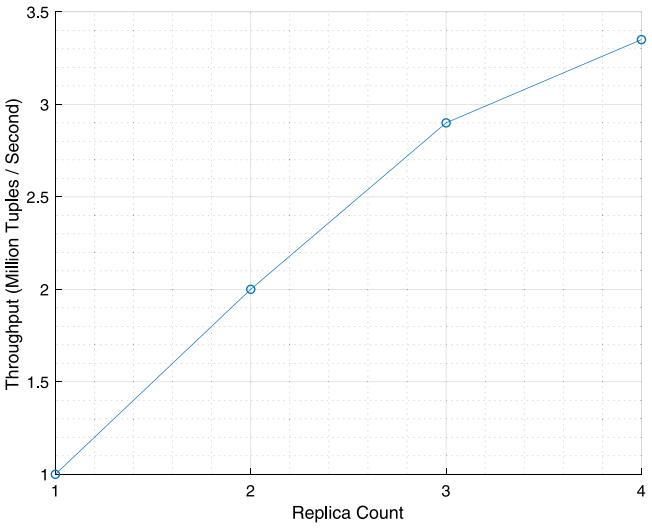


Fig. 14. Adjusting the *Split utility threshold* for non-balanced operator costs.

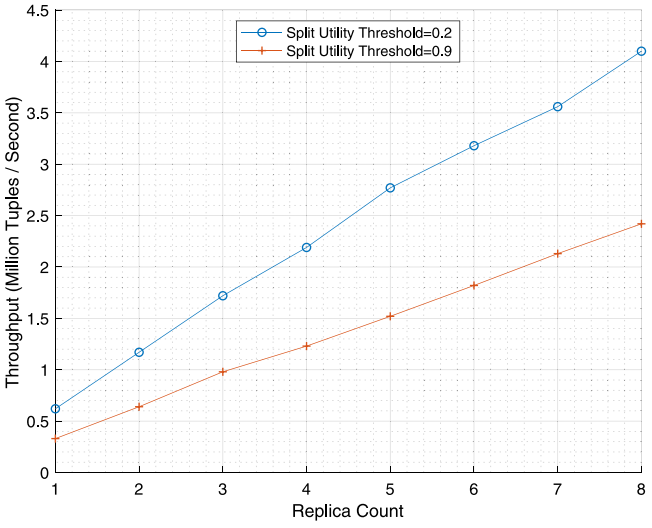


Fig. 15. Adjusting the *Split utility threshold* for balanced operator costs.

We demonstrate this behavior in the following experiments performed on a 2-operator region (see Fig. 13). First, we run this region with unbalanced operator costs: 0.75 and 0.10, i.e., pipeline replica threads spend 75% and 10% of their CPU time on the first and second operators respectively. In this setting, we use 2 different values for the *split utility threshold*: 0,2 and 0,1. Fig. 14 displays how the adaptation algorithm scales the data flow graph by adding new replicas. When the *split utility threshold* is 0.2, the algorithm solely creates new replicas and resolves the bottleneck at 4 steps. On the other hand, when the *split utility threshold* is 0.1, before adding a new replica, the algorithm creates a new pipeline and reverts it at each step because increasing the degree of pipeline parallelism is not beneficial for this unbalanced operator cost distribution. We run this data flow graph also with a balanced operator cost distribution: 0.45 and 0.40, and the *split utility threshold* values: 0.2 and 0.9. When *split utility threshold* is 0.2, the algorithm creates a new pipeline first, then proceeds with adding new replicas. However, it does not create a new pipeline when *split utility threshold* is 0.9. Fig. 15 shows that using a large *split utility threshold* value restrains the potential speedup achievable by the algorithm. When no new pipeline is created,

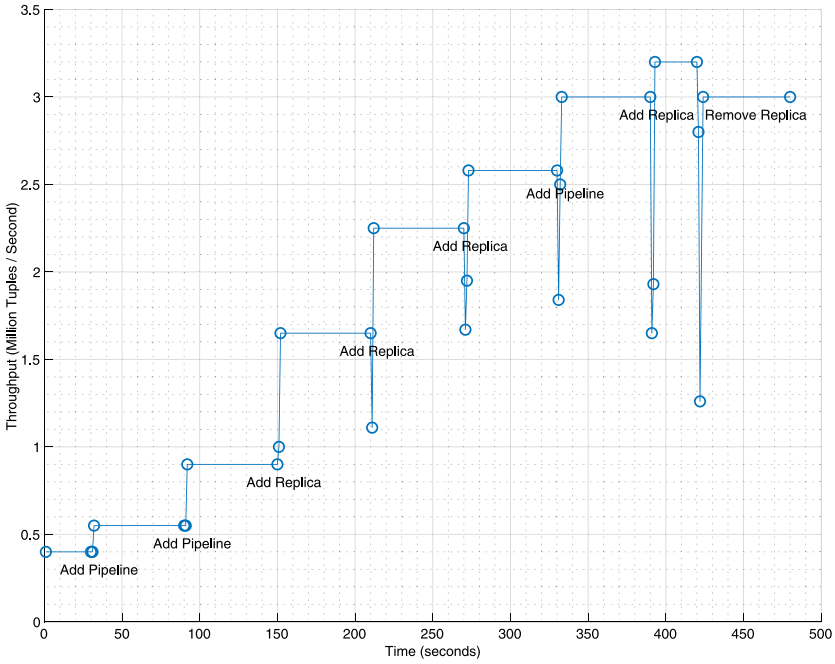


Fig. 16. Throughput of the data flow graph throughout parallelism changes.

the execution congests after 8 replicas and the achieved speedup becomes 40% less than the lower-threshold case.

