C-STREAM: A COROUTINE-BASED ELASTIC STREAM PROCESSING ENGINE

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

By

Semih Şahin

June, 2015

C-Stream: A Coroutine-based Elastic Stream Processing Engine
By Semih Şahin
June, 2015

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Buğra Gedik(Advisor)

Assoc. Prof. Dr. Hakan Ferhatosmanoğlu

Assist. Prof. Dr. Gültekin Kuyzu

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Graduate School

ABSTRACT

C-STREAM: A COROUTINE-BASED ELASTIC STREAM PROCESSING ENGINE

Semih Şahin
M.S. in Computer Engineering
Advisor: Assoc. Prof. Dr. Buğra Gedik
June, 2015

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 is naturally captured by the stream processing model of computation, such as pipeline, task, and data parallelism. In this thesis, 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 stream processing operators, C-Stream provides an interface wherein each operator has its own control loop and rely on data availability APIs to decide when to perform its computations. The self-control based model significantly simplifies development of operators that require multi-port synchronization. Second, C-Stream contains a multi-threaded dynamic scheduler that manages the 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 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. 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.

Keywords: Stream processing, Big data, Coroutine.

ÖZET

C-STREAM: EŞ PROGRAM TABANLI ESNEK AKAN VERİ İŞLEME MOTORU

Semih Şahin
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Doç. Dr. Buğra Gedik
Haziran, 2015

Akan veri işleme, canlı veriyi havada işleme üzerine olan bir programlama paradigmasıdır. Bu paradigma içinde barındırdığı ardışık düzen, veri ve görev paralelleştirme methodlarını kullanarak, uygulamaların birim zamanda üretilen iş miktarını arttırmasına yada birim iş parçası başına duşen ortalama işlem süresinin azalmasına olanak saglar. Bu tez çalışmasında, elastik akan veri işleme motoru olan C-Stream dizayn ve uygulamaları geliştirilmiştir. İlk olarak C-Stream, literatürdeki çalışmaların çoğunluğunun benimsediği olaya-dayalı işleç geliştirme methodunun aksine eş-program tabanlı işleç geliştirme modelini sunmaktadır. Bu modelde her işleç, kendi kontrol döngüsüne sahip olmakta, veri erişilebilirlik uygulama programi arabirimi (UPA) ile veri işleme zamanını kontrol edebilmektedir. Bu model çok-portlu işleç geliştirme sürecini basitleştirmektedir. Ikinci olarak, C-Stream işleçlerin çalışmasını kontrol eden, çok izlekli dinamik zamanlayıcı barındırmaktadır. Bu zamanlayıcı, eklentiler ile de özelleştirilebilmektedir. Eklentilerden bagımsız olarak, zamanlayıcı geri-baski problemini çozer, işleçlerin veri erişilebilirlik UPA'sına ulaşımını saglar, işleçlerin çalısma esnasında durdurulması ve sonlanmasını kontrol eder. Son olarak, C-Stream elastik paralelleştirme ozelliğine sahiptir. Dinamik olarak aktif çalısan izlek sayısını kontrol etmekle beraber, uygulamada tıkanmaya sebep olan işleçleri tespit ederek, onların kopya sayısını arttırır ve tıkanmayı ortadan kaldırır. Yaptığımız deneyler gostermektedir ki, C-Stream ölçeklenebilir, özelleştirilebilir ve esnek paralelleştirme ozelliğine sahip bir akan veri işleme uygulaması geliştirme motorudur.

Anahtar sözcükler: Akan veri isleme, Buyuk veri, es-program.

Acknowledgement

First and foremost, I owe my deepest gratitude to my supervisor, Assoc. Prof. Dr. Buğra Gedik for his encouragement, motivation, guidance and support throughout my studies.

Special thanks to Assoc. Prof. Dr. Hakan Ferhatosmanoğlu and Assist. Prof. Dr. Gültekin Kuyzu for kindly accepting to be in my committee. I owe them my appreciation for their support and helpful suggestions.

I would like to thank to my father Ertuğrul, my mother Öznur and my brother Ali for always being cheerful, motivating and supportive. None of this would have been possible without their love. I am tremendously grateful for all the selflessness and the sacrifices they have made on my behalf. I am sure that, they are proud of me for this work.

I consider myself to be very lucky to have the most valuable friends Bahadır, Said, Salih, Selahattin, Süheyl, Mustafa, Abdurrahman and Doğukan. I would also like to thank to my special office mates for sharing their knowledge and supporting me all the time.

