Language Runtime and Optimizations in IBM Streams

Scott Schneider
IBM Research
scott.a.s@us.ibm.com

Buğra Gedik
Bilkent University
bgedik@cs.bilkent.edu.tr

Martin Hirzel
IBM Research
hirzel@us.ibm.com

Abstract

Stream processing is important for continuously transforming and analyzing the deluge of data that has revolutionized our world. Given the diversity of application domains, streaming applications must be both easy to write and performant. Both goals can be accomplished by high-level programming languages. Dedicated language syntax helps express stream programs clearly and concisely, whereas the compiler and runtime system of the language help optimize runtime performance. This paper describes the language runtime for the IBM Streams Processing Language (SPL) used to program the distributed IBM Streams platform. It gives a system overview and explains several language-based optimizations implemented in the SPL runtime: fusion, thread placement, fission, and transport optimizations.

1 Introduction

The increase in available data, commonly referred to as *big data*, has caused renewed exploration in systems for data management and processing. Processing this larger volume of data in a timely manner has necessitated moving away from the data-at-rest model, where data is archived in a database, and external applications query and process that data. In order to handle large volumes of data in real time, systems must exploit multiple levels of parallelism at scale.

The MapReduce [9] programming model was widely adopted as a solution in industry to the big data problem. While it brought parallel and distributed programming out of the niche of high performance computing, the model and its implementations have several deficiencies that make it ill-suited for handling online big data. First, the programming model is limited, as all computations must be expressed as *map* and *reduce* operations. In theory, one can express any arbitrary computation with sequences of such operations, but in practice the result may be difficult to understand, and will not necessarily perform well. Second, the design for MapReduce systems were inherently batch-based, which is incongruous with continuous, online data processing. Finally, MapReduce was still a data-at-rest solution: the data was stored in a shared file system prior to running any jobs.

Distributed stream processing is a more appropriate solution for online big data processing. Stream processing systems are designed to contend with continuously arriving data that must be processed quickly. Distributing such computations across a cluster enables the scalability required to deal with large volumes of data. Just as important as the underlying system is the programming model exposed to programmers. The stream processing programming model naturally exposes parallelism that is easily exploitable by the underlying runtime system.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

```
 1  stream<CDR> Calls = TCPSource() {
 2    param role: client; address: "1.2.3.0";
 3  }
 4  stream<CDR> UniqueCalls = Deduplicate(Calls) {
 5    window Calls: sliding, time(3600.0);
 6  }
 7  stream<Customer> Customers = TCPSource() {
 8    param role: client; address: "1.2.3.1";
 9  }
10  stream<CDR, tuple<rstring fromName>> Enriched =
11   Enricher(UniqueCalls; Customers) {
12  }
13  stream<rstring fromName, float64 avgLen> Stats =
14   Aggregate(Enriched) {
15    window Enriched: sliding, time(300.0);
16    output Stats: avgLen = Average(len);
17  }
18  () as Visualize = Dashboard(Stats) {
19  }
20  () as Persist = DBSink(Enriched) {
21    param address: "1.2.3.2"; table: "calls";
22  }
```

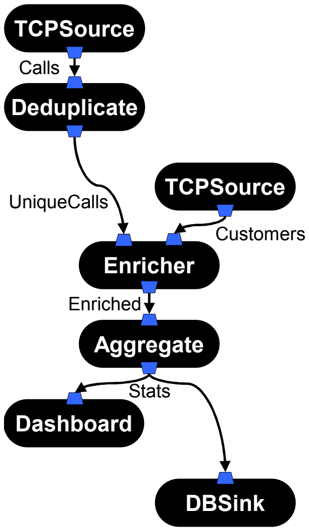

Figure 1: SPL code (left) and corresponding stream graph (right) for telecommunications example.

By allowing programmers to define their applications as independent operators that communicate over streams, distributed stream processing is the full realization of bringing parallel programming to application developers.

This paper presents the runtime for SPL, a stream processing language that targets the IBM Streams [15] platform for distributed stream processing. The SPL runtime was designed with performance as a goal: it supports low-latency, high-throughput streaming applications that execute continuously. SPL is a programming language designed to naturally expose task, pipeline, and data parallelism. The runtime system for SPL exploits such parallelism on hardware ranging from a single machine with many cores to many single-core machines.

Prior publications related to SPL focused on the language design [13] or specific optimizations applied in a streaming context [11, 20]. This paper is the first to focus on the SPL runtime system itself. It identifies the requirements for the SPL runtime, which are focused on the core semantics of the language and delivering high performance that is scalable and configurable. It presents the runtime system that meets those requirements and details its optimizations. These optimizations are possible because of the semantics of the stream programming model realized in SPL.