Last, we investigate the *throughput increase threshold* by repeating the experiment shown in Fig. 12 for three different values of the parameter: 0.1, 0.2, and 0.4. As we describe in Section 4.3, the *throughput increase threshold* parameter is used for evaluating the profitability of parallelism changes. A parallelism change is reverted if it could not increase the throughput sufficiently. Fig. 12 tells that setting a high value for the threshold restricts the adaptation algorithm. When the computation is cheap, the contribution of the adaptation algorithm to the overall performance is limited. For cheap operators, increasing the threshold reduces the speedup values drastically. The effect of the threshold's value is inversely correlated with the computation cost. When the computation becomes more costly, the algorithm achieves similar speedups for the threshold values 0.1 and 0.2. On the other side, we observe that 0.4 is very restrictive for the threshold. At early stages, when the degree of parallelism is low, adding a new pipeline or a new replica results in significant improvements in the throughput. As the algorithm proceeds and the runtime utilizes a higher degree of parallelism compared to the initial state, the ratio of increase in the throughput after new changes is not as large as in the early stages. Therefore, setting a large value for the *throughput increase threshold* causes the algorithm to revert and blacklist beneficial parallelism changes because the ratio of increase in the throughput is considered to be insufficient.

There is also the *workload change threshold* to adjust the sensitivity of the adaptation algorithm to changes in the workload. If a small value is set, minor workload changes can affect the adaptation algorithm easily and cause the algorithm to repeat its past nonprofitable parallelism changes. If a large value is set, it can reduce the dynamicity of the adaptation algorithm. We perform no experiment for the *workload change threshold* parameter because its semantics is easy to grasp.

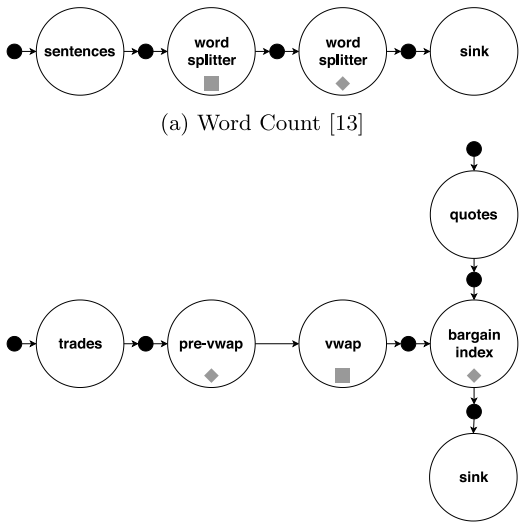
5.3. Investigation of the overhead of the adaptation algorithm

Before going into the real-world application benchmarks, we study how much overhead our adaptation solution imposes to the run-time. We run a data flow graph in which a source operator is connected to a data-parallel region of 5 operators with the following multiplication counts: 1024, 32, 32, 1024, 1024. We record the throughput of the source operator.

Fig. 16 depicts the change in the throughput over the course of the execution. In this figure, we mark the throughput at the time of and a few seconds after a parallelism change. In the beginning, the adaptation algorithm creates a new pipeline. There are only 2 threads at this stage, one thread for the source operator and another thread for the single pipeline of the data parallel region. Only the latter is paused and the region is divided into 2 pipelines. Since the number of threads is small, this parallelism change is applied promptly and the throughput is not hurt. Nevertheless, the throughput does not increase right after the parallelism change. It takes 2 s to see the benefit of the new pipeline. The same situation also goes for the second and third parallelism changes. The throughput does not decrease while applying these parallelism changes, but it still takes 2 s to see the effect of the increased degree of parallelism. After the third parallelism change, the region has 3 pipelines and 2 replicas, utilizing 6 threads in total. The first pipeline executes the first, second and the third operators, the second pipeline executes the fourth operator, and the third pipeline executes the last operator.

Before adding the third replica, the adaptation algorithm pauses all 6 threads of the data parallel region, re-distributes tuples and key–value state objects, resumes the paused threads, and starts 3 new threads for the new replica of the pipelines. The throughput drops more than 30% after this change and it takes 3 s to see an improvement in the throughput. We observe a similar pattern for the following parallelism changes.

Since the current implementation of Joker runtime is not distributed, the realization of parallelism changes does not involve serialization of tuples and key–value state objects, and sending them through the network. Instead, references of these objects are moved between threads. If the degree of data parallelism is changed, re-hashing is performed as well. Because of the lack of the serialization and networking overhead, Joker is able to apply parallelism changes are under a second. However, it takes around 2 to 3 s for the execution to stabilize afterwards. We



(a) Word Count [13]

(b) Finance [12]

(c) Network Monitoring [14]

Fig. 17. Data flow graphs for the application kernels.

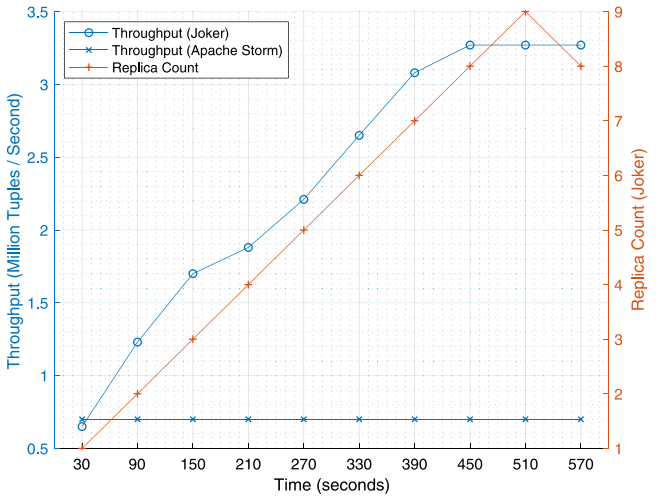


Fig. 18. Scaling the Word Count application.

conclude that parallelism changes have a non-negligible overhead on the execution when the degree of parallelism is high, and this overhead must be taken into consideration while adjusting the adaptation algorithm parameters.