Contents

1 Introduction **1**

2 Programming Model **4**

2.1 Basic Concepts 4

2.2 Flow Composition 5

2.3 Operator Development 7

3 Runtime **11**

3.1 Execution Model 11

3.2 Scheduler 12

3.2.1 Overview 12

3.2.2 Base Scheduler Algorithm 13

3.2.3 Scheduler Plug-Ins 21

4 Adaptation **23**

4.1 Dynamic Thread Pool Size 23

4.2 Elastic Data Parallelism 24

4.2.1 Bottleneck Detection 25

4.2.2 Replica Controller 26

5 Experiments **27**

5.1 Base Experiments 27

5.2 Adaptation Experiments 34

6 Related Work **37**

7 Conclusion **40**

List of Figures

2.1 Example flow graph from Figure 2.1. 7

4.1 Elastic data parallelism. 24

5.1 Application topologies. 28

5.2 # of threads vs. throughput 29

5.3 # of threads vs. latency 30

5.4 Data parallel cost experiments 32

5.5 Data parallel application size experiments 33

5.6 Chain selectivity experiments 33

5.7 Adaptation 1 busy experiment 34

5.8 Adaptation 2 busy experiment 35

5.9 Adaptation data parallel experiment 36

Chapter 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 [1], telecommunications [2], cyber-security [3], and health-care [4] to name a few.

A streaming application is typically represented as a graph of streams and operators [5], where operators are generic data manipulators and streams connect operators to each other using 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 is naturally captured by the streaming model of computation [6], 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 parallelism. Second, we need a flexible scheduler that can dynamically schedule operators to take advantage of pipeline, task, and data parallelism in a transparent manner. Furthermore, the scheduler should be configurable so that we can adjust the trade-off between low latency and high throughput. Last, but not the 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 thesis, 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 control loop and rely on data availability 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 intra-operator optimizations such as batching. Second, C-Stream contains a multi-threaded dynamic scheduler that manages the 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 on 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 (in terms of scheduling goals), and can resolve bottlenecks by dynamically adjusting the level of data parallelism used (elasticity).

[1]This requires state migration and ordering support, which is not yet implemented in our prototype, but have been implemented in other systems [7].

In summary, this thesis makes the following contributions:

- We propose an operator development API that facilitates sequential implementations, significantly simplifying development of multi-port operators that otherwise require explicit synchronization.
- We develop a flexible scheduler and accompanying runtime machinery for executing operators that are implemented as co-routines, using multiple threads.
- We present techniques for elastic executing, 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 thesis is organized as follows. Chapter 6, discusses related work. Chapter 2 overviews the programming model and the operator development APIs used by C-Stream. Chapter 3 describes the co-routine based runtime, the multi-threaded scheduler, and the custom scheduler plug-ins we have developed for it. Chapter 4 explains how Stream-C achieves elasticity. Chapter 5 presents our experimental evaluation and Chapter 7 concludes the thesis.

Chapter 2

Programming Model

In this chapter, we first give a brief overview of the basic concepts in stream processing. We then describe the programming model used by C-Stream. The latter has two aspects: *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 an input port 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 could be multiple stream connections originating from an output port, called a *fan-out*. Similarly, there could be multiple stream connections destined to an input port, called a *fan-in*.

Three major kinds of parallelism are inherently present within streaming applications.

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, where 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 importantly, a merge operation after the processing, in order to re-establish the original tuple order. Data parallelism can be applied to *stateless* as well as *partitioned stateful* operators [8]. Stateless operators are those that do not maintain state across tuples. Partitioned operators do maintain state, 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.

Stream-C takes advantage of all these forms of parallelism, which we cover in Chapter 4.

2.2 Flow Composition

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 we cover in detail in the next section.

Stream-C supports flow composition using an API-based approach, employing the `C++11` language. Listing 2.1 shows how a simple streaming application is composed using these APIs. Figure 2.1 depicts the same application in graphical