2 Background on Streaming

This section reviews core streaming concepts as embodied by SPL. It starts with an example application, then introduces development-time concepts, followed by runtime concepts, and wraps up with a discussion of alternative design choices.

Figure 1 shows a simplified version of the real-world telecommunications application presented by Bouillet et al. [6]. The first `TCPSource` in lines 1–3 ingests call detail records (CDRs) from an external system such as a telco switch. The `Deduplicate` in lines 4–6 drops duplicate CDRs in a 1-hour sliding window. The second `TCPSource` in lines 7–9 watches changes to customer information from an external system such as a subscriber database. The `Enricher` in lines 10–12 augments each CDR with a customer name, by buffering `Customers` information in memory and using it to look up names by phone numbers. The `Aggregate` in lines 13–17 computes statistics over a 5-minute sliding window; for simplicity, in this example, the aggregate statistics are just the average call length. The `Dashboard` in lines 18–19 visualizes aggregate statistics for online monitoring, whereas the `DBSink` in lines 20–22 persists them for offline analysis.

The code in Figure 1 exemplifies the development-time view of an application by describing the static structure of an SPL stream graph. Each vertex of the graph is an *operator invocation*, such as the first `TCPSource`. An operator invocation instantiates an operator, configures it (e.g. with a **`param`** clause), and connects it to streams (e.g. the `Calls` stream). *Streams* are the edges of the directed graph of operator instances. *Ports* are the points where streams connect to operator instances. Each operator instance can have zero or more input ports and zero or more output ports, and each port can be connected to zero or more streams. An operator, such as `TCPSource`, is a template that can be instantiated multiple times. Different invocations of the same operator can be configured differently; the specifics for how SPL supports this configuration are not relevant to this paper and can be found elsewhere [13]. What is important is that SPL poses no restrictions on permitted topologies, which can have multiple roots (sources) and leaves (sinks) and may even be cyclic. This flexibility makes it possible to support a broad set of applications. However, once deployed, the stream graph is fixed, to support static optimization.

The runtime view of SPL adheres to the semantics for the dynamic behavior of an SPL application. At runtime, streams send tuples between operator instances. Most non-source operators only activate when a tuple arrives at an input port. (Source operators activate based on external triggers; from the perspective of the application, they appear to activate spontaneously.) Note that each tuple causes a separate activation that consumes the tuple that triggered it; as a corollary, ports fire independently. The per-tuple activation semantics minimize the need for synchronization and are formalized in the Brooklet calculus [22]. An operator activation typically modifies operator-local state (if any) and submits zero or more tuples on output ports (if any), and then the operator becomes passive again waiting for the next activation. State is in-memory and operator-local, and thus state access requires no inter-operator coordination, avoiding a performance bottleneck for distributed deployments. When multiple streams converge on a single port, their tuples are interleaved in an unspecified order. When multiple streams originate from a single port, they all carry the same tuples.

Some streaming languages, such as CQL [5], focus primarily on relational operators. In contrast, SPL has an extensive operator library of which relational operators make up only a small fraction. This library, and the support for user-defined operators, provide an ecosystem for SPL applications in diverse domains. Some other streaming languages, such as StreamIt [12], focus primarily on operators with statically known selectivity. The *selectivity* of an operator is the number of tuples consumed and produced in an activation. In SPL, activations of non-source operators consume exactly one tuple, but the number of tuples produced varies per activation and is not known statically. Again, this design choice was important for SPL to work in many domains.

Overall, SPL provides the generality needed to address many applications and run on a distributed system, while retaining enough static information for language-based optimizations as described later in this paper.

3 Requirements

The runtime system for a streaming language has two primary responsibilities: to enforce the semantics of the programming language and to deliver high performance.

3.1 Enforcing Semantics

The responsibility of enforcing the semantics of the programming model belongs primarily to the compiler. But it is the language runtime that provides the streaming primitives that the compiler targets. A runtime for SPL has the following requirements:

- *Operator-state protection:* Operator state is exclusively local to that operator. The runtime is responsible for enforcing operator-state protection, even if multiple operators happen to execute in the same address space.

- *Asynchronous tuple-at-a-time:* Operators must be able to asynchronously yet safely process individual tuples. The runtime is responsible for delivering tuples to operators while preventing data races and deadlock.