5.4. Application benchmarks

In this section, we run benchmarks with three real-world stream processing application kernels from previous works, shown in Fig. 17.

These application kernels are:

- *Word Count*: In this application, the source operator generates sentences which are split into words by the second operator. Then, the words are forwarded to an operator that counts occurrences of each word and emits an output tuple when it increments the number of occurrences of a word. The input is a data set of random English words. In this application, the Word Splitter operator forms a stateless region and the Word Counter operator forms a partitioned stateful region.
- *Finance*: One branch of the application calculates volume weighted average price of the trades. Another branch emits quote values which are joined with the average prices. As the output, bargain indices are calculated for profitable quotes. For input data, it cycles through a real-world stock market data set.
- *Network Monitoring*: Public-facing servers constantly receive malicious login attempts through ssh service. This application, as described in [17], parses Linux system logs to detect break-in attempts. We combine system logs of a server with synthetic login attempts to generate the input.

For the Word Count and Finance applications, we plot the throughput along with the number of replicas for the data parallel regions as a function of time. The throughput is reported until the adaptation algorithm locates a parallelization configuration which saturates the whole input throughput, i.e., pipeline replica thread of the source operator fully utilizes its CPU.

Fig. 18 plots the throughput achieved for the Word Count application. The adaptation algorithm increases the replica count of the Word Counter operator until the speedup goes up to 5x. Although the Word Counter operator is evaluated as bottleneck with 8 replicas towards the end of the execution, adding a new replica does not help anymore since the input load is already saturated with 8 replicas.

We run the Word Count application also on Apache Storm[10] to compare Joker against the state of the art stream processing engines. Since Joker currently runs as a single-node engine, Apache Storm is also deployed as a single-node cluster on our server. In order to achieve the highest performance in Apache Storm, the Word Count application is executed in a single worker process,[11] and reliable tuple processing is disabled.[12] In this setting, we hand-optimized task counts of the application in Apache Storm.

Fig. 18 shows that Joker's performance is comparable to Apache Storm. In this experiment, Joker runs only a single replica of the source operator. Its thread fully utilizes its CPU and emits 3.2 Mtps (million tuples per second) when the downstream is executed by 9 threads. Joker can achieve higher throughput values if more source threads are added. On the other hand, Apache Storm processes 0.7 Mtps with 22 threads. The throughput achieved by Joker runtime is 4.5 times higher than Apache Storm. It is worth mentioning that Apache Storm bears several components that are required for production usage and not implemented in Joker, and these components bring an overhead to the execution. Apache Storm could achieve similar performance to Joker when it runs in the cluster mode.

Joker outperforms Apache Storm for the Word Count application which consists of only 3 operators. The other 2 real-world

10 Apache Storm 1.2.2 http://storm.apache.org/releases/1.2.2/index.html.

11 The parallelism model of Apache Storm http://storm.apache.org/releases/1.2.2/Understanding-the-parallelism-of-a-Storm-topology.html.

12 Reliable tuple processing in Apache Storm http://storm.apache.org/releases/1.2.2/Guaranteeing-message-processing.html.

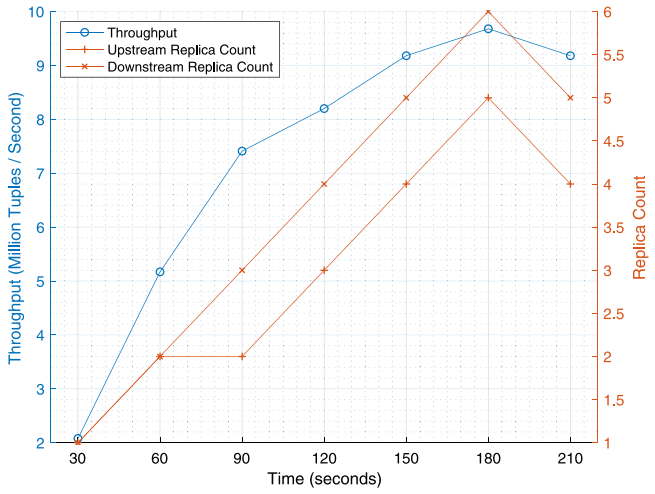


Fig. 19. Scaling the finance application.

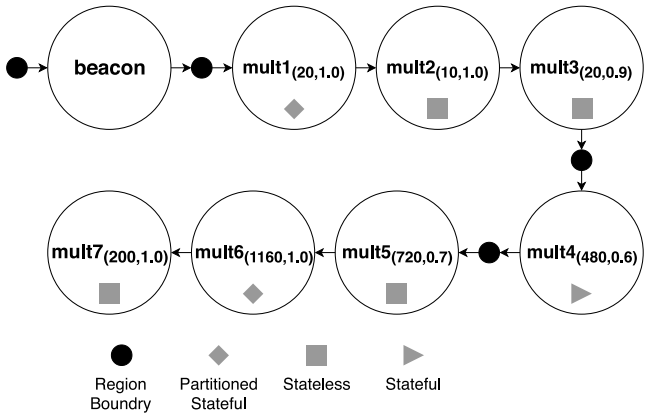


Fig. 21. The data flow graph used in the model-based experiments.

application kernels in this section consist of more complex data flow graphs. We reckon that results would be similar for those application kernels, hence do not run them on Apache Storm.

