# C-Stream: A Co-routine-Based Elastic Stream Processing Engine

SEMİH ŞAHİN, Georgia Tech
BUĞRA GEDİK, Bilkent University

Stream processing is a computational paradigm for on-the-fly processing of live data. This paradigm lends itself to implementations that can provide high throughput and low latency by taking advantage of various forms of parallelism that are naturally captured by the stream processing model of computation, such as pipeline, task, and data parallelism. In this article, we describe the design and implementation of *C-Stream*, which is an elastic stream processing engine. C-Stream encompasses three unique properties. First, in contrast to the widely adopted event-based interface for developing streaming operators, C-Stream provides an interface wherein each operator has its own driver loop and relies on data availability application programming interfaces (APIs) to decide when to perform its computations. This self-control-based model significantly simplifies the development of operators that require multiport synchronization. Second, C-Stream contains a dynamic scheduler that manages the multithreaded execution of the operators. The scheduler, which is customizable via plug-ins, enables the execution of the operators as co-routines, using any number of threads. The base scheduler implements back-pressure, provides data availability APIs, and manages preemption and termination handling. Last, C-Stream varies the degree of parallelism to resolve bottlenecks by both dynamically changing the number of threads used to execute an application and adjusting the number of replicas of data-parallel operators. We provide an experimental evaluation of C-Stream. The results show that C-Stream is scalable, highly customizable, and can resolve bottlenecks by dynamically adjusting the level of data parallelism used.

CCS Concepts: • **Information systems → Stream management**; • **Software and its engineering →** *Scheduling*; • **Computing methodologies → Parallel programming languages**;

Additional Key Words and Phrases: C-Stream, elastic stream processing engine

**ACM Reference format:**
Semih Şahin and Buğra Gedik. 2018. C-Stream: A Co-routine-Based Elastic Stream Processing Engine. *ACM Trans. Parallel Comput.* 4, 3, Article 15 (April 2018), 27 pages.
https://doi.org/10.1145/3184120

## 1 INTRODUCTION

As the world becomes more instrumented and interconnected, the amount of live data generated from software and hardware sensors increases exponentially. Data stream processing is a computational paradigm for on-the-fly analysis of such streaming data at scale. Applications of streaming can be found in many domains, such as financial markets (Zhang et al. 2009), telecommunications

Authors' addresses: S. Şahin is a graduate student with the College of Computing at Georgia Institute of Technology, 266 Ferst Drive, Atlanta, GA 30332, USA; email: semihsahin@gatech.edu. B. Gedik is a faculty at the Computer Engineering Department at Bilkent University, Ankara 06800, Turkey; email: bgedik@cs.bilkent.edu.tr.

https://doi.org/10.1145/3184120

(Zerfos et al. 2013), cyber-security (Schales et al. 2014), and health care (Garg et al. 2010), to name a few.

A streaming application is typically represented as a graph of streams and operators (Hirzel et al. 2013), in which operators are generic data manipulators and streams connect operators to each other using first in first out (FIFO) semantics. In this model, the data is analyzed as it streams through the set of operators forming the graph. The key capability of streaming systems is their ability to process high-volume data sources with low latency. This is achieved by taking advantage of various forms of parallelism that are naturally captured by the streaming model of computation (Hirzel et al. 2014), such as pipeline, task, and data parallelism.

While streaming applications can capture various forms of parallelism, there are several challenges in taking advantage of them in practice. First, the operators, which are the building blocks of streaming applications, should be easy to develop and preferably sequential in nature, saving the developers from the complexities of parallel programming. Second, streaming systems need a flexible scheduler that can dynamically schedule operators to take advantage of various forms of parallelism in a transparent manner. Furthermore, the scheduler should be configurable so that we can adjust the trade-off between latency and high throughput. Last, but not least, the stream processing system should be elastic in the sense that the level and kind of parallelism applied can be adjusted depending on the resource and workload availability.

In this article, we describe the design and implementation of *C-Stream*, which is an elastic stream processing engine. C-Stream addresses all of the aforementioned challenges. First, in contrast to the widely adopted event-based interface for developing stream processing operators, C-Stream provides an interface wherein each operator has its own driver loop and relies on data availability application programming interfaces (APIs) to decide when to perform its computations. This model significantly simplifies development of multi-input port operators that otherwise require complex synchronization. Furthermore, it enables intraoperator optimizations, such as batching. Second, C-Stream contains a dynamic scheduler that manages the multithreaded execution of the operators. The scheduler, which is customizable via plug-ins, enables the execution of the operators as co-routines, using any number of threads. The base scheduler implements back-pressure, provides data availability APIs, and manages preemption and termination handling. Scheduler plug-ins are used to implement different scheduling policies that can prioritize latency or throughput. Last, C-Stream provides elastic parallelization. It can dynamically adjust the number of threads used to execute an application and can also adjust the number of replicas of data-parallel operators to resolve bottlenecks. For the latter, we focus on stateless operators, but the techniques also apply for partitioned parallel operators[1]. Finally, we have evaluated our system using a variety of topologies under varying operator costs. The results show that C-Stream is scalable (with increasing number of threads), highly customizable (with respect to latency and throughput trade-offs), and can resolve bottlenecks by dynamically adjusting the level of data parallelism used (providing elasticity).

In summary, this article makes the following contributions:

- We propose an operator development API that facilitates sequential implementations, significantly simplifying the development of multi-input-port operators that otherwise require explicit synchronization.
- We develop a flexible scheduler and accompanying runtime machinery for executing operators as co-routines, using multiple threads.

[1]This requires state migration and ordering support (Schneider et al. 2015), which is not yet implemented in our prototype.

- We present techniques for elastic execution, including the adjustment of the level of parallelism used and the number of operator replicas employed.
- We provide a detailed evaluation of our system to showcase its efficacy.

The rest of this article is organized as follows. Section 2 overviews the programming model and the operator development APIs used by C-Stream. Section 3 describes the co-routine-based runtime, the multithreaded scheduler, and the custom scheduler plug-ins that we have developed for it. Section 4 explains how Stream-C achieves elasticity. Section 5 presents our experimental evaluation. Section 6 discusses related work and Section 7 contains our conclusions.

## 2 PROGRAMMING MODEL

In this section, we first give a brief overview of the basic concepts in stream processing. We then describe the programming model used by C-Stream, with a focus on *flow composition* and *operator development*.

### 2.1 Basic Concepts

A streaming application takes the form an operator flow graph. Operators are generic data manipulators that are instantiated as part of a flow graph, with specializations (e.g., parameter configurations and port arity settings). Operators can have zero or more input and output ports. An operator with only output ports is called a *source* operator and an operator with only input ports is called a *sink* operator. Each output port produces a *stream*, that is, an ordered series of tuples. An output port is connected to an input port via a *stream connection*. These connections carry tuples from the stream, providing FIFO semantics. There can be multiple stream connections originating from an output port, called a *fan-out*. Similarly, there can be multiple stream connections destined to an input port, called a *fan-in*.

Three major kinds of parallelism are inherently present in streaming applications; Stream-C takes advantage of all three (see Section 4):

- *Pipeline parallelism*: As one operator is processing a tuple, its upstream operator can process the next tuple in line at the same time.
- *Task parallelism*: A simple fan-out in the flow graph gives way to task parallelism, in which two different operators can process copies of a tuple at the same time.
- *Data parallelism*: This type of parallelism can be taken advantage of by creating replicas of an operator and distributing the incoming tuples among them so that their processing can be parallelized. This requires a split operation, but more important, a merge operation after the processing in order to reestablish the original tuple order. Data parallelism can be applied to *stateless* as well as *partitioned stateful* operators (Gedik et al. 2014). Stateless operators are those that do not maintain the state across tuples. Partitioned operators maintain the state across tuples, but the state is partitioned based on the value of a *key* attribute. In order to take advantage of data parallelism, the streaming runtime has to modify the flow graph behind the scenes (Hirzel et al. 2014).

There are two aspects of developing a streaming application. The first is to compose an application by instantiating operators and connecting them via streams. This is called *flow composition*. It is a task typically performed by the streaming application developer. The second is *operator development*, which involves creating reusable streaming analytics. It is a task typically performed by a library developer.

```
 1  Flow flow("sample␣application");
 2
 3  // create the operators
 4  auto& names = flow.createOperator<FileSource>("name_source")
 5    .set_fileName("data/names.dat")
 6    .set_fileFormat({{"id",Type::Integer},{"name",Type::String}});
 7
 8  auto& values = flow.createOperator<TCPSource>("value_source")
 9    .set_address("my.host.com", 44000)
10    .set_dataFormat({{"id",Type::Integer},{"value",Type::Integer}});
11
12  auto& filter = flow.createOperator<Filter>("empty_filter")
13    .set_filter(MEXP1( t_.get<Type::String>("name") != "" ));
14
15  auto& combiner = flow.createOperator<Barrier>("combiner", 2);
16
17  auto& sink = flow.createOperator<FileSink>("file_sink")
18    .set_fileName("data/out.dat")
19    .set_fileFormat({{"id",Type::Integer},{"name",Type::String},{"value",Type::Integer}});
20
21  // create the connections
22  flow.addConnections( (names,0) >> (0,filter,0) >> (0,combiner) );
23  flow.addConnections( (values,0) >> (1,combiner,0) >> (0,snk) );
24
25  // configure the runner
26  FlowRunner & runner = FlowRunner::createRunner();
27  runner.setInfrastructureLogLevel(Info);
28  runner.setApplicationLogLevel(Trace);
29
30  // run the application and wait for completion
31  int numThreads = 2;
32  runner.run(flow, numThreads);
33  runner.wait(flow);
```

Listing 1. Flow composition in C-Stream.