- *Ordered streams:* Operators must be able to send tuples over streams to other operators for asynchronous processing. The runtime must deliver tuples to the operators that consume the stream even if the consuming operators are on a separate machine. The runtime must also maintain tuple order on streams: if an operator submits tuple *a* before tuple *b*, all operators that receive tuples from that stream must receive *a* before *b*.

- *Communication across applications*: Stream programs must be able to choose to publish or subscribe to streams from other stream programs. The runtime is responsible for matching publication and subscription specifications as new applications enter the system, and for delivering the appropriate tuples.

3.2 Delivering High Performance

From an implementation perspective, delivering high performance is at odds with enforcing language semantics: the simplest means to enforce the semantics tend to result in unacceptable performance. The following requirements are needed for SPL to deliver high performance, and will determine the runtime optimizations:

- *High throughput:* The primary performance metric for most SPL applications is throughput: tuples processed per second.

- *Low latency:* Tuple processing must not incur undue latency for any individual tuple. This requirement means that aggressively optimizing for throughput via large batches is not acceptable.

- *Continual processing:* Applications must be able to execute indefinitely, without the loss of performance. The runtime must be designed such that a single application can process data continuously for months.

- *System independence:* The abstractions provided in SPL allow any given application to map to any arbitrary distributed system. The SPL runtime must deliver on this promise, in both directions. The runtime must provide the ability for the same application to execute on many different kinds of distributed systems, and, given a particular distributed system, the runtime must be able to handle any arbitrary SPL application.

- *Parallel execution:* Operators in an application must be able to execute across a distributed cluster, in parallel. Parallelism is one of the means through which the runtime delivers high throughput, so any decision that limits parallelism must improve performance in some other way.

- *Explicit user control:* Experts with a deep understanding of the underlying distributed system—and how the abstractions in SPL map to that system—need to be able to control how their applications are deployed. That control is required both for influencing the optimizations in the runtime (such as parallelism or cheap communication) and for dealing with the constraints of a particular system (such as which machines in a cluster are allowed to access remote data sources).

4 System Overview

Creating and executing distributed streaming applications is more involved than the typical compile-and-execute model for general-purpose languages. This section gives a brief overview of the system as a whole, including the artifacts that are introduced in each stage of the application life cycle.

As a platform for distributed and parallel applications, IBM Streams must provide services such as name resolution, application life-cycle management, and scheduling. However, platform services are outside of the scope of this paper, which focuses on the SPL runtime.

Compilation. The primary entity in the SPL runtime is the *processing element*, or PE. Multiple operators can execute inside a PE, and determining which operators will execute together in the same PE is called *fusion*.

The compiler is responsible for operator fusion. The two main artifacts produced by compiling an SPL application are the PEs and the ADL (application description language) file. The optimization aspects of fusion are covered in Section 6.1. From the system's perspective, the PEs are dynamic libraries that contain the code for all of the operators fused into that PE. The ADL contains a meta-description of the entire application, including all of the PEs and the operators they contain. The connections between all operators within each PE, and between all PEs, are fully represented in the ADL.

Developers can annotate operator invocations to parallelize arbitrary sub-graphs. The compiler recognizes these annotations, but it does not perform the parallel expansion. Instead, it records in the ADL which regions of the stream graph should be parallelized at job submission time.

Job submission. SPL applications start executing when the ADL for the application is submitted to the Streams platform. Parallel expansion occurs at job submission, using the information from the ADL to indicate which portions of the application should be parallelized. The transformation process produces the PADL (physical ADL), which is the final representation of the stream graph that will execute.

The transformation process replicates all relevant operators and streams, and is responsible for connecting the replicated streams back into the unparallelized portions of the application. Because fusion happened at compile time, the parallel expansion cannot change which operators are in which PEs. There are two means by which it can achieve parallelism: replicate an entire PE, or replicate operators within a PE and inject threaded ports to ensure parallelism. In both cases, the PE binaries remain unchanged; the replication happens entirely in the stream graph representation in the PADL. This late-stage transformation is enabled by the separation between the high-level description of the application in the ADL and the actual code that executes in the PE binaries.

From the PADL, the Streams platform creates an AADL (augmented ADL) for each PE, which details what part of the stream graph that PE is responsible for. Finally, the platform is responsible for scheduling the PEs on the available hosts.

Execution. The Streams platform launches all of the PEs in the SPL application. Upon start-up, the PEs refer to their AADL to know which operators to start, how those operators are connected to each other, how those operators are connected to the input and output ports of the PE itself, and which connections to establish with the other PEs in the application. PEs created through the parallel expansion will execute the same PE binary, and operators replicated inside of PEs will simply instantiate the same operator multiple times.