Fig. 19 plots the throughput achieved for the Finance application. There are 2 source operators and 2 partitioned stateful regions in this application. The upstream partitioned stateful region consists of the two operators that calculate average trade prices and the downstream region consists of a single operator that calculates bargains. The adaptation algorithm manages to scale the application by resolving bottlenecks in these regions together. It achieves a linear speedup of 4.4x with 4 replicas for the upstream region and 6 replicas for the downstream region. Although there are two operators in the upstream region, the adaptation algorithm cannot achieve any speedup by increasing its degree of pipeline parallelism. This is because the second operator is very cheap and running it on a separate pipeline is not profitable. In the final parallelization configuration, the 2 source operators are able to emit 9 Mtps in total.

In the Network Monitoring application, Auth Failure Filter and Failure Parser operators form a stateless region, and Failure Window, Cut Off and Diff operators form a partitioned stateful region. In the second region, only the first operator is partitioned stateful and other operators are stateless. Fig. 20 walks through the adaptation steps of the Network Monitoring application. In this experiment, the adaptation algorithm achieves only 2.86x speedup because the overall application complexity is very low. With the initial parallelization configuration, which is shown in Fig. 20a, the runtime is able to process 2.1 Mtps. In this configuration, the partitioned stateful region is evaluated as bottleneck and a new replica is created, as shown in Fig. 20b. The new parallelization configuration achieves 2.3x speedup. The partitioned stateful region is still a bottleneck in the new configuration. Since it is able to process more tuples in the new configuration, it also causes the stateless region to turn into a bottleneck. In this case, the adaptation algorithm applies a pipeline split to the upstream region and adds a new replica to the downstream region in a single step. Fig. 20c shows the resulting parallelization configuration, which leads to 2.86x speedup compared to the initial throughput. Joker processes 6 Mtps with this parallelization configuration.

5.5. Comparison to model based solutions

Gedik et al. [11] present a model-based solution to optimize the throughput of streaming applications. Their solution defines

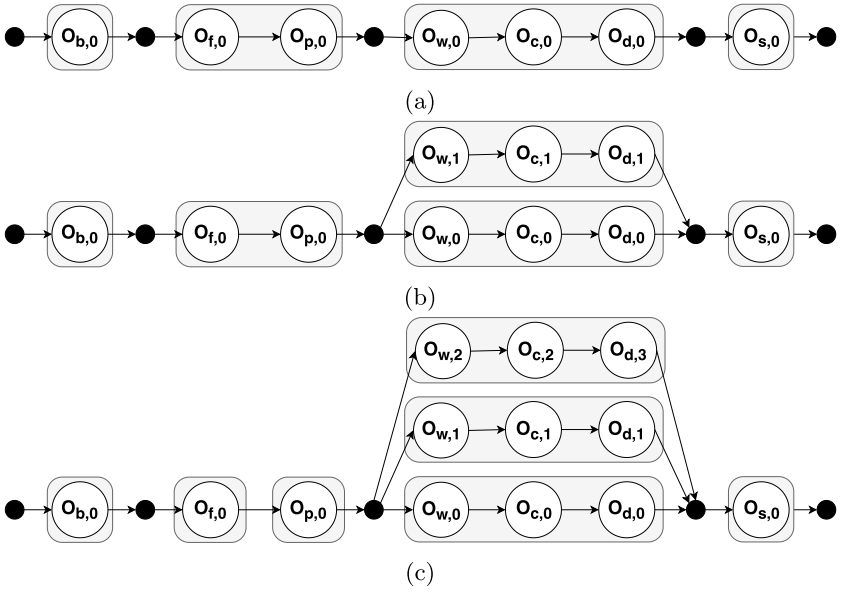


Fig. 20. Adaptation steps of the network monitoring application.

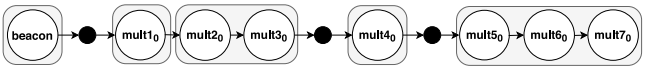


Fig. 22. Sel. factor: 0.1, the *pipelined fission model*.

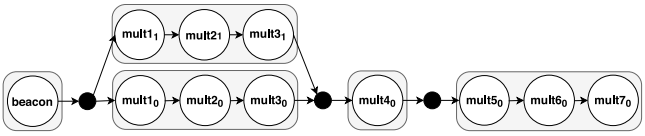


Fig. 23. Sel. factor: 0.1, Joker.

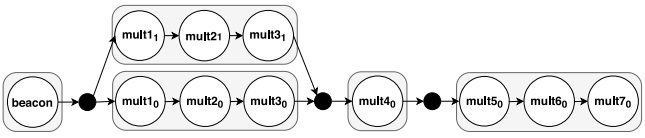


Fig. 24. Sel. factor: 0.2, the *pipelined fission model*.

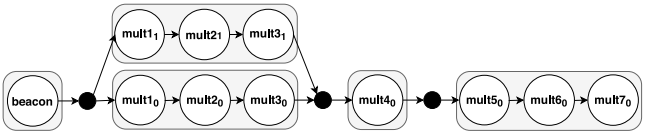


Fig. 25. Sel. factor: 0.2, Joker.