form.

```
Flow flow("sample␣application");

// create the operators
auto& names = flow.createOperator<FileSource>("name_source")
  .set_fileName("data/names.dat")
  .set_fileFormat({{"id",Type::Integer},{"name",Type::String}});

auto& values = flow.createOperator<TCPSource>("value_source")
  .set_address("my.host.com", 44000)
  .set_dataFormat({{"id",Type::Integer},{"value",Type::Integer}});

auto& filter = flow.createOperator<Filter>("empty_filter")
  .set_filter(MEXP1( t_.get<Type::String>("name") != "" ));

auto& combiner = flow.createOperator<Barrier>("combiner", 2);

auto& sink = flow.createOperator<FileSink>("file_sink")
  .set_fileName("data/out.dat")
  .set_fileFormat({{"id",Type::Integer},
                   {"name",Type::String},{"value",Type::Integer}});

// create the connections
flow.addConnections( (names,0) >> (0,filter,0) >> (0,combiner) );
flow.addConnections( (values,0) >> (1,combiner,0) >> (0,snk) );

// configure the runner
FlowRunner & runner = FlowRunner::createRunner();
runner.setInfrastructureLogLevel(Info);
runner.setApplicationLogLevel(Trace);

// run the application and wait for completion
int numThreads = 2;
runner.run(flow, numThreads);
runner.wait(flow);
```

Listing 2.1: Flow composition in C-Stream.

A `Flow` object is used to hold the data flow graph. Operators are created using the `createOperator` function of the `Flow` object. 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. Operators are configured via their `set_` methods,

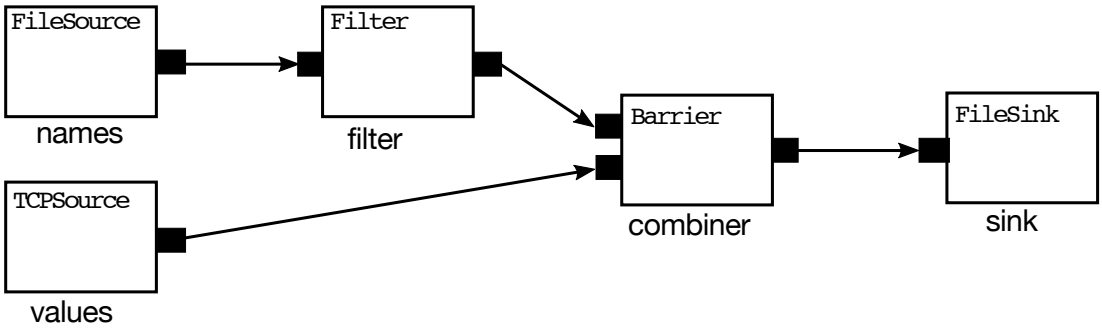


Figure 2.1: Example flow graph from Figure 2.1.

which are specific to each operator kind. The parameters to operators can also be lambda expressions, such as the `filter` parameter of the `Filter` operator. 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 `createConnections` function of the `Flow` object. The `>> C++` operator is overloaded to create chains of connections. For instance, `(names,0) >> (0,filter,0) >> (0,combiner)` represents a chain of connections, where 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. 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.

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 the application developers to pick and configure operators from a pre-existing set of cross-domain and domain specific

operators.

Problems with the Event-driven Programming Model

The classical approach to operator development has been to use an event-driven model, where a new operator is implemented by extending a framework class and overriding a tuple processing function to implement the custom operator logic. Examples abound [9, 10, 5].

However, the event-driven approach has several disadvantages. 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 one. 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 empty internal buffer receives a tuple, then the operator can produce an output tuple. More important than the increased complexity of implementation, there is also the problem of limiting memory use and/or creating back-pressure. Consider the case when 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 problems arise, such as how long to block.

Second, the even-driven approach makes it more difficult to implement intra-operator batching optimizations, as tuples arrive one at a time. Finally, in the presence of multi-input port operators, termination handling becomes more difficult. One way to handle termination is to rely on punctuations [11], which are out-of-band signals within a stream. One kind of puctuation is a *final marker* punctuation that indicates no more tuples are to be received from a stream. A

multi-input port operator would track these punctuations from its input ports to determine when all its ports are closed.

Self-control based Programming Model

C-Stream uses a self-control based programming model, where each operator runs its own control 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 due to a termination request or due 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 waiting state until the desired number of tuples are available from the input ports. The wait ends when the requested data is available or when the system knows 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 2.2 shows how a barrier operator is implemented in C-Stream. We focus on the `process` method, which contains the control loop of the operator. The first thing the operator does is to setup a wait specification, which contains the number of tuples the operators 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 spec is set up, the operator enters into its main loop. The `context` object is used to check whether an explicit shutdown is requested. If not, the operators 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 closed ports, then the barrier operator completes, as it cannot produce any additional output anymore. Otherwise it pops one tuple from each input port, combines them into a new output tuple and pushes this new tuple into the output