Cancellation. Unlike applications in general-purpose languages, streaming applications are designed to execute indefinitely. For that reason, users must explicitly tell the Streams platform to cancel a particular job. When a PE receives a cancellation notification from the platform, it informs the operators it is responsible for, so they can safely clean up their resources.

5 The SPL Runtime

The SPL runtime manages the life-cycle and execution of the operators that are contained within the same PE. It also interacts with the larger Streams runtime to participate in application life-cycle management, dynamic connection management, metrics collection, and remote debugging support.

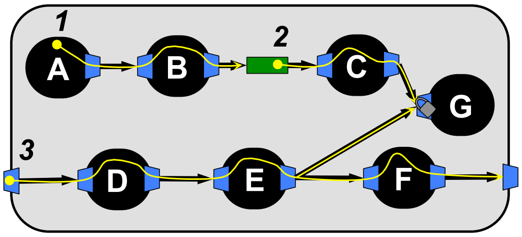

Figure 2: PE with three threads: a thread in a source operator, a threaded port, and a thread from the PE input port.

5.1 PE Execution

Operators within the same PE are executed as a single operating system process. The system component responsible for inter-PE communication is called the *transport*. The SPL runtime can use multiple threads within a PE to execute the PE's operators, as shown in Figure 2. In particular, source operators and input ports that are fed by the transport (PE input ports) are driven by dedicated threads. These threads execute the operator graph that is downstream of their associated source operators or PE input ports. The stream connections within a PE are implemented via function calls, using simple reference passing to avoid costly serialization. Tuples that go through inter-PE connections are buffered within the transport, whereas those that go through intra-PE connections implemented by function calls are not buffered. Further parallelism is achieved within a PE via the use of *threaded ports*. A threaded port is an input port within a PE that maintains a tuple buffer and uses a dedicated thread to execute its downstream operator graph. These threaded ports can be inserted manually by the application developer, as well as automatically by the SPL runtime [23]. In addition to these, individual operators can also request one or more SPL runtime managed threads for executing asynchronous tasks.

5.2 Operator Execution

The SPL runtime and the user-defined operators interact via an event-driven model. Operators handle tuples by implementing a tuple-handler function. They can submit tuples to their output ports, either as part of the tuple-handler function in reaction to a tuple arrival, or as part of the asynchronous tasks they execute. SPL also supports *punctuations*, which are out-of-band signals embedded within the tuple flow. Punctuations are handled via handler functions just like tuples. They can also be submitted to output ports. Two kinds of punctuations are supported: *window* punctuations and *final* punctuations.

Window punctuations are used to mark window boundaries within a stream. They enable custom windowing semantics, where the boundaries of the windows are not determined by a predefined windowing policy, but instead they are determined based on the presence of window punctuations in the stream.

Final punctuations are used to handle application termination. Receiving a final punctuation on an input port indicates that no tuples are to be received from that input port in the future. The SPL runtime manages the creation and forwarding of final punctuations automatically. Operators can opt to handle final punctuations in order to perform finalization tasks.

5.3 Window Management

SPL offers windowing syntax for any operator, not limited to relational ones. The SPL runtime facilitates the implementation of such windowed operators by providing a windowing API. In particular, the SPL runtime maintains windows in-memory, provides access to window contents, and lets user-defined operators register callback functions to handle various windowing events. SPL supports *tumbling* and *sliding* windows, including *partitioned* varieties. Tumbling windows are non-overlapping, whereas sliding windows are potentially overlapping. Partitioned windows maintain independent windows for different sub-streams based on a partitioning attribute. Windows are configured via *window eviction* and *window trigger* policies. SPL supports time-based, count-based, and attribute-delta based eviction and trigger policies [10]. For a tumbling window, the eviction policy specifies when to flush the window, such as after every 10 tuples or after the timestamp attribute increases by 10 units. For a sliding window, the eviction policy specifies when to evict old tuples from the window, such as when the window size grows beyond 10 (as a count or based on a timestamp attribute). For a sliding window, the trigger policy specifies when to process the window contents, such as after every 2 tuples, or after the timestamp attribute increases by 2 time units. Tumbling windows do not have trigger policies, as they trigger when the window is flushed.