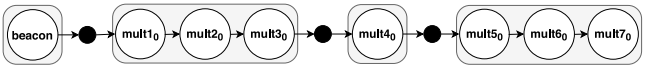


Fig. 26. Sel. factor: 0.4, the *pipelined fission model*.

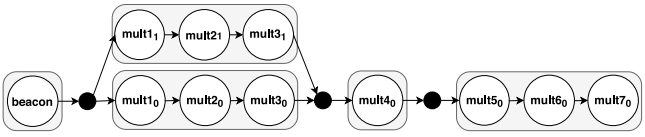


Fig. 27. Sel. factor: 0.4, Joker.

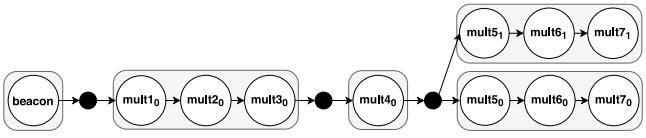


Fig. 28. Sel. factor: 0.8, the *pipelined fission model*.

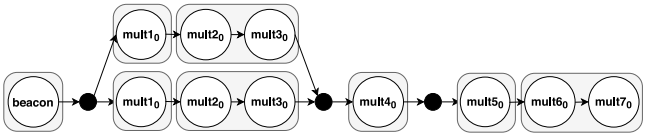


Fig. 29. Sel. factor: 0.8, Joker.

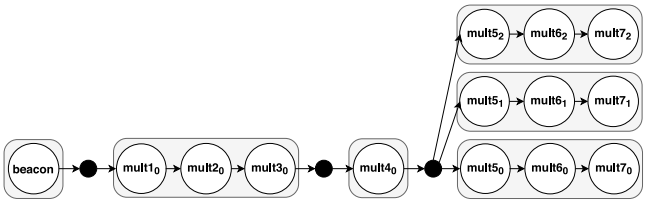


Fig. 30. Sel. factor: 1.0, the *pipelined fission model*.

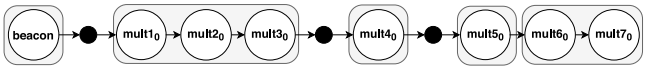


Fig. 31. Sel. factor: 1.0, Joker.

the throughput given a parallelization configuration, and proposes a heuristic approach to find a close-to-optimal parallelization configuration.

The model in the *pipelined fission solution* [11] contains 2 parameters related to the stream processing runtime: *thread switching overhead* and *replication cost factor*. These parameters are used for estimating the overhead incurred by the application of pipeline and data parallelism. Their values are calculated only once for a stream processing runtime and an execution environment. Besides these global parameters, the model requires 2 additional parameters related to each streaming application: *operator cost* and *selectivity*.

Our parallel execution model extends the parallel execution model developed in [11]. However, our adaptation algorithm works online. It does not require a model to estimate the throughput. Instead, it measures the actual throughput at run-time, and gradually increases the degree of pipeline and data parallelism to resolve bottlenecks. We use the data flow graph shown in Fig. 21 to compare our solution with the *pipelined fission solution*. In this data flow graph, there is a single source operator and 7 multiplication operators. The multiplication operators have different state types in order to create multiple regions. They also vary in multiplication counts and selectivity values. To emulate selectivity, an operator draws a random number for each incoming tuple and compares it to the provided selectivity value to determine if the input tuple should be forwarded to downstream. For instance, *"mult3"* performs 20 multiplications for each input tuple and its ratio of the number of output tuples to input tuples is 0.9. We generate different versions of the data flow graph by multiplying operators' selectivity values with factors of 0.1, 0.2, 0.4, 0.8, and 1.0. When selectivity values are small, less number of tuples arrive at the downstream operators and the upstream operators become more costly. The downstream operators receive more tuples and become more costly as the selectivity values increase.

We first determine the value of the thread switching overhead and replication cost factor for Joker runtime by following the procedure described in [11]. We provide the thread switching overhead, the replication cost factor, operator costs, and selectivity values to the *pipelined fission solution* to generate a parallelization configuration for each selectivity factor. Then, we disable the adaptation algorithm and run the data flow graph in Joker directly with those parallelization configurations. We also run the data flow graph in Joker with the adaptation algorithm.

Figs. 22 to 31 display the parallelization configurations generated by the *pipelined fission solution* and Joker's adaptation algorithm, and Fig. 32 reports the throughput values achieved by those configurations. The *pipelined fission solution* and the adaptation algorithm parallelize only the upstream region when the selectivity factor is small. When it is 0.1, the adaptation algorithm increases the degree of data parallelism while the *pipelined fission solution* increases the degree of pipeline parallelism, and the adaptation algorithm achieves higher throughput. However, when the selectivity factor is 0.2, both solutions increase the degree of data parallelism. The adaptation algorithm continues to generate the same configuration when the selectivity factor is

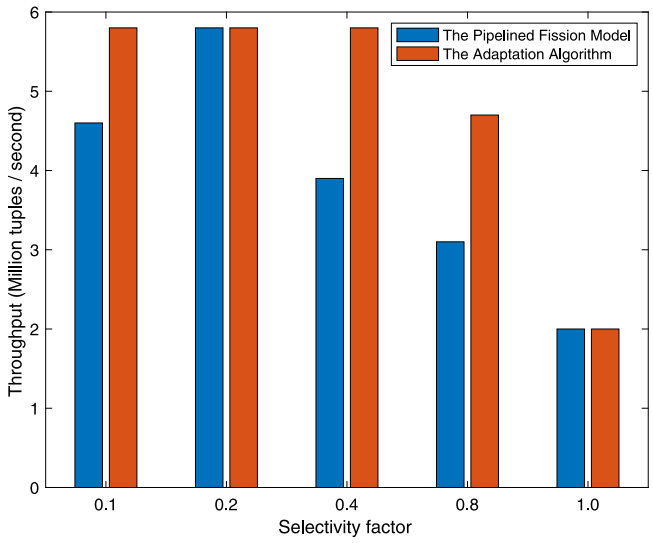


Fig. 32. Comparison of Joker and the *pipelined fission model*.