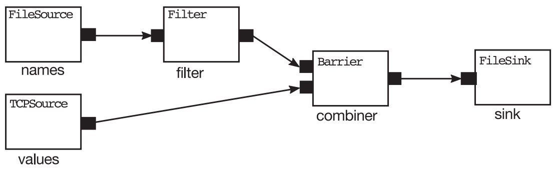


Fig. 1. Example flow graph from Listing 1.

## 2.2 Flow Composition

Stream-C supports flow composition using an API-based approach, employing the C++11 language. Listing 1 shows how a simple streaming application is composed using these APIs. Figure 1 depicts the same application in graphical form.

A `Flow` object is used to hold the dataflow graph (Line 1). Operators are created using the `createOperator` function of the `Flow` object (Line 4). This function takes the operator kind as a template parameter and the runtime name of the operator instance being created as a parameter. Optionally, it takes the arity of the operator as a parameter as well. For instance, the instance of the `Barrier` operator referenced by the `combiner` variable is created by passing the number of input ports, 2 in this case, as a parameter (Line 15). Operators are configured via their `set_` methods, which are specific to each operator type. The parameters to operators can also be lambda expressions, such as the `filter` parameter of the `Filter` operator (Line 13). Such lambda expressions can reference input tuples (represented by the `t_` variable in the example code).

The connections between the operator instances are formed using the `addConnections` function of the `Flow` object (Lines 22 and 23). The `>>` C++ operator is overloaded to create chains of connections. For instance, `(names,0) >> (0,filter,0) >> (0,combiner)` represents a chain of connections in which the output port `0` of the operator instance referenced by `names` is connected to the input port `0` of the one referenced by `filter` and the output port `0` of the latter is connected to the input port `0` of the operator instance referenced by `combiner`.

The flow is run via the use of a `FlowRunner` object (Line 26). The `run` method of the `FlowRunner` object takes the `Flow` object as well as the number of threads to be used for running the flow as parameters (Line 32).

## 2.3 Operator Development

The success of the stream processing paradigm depends, partly, on the availability of a wide range of generic operators. Such operators simplify the composition of streaming applications by enabling application developers to pick and configure operators from a preexisting set of cross-domain and domain-specific operators.

*Problems with the Event-Driven Programming Model.* The classical approach to operator development has been the event-driven model, in which a new operator is implemented by extending a framework class and overriding a tuple processing function to specify the custom operator logic. Examples abound (Storm 2015; S4 2015; Samza 2015; Hirzel et al. 2013).

However, the event-driven approach has several shortcomings. First, it makes the implementation of multi-input port operators that require synchronization difficult. Consider the implementation of a simple `Barrier` operator, whose goal is to take one tuple from each of its input ports and combine them into a single-output tuple. It is an operator that is commonly used at the end of task parallel flows. Recall that in the event-based model, the operator code executes as a result of tuple arrivals. Given that there is no guarantee about the order in which tuples will arrive from different input ports, the operator implementation has to keep an internal buffer per input port in order to implement the barrier operation. When the last remaining empty internal buffer receives a tuple, then the operator can produce an output tuple. More important than the increased complexity of implementation, there are also problems related to bounding the memory usage and/or creating back-pressure. Consider the case in which one of the input ports is receiving data at a higher rate. In this case, the internal buffer will keep growing. In order to avoid excessive memory usage, the operator has to block within the tuple handler function, which is an explicit form of creating back-pressure. Once blocking gets into the picture, then complex synchronization issues arise, such as determining how long to busy wait or using a wait/signal-style blocking synchronization.

Second, the event-driven approach makes it more difficult to implement intraoperator batching optimizations, as tuples arrive one at a time. Finally, in the presence of multi-input port operators, termination handling becomes more involved as well. One way to handle termination is to rely on punctuations (Andrade et al. 2014), which are out-of-band signals within a stream. One kind of punctuation is a *final marker* punctuation indicating that no more tuples are to be received from an input port. A multi-input port operator would track these punctuations from its input ports to determine when all of its ports are closed.

*Self-control-Based Programming Model.* C-Stream uses a self-control-based programming model in which each operator runs its own driver loop inside a `process` function. An operator completes its execution when its control loop ends, i.e., when the process function returns. This happens typically owing to a termination request or owing to no more input data being available for processing.

A typical operator implementation in C-Stream relies on data availability API calls to block until all the input data it needs is available for processing. A data availability call requests the runtime system to put the operator into the waiting state until the desired number of tuples are available

```
class Barrier : public Operator
{
public:
    Barrier(std::string const& name, int const numInputs)
        : Operator(name, numInputs, 1)
    {}

    void process(OperatorContext& context)
    {
        unordered_map<InputPort*, size_t> waitSpec;
        for (auto iport : context.getInputPorts())
            waitSpec[iport] = 1;
        auto& oport = *context.getOutputPorts().front();
        while (!context.isShutdownRequested()) {
            Status status = context.waitOnAllPorts(waitSpec);
            if (status == Status::Over) break;
            Tuple resultTuple;
            for (auto iport : context.getInputPorts())
                resultTuple.append(iport->popFrontTuple());
            oport.pushTuple(resultTuple);
        }
    }
};
```

Listing 2. Barrier operator implementation in C-Stream.

from the input ports. The wait ends when the requested data is available or when the runtime determines that the data will never be available. The latter can happen if one or more of the ports on which data is expected close before there are enough tuples to serve the request. An input port closes when all of its upstream operators are complete.

Listing 2 shows how a barrier operator is implemented in C-Stream. We focus on the `process` function, which contains the driver loop of the operator. The first thing the operator does is to set up a wait specification, which contains the number of tuples that the operator needs from each one of the input ports. For the barrier operator, the specification contains the value 1 for each one of the input ports. After the wait specification is set up, the operator enters its driver loop. The `context` object is used to check whether an explicit shutdown is requested. If not, the operator passes the wait specification to the `waitOnAllPorts` data availability API call in order to wait until at least one tuple is available from each one of the input ports. If the call reports that the request cannot be satisfied due to closed ports, then the barrier operator completes, as it cannot produce any additional output. Otherwise, it pops one tuple from each input port, combines them into a new output tuple, and pushes it into the output port.

To compare, an alternative implementation in a hypothetical event-driven programming model, inspired by SPL's Barrier operator implementation in IBM Streams (Gedik and Andrade 2012), is given in the Appendix. This alternative implementation is not only more complex in terms of the number of lines but also involves complex multithreading logic that relies on mutexes and conditional variables.

## 3 RUNTIME

In this section, we describe the runtime of C-Stream. We first explain the basic execution model used by C-Stream and then provide the algorithms that constitute the base scheduler.

### 3.1 Execution Model

The most straightforward way to support the programming model provided by C-Stream for operator development is to execute each operator as a separate thread. However, it is known that this kind of execution model does not scale with the number of operators (Carney et al. 2003). Instead, C-Stream executes each operator as a co-routine. This way, each operator has a stack of its own and the runtime system can suspend/resume the execution of an operator at well-controlled points within its `process` function. In particular, C-Stream can suspend the execution of an operator at two important points within the operator's processing logic: (1) data availability calls, and (2) tuple submission calls. These are also the points where the operator may need blocking, as there may not be sufficient data available for processing in the input ports or there may not be sufficient space available for submitting tuples to downstream input ports. One of the big advantages of co-routines compared to threads is that they can be suspended/resumed completely at the application level and with little overhead[2].

C-Stream executes operators using a pool of worker threads. When an operator blocks on a data availability call or on a tuple submission call, the scheduler assigns a new operator to the thread. We cover the details of how the thread pool size is adjusted in Section 4, in which we introduce elastic parallelism in C-Stream.

## 3.2 Scheduler

C-Stream has a pluggable scheduler. The scheduler provides the following base functionality irrespective of the plug-in used to customize its operation: data availability, back-pressure, preemption, and termination.

*A note on locks*: For the sake of simplicity, we have not detailed the locking scheme used by the scheduler in our descriptions of the algorithms. In practice, scheduler operations make use of reader/writer locks. The common case in which operators need not be blocked/unblocked is handled by just holding reader locks, avoiding congestion.

**ALGORITHM 1:** OperatorContext::waitForAll(*waitSpec*)

```
Param: waitSpec, wait specification mapping input ports to the # of tuples to wait for
Result: Over, if the request can never be satisfied; Done, if the wait specification is satisfied
begin
    needToWait ← true
    while needToWait do
        allAvailable ← true
        foreach (iport, count) in waitSpec do
            if |iport| < count then
                allAvailable ← false
                break
        if allAvailable then needToWait ← false
        else
            foreach (iport, count) in waitSpec do
                if iport.isClosed() and |iport| < count then return Over
        if needToWait then scheduler.markReadBlocked(this, waitSpec, Conj)
        else scheduler.checkForPreemption(this)
    return Done
```

### 3.2.1 *Data Availability.*

The scheduler supports data availability APIs by tracking the status of the wait requests of the operators. It puts the operators into the *Ready* or *Waiting* state depending on the availability of their requested number of tuples from the specified input ports. Such requests could be conjunctive (e.g., one from each input port) or disjunctive (e.g., one from any

[2]The *boost co-routines* library we use can context switch in 33 cycles on a modern 64-bit Intel processor; see http://www.boost.org/doc/libs/1_58_0/libs/coroutine/doc/html/coroutine/performance.html.

port). However, data availability has to also consider the termination scenarios. While an operator may be waiting for availability of data from an input port, that data may never arrive, as the upstream operator(s) may never produce any additional items due to termination. The scheduler tracks this via the *Completed* operator state.