5.4 Back-Pressure Management

The SPL runtime implements back-pressure to handle potential differences in the processing rates of operators. When an operator is faster than those downstream of it, submit calls will eventually block, as the downstream operators' input port buffers will be full. This will in turn slow down the operator at hand. As time progresses, the back-pressure will propagate further upstream. It will eventually reach source operators, and through them, external sources. Via the use of back-pressure, streaming operators naturally throttle themselves to avoid continuously growing buffers, without the need for shedding any tuples. Since the SPL runtime implements tuple submissions via function calls within a PE, back-pressure manifests at the boundaries where tuple submissions go through a buffer. These include tuple submissions to PE output ports (that go into the transport buffers) and tuple submissions to output ports that are connected to threaded ports (that go into the threaded port buffers).

SPL allows feedback loops in its flow graphs, where a downstream operator can produce an output that is fed back into the input port of an upstream operator. Such feedback loops create cycles in the flow graph, yet arbitrary cycles can cause deadlocks in the presence of back-pressure. To avoid deadlocks, SPL only allows feedback connections into *control ports*. A control port is a special kind of input port with the restriction that it cannot trigger the production of output tuples. Typically, control ports consume the incoming tuples to update the operator's internal state.

5.5 Consistent Regions

SPL applications can achieve fault tolerance through user-applied *consistent regions* [8]. Tuples in consistent regions are guaranteed to be processed at least once, even in the presence of operator and PE failure. The SPL runtime achieves this guarantee with a combination of operator state checkpointing and tuple replay. Source operators in consistent regions periodically send out special punctuations that inform operators that it is time to checkpoint their local state. Because streams are ordered, when an operator checkpoints its local state, it is guaranteed that the state contains the result of all tuples prior to the punctuation. The accumulated application state across all operators after they have all finished checkpointing is a consistent view of the application's state.

In the event of a failure, the platform notifies the source operators in consistent regions. The source operators then send out another special punctuation that tells all operators in the region to discard their current state, and reload their state from their last checkpoint. Following that punctuation is a replay of tuples that came after the last checkpoint. Through failure tracking, checkpointing, and a specialized protocol, the SPL runtime is able to guarantee at-least-once tuple processing. If the operators in the consistent region do not have externally visible behavior that cannot be rolled back, then from an operator developers perspective, this guarantee becomes exactly-once.

5.6 Dynamic Connections

A typical stream connection is established between an operator output port and an operator input port, based on the connection specification defined within an SPL program. Such connections are considered *static*. A complementary form of connections are *dynamic connections*, where the exact endpoints are established at runtime, subject to constraints specified in an SPL program at compile-time. Dynamic connections enable a few use cases that cannot be satisfied by static connections. One such use case is incremental deployment of applications, where an application is deployed in piecemeal fashion, adding new components as the application evolves. Another example is dynamic discovery of sources and sinks, where an application is designed to consume/produce data from/to a variable set of producers and consumers. These producers and consumers can be other applications sharing the same runtime instance. As a concrete example, in an operational monitoring application, new log sources (producers) as well as new analytic applications (consumers) could be added/removed at runtime via the use of dynamic connections.

SPL supports dynamic connections via *export properties* and *import specifications*. An output port that produces a stream can export it by associating a list of export properties with the stream. Dually, an input port that consumes streams can import them by providing an import specification. Import specifications are Boolean expressions that make use of export properties and basic arithmetic and logical operations on them. Both export properties and import specifications can either be defined within SPL programs or dynamically changed via runtime APIs. Based on export properties and import specifications, the Streams runtime performs continuous matching to determine changes on the dynamic connections. When such changes are detected, it coordinates with the SPL runtime to establish new connections and/or tear down existing ones to keep the dynamic connections up to date. Changes in the dynamic connections can happen due to changes in the list of SPL applications running within a Streams instance, or due to changes in the export properties or import specifications of existing SPL applications.

5.7 Dynamic Filters

Dynamic connections enable operators to subscribe to streams on demand. However, once a stream is subscribed via an import specification, its entire contents are received, since the matching is on stream-level export properties and not on tuple-level attributes. To support subscribing to a selective subset of imported streams, SPL supports *dynamic filters*. Dynamic filters, which can be specified together with import specifications, are Boolean expressions defined on tuple attributes. These filters are shipped by the Streams runtime to the PEs that are producing the exported streams and are evaluated by the SPL runtime to perform the filtering.

6 Runtime Optimizations

The SPL runtime implements several optimizations, with a particular focus on maximizing the throughput of applications by taking advantage of parallelization and distribution opportunities.

6.1 Fusion

The fusion optimization aims at grouping operators into PEs, so that the stream-processing application can be distributed over multiple hosts. Since process migration is costly, SPL performs fusion at compile-time. However, profiling data is collected during runtime and earlier runs guide the fusion decisions based on this profiling data. The profile-optimize cycle can be iterated to improve accuracy.