0.4, but the *pipelined fission solution* keeps the initial configuration in this case. When the selectivity factor is 0.8, the *pipelined fission solution* increases the degree of data parallelism for the downstream region, while the adaptation algorithm optimizes both the upstream and downstream regions. In the last experiment, the stateful region in the middle becomes a non-resolvable bottleneck and restrain the throughput for both solutions.

Fig. 32 shows that Joker presents comparable results to the model-based *pipelined fission solution* and utilizes a higher degree of parallelism. We conclude that although the *pipelined fission solution* is effective for optimizing streaming applications, it can produce sub-optimal results, for instance when the model parameter estimations become incompatible with the runtime because of the difficulty in covering the runtime's execution characteristics.

5.6. Evaluation summary

The experimental results presented in this section show that the organic adaptation solution is able to:

- transparently and non-disruptively optimize parallelization of non-trivial data flow graphs that involve partitioned stateful computations,
- resolve multiple bottlenecks at a time,
- achieve higher speedup values by joint optimization of pipeline and data parallelism,
- adapt to changes in the workload, and discover an effective parallelization configuration without causing overshoot.
- present comparable performance to the state of the art stream processing engines and model-based optimization solutions.

6. Related work

Recent research efforts are focused on the dynamic application of various optimization techniques to achieve automatic elasticity at run-time. Realizing online optimizations involves several challenges that need to be addressed by stream processing engines, such as low-overhead profiling, simple cost models, effective performance metrics, profitability evaluation strategies, and safe operator state migrations. Hirzel et al. [18] present a catalog of stream processing optimizations. Based on their classification, we review several studies here and investigate how they apply the fusion and fission techniques for elastic stream processing.

Fusion is a technique that trades communication cost against pipeline parallelism [18]. A number of studies investigate how fusion can be leveraged for elasticity. Tang et al. [24] develop a greedy control algorithm that continuously changes pipeline parallelism configuration of a given data flow graph to improve throughput. Lohrmann et al. [21] utilize dynamic task chaining and adaptive output buffer sizing to trade high throughput for low latency. Their solution continuously arranges chains of operators that can be executed by the same thread without causing a CPU bottleneck.

The fission technique realizes data parallelism by replicating an operator and splitting the input stream over its replicas [18]. Even though fission can be applied to stateless operators easily, handling stateful operators involves rigorous tasks, such as preserving correctness, performing transparent state migrations, and minimizing migrated state. Many of the earlier works apply fission in a dynamic way to adjust the degree of data parallelism with respect to a performance metric [4–10,12,14,15,20,23,25,26].

Lohrmann et al. [20] satisfy latency constraints with minimum resource consumption by adjusting the number of replicas for stateless operators. Stela [26] runs on top of Apache Storm and adjusts the degree of data parallelism for stateless operators. When scale-up is requested by the user, it picks the operator which would lead to the highest amount of increase in the throughput. Hidalgo et al. [15] build an elastic data parallelism solution on top of S4 [22] for stateless operators. In their solution, the system initially overloads an operator to measure the average number of events the operator can process in a period of time and assumes it to be constant during the execution. Cardellini et al. [4] build a prototype solution on top of Apache Storm that jointly optimizes placement and parallelism of the operators. Şahin et al. [13] develop a co-routine based elastic stream processing engine. They model operators as co-routines and use a thread pool to run the operators. Their elasticity algorithm extends the thread pool if the threads have a high average utilization value, and adjusts the degree of data parallelism only for stateless operators. ESC [23] is an elastic stream processing solution implemented in Erlang.[13] It realizes elasticity by acquiring and releasing machines and changing the number of operator replicas. Although its programming interface contains a state management primitive, ESC does not support partitioned stateful computations and operator state rebalancing. Cardellini et al. [5] extend Apache Storm with a new set of APIs to enable stateful computations and automatic elasticity by changing the number of operator instances at run-time. In their solution, it is the user's responsibility to define the minimum unit of migratable state. Different from these studies, Joker is able to jointly optimize pipeline and data parallelism for partitioned stateful streaming computations in a transparent and safe manner without requiring user intervention.

StreamCloud [14], MillWheel [2], SEEP [6], Gedik et al. [12], and ChronoStream [25] introduce state management primitives to enable elasticity in the presence of partitioned stateful operators. In addition to elasticity, MillWheel [2], SEEP [6], and ChronoStream [25] make use of their state management capabilities to achieve fault tolerance.

StreamCloud [14] is built on top of Borealis [1]. It splits a query into sub-queries based on stateful operators and deploys each sub-query to a different set of nodes to achieve pipeline parallelism. In addition, StreamCloud computes hash values for tuples based on the semantics of a sub-cluster's stateful operator and maps them to buckets, that are distributed to nodes of the

[13] Erlang website: https://www.erlang.org retrieved in March 2019.

sub-cluster, to achieve data parallelism. StreamCloud can acquire or release nodes for a sub-cluster, or move ownership of buckets between nodes in the sub-cluster to resolve bottlenecks. Even though StreamCloud supports partitioned-stateful computations, it compiles high-level queries into a set of relational algebra operators. Additionally, its query splitting strategy imposes a static pipeline parallelism configuration. In contrast, the operator development API we introduce exposes a set of state-management primitives to enable arbitrary stateful computations. Moreover, we offer a more advanced parallel execution model that can apply data parallelism to multiple operators at once and has more opportunities for pipeline parallelism. Last, Joker runtime is able to optimize the degree of data and pipeline parallelism together.