*3.2.2 Back-pressure.* The scheduler handles back-pressure by putting limited size buffers on operator input ports. When an operator submits a tuple, the runtime system checks if space is available in the downstream input port buffers. In the case of space unavailability, the operator doing the submission will be put into the *Waiting* state until there is additional space in the downstream input ports to enable progress. Care needs to be taken for handling termination. If the downstream input port is attached to an operator that has moved to the *Completed* state owing to a global termination request, then the current operator should be put back into the Ready state so that it can terminate as well (avoiding a potential deadlock when the downstream input port is full). Another important case to handle is flows that involve cycles. Back-pressure along a cyclic path can cause deadlock. C-Stream handles this by limiting the feedback loops in the application to control ports—input ports that cannot result in production of new tuples but can change the internal state of an operator.

*3.2.3 Termination.* C-Stream handles termination using two mechanisms. The first is the *Completed* state for the operators, as we have outlined earlier. The second is the notion of *closed* input ports. An input port closes when its upstream operator(s) have all moved into the *Completed* state. In order for an operator to move into the *Completed* state, it needs to exit its driver loop. That typically happens when an explicit shutdown is requested or when all of the input ports are closed, that is, no more tuples can be received from them. An operator moving into the *Completed* state may result in unblocking some downstream operators that are waiting on data availability; these operators can learn about the unavailability of further data via their input ports' closed status.

*3.2.4 Preemption.* To prevent operators from starving and to provide low latency, C-Stream's base scheduler uses a quanta-based approach. As part of tuple availability and tuple submission calls, operators are checked for preemption. If the operator has used up its quanta, it is preempted. Depending on the scheduler plug-in, the next operator is selected for execution. Furthermore, C-Stream maintains per operator and per input port statistics, such as the amount of time that each operator has been executed over the recent past or how long tuples have waited on input port buffers on average. Such statistics can be used by scheduler plug-ins to implement more advanced preemption policies.

*3.2.5 Base Scheduler Algorithm.* We now describe the base scheduler algorithm. Recall that there are two points at which the operator code interacts with the scheduler. These are the data availability calls and tuple submission calls. We start our description of the algorithm from these.

**Data availability calls:** The operator context object provides two data availability calls: *waitForAll* (conjunctive wait) and *waitForAny* (disjunctive wait). The pseudo-code for these is given in Algorithms 1 and 2, respectively.

The *waitForAll* call takes a wait specification as a parameter, which maps ports to the number of tuples to wait from them. It blocks until the specified number of tuples are available from the input ports and returns *Done*. However, if at least one of the ports on which we are waiting tuples is closed without having the sufficient number of tuples presents, then the call returns *Over*. The closed status of a port is determined using the *isClosed* call on the input port, which returns *true* when all the upstream operators of a port are in the *Completed* state. The completion of operators typically propagates from the source toward the sinks. For example, in a typical chain topology, the source operator will move into the *Completed* state when it exits from its driver loop, typically

**ALGORITHM 2:** OperatorContext::waitForAny(*waitSpec*)

```
Param: waitSpec, wait specification mapping input ports to the # of tuples to wait for
Result: Over, if the request can never be satisfied; Done, if the wait specification is satisfied
begin
    needToWait ← true
    while needToWait do
        oneAvailable ← false
        foreach (iport, count) in waitSpec do
            if |iport| ≥ count then
                oneAvailable ← true
                break
        if oneAvailable then needToWait ← false
        else
            cannotSatisfy ← true
            foreach (iport, count) in waitSpec do
                if not iport.isClosed() or |iport| ≥ count then
                    cannotSatisfy ← false
                    if |iport| ≥ count then needToWait ← false
                    break
            if cannotSatisfy then return Over
        if needToWait then scheduler.markReadBlocked(this, waitSpec, Disj)
        else scheduler.checkForPreemption(this)
    return Done
```

owing to its source data being depleted or to a global shutdown request. The source operator moving into the *Completed* state will cause the downstream operator to receive an *Over* status if it was waiting for data on its input port, unblocking it so that it can exit its driver loop as well.

In the case that we need to wait, this is achieved by making a *markReadBlocked* call to the scheduler, asking it to put the operator into the *ReadBlocked* state. This is also a blocking call. The outer while loop in the algorithm ensures that the wait spec is reevaluated after the scheduler returns. This is important because if the return from the scheduler call is due to termination or port closure, the request may never be satisfied, in which case *Over* is returned.

Finally, in the case that we do not need to wait, we still make a call to the scheduler, named *checkForPreemption*. This is to check whether the operator should be preempted or not. The scheduler simply forwards this call to the scheduler plug-in, which decides whether the operator should be preempted.

The *waitForAny* call is similar in nature but returns *Over* only when none of the ports can ever satisfy the request.

**Tuple submission calls:** Output ports handle the tuple submissions, pseudo-code of which is given in Algorithm 3. To implement back-pressure, tuple submissions must block if at least one of the downstream input ports is full (the number of tuples is equal to *maxQueueSize*). However, there are two cases in which the input port sizes may go slightly over the limit.

The first is the shutdown case, in which a request for shutdown has been made. In this case, the tuple should be enqueued into the downstream ports right away, moving the control back to the operator's processing loop so that it can detect the shutdown request and return from its process function. This will enable the runtime to move the operator into the *Completed* state.

**ALGORITHM 3:** OutputPort::pushTuple(*tuple*)

```
Param: tuple, tuple to be pushed to all subscriber input ports
begin
    needToWait ← true
    while needToWait do
        waitSpec ← {}
        if not isShutdownRequested() then
            foreach iport in this.subscribers do
                if $|iport| \geq maxQueueSize$ then waitSpec.add(iport)
        if $|waitSpec| = 0$ then
            needToWait ← false
            foreach iport in this.subscribers do
                iport.pushTuple(tuple)
        if needToWait then scheduler.markWriteBlocked(this.oper, waitSpec)
        else scheduler.checkForPreemption(this.oper)
```

**ALGORITHM 4:** Scheduler::markReadBlocked(*oper*, *waitSpec*, *mode*)

```
Param: oper, operator to be blocked
Param: waitSpec, the wait specification of the operator
Param: mode, the mode of the blocking (Conj or Disj)
begin
    if mode = Conj then
        foreach (iport, count) in waitSpec do
            if iport.isClosed() then return
    else
        allClosed ← true
        foreach (iport, count) in waitSpec do
            if not iport.isClosed() then allClosed ← false break
        if allClosed then return
    waitCond ← oper.readWaitCondition()
    waitCond.setMode(mode)
    waitCond.setCondition(waitSpec)
    if not waitCond.isReady() then updateOperState(oper, ReadBlocked)
    else updateOperState(oper, Ready)
    oper.yield()
```

The other case happens when other operators that are being run by different threads have submitted tuples between our check of the queue sizes and doing the actual enqueuing of the tuple. This results in temporarily exceeding the queue size limit. However, this is a small compromise that avoids the need to hold multiport locks. The queue sizes would quickly go down once the upstream operators eventually move into the *WriteBlocked* state.

The output port uses the *markWriteBlocked* scheduler function for moving operators into the *WriteBlocked* state. If blocking is not needed due to back-pressure, the preemption is checked via the *checkForPreemption* scheduler function.

**Moving operators into the blocked state:** The scheduler uses the *markReadBlocked* and *markWriteBlocked* functions to move operators into blocked state, whose pseudo-codes are give in Algorithms 4 and 5, respectively.

**ALGORITHM 5:** Scheduler::markWriteBlocked(*oper*, *waitSpec*)

```
Param: oper, operator to be blocked
Param: waitSpec, set of ports that are full
begin
    if isShutdownRequested() then return
    waitCond ← oper.writeWaitCondition()
    waitCond.setCondition(waitSpec)
    if not waitCond.isReady() then updateOperState(oper, WriteBlocked)
    else updateOperState(oper, Ready)
    oper.yield()
```

**ALGORITHM 6:** Scheduler::markCompleted(*oper*)

```
Param: oper, operator to be moved into completed state
begin
    updateOperState(oper, Completed)
    foreach downOper in oper.subscribers() do
        if downOper.state() = ReadBlocked then updateOperState(downOper, Ready)
    if isShutdownRequested() then
        foreach upOper in oper.publishers() do
            if upOper.state() = WriteBlocked then updateOperState(upOper, Ready)
```

In the *markReadBlocked* function, the scheduler quickly rechecks whether the port closures would prevent the scheduler from moving the operator into the blocked state. For conjunctive wait specifications, this happens when any one of the ports is closed and for disjunctive ones when all of the ports are closed. Otherwise, the wait specification of the operator is recorded as part of the operator's scheduling state (*waitCond* variable). Then, the wait condition is reevaluated ($waitCond.isReady()$ call in the pseudo-codes), as the state of the input ports may have changed since the time operator context has detected that it should ask the scheduler to block. If this reevaluation still indicates the need to block, then the scheduler updates the operator state to *ReadBlocked*. Otherwise, it sets it to *Ready*. In both cases co-routine $yield()$ is called on the operator as the last step. Thus, the yield moves the control back to the worker thread, which will ask the scheduler for an available operator to execute. The scheduler will forward this request to the scheduler plug-in, which will pick one of the ready operators for execution.

In the *markWriteBlocked* function, we first check if a shutdown is requested and, if so, we return. This avoids a potential deadlock for the case in which a subscribing input port is full and its associated operator has already terminated. Otherwise, we record the wait specification of the operator as part of the operator's scheduling state (*waitCond* variable), and reevaluate it ($waitCond.isReady()$) to make sure that it is safe to block the operator. Then, the operator's scheduling state is updated and $yield()$ is called, as before.

**Moving operators into the completed state:** An operator moves into the *Completed* state when it exits its process function, at which point the *markCompleted* function of the scheduler is called. The pseudo-code for this function is given in Algorithm 6. As can be seen, the scheduler simply moves the operator into the *Completed* state has to consider two important scenarios.