Fusion is a graph-partitioning problem, where the goal is to minimize the volume of data flow between PEs, while keeping the total cost of operators within a PE under a limit. Minimizing the volume of data flow between PEs minimizes the costly transmission of tuples across PEs, since stream connections are implemented as function calls within a PE. Limiting the total cost of operators within a PE avoids overloading a single host and makes it possible to utilize multiple hosts. The partitioning of the application flow graph for fusion can be implemented bottom-up, starting with one operator per PE and iteratively merging PEs; or top-down, starting from a single PE and iteratively dividing PEs. SPL's *auto-fuser* takes the latter approach, which is shown to have better performance [17] and can be easily adapted to work in the presence of the fission optimization in Section 6.3 [20].

SPL also enables application developers to explicitly request fusion via PE-level *co-location*, *ex-location*, and *isolation* directives. Co-location places a group of operators into the same PE. Ex-location enforces that a group of operators pair-wise do not share their PEs. Isolation runs an operator inside a PE by itself, with no other operators present. SPL's auto-fuser respects these fusion constraints.

6.2 Intra-PE Thread Placement

The intra-PE thread placement optimization aims to take advantage of multiple cores on a single host for executing operators within a PE. It can exploit both pipeline and task parallelism inherently present in streaming applications. In SPL, threaded ports perform thread placement. However, it is difficult to find a close-to-optimal configuration by hand, because it depends on the per-tuple costs and selectivities of operators. These properties are difficult to guess at development time. Furthermore, the number of possible placements increases combinatorially with the number of input ports and hardware threads available in the system. SPL solves this problem via an *auto thread placer*[1] that can automatically insert threaded ports as the application is executing [23].

The auto thread placer is a runtime component that incorporates a profiler and an optimizer. The profiler uses an application-level operator stack to track thread execution and periodically samples this stack to measure operator costs and thread utilizations. The optimizer uses these values to find bottleneck threads and decides where to insert threaded ports to maximize the application throughput. Additional runtime machinery is used to put these decisions into effect with minimal disruption to the active data flow. The process is iterative, where at each iteration additional threaded ports are added until no further improvements are possible.

The key insight used by SPL's auto thread placer is that, at each step, additional threaded ports decrease the workload of all of the highly utilized threads, as otherwise the optimization process will get stuck at a local minimum. This is particularly due to the dependence of the throughput on the slowest component of a pipeline. Another important consideration is that, sometimes, adding new threaded ports may not improve performance due to external effects, such as globally shared resources like files, locks, and databases. The auto thread placer monitors the achieved performance after changes in the threaded port configuration, in order to rescind ineffective changes. It also uses a blacklist to avoid them in the future.

6.3 Fission

Fission is an optimization that exploits data parallelism. To apply fission, a region of the application graph is replicated, the data is distributed over these replicas via a *split* operator, and the results from the replicas are re-ordered via a *merge* operator. In Streams, fission can be user-defined or automatic[2]. In user-defined fission, the application developer annotates the region that will take advantage of data parallelism, called the *parallel region*, and specifies the number of replicas. The runtime system handles the actual instantiation of the replicas, the distribution of tuples over the replicas, and the re-ordering at the end to maintain the sequential semantics.

Auto-fission both detects parallel regions and determines the number of parallel channels automatically, without involving the application developer. Auto-fission requires static code analysis to determine when the optimization is safe and runtime support to maintain that safety. The SPL compiler locates data-parallel regions by analyzing operator models as well as the configurations of the individual operator instances in the SPL

[1] Auto thread placer is available in a research version of the system [23].

[2] Auto-fission is available in a research version of the system [11, 20].

program [20]. It uses a left-to-right heuristic to consider operators in the graph and merges as many consecutive operators as possible into a parallel region to minimize parallelization overhead. The left-to-right heuristic is motivated by the observation that most streaming applications apply progressive filtering. Operators can be combined into parallel regions if they are suitable for data parallelism and their partitioning keys are compatible. Only operators that are either stateless or partitioned stateful can be used for data parallelism.