ChronoStream [25] achieves elastic stream processing for stateful computations by treating the computation state as a first-class citizen. It divides the computation state into slices which are distributed and checkpointed in the cluster. For elasticity, it is able to update the mapping of computation slices to threads of a worker process. However, ChronoStream does not perform automatic scaling. Furthermore, ChronoStream forms its state management primitives on a per-partitioning key basis. Operators register state objects which provide basic methods for state manipulation. On the other hand, our operator development API offers a unified state management API for both stateful and partitioned stateful operators, and our adaptation algorithm realizes automatic scaling.

SEEP [6] checkpoints the externalized operator state on upstream operators periodically and uses that state to scale out stateful operators. Its parallelism model is limited as it deploys a single operator on each node and scales one operator at a time. SEEP operators maintain processing state locally and expose it to the engine via an interface. On the contrary, Joker runtime takes the responsibility of managing state objects. When a stateful or partitioned stateful operator is invoked, the corresponding state object is passed to the operator. This approach opens the door for Joker runtime to transparently apply various state management optimizations, such as rebalancing operator state between operator replicas. Moreover, Joker can scale up multiple operators at once while performing optimizations.

Gedik et al. [12] introduce an elastic scaling policy for SPL applications [16]. Their solution changes the degree of data parallelism in the presence of partitioned stateful operators. It contains a key–value based state management interface and employs compile-time rewrite techniques on user code to utilize its state API. By this way, it performs state migrations transparently. Its control algorithm verifies the profitability of auto-scaling decisions. Nonprofitable parallelism changes are reverted and blacklisted. The control algorithm can also adapt to changes in the workload. If it detects a workload change, it forgets past nonprofitable decisions.

Floratou et al. [10] introduce the notion of *self-regulating* streaming systems. They build a solution on top of Apache Heron, called *Dhalion*. Dhalion is built with a modular and extensible architecture. With its default policies, Dhalion meets throughput SLOs by automatically adjusting the degree of data parallelism for Heron spouts and bolts, in the presence of workload variations. Dhalion's policies also resolve problems in the execution, such as slow Heron instances and data skew among bolts. It moves slow Heron instances to new containers and updates the hash function when it discovers a data skew problem among bolts. Although Dhalion enables users to develop custom policies, it does not handle stateful computations because of the lack of state support in the Heron API.

Gedik et al. [11] present a model-based solution to optimize the throughput of streaming applications. They follow a profile-guided optimization approach to find a parallelization configuration that achieves close-to-optimal throughput. Their model requires 2 global parameters related to the stream processing runtime and 2 additional parameters related to each modeled streaming application. However, it cannot adapt to workload changes that occur at run-time. Moreover, their solution works only on chain topologies that do not contain branches. We extend their parallel execution model to build an online, model-free, and adaptive parallelization optimization algorithm. We present a solution that contains both an optimization algorithm and an accompanying runtime. Our solution is also able to work on non-trivial topologies that contain branches.

Programming interfaces play an important role in the realization of performance optimizations. For instance, since stateful computation is not a first-class citizen in Apache Storm, either elasticity solutions are built only for stateless computations, or Apache Storm's operator API is extended to support computation state. Likewise, StreamCloud [14] supports stateful computations only for a set of relational algebra operators. However, our operator development API unifies the development of different types of operators. It supports arbitrary stateful computations, and enables transparent operator state rebalancing and migrations while scaling stateful streaming computations.

Furthermore, the majority of the studies focus on dynamic fission of a single operator at a time [6–9], with a few exceptions, such as StreamCloud [14] and Gedik et al. [12]. Although StreamCloud [14] can apply fission on a chain of operators, its pipeline configurations are not dynamic. Moreover, elasticity is studied by optimizing a single parallelization technique [6,12,14,24]. On the other hand, our parallel execution model identifies sequences of operators that are amenable to data parallelism as a whole. It can also apply different pipeline parallelism configurations to these operator sequences. To the best of our knowledge, our online parallelization optimization algorithm is the first solution that addresses elasticity by applying fusion and fission optimizations in a joint manner. It is a complete solution that resolves multiple bottleneck operators in a single step, optimizes the degree of parallelism with multiple techniques, and tracks the profitability of its optimization decisions.

7. Conclusion

In this paper, we presented an online parallelization optimization algorithm that resolves bottlenecks of streaming applications. To the best of our knowledge, our algorithm is the first solution that optimizes the degree of pipeline and data parallelism in a joint manner. We defined an operator development API and a flexible parallel execution model to form a basis for the optimization algorithm. The operator interface we introduced unifies the development of stateless, stateful, and partitioned stateful operators, source and sink operators, and operators with multiple input/output ports. We developed Joker, a single-host elastic stream processing runtime that scales streaming applications in a safe, transparent, dynamic, and automatic manner. Joker runtime capitalizes on the information revealed by the operator development API to parallelize streaming applications transparently. Joker employs the runtime components to carry out online changes in the parallelization configuration of streaming applications. It characterizes the performance of a parallelized execution, identifies bottlenecks, and optimizes the degree of parallelism in a greedy manner. Most importantly, the parallelization optimization algorithm is able to safely handle partitioned stateful computations that are involved in the vast majority of real-world streaming applications. Our extensive experiments showcase the effectiveness of our solution.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.jpdc.2019.10.012.