First, if there are subscribers to the output ports of the operator that are in the *ReadBlocked* state, these downstream operators may never satisfy their wait specifications owing to the completion of this operator. For this purpose, the scheduler puts them into the *Ready* state. Recall from

**ALGORITHM 7:** Scheduler::markInputPortWritten(*iport*)

```
Param: iport, input port that is written
begin
    foreach oper in operators.readBlockedOn(iport) do
        waitCond ← oper.readWaitCondition()
        if waitCond.isReady() then updateOperState(oper, Ready)
        else if waitCond.isReady(iport) then
        waitCond.remove(iport)operators.removeReadBlockedOn(iport)
```

**ALGORITHM 8:** Scheduler::markInputPortRead(*iport*)

```
Param: iport, input port that is read
begin
    foreach oper in operators.writeBlockedOn(iport) do
        waitCond ← oper.writeWaitCondition()
        if waitCond.isReady() then updateOperState(oper, Ready)
        else if waitCond.isReady(iport) then writeBlockedOperators.remove(iport)
```

Algorithms 1 and 2 that once *markReadBlock* returns, the operator will reevaluate whether it should return *Over* or *Done* to the user code or go back to the blocked state via another *markReadBlocked* call to the scheduler.

Second, if there is a pending termination request and there are publishers to the input ports of the operator that are in the *WriteBlocked* state, these upstream operators may never unblock as the completed operator will no longer process any tuples from its input ports. For this purpose, the scheduler puts them into the *Ready* state. Recall from Algorithm 3 that once the *markWriteBlock* returns, the operator will see the shutdown request and push the tuple to the downstream buffers right away.

**Moving operators into the ready state:** A tuple being pushed into an input port buffer can potentially unblock the operator associated with that input port. This can happen if the operator is in the *ReadBlocked* state and its wait specification includes the port in question. Similarly, a tuple popped from an input port can potentially unblock upstream operators that are pushing tuples to this input port. This can happen if the upstream operators are in the *WriteBlocked* state and their wait specifications include the port in question. These cases are handled by the *markInputPortWritten* and *markInputPortRead* functions of the scheduler, whose pseudo-codes are given in Algorithms 7 and 8. These two functions are called by the *pushTuple* and *popTuple* functions of the input ports, respectively. An input port's *pushTuple* function gets called by an output port's *pushTuple* function, as was shown earlier in Algorithm 3. An input port's *popTuple* function is called by the user code implementing an operator, as was shown earlier in Listing 2.

The *markInputPortWritten* function iterates over the *ReadBlocked* operators whose wait specifications include the input port that is written. It reevaluates their wait specification and, if satisfied, puts them into *Ready* state. Otherwise, it checks whether the part of the wait specification related to the input port that is written is satisfied and, if so, removes that input port from the waiting specification of the operator. The *markInputPortRead* works similarly, with the exception that it does not remove the current input port from the wait specification when the condition is partially satisfied. This is because input ports can have multiple publishers; thus, the availability of space has to be reevaluated the next time.

*3.2.6 Scheduler Plug-Ins.* The scheduler consults the scheduler plug-in to decide on (i) whether an operator needs to be preempted or not and (ii) which *Ready* operator a thread should execute next. To help plug-ins implement this functionality, the scheduler makes available the following information:

- last scheduled time of operators,
- status of input port buffers (including enqueue times of the tuples),
- recent read rates of input ports,
- recent write rates of output ports, and
- fraction of conjunctive and disjunctive wait calls made by the operator[3].

Using this information, different scheduler plug-ins can be implemented for different goals, such as low latency and high throughput. We have developed the following schedulers, all using a configurable quanta-based preemption:

- **RandomScheduling:** Pick uniformly at random among *Ready* operators.
- **MaxQueueLengthScheduling:** Pick the operator with the maximum input queue size, with the exception that if there is a source operator in the *Ready* state, then pick it. Ties are broken randomly.
- **MinLatencyScheduling:** Scheduling decision is based on the timestamp of the front tuple in the input port buffers of *Ready* operators. The operator whose front tuple has the minimum timestamp value is picked. For source operators, their last scheduled time is used.
- **LeastRecentlyRunScheduling:** Pick the least recently scheduled operator among the *Ready* ones.
- **MaxRunningTimeScheduling:** Scheduling decision is based on the estimation of how long an operator can execute. To compute that, statistics such as port buffer fullness, read rate from input ports and write rate to output ports are used. The execution time of the operator is computed as the minimum of how long it can read from its input port buffers (buffer tuple count divided by operator's read rate from the input port) and how long it can write to input port buffers of its subscriber operators (available space in port buffer of subscriber operator divided by operator's write rate to the output port).

## 4 ADAPTATION

In this section, we describe the adaptation capabilities of C-Stream, which include two main functionalities: (*i*) adjusting the number of threads used to schedule the operators and (*ii*) using data parallelism to resolve bottlenecks.

### 4.1 Dynamic Thread Pool Size

C-Stream adjusts the number of threads used to schedule the operators based on a metric called *average utilization*. The controller that manages thread-pool size tracks this metric periodically, using an adaption period of $\Delta$ seconds. At the end of each adaptation period, for each worker thread a utilization value is computed as the fraction of time that the thread has spent running operators during the last period. The average utilization, denoted by $U$, is then computed over all threads and gives a value in range $[0, 1]$. A *low threshold* $U_l$ is used to decrease the number of threads when the average utilization is low (threads are mostly idle). That is, if $U < U_l$, then the number of threads is decreased. A *high threshold*, $U_h > U_l$, is used to increase the number of threads when average utilization is high. That is, if $U > U_h$, then the number of threads is increased. $\Delta$,

[3]In practice, most operators make use of only one of these calls.

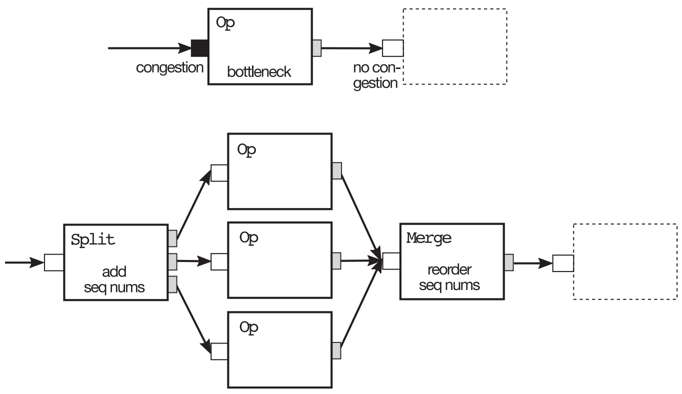


Fig. 2. Elastic data parallelism.

**ALGORITHM 9:** DetectBottleneck(*candidates*)

```
Param: candidates, list of single input/output operators; $\tau$ congestion threshold parameter
Result: bottleneck operator if exists, null otherwise
begin
    foreach $op \in candidates$ do
        if not $op.isReplicated$ then
            if $op.iport.writeBlockedRatio \geq \tau$ and
                    $op.oport.writeBlockedRatio < \tau$ then return $op$
        else
            $avgIPortWriteBlockedRatio =$
                    $\sum_{op' \in op.replicas} op'.iport.writeBlockedRatio / |op.replicas|$
            if $avgIPortWriteBlockedRatio \geq \tau$ and
                    $op.oport.writeBlockedRatio < \tau$ then return $op$
    return null
```

$U_l$, and $U_h$ are configurable parameters of the adaptation algorithm and C-Stream increases or decreases the thread counts by 1 at each adaptation period.

## 4.2 Elastic Data Parallelism

C-Stream applies elastic data parallelism, wherein streaming operators are replicated to resolve bottlenecks. For this purpose, C-Stream first detects bottleneck operators and then increases the number of replicas for them. Increasing the number of replicas enables the replicated operators to be executed by more than one thread. Also, it enables the bottleneck processing task to receive additional scheduling time.

Figure 2 illustrates how C-Stream uses data parallelism to resolve bottlenecks. In the upper part of the figure, we see an operator that is determined as the bottleneck. Note that its downstream input port is not experiencing congestion, yet its input port does. In the bottom part of the figure, we see that the bottleneck is resolved by replicating the operator in question. This is achieved by using split and merge operators before and after the bottleneck operator, respectively. The split operator partitions the stream over the replicas. It also assigns sequence numbers to the tuples so that these tuples can be ordered later by the merge operator. If the bottleneck operator is a partitioned stateful one, the splitting can be performed using hashing on the partitioning key. Otherwise, it will be a round-robin distribution.

*4.2.1 Bottleneck Detection.* Stream-C performs bottleneck detection based on a simple principle: an operator that has no downstream input ports that are congested, yet at least one input port that is congested is a bottleneck operator. The former condition makes sure that we do not include operators that are blocked due to back-pressure in our definition. The second condition simply finds operators that are not processing their input tuples fast enough and, thus, are bottlenecks.

To define congestion, we use a metric called *write blocked ratio*. For an input port, it is the fraction of time that the port buffer stays full. For an output port, we define it as the maximum write blocked ratio of the subscribing downstream input ports. Algorithm 9 describes how bottleneck operators are found using these metrics.

Stream-C applies data parallelism only for operators with single-input and single-output ports. Bottleneck operators are selected from candidate operators with this property. Furthermore, congestion threshold parameter $\tau$ is used to determine if a port is congested. Concretely, a port is congested if and only if its write blocked ratio is greater than or equal to $\tau$.

Among the candidates' operators, if an operator is not replicated then it is a bottleneck if and only if its input port is congested and its output port is not congested. If an operator is replicated, then the same rule applies, with the exception that the write blocked ratio for the input port is computed by averaging it over all of the replicas of the operator. There is no change for the computation of the output port write blocked ratio, as there is only a single downstream input port subscribing to the output ports of the replicas, which is the input port of the merge operator.