Auto-fission automatically discovers the degree of parallelism that achieves the best throughput, and adapts to changes in workload and resource availability. For this purpose a control algorithm is implemented within the splitters [11]. It uses throughput and congestion metrics to adjust the number of channels for the parallel region. The basic principle behind the control algorithm is to increase the number of channels until the congestion goes away. However, if the congestion is due to a downstream bottleneck that cannot be resolved by the parallel region at hand, then this situation is detected by the lack of improvement in the throughput in response to an increase made in the number of channels. Various additional mechanisms are employed to satisfy *SASO* properties: stability (no oscillations), accuracy (close to optimal throughput), settling time (number of channels is set quickly), and overshoot (no excessive resource consumption). In the presence of partitioned stateful operators, auto-fission requires support for state migration. Migration is needed whenever the number of channels changes, as some partitions are assigned to new operators. SPL addresses this issue by automatically managing operator state via a key-value store [11], using consistent hashing [16] to minimize the amount of data migrated.

6.4 Transport Optimizations

The Streams runtime provides various transport options, including InfiniBand for high-performance network hardware, TCP for general-purpose inter-host PE communication, and Unix domain sockets for intra-host PE communication. Various configuration options are provided related to buffering of tuples by the transport as well as thread usage for receiving tuples, in order to adjust the trade-off between latency and throughout.

The SPL runtime uses serialization and deserialization to transform between in-memory and on-the-wire representation of tuples. For highly performance-sensitive applications, this conversion may introduce significant overhead. Given SPL's dynamically-sized types (strings, lists, maps, and sets), these transformations are necessary in the general case. The SPL runtime implements an optimization called *façade tuples* to eliminate this overhead when the tuples involved contain only fixed-size types. The SPL language's support for fixed-size types includes bounded strings and bounded versions of lists, maps, and sets, in addition to the regular primitive types. Fixed-size types always occupy space corresponding to their maximum size, irrespective of their current effective size. The façade tuple optimization uses the same on-the-wire and in-memory representation for tuples that contain only fixed-size attributes. On the down-side, accessing façade tuple attributes might result in unaligned memory access, which may be unavailable in some systems and slightly slower in others.

7 Related Work

The first main topic of this paper is the distributed runtime system for SPL. Here, we compare SPL's runtime to other streaming runtimes.

Like SPL, TelegraphCQ [7] and CQL [5] enable continuous dataflow processing. Furthermore, like SPL, CQL has a language-centric design. However, both TelegraphCQ and CQL focus on relational stream queries, whereas a primary objective of SPL is support for operators beyond the relational domain. Furthermore, unlike SPL, TelegraphCQ and CQL lack distributed runtimes. Borealis pioneered distributed stream-relational systems [1]. However, it did not have a language-centric design. Therefore, unlike SPL, Borealis does not offer language-based optimizations. Another streaming platform with a language-centric design is StreamIt [12]. It does not emphasize a relational approach and supports distribution. However, unlike SPL, StreamIt only allows a restrictive set of topology combinators, ruling out commonly-needed cases such as multiple sources or sinks. Furthermore, unlike SPL, StreamIt focuses on operators with statically known selectivity. Microsoft StreamInsight is a streaming platform that derives from earlier stream-relational systems [4]. However, by using LINQ (language-integrated queries), it augments its relational foundation with user-defined code. Unlike SPL, StreamInsight was not designed with a distributed runtime in mind.

Recently, there has been a flurry of new streaming platforms that primarily focus on distribution: Google Millwheel [2], Spark Streaming with its micro-batch approach [25], Microsoft Naiad with its timely dataflow approach [18], and Twitter Storm [24]. Like SPL, they advance the state of the art for scalable and resilient distribution. However, none of them use a language-centric design, which means that unlike SPL, they do not offer much in the way of language-based optimization.

The second main topic of this paper is language-based optimizations for SPL. Here, we review streaming optimizations work that is closely related to SPL. For a comprehensive overview, see our survey paper [14]. Optimization algorithms must tackle two challenges, safety and profitability. Safety ensures that the optimized application produces the same results as the original code, and profitability ensures that it runs faster or uses less resources or scales to bigger work-loads.

Fusion combines operators to avoid the overhead of serialization and transport. There are variants of fusion depending on whether the operators are only combined in a single process or also in a single thread [23]. Fusion safety tends to be easy to establish. COLA offers a sophisticated solution to fusion profitability in the context of SPL [17]. Languages that focus on streaming with statically known selectivity solve fusion profitability even more comprehensively [21]. Fission introduces data parallelism by replicating an operator or even an entire subgraph of the stream graph. Fission is the killer optimization for StreamIt [12]. In the context of SPL, we have researched both fission safety [20] and fission profitability [11]. Fission is so important for performance that recent streaming platforms design partitioning deeply into their semantics to make fission the default [2, 18, 24, 25]. Transport optimizations reduce the overheads for sending tuples between distributed streaming operators across process or machine boundaries. The SPL runtime includes a highly optimized transport fabric with good defaults, but can be further tuned for extreme situations [19]. Many other distributed streaming systems start out with higher transport overheads, which can be optimized by reducing threads, serialization, etc. [3].