Acknowledgment

This work was supported by the Scientific and Technical Research Council of Turkey (TUBITAK) under Grant No. 114E472.

References

[1] D.J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, S. Zdonik, The design of the Borealis stream processing engine, in: Cidr, Vol. 5, 2005, pp. 277–289.
[2] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, S. Whittle, Millwheel: fault-tolerant stream processing at internet scale, Proc. VLDB Endow. 6 (11) (2013) 1033–1044.
[3] H.C. Andrade, B. Gedik, D.S. Turaga, Fundamentals of Stream Processing: Application Design, Systems, and Analytics, Cambridge University Press, 2014.
[4] V. Cardellini, V. Grassi, F. Lo Presti, M. Nardelli, Optimal operator replication and placement for distributed stream processing systems, ACM SIGMETRICS Perform. Eval. Rev. 44 (4) (2017) 11–22.
[5] V. Cardellini, M. Nardelli, D. Luzi, Elastic stateful stream processing in storm, in: 2016 International Conference on High Performance Computing & Simulation (HPCS), IEEE, 2016, pp. 583–590.
[6] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, P. Pietzuch, Integrating scale out and fault tolerance in stream processing using operator state management, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, ACM, 2013, pp. 725–736.
[7] T. De Matteis, G. Mencagli, Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing, in: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, 2016, p. 13.
[8] T. De Matteis, G. Mencagli, Elastic scaling for distributed latency-sensitive data stream operators, in: 2017 25th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), IEEE, 2017a, pp. 61–68.
[9] T. De Matteis, G. Mencagli, Proactive elasticity and energy awareness in data stream processing, J. Syst. Softw. 127 (2017b) 302–319.
[10] A. Floratou, A. Agrawal, B. Graham, S. Rao, K. Ramasamy, Dhalion: Self-regulating stream processing in heron, in: Proceedings of the 2017 VLDB Endowment 10.
[11] B. Gedik, H. Özsema, Ö. Öztürk, Pipelined fission for stream programs with dynamic selectivity and partitioned state, J. Parallel Distrib. Comput. 96 (2016) 106–120.
[12] B. Gedik, S. Schneider, M. Hirzel, K.-L. Wu, Elastic scaling for data stream processing, IEEE Trans. Parallel Distrib. Syst. 25 (6) (2014) 1447–1463.
[13] B. Gedik, et al., C-stream: a co-routine-based elastic stream processing engine, ACM Trans. Parallel Comput. (TOPC) 4 (3) (2018) 15.
[14] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, P. Valduriez, Streamcloud: An elastic and scalable data streaming system, IEEE Trans. Parallel Distrib. Syst. 23 (12) (2012) 2351–2365.
[15] N. Hidalgo, D. Wladdimiro, E. Rosas, Self-adaptive processing graph with operator fission for elastic stream processing, J. Syst. Softw. 127 (2017) 205–216.
[16] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé, et al., Ibm streams processing language: Analyzing big data in motion, IBM J. Res. Dev. 57 (3/4) (2013) 7–1.
[17] M. Hirzel, S. Schneider, B. Gedik, Spl: An extensible language for distributed stream processing, ACM Trans. Program. Lang. Syst. (TOPLAS) 39 (1) (2017) 5.
[18] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, R. Grimm, A catalog of stream processing optimizations, ACM Comput. Surv. 46 (4) (2014) 46.
[19] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, D. Lewin, Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web, in: International Symposium on Theory of Computing (ACM STOC), 1997.
[20] B. Lohrmann, P. Janacik, O. Kao, Elastic stream processing with latency guarantees, in: 2015 IEEE 35th International Conference on Distributed Computing Systems (ICDCS), IEEE, 2015, pp. 399–410.
[21] B. Lohrmann, D. Warneke, O. Kao, Nephele streaming: Stream processing under qos constraints at scale, Cluster Comput. 17 (1) (2014) 61–78.
[22] L. Neumeyer, B. Robbins, A. Nair, A. Kesari, S4: Distributed stream computing platform, in: 2010 IEEE International Conference on Data Mining Workshops (ICDMW), IEEE, 2010, pp. 170–177.
[23] B. Satzger, W. Hummer, P. Leitner, S. Dustdar, Esc: Towards an elastic stream computing platform for the cloud, in: 2011 IEEE International Conference on Cloud Computing (CLOUD), IEEE, 2011, pp. 348–355.
[24] Y. Tang, B. Gedik, Autopipelining for data stream processing, IEEE Trans. Parallel Distrib. Syst. 24 (12) (2013) 2344–2354.
[25] Y. Wu, K.-L. Tan, Chronostream: Elastic stateful stream computation in the cloud, in: 2015 IEEE 31st International Conference on Data Engineering (ICDE), IEEE, 2015, pp. 723–734.
[26] L. Xu, B. Peng, I. Gupta, Stela: Enabling stream processing systems to scale-in and scale-out on-demand, in: 2016 IEEE International Conference on Cloud Engineering (IC2E), IEEE, 2016, pp. 22–31.

Basri Kahveci is a graduate student in the Department of Computer Engineering, Bilkent University, Turkey. He holds a M.Sc. degree from the same department. His research interests are in stream processing systems.

Buğra Gedik is an Associate Professor in the Department of Computer Engineering, Bilkent University, Turkey. He holds a Ph.D. degree in Computer Science from Georgia Institute of Technology. His research interests are in data-intensive distributed systems.