*4.2.2 Replica Controller.* C-Stream has a replica controller that adjusts the number of replicas of operators to improve the throughput. Every adaptation period, it runs the bottleneck detection procedure to locate a bottleneck operator. If there are no bottleneck operators (all input ports have write block ratios that are below the congestion threshold $\tau$), then it does not take any action. If there are bottleneck operators, then their number of replicas are incremented by one. Increasing the number of replicas for an operator results in the operator being scheduled more often. We do not limit the number of replicas explicitly, assuming that additional computational resources result in improving the throughput. To handle IO bound operators, simple blacklisting techniques can be applied (Tang and Gedik 2013).

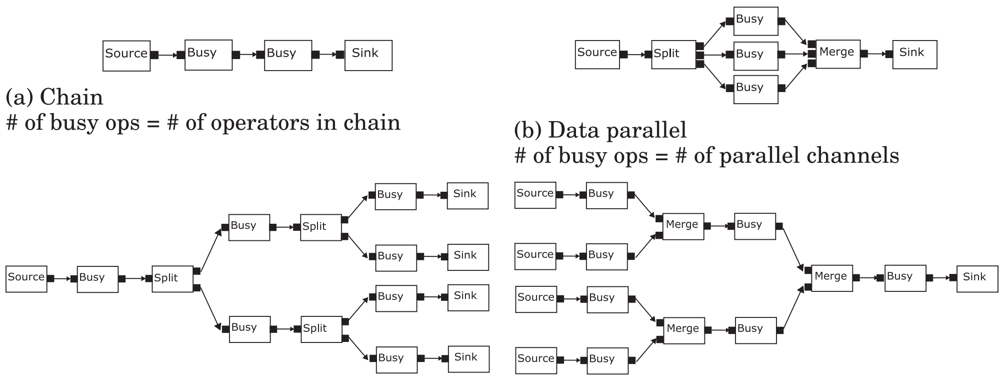
## 5 EXPERIMENTS

In this section, we present our experimental results. First, we provide base experiments studying the performance and scalability of C-Stream under varying topologies, application sizes (number of operators), operator costs, and scheduler plug-ins. Second, we provide experiments showcasing the effectiveness of our adaptation module.

All of our experiments were performed on a host with $2 \times 2$ GHz Intel Xeon processors, each containing 6 cores. In total, we have a machine with 12 cores, running Linux with kernel version 2.6. In the base experiments, the default value for the number of threads is set as 4 and the default selectivity is set as 1, even though we experiment with varying values for both. In adaptation experiments, the default selectivity value is 1, and the default scheduler plug-in is *RandomScheduling*. In all of our experiments, quanta value is set as 50 milliseconds.

### 5.1 Base Experiments

Our base evaluations are performed on applications with varying topologies under varying application sizes, operator costs, and selectivity values. The selectivity of an operator is the number of tuples that it produces for each tuple that it consumes. Usually, this is a number in the range [0, 1], representing operators performing filtering.



(a) Chain
# of busy ops = # of operators in chain

(b) Data parallel
# of busy ops = # of parallel channels

(c) Tree
# of busy ops = $2^n - 1$, $n$ is the tree height

(d) Reverse tree
# of busy ops = $2^n - 1$, $n$ is the tree height

Fig. 3. Application topologies.

For this purpose, we generate parameterized topologies, which consist of *chain*, *data parallel*, *tree*, and *reverse tree* topologies. Structures of these topologies are shown in Figure 3. In these experiments, our adaptation module is disabled and we use throughput and average latency as performance metrics to evaluate scalability of the system as well as the impact of different scheduling plug-ins on these metrics.

In all of our experiments, we use busy operators that are parametrized by cost and selectivity values. For each incoming tuple, a busy operator performs a busy loop for the time specified by its cost parameter and forwards the tuple with the probability specified by its selectivity parameter. We have 12 busy operators in our chain and data parallel experiments. In tree and reverse tree experiments, we set the tree depth to 6, and branching factor to 2, resulting in 63 busy operators in total. Unless otherwise stated, costs of the busy operators are equal to 100 microseconds per tuple.

**Number of threads:** In our first experiment, we show the effect of the number of threads on the throughput and latency for each scheduler plug-in and for each topology. For the chain topology, as we increase the number of threads, throughput increases linearly and average latency decreases, as shown in Figures 4(a) and 5(a). Throughput that we obtain from different scheduler plug-ins are nearly the same. The reason is that, since we have 12 busy operators of equal cost and 12 threads at most, roughly speaking, all operators require the same amount of scheduling, which is a scheduling requirement that can be satisfied by all the scheduler plug-ins with ease. Despite having similar throughput, we observe that the *LeastRecently* plug-in provides the best latency results.

Figures 4(b) and 5(b) show the effect of the number of threads on throughput and latency, respectively, for the data parallel topology. While throughput increases as we increase the number of threads, it starts decreasing after some value between 9 and 11 threads, depending on the scheduler plug-in used. The reason is that the merge operator eventually becomes a bottleneck since it is sequential. Having more threads than actually needed makes the problem worse owing to the scheduling overhead.

After closer examination, we have found that significant drops in performance are due to having too many threads. These threads pick up operators that were recently executed just because they are again in the ready state after little space opens up in their downstream port buffers. However, once these operators resume execution, they would quickly move into the waiting state after doing just a little work, causing significant scheduling overhead. This experiment shows the importance of setting the number of threads correctly, which we address via our adaptation module. We study

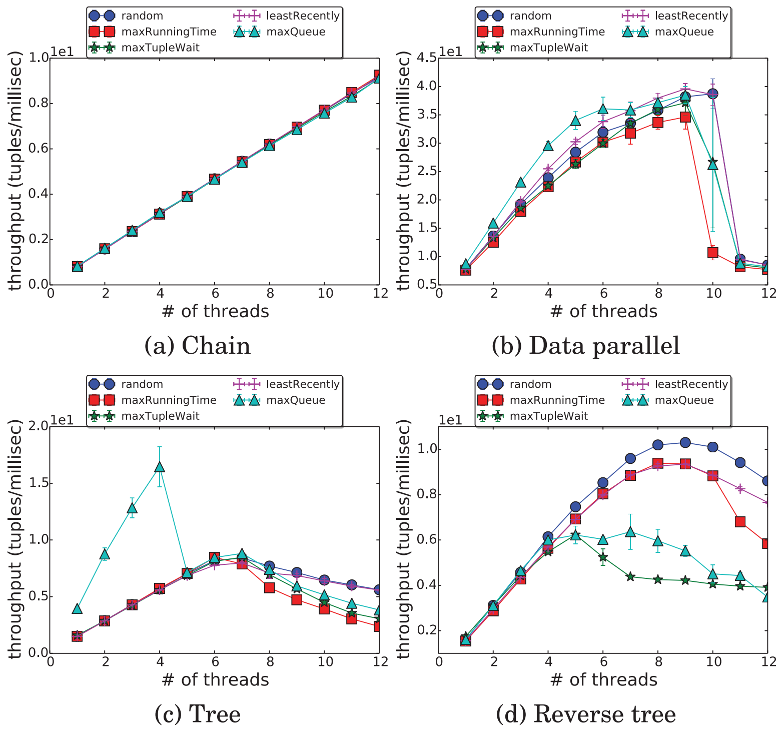
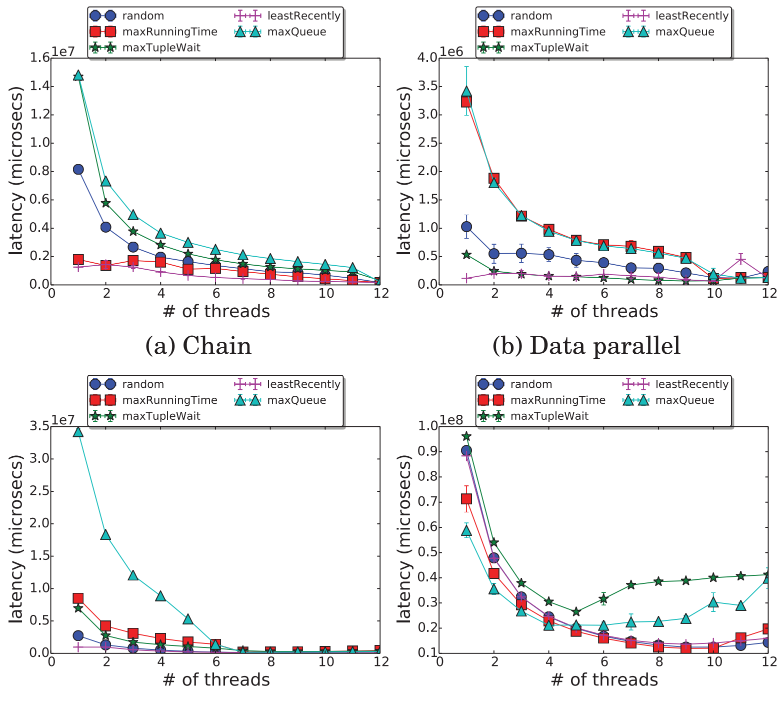


Fig. 4. Number of threads versus throughput.

the effectiveness of our adaptation module in Section 5.2. From these experiments, we also observe that *MaxQueue* provides slightly higher throughput compared to other alternatives but at the cost of significantly increased latency, especially for small numbers of threads. *MaxTupleWait* and *LeastRecently* plug-ins provide the lowest latencies.