8 Conclusion

This paper describes the SPL language runtime and its optimizations. The SPL runtime provides the system support for hosting a graph of operators on multiple cores and multiple machines while enforcing the semantics of the programming language. Furthermore, the SPL runtime supports several language-based optimizations: fusing operators in the same operating-system process to reduce communication cost; placing multiple threads into such a process to increase intra-machine parallelism; using fission to replicate subgraphs of operators to increase inter-machine parallelism; and optimizing the transport to eliminate serialization overheads. The SPL runtime enables both user-directed and fully-automated variants of these optimizations.

References

[1] D. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Innovative Data Systems Research Conference (CIDR)*, 2005.

[2] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-tolerant stream processing at internet scale. In *Very Large Data Bases (VLDB) Industrial Track*, pages 734–746, 2013.

[3] S. Akram, M. Marazakis, and A. Bilas. Understanding and improving the cost of scaling distributed event processing. In *International Conference on Distributed Event-Based Systems (DEBS)*, pages 290–301, 2012.

[4] M. Ali, B. Chandramouli, J. Goldstein, and R. Schindlauer. The extensibility framework in Microsoft StreamInsight. In *International Conference on Data Engineering (ICDE)*, pages 1242–1253, 2011.

[5] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *Journal on Very Large Data Bases (VLDB J.)*, 15(2):121–142, 2006.

[6] E. Bouillet, R. Kothari, V. Kumar, L. Mignet, S. Nathan, A. Ranganathan, D. S. Turaga, O. Udrea, and O. Verscheure. Experience report: Processing 6 billion CDRs/day: From research to production. In *Conference on Distributed Event-Based Systems (DEBS)*, pages 264–267, 2012.

[7] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Conference on Innovative Data Systems Research (CIDR)*, 2003.

[8] G. J. da Silva. Guaranteed tuple processing in InfoSphere Streams v4 with consistent regions. `https://developer.ibm.com/streamsdev/2015/02/20/processing-tuples-least-infosphere-streams-consistent-regions/`. Retrieved December, 2015.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Operating Systems Design and Implementation (OSDI)*, pages 137–150, 2004.

[10] B. Gedik. Generic windowing support for extensible stream processing systems. *Software: Practice & Experience (SP&E)*, 44(9):1105–1128, 2014.

[11] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu. Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 25(6):1447–1463, 2014.

[12] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 151–162, 2006.

[13] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé, and K.-L. Wu. IBM Streams Processing Language: Analyzing big data in motion. *IBM Journal of Research and Development*, 57(3/4):7:1–7:11, 2013.

[14] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4), Apr. 2014.

[15] IBM Streams. `http://ibmstreams.github.io/`. Retrieved September, 2015.

[16] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Symposium on the Theory of Computing (STOC)*, pages 654–663, 1997.

[17] R. Khandekar, K. Hildrum, S. Parekh, D. Rajan, J. Wolf, K.-L. Wu, H. Andrade, and B. Gedik. COLA: Optimizing stream processing applications via graph partitioning. In *International Middleware Conference*, pages 308–327, 2009.

[18] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Symposium on Operating Systems Principles (SOSP)*, pages 439–455, 2013.

[19] Y. Park, R. King, S. Nathan, W. Most, and H. Andrade. Evaluation of a high-volume, low-latency market data processing system implemented with IBM middleware. *Software: Practice & Experience (SP&E)*, 42(1):37–56, 2012.

[20] S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu. Auto-parallelizing stateful distributed streaming applications. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 53–64, 2012.

[21] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe. Cache aware optimization of stream programs. In *Languages, Compiler, and Tool Support for Embedded Systems (LCTES)*, pages 115–126, 2005.

[22] R. Soulé, M. Hirzel, R. Grimm, B. Gedik, H. Andrade, V. Kumar, and K.-L. Wu. A universal calculus for stream processing languages. In *European Symposium on Programming (ESOP)*, pages 507–528, 2010.

[23] Y. Tang and B. Gedik. Autopipelining for data stream processing. *Transactions on Parallel and Distributed Systems (TPDS)*, 24(12):2344–2354, Dec. 2013.

[24] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm @twitter. In *International Conference on Management of Data (SIGMOD)*, pages 147–156, 2014.

[25] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Symposium on Operating Systems Principles (SOSP)*, pages 423–438, 2013.