port.

```
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 2.2: Barrier operator implementation in C-Stream.

Chapter 3

Runtime

In this chapter 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. We end this chapter with scheduler plug-in we implemented.

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 [12]. 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, 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 they 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[1].

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 Chapter 4, where 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, termination.

3.2.1 Overview

Data availability: The scheduler supports data availability APIs by tracking the status of the wait specifications of the operators. It puts the operators into *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 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.

Backpressure: The scheduler handles backpressure by putting limited size

[1]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`

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 *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 due to a global termination request, then the operator should be put back into the Ready state, so that it can terminate as well (avoiding a deadlock). Another important case 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 the operator.

Termination: C-Stream handles termination using two mechanisms. The first one is the *Completed* state for the operators, as we have outlined earlier. The second one is the notion of closed input ports. In order for an operator to move into the *Complete* state, it needs to exit its main control loop, and for most operators, that happens when either an explicit shutdown is requested, or when the input ports are closed, that is no more tuples can be received from them. An operator moving into *Complete* state may result in unblocking some downstream operators that are waiting on data availability, and these operators can learn about the unavailability of further data via their input ports' closed status.

Preemption: C-Stream's base scheduler uses a quanta based approach to preempt operators in order to provide low latency. Furthermore, C-Stream maintains per operator and per input port statistics, such as the amount of time each operator has been executed over the recent past or how much tuples have waited on input port buffers. Such statistics can be used by scheduler plug-ins to implement more advanced preemption policies.

3.2.2 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

Algorithm 1: OperatorContext::waitForAllPorts(*waitSpec*)

```
Param : waitSpec, wait specification that maps an input port to the number of
        tuples to wait on that port
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
```

availability calls and the tuple submission calls. We start our description of the algorithm from these.

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

The *waitForAllPorts* 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 *Completed*

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

```
Param : waitSpec, wait specification that maps an input port to the number of
        tuples to wait on that port
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
```

state. The completion of operators typically propagate from the source towards the sinks. For example, in a typical chain topology, the source operator will move to the *Completed* state when it exits from its main loop, typically due to its source data being depleted or due to a global shutdown request. Note that a source operator cannot be waiting on an input port, as it does not have any. The source operator moving into *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 main loop as well.

In the case where we need to wait, this is achieved by making a

Algorithm 3: OutputPort::push(*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 subscribers do
                if |iport| ≥ maxQueueSize then
                    waitSpec.add(iport)
        if |waitSpec| = 0 then
            needToWait ← false
            foreach iport in subscribers do
                iport.pushTuple(tuple)
        if needToWait then
            scheduler.markWriteBlocked(oper, waitSpec)
        else
            scheduler.checkForPreemption(oper)
```

markReadBlocked call to the scheduler, asking it to put the operator into *Read-Blocked* state. This is also a blocking call. The outer while loop in the algorithm ensures that the return from the scheduler call is not due to termination of port closure.

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. In both algorithms, the check for the *Over* state is done using a separate loop to avoid checking whether a port is closed in the fast path.

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 subscriber input ports are full (the number of tuples is equal to *maxQueueSize*) However, there are two cases in which the

input port sizes may go over slightly over the limit.

The first is the shutdown case, where 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 method. This will enable the runtime to move the operator into the *Completed* state.

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

The output port uses the $markWriteBlocked$ scheduler function for moving operators to the *WriteBlocked* state. If no blocking is needed due to back-pressure, the preemption is checked via the $checkForPreemption$ scheduler method.

Moving operators into blocked state: The scheduler uses the $markReadBlocked$ and $markWriteBlocked$ methods to move operators into blocked state, whose pseudo-codes are give in Algorithms 4and 5, respectively. The scheduler methods are executed while holding a scheduler lock.

In the $markReadBlocked$ method, the scheduler quickly re-checks if the port closures should prevent the scheduler from moving the operator into blocked state. For conjunctive wait specifications, this happens when any one of the ports are closed; and for disjunctive ones, when all of the ports are closed. Otherwise, the wait specification of the operator is recorded as part of scheduler state ($waitCond$ variable). Then the wait condition is re-evaluated ($waitCond.isReady()$ call in the pseudo-codes), as between the time the operator context has detected that it should ask the scheduler to block and the time of the actual call to block, the state of the input ports may have changed. If this re-evaluation still indicates