Figures 4(c) and 5(c) show the effect of the number of threads on throughput and latency, respectively, for the tree topology. Similar to data parallel topology, throughput increases only up to a certain number of threads, after which a downward trend in throughput starts. However, unlike the data parallel scenario, the decrease in throughput is less steep. In the tree topology, the input rates of operators decrease as we go deeper down in the tree since the tuples are distributed nondeterministically across the downstream ports. Concretely, if the input rate for an operator is $r$, then the input rate for its downstream operators is $r/b$, where $b$ is the branching factor. This causes upstream busy operators to become bottlenecks. The *MaxQueue* plug-in provides higher throughput compared to other alternatives, but only up to 4 threads, reaching as high as 3 times the throughput of *Random*. However, this comes at the cost of increased latency, as high as 3.5 times that of *Random*. Lowest latency is again provided by the *LeastRecently* plug-in.

Figures 4(d) and 5(d) show the effect of the number of threads on throughput and latency, respectively, for the reverse tree topology. Results are similar to the data-parallel and tree scenarios, wherein throughput increases up to a certain number of threads and then starts decreasing. It is surprising that *Random* plug-in provides the best throughput (10% higher than other alternatives). While *MaxQueue* has shown solid performance for all other topologies with respect to throughput, it performs poorly for the reverse topology. In particular, the highest throughout it could achieve is 40% lower than that of *Random*. At peak throughput, latencies provided by different plug-ins are close to each other, except for *MaxTupleWait*, which shows higher latency.



(a) Chain (b) Data parallel (c) Tree (d) Reverse tree

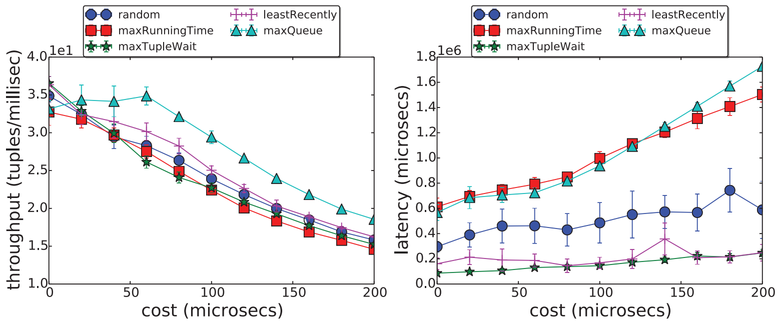
Fig. 5. Number of threads versus latency.

We summarize our observations from these experiments as follows:

- Stream-C, without elastic adaptation, scales well with increasing threads only up to a certain point. For certain topologies, such as data-parallel topology, the throughput significantly decreases if the number of threads becomes higher than the ideal.
- While the *MaxQueue* scheduler plug-in can provide improved throughput for certain topologies, such as data-parallel and tree topologies, the *Random* is quite robust across all topologies in terms of throughput.
- While the *LeastRecently* scheduler plug-in can provide improved latency for certain topologies—such as chain, data-parallel, and tree topologies—the *Random* is quite robust across all topologies in terms of latency.

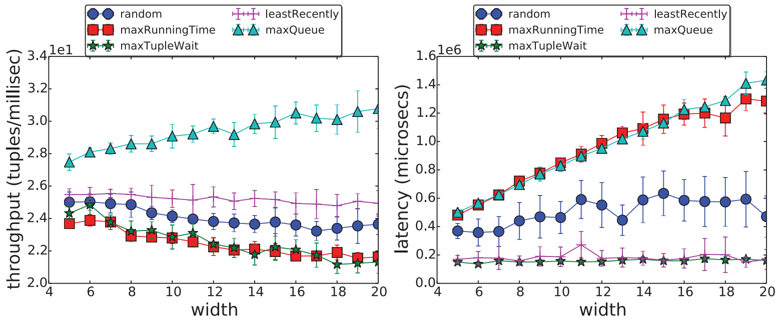
**Operator Cost:** In this experiment, we show the effect of busy operator costs on throughput and latency, using data-parallel topology of 12 busy operators. We fix the number of threads to 4. Figure 6(a) shows that the throughput decreases as we increase the cost of the busy operators and the decrease in throughput is linear in the increase in operator costs. Figure 6(b) shows that latency increases as we increase the busy operator cost and, again, we see a mostly linear trend. The only exception is the *MaxQueue* scheduler plug-in, whose initial rate of latency increase shows an increasing trend, but as the operator cost increases, the rate of increase stabilizes. Furthermore, its rate of latency increase is higher than other plug-ins.

**Application Size:** This experiment shows the effect of application size (in terms of the number of operators) on throughput and latency for the data-parallel topology. Figure 7(a) shows that for most of the scheduler plug-ins, throughput does not change significantly since the number of



(a) Cost vs. tput. (b) Cost vs. latency

Fig. 6. Data-parallel cost experiments.



(a) Width vs. tput. (b) Width vs. latency

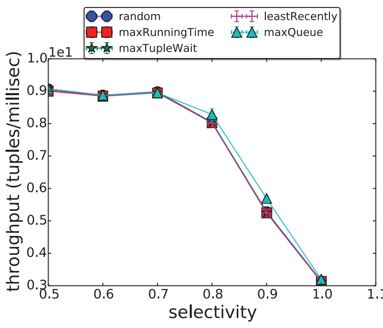
Fig. 7. Data parallel application size experiments.

data parallel operators is at least equal to the number of threads, which is 4. The *MaxQueue* plug-in shows increasing throughput as a result of an increasing number of data parallel operators, whereas others show a slight decrease. The slight decrease can be explained by increased operator management and scheduling overhead. The reason for the increase in *MaxQueue* plug-in's performance is a surprising one: an increased number of data-parallel operators results in smaller input queue sizes for them; this, in turn, increases the amount of scheduling time that the merger gets. Figure 7(b) shows the effect of the number of data-parallel operators on latency. We observe that *MaxQueue* and *MaxRunningTime* have linearly increasing latencies, whereas other plug-ins show more stable results.

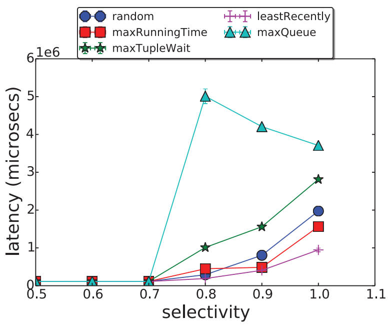
**Selectivity:** In this experiment, we show the effect of operator selectivity on throughput and latency using the chain topology of 12 busy operators. Each busy operator has the same cost and same selectivity value. Selectivity determines the number of output tuples generated by an operator per input tuple consumed. As shown in Figure 8(a), throughput decreases and, as shown in Figure 8(b), latency increases as we increase operator selectivity.

## 5.2 Adaptation Experiments

In this section, we look at the performance of the elastic parallelization module of C-Stream. First, we perform our experiments using the chain topology. The chain topology with adaptation is similar to the data-parallel topology, but the number of replicas for bottleneck operators (i.e., busy operators) is adjusted automatically. Furthermore, with adaptation, we resolves bottlenecks in multiple

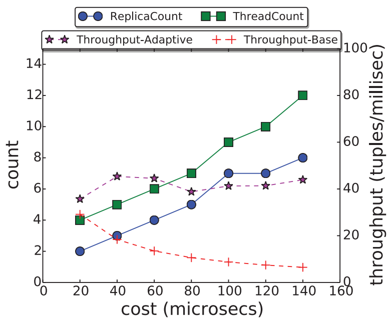


(a) Selectivity vs. throughput

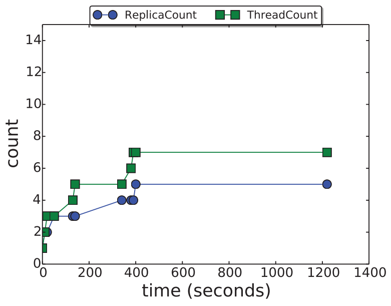


(b) Selectivity vs. latency

Fig. 8. Chain selectivity experiments.



(a) Cost vs. thread, replica count and throughput



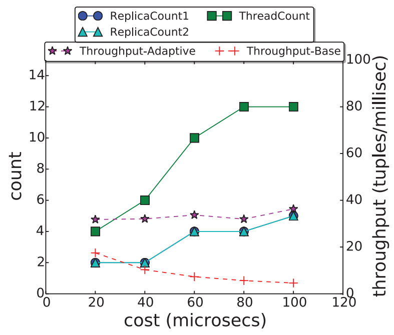
(b) Time vs. thread, replica count

Fig. 9. Adaptation 1 busy experiment.

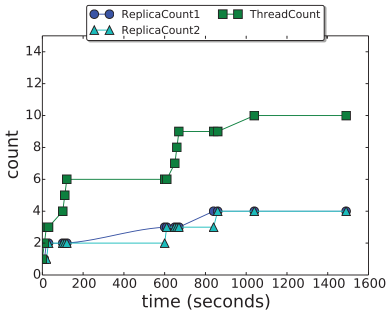
busy operators. In these experiments, we compare throughput values obtained from elastic scaling against the throughput values from the single-thread, single-replica scenario. Next, we compare the throughput values obtained from elastic scaling against the case in which the number of threads and replicas are set manually to obtain the best throughput. For all the experiments in this section, the maximum number of threads is set to 12. The scheduler plug-in used is *Random*. $\tau$ (congestion threshold) is set to 0.01, low-threshold $U_l$ is set to 0.90, and high-threshold $U_h$ is set to 0.95. Our adaptation period is 10 seconds.

We have a single busy operator in our first experiment. The throughput results from this experiment are presented in Figure 9(a), where the $x$ axis represents the busy operator cost, y axis the final thread and replica count, and the secondary $y$ axis the throughput. The figure shows that both the number of threads used and the number of replicas of the busy operator increase as the cost of the busy operator is increased. Furthermore, while the throughput value decreases dramatically in the single-thread, single-replica scenario, the adaptation module of C-Stream prevents that, and throughput values remain stable as the operator cost increases.

Figure 9(b) plots the number of operator replicas and thread count, as a function of time, for the operator cost of 80 microseconds. The time represents the duration since the start of the application. The figure shows that the numbers of threads and replicas increase and eventually stabilize. In this particular case, stabilization happens at 5 replicas and 7 threads. For the sake of brevity, further datapoints are not shown.



(a) Cost vs. thread, replica count and throughput



(b) Time vs. thread, replica count

Fig. 10. Adaptation 2 busy experiment.

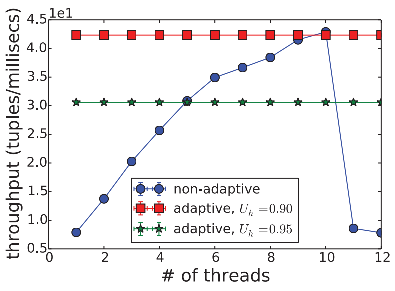
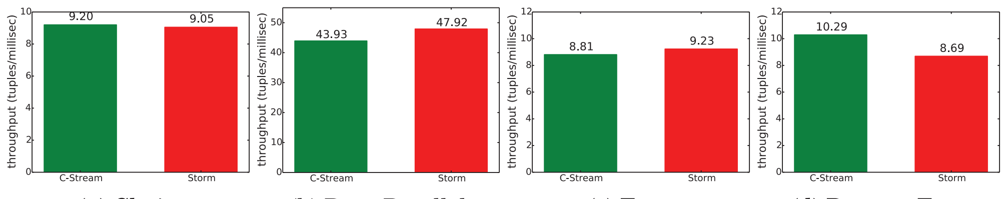


Fig. 11. Adaptation data-parallel experiment.

In the second experiment, we use 2 busy operators with the same cost. The throughput results from this experiment are presented in Figure 10(a). This figure shows that, together with the number of active threads, the number of replicas for each of the busy operators increases as the costs of the busy operators increase. Also, it shows that while throughput value decreases in the base case, the adaptation module of C-Stream maintains a stable throughput.

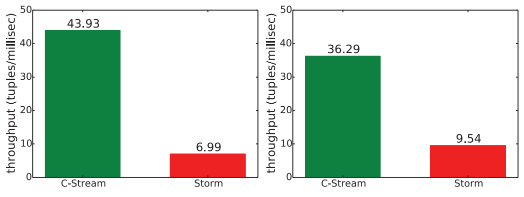
Figure 10(b) plots the number of operator replicas and thread count, as a function of time, for the operator cost of 60 microseconds. It shows that the adaptation module resolves the bottleneck and increments the replica count for one of the busy operators first and increments the replica count for the other busy operator next; this pattern continues until there is no congestion left in the flow. The congestion goes away when both busy operators gain 4 replicas. The number of total threads stabilizes at 10.

In the third adaptation experiment, we use a data-parallel topology of 12 busy operators for evaluating the effectiveness of the adaptation module with respect to adjusting the number of threads. High threshold $U_h$ is tried with two different values in this experiment: 0.95 and 0.90. Figure 11 plots the throughput obtained as a function of the number of threads for the case in which the adaptation module is disabled and the final throughput achieved via the adaptation module for different high thresholds. The figure shows that setting $U_h$ to 0.90 increases the number of threads more aggressively than the case of $U_h = 0.95$, and obtains the maximum throughput achievable with the system.



(a) Chain (b) Data Parallel (c) Tree (d) Reverse Tree

Fig. 12. C-Stream versus Storm, base experiments—adaptation disabled.



(a) 1-Busy Operator (b) 2-Busy Operators

Fig. 13. C-Stream versus Storm—adaptation enabled.

## 5.3 C-Stream vs. Apache Storm

In this section, we compare the performance of C-Stream with Apache Storm version 1.1.0. The goal of the comparison is to see if C-Stream introduces any overhead beyond that of a production-ready system when using the same number of threads. We should note that the parallelism level is set for the entire topology in C-Stream. In contrast, Storm creates one thread (executor) for each operator in the topology.

In the first set of experiments, we disabled the elastic parallelism module in C-Stream as we did in Section 5.1 and evaluated the performance of both systems in 4 different topologies: chain, data parallel, tree, and reverse tree. We have 12 busy operators in our chain and 8 busy operators in the data-parallel topology. In the tree and reverse-tree experiments, we set the tree depth to 6 and branching factor to 2, resulting in 63 busy operators in total. We set the cost of busy operators to 100 microseconds and selectivity to 1.

Figure 12 shows that C-Stream presents comparable results to Storm. Results also show that the programming model that we present in C-Stream does not create any overhead while making operator development easier compared to a popular open-source stream processing system that follows the event-based model.

In the second set of experiments, we compare C-Stream, with the adaptation module enabled, to Storm, using the same setup we used in Section 5.2. In the first scenario, we have a chain topology with 1 busy operator of cost 100 microseconds. Figure 13(a) shows that C-Stream outperforms Storm by 6.2$x$ by increasing the number of replicas and threads dynamically. Similarly, in the second scenario, in which we have 2 busy operators of cost 140 microseconds, C-Stream outperforms Storm by 3.8$x$.

## 6 RELATED WORK

Stream processing has been an active area of research and development over the last decade. Systems such as STREAM (Arasu et al. 2003), Borealis (Abadi et al. 2005), TelegraphCQ (Chandrasekaran et al. 2003), IBM InfoSphere Streams (Gedik and Andrade 2012), and

StreamBase (2015) are examples of academic and industrial stream processing middleware. In many cases, these systems provide declarative languages for developing streaming applications as well. StreamIt (Thies et al. 2002), Aspen (Upadhyaya et al. 2007), and SPL (Hirzel et al. 2013) are domain-specific programming languages for high-level stream programming, which shield users from the complexity of parallel and distributed systems. There are also open-source stream processing platforms such as Storm (2015), Apache S4 (S4 2015), and Apache Samza (Samza 2015).

Storm (2015), Apache S4 (S4 2015), Apache Samza (Samza 2015), SPL (Hirzel et al. 2013), and many other systems adopt an event-driven model for operator development. In this model, process function of an operator is fired for each incoming tuple. The problem with this approach is that development of multiport operators requires additional effort to provide port synchronization and to handle back-pressure resulting from the difference in incoming stream rates at the input ports. In C-Stream, operators have their own driver loop and tuple access is orchestrated via our data availability API. To manage operator termination, punctuations are used in InfoSphere Streams (Andrade et al. 2014). In C-Stream, termination is handled by our base scheduler, separating the termination control from the operator's tuple execution logic. These features significantly simplify operator development in C-Stream.

Scheduling relies on operator grouping in SPADE (Gedik et al. 2008), in which a set of operators are grouped and assigned to a processing element. Within a single processing element there could be multiple threads, but the assignment of operators to threads is not dynamic. In their work on Aurora, Carney et al. (2003) use superboxes for scheduling, which are sequences of operators that are executed as an atomic group. However, no results are given on throughput scalability with increasing number of worker threads. In C-Stream, our scheduling relies on using one co-routine per operator while keeping the number of worker threads flexible. Furthermore, C-Stream supports elastic parallelization, adjusting the number of threads to resolve bottlenecks.

The StreamIt compiler auto-parallelizes operators using a round robin split to guarantee ordering, but only for stateless operators with static selectivity. In Schneider et al. (2015) and Gedik et al. (2014), stateful operators are parallelized by partitioning the state by keys. Similar techniques can also be found in distributed database systems (DeWitt et al. 1990; Graefe 1990). It is also the main technique behind the success of batch processing systems such as Map/Reduce (Dean and Ghemawat 2008) and Isard et al. (2007). Brito et al. (2008) describe how to apply auto-parallelization using software transactional memory, but only if selectivity is 1 and memory is shared.

To exploit the parallelization opportunities contained within streaming applications, an auto-pipelining solution is proposed in Tang and Gedik (2013) for multicore processors. While the assumptions and goals are similar to that of C-Stream's, this technique cannot take advantage of data parallelism.

In Storm (2015), data parallelism can be achieved by requesting multiple copies of operators. However, preserving order is left to the developers. In S4 (2015), creating processing element replicas enables data parallelism. Again, safety is left to the developer. In C-Stream's elastic parallelization module, split and merge operators are automatically inserted before and after operator replicas to maintain the tuple order.

Elasticity under distributed streaming environments or in the Cloud introduces additional research issues, which is beyond the scope of this work. It requires operator to machine mapping (placement), and operator state migration as a result of scaling in and out. FUGU (Heinze et al. 2013) is an allocation component for distributed complex event processing systems, which is able to elastically scale in and out under varying system loads. In Heinze et al. (2014), auto-scaling techniques are presented on top of FUGU, including local thresholds, global thresholds, and reinforcement learning. StreamMine3G (Martin et al. 2014) is another elastic stream processing system that supports both vertical and horizontal scalability. Stormy (Loesing et al. 2012), on the other hand, is an elastic stream processing engine running in the Cloud. As part of elasticity, migration protocols are proposed for operators in Heinze et al. (2014) and Gedik et al. (2014). In Gedik et al. (2014), an incremental migration protocol that relies on consistent hashing is proposed. In Heinze et al. (2014), a migration algorithm that aims at reducing migration latency is proposed. It is based on operator movement cost estimation. In contrast to these systems, C-Stream is a single-node, multicore stream processing system, with a focus on flexible scheduling and elastic streaming execution.