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
    // while holding the scheduler lock
    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 iport.isClosed() ≠ true then
                allClosed ← false break
        if allClosed then
            return
    waitCond ← oper.getReadWaitCondition()
    waitCond.setMode(mode)
    waitCond.setCondition(waitSpec)
    if waitCond.isReady() = false then
        updateOperState(oper, ReadBlocked)
    else
        updateOperState(oper, Ready)
    oper.yield()
```

the need to block, then the scheduler updates the operator state to *ReadBlocked*. Otherwise, it sets it to *Ready*. In both cases *yield*() is called on the operator as the last step. Recall that operators are co-routines. 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* method, we first check if a shutdown is requested and if so, we return. This avoids a potential deadlock if a subscribing input port whose operator has completed is full. Otherwise, we record the wait specification of the operator as part of scheduler state (*waitCond* variable), and re-evaluate it (*waitCond.isReady*()) to make sure it is safe to block the operator. Then the operator's scheduling state is updated and *yield*() is called as before.

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

```
Param : oper, operator to be blocked
Param : waitSpec, set of ports that are full
begin
    // while holding the scheduler lock
    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
    // while holding the scheduler lock
    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)
```

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

First, if the are subscribers to the output ports of the operator that are in *ReadBlocked* state, these subscribers may never satisfy their wait specifications due to the completion of this operator. For this purpose, the scheduler puts them into *Ready* state. Recall from Algorithms 1 and 1 that once *markReadBlock* returns, the operator will re-evaluate whether is should return *Over* or *Done* to

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

```
Param : iport, input port that is written
begin
    // while holding the scheduler lock foreach oper in
    operators.readBlockedOn(iport) do
        waitCond ← oper.readWaitCondition()
        if waitCond.isReady() then
            updateOperatorState(oper, Ready)
        else if waitCond.isReady(iport) then
            waitCond.remove(iport)
```

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

```
Param : iport, input port that is read
begin
    // while holding the scheduler lock foreach oper in
    operators.writeBlockedOn(iport) do
        waitCond ← oper.writeWaitCondition()
        if waitCond.isReady() then
            updateThreadState(oper, Ready)
```

the user code, or go back to 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 *WriteBlocked* state, these publisher may never unblock as the completed operator will not process any tuples from its input ports anymore.

For this purpose, the scheduler puts them into *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 ready state: The *popTuple* and *pushTuple* methods of the input ports go through the scheduler so as to unblock operators when needed. A tuple being pushed into an input port buffer can potentially unblock operators that are in *ReadBlocked* state and whose wait specifications include the port in question. Similarly, a tuple popped from an input port can potentially unblock operators that are in *WriteBlocked* state. These cases are handled by

the $markInputPortWritted$ and $markInputPortRead$ methods of the scheduler, whose pseudo-codes are given in Algorithms 7 and 8.

The $markInputPortWritten$ method iterates over the *ReadBlocked* operators whose wait specifications include the input port that is written. It re-evaluates their wait specification and if satisfied, puts them into *Ready* state. Otherwise, it checks whether the part of the wait specification that is about 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 form the wait specification when the it is partially satisfied. This is because input ports can have multiple publishers, so the the availability of space has to be re-evaluated the next time.

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 of not blocking/unblocking operators is handled by just using reader locks, avoiding congestion.

3.2.3 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 these functionality, the scheduler makes available the following information: Last scheduled time of operators, input port buffers, recent read rates of input ports, recent write rates of output ports, enqueue time of tuples waiting in input port buffers, and fraction of conjunctive and disjunctive wait calls made by the operator. Using these statistics, different scheduler plug-ins can be implemented considering different QoS such as low latency or high throughput.

We have developed the following schedulers, all using a configurable quanta

based preemption:

- **RandomScheduling**: The operator to be scheduled is selected randomly among *Ready* operators.
- **MaxQueueLengthScheduling**: The operator to be scheduled is the one with the maximum input queue size, with the exception that if there is a source operator in *Ready* state, then it is scheduled. Ties are broken randomly.
- **MinLatencyScheduling**: Scheduling decision is based on the timestamp of the front tuple in the input ports buffers of *Ready* operators. The operator whose front tuple has the minimum timestamp value is scheduled. For source operators, these values are set to operators' last scheduled time.
- **LeastRecentlyRunScheduling**: Among the *Ready* operators, the least recently scheduled is selected.
- **MaxRunningTime**: 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).

Chapter 4

Adaptation

In this chapter we describe the adaptation capabilities of C-Stream, which include two main functionalities: *i*) adjusting the number of threads used to schedule the operators, *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 pool size controller tracks this metric periodically, using an adaption period of, Δ. At the end of each adaptation period, for each worker thread a utilization value is computed as the fraction of time the thread run 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). I.e., if $U < U_l$, then the number of threads is decreased. A *high threshold*, $1 