In the general area of auto-parallelization (not specific to streaming), dynamic multithreaded concurrency platforms—such as Cilk++ (2015), OpenMP (2015), and x10 (Charles et al. 2005)—decouple expressing a program's innate parallelism from its execution configuration. OpenMP and Cilk++ are widely used language extensions for shared memory programs, in which parallel execution in a program is expressed at development time and the system takes advantage of it at runtime. Kremlin (Garcia et al. 2011) is an auto-parallelization framework that complements OpenMP. Kremlin recommends to programmers a list of regions for parallelization, which is ordered by achievable program speedup.

## 7 CONCLUSION

In this article, we presented C-Stream—a co-routine-based scalable, highly customizable, and elastic stream processing engine. Unlike traditional event-based stream processing operators, C-Stream operators have their own driver loop and can decide when to perform their computations by employing data availability APIs provided by the runtime. This property of C-Stream simplifies the development of operators that require multiport synchronization. As part of C-Stream, we introduced a customizable scheduler that handles back-pressure, provides data availability APIs, and manages preemption and termination handling. It can be configured via plug-ins to achieve different goals, such as high throughput or low latency. We described the adaptation module in C-Stream, which adjusts the level of parallelism by detecting bottleneck operators, incrementing the replica counts until bottlenecks are resolved, and adjusting the number of threads used. We showcased the effectiveness of C-Stream via an extensive experimental evaluation.

## APPENDIX

```
class Barrier : public Operator {
public:
    Barrier(string const& name, int numInputs, int bufferCapacity)
        : Operator(name, numInputs, 1), buffers_(TupleBuffer(), numInputs),
          bufferCapacity_(bufferCapacity), numNonEmptyQueues_(0), numPortsCompleted_(0) {}
    void process(Tuple tuple, int port) {
        LockGuard guard(mutex_);
        TupleBuffer& buffer = buffers_[port];
        // create back-pressure by blocking if the buffer is full
        while (!isShutdownRequested() &&
                 numPortsCompleted_ < getNumInputPorts() &&
                 buffer.size() == bufferCapacity_) {
            notFullCV_.wait(mutex_);
            if (getShutdownRequested() ||
                  numPortsCompleted_ == getNumInputPorts())
                return;
        }
        // insert the new tuple
      if (buffer.isEmpty())
            numNonEmptyQueues_++;
        buffer.push(tuple);
        // perform the barrier if no empty buffers
        if (numNonEmptyQueues_ != getNumInputPorts())
            return;
        bool needSignal = false;
        Tuple resultTuple;
        for (Buffer& oBuffer : buffers_) {
            if (oBuffer.size() == bufferCapacity_)
                needSignal = true;
            resultTuple.append(oBuffer.pop());
            if (oBuffer.isEmpty())
                numNonEmptyQueues_--;
        }
        // wake-up blocked threads if needed
        if (needSignal)
            notFullCV_.broadcast();
    }
    void process(Punctuation punc, int port) {
        LockGuard guard(mutex_);
        if(punc == Punctuation::Final) {
            numPortsCompleted_++;
            if (numPortsCompleted_ == getNumInputPorts())
                notFullCV_.broadcast();
        }
    }
    void shutdown() {
        notFullCV_.broadcast();
    }
private:
    // tuple buffers
    vector<TupleBuffer> buffers_;
    // synchronization variables
    Mutex mutex_;
    ConditionVariable notFullCV_;
    // bookkeeping variables
    int bufferCapacity_;
    int numNonEmptyQueues_;
    int numPortsCompleted_;
};
```

Listing 3. Barrier operator—event-driven implementation, assuming that different input ports are driven by different threads.

## REFERENCES

Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Er Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. 2005. The design of the Borealis stream processing engine. In *Biennial Conference on Innovative Data Systems Research (CIDR'05)*. 277–289.

Henrique Andrade, Buğra Gedik, and Deepak Turaga. 2014. Application development–data flow programming. In *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press, New York, NY.

Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. 2003. STREAM: The Stanford stream data manager. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*. 665–665.

Andrey Brito, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. 2008. Speculative out-of-order event processing with software transaction memory. In *Proceedings of the ACM International Conference on Distributed Event-based Systems (DEBS'08)*. 265–275.

Don Carney, Uğur Çetintemel, Alex Rasin, and Stan Zdonik. 2003. Operator scheduling in a data stream manager. In *Proceedings of the International Conference on Very Large Databases (VLDB'03)*.

Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. 2003. TelegraphCQ: Continuous dataflow processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*. 668–668.

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*. 519–538.

Cilk++. 2015. Cilk++. Retrieved May 2015 from https://cilkplus.org.

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Communications of the ACM* 51, 1, 107–113.

David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-I Hsiao, and Rick Rasmussen. 1990. The gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering* 2, 1, 44–62.

Saturnino Garcia, Donghwan Jeon, Christopher M. Louie, and Michael Bedford Taylor. 2011. Kremlin: Rethinking and rebooting gprof for the multicore age. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. 458–469.

Manoj K. Garg, Duk-Jin Kim, Deepak S. Turaga, and Balakrishnan Prabhakaran. 2010. Multimodal analysis of body sensor network data streams for real-time healthcare. In *Multimedia Information Retrieval*. 469–478.

Buğra Gedik and Henrique Andrade. 2012. A model-based framework for building extensible, high performance stream processing middleware and programming language for IBM infosphere streams. *Software: Practice and Experience Journal* 11, 42.

Buğra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. 2008. SPADE: The system s declarative stream processing engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*. 1123–1134.

Buğra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. 2014. Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 6, 1447–1463.

Goetz Graefe. 1990. Encapsulation of parallelism in the volcano query processing system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'90)*. 102–111.

Thomas Heinze, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. 2014. Latency-aware elastic scaling for distributed data stream processing systems. In *Proceedings of the ACM International Conference on Distributed Event-based Systems (DEBS'14)*. 13–22.

Thomas Heinze, Yuanzhen Ji, Yinying Pan, Franz Josef, Grueneberger Zbigniew, and Jerzak Christof Fetzer. 2013. Elastic complex event processing under varying query load. In *International Workshop on Big Dynamic Distributed Data (BD3'13)*. 25.

Thomas Heinze, Valerio Pappalardo, Zbigniew Jerzak, and Christof Fetzer. 2014. Auto-scaling techniques for elastic data stream processing. In *Proceedings of the ACM International Conference on Distributed Event-based Systems (DEBS'14)*. 318–321.

Martin Hirzel, Henrique Andrade, Buğra Gedik, Gabriela Jacques-Silva, Rohit Khandekar, Vibhore Kumar, Mark Mendell, Howard Nasgaard, Scott Schneider, Robert Soule, and Kun-Lung Wu. 2013. Streams processing language: Analyzing big data in motion. *IBM Journal of Research and Development* 57, 3/4, 7:1–7:11.

Martin Hirzel, Robert Soule, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A catalog of streaming optimizations. *ACM Computing Surveys* 4, 46.

Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'07)*. 59–72.

Simon Loesing, Martin Hentschel, Tim Kraska, and Donald Kossmann. 2012. Stormy: An elastic and highly available streaming service in the cloud. In *EDBT/ICDT Workshops*. 55–60.

André Martin, Andrey Brito, and Christof Fetzer. 2014. Scalable and elastic realtime click stream analysis using streammine3G. In *DEBS'14*. 198–205.

OpenMP. 2015. OpenMP. Retrieved May 2015 from http://openmp.org.

S4. 2015. S4 project. Retrieved May 2015 from http://incubator.apache.org/s4.

Samza. 2015. Apache Samza project. Retrieved May 2015 from http://incubator.apache.org/samza.

Douglas L. Schales, Mihai Christodorescu, Josyula R. Rao, Reiner Sailer, Marc Ph. Stoecklin, Wietse Venema, and Ting Wang. 2014. Stream computing for large-scale, multi-channel cyber threat analytics. In *IEEE International Conference on Information Reuse and Integration (IRI'14)*. 8–15.

Scott Schneider, Martin Hirzel, and Buğra Gedik an Kun-Lung Wu. 2015. Safe data parallelism for general streaming. *IEEE Transactions on Computers* 64, 2, 504–517.

Storm. 2015. Apache Storm project. Retrieved May 2015 from http://storm-project.net/.

StreamBase. 2015. Tibco Streambase. Retrieved May 2015 from http://www.streambase.com.

Yuzhe Tang and Buğra Gedik. 2013. Autopipelining for data stream processing. *IEEE Transactions on Parallel and Distributed Systems* 24, 12, 2344–2354.

William Thies, Michal Karczmarek, and Saman P. Amarasinghe. 2002. StreamIt: A language for streaming applications. In *Proceedings of the International Conference on Compiler Construction (CC'02)*. 179–196.

Gautam Upadhyaya, Vijay S. Pai, and Samuel P. Midkiff. 2007. Expressing and exploiting concurrency in networked applications in Aspen. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'07)*. 13–23.

Petros Zerfos, Mudhakar Srivatsa, Hao Yu, D. Dennerline, Hubertus Franke, and Dakshi Agrawal. 2013. Platform and applications for massive-scale streaming network analytics. *IBM Journal of Research and Development* 57, 3/4, 11:1–11:13.

Xiaolan J. Zhang, Henrique Andrade, Buğra Gedik, Richard King, John Morar, Senthil Nathan, Yoonho Park, Raju Pavuluri, Edward Pring, Randall Schnier, Philippe Selo, Michael Spicer, Chitra Venkatramani, Andy Frenkiel, Wim De Pauw, Michael Pfiefer, Paul Allen, Norman Cohen, and Kun-Lung Wu. 2009. Implementing a high-volume, low-latency market data processing system on commodity hardware using IBM middleware. In *Workshop on High-Performance Computational Finance at Supercomputing*.

Received September 2015; revised May 2017; accepted January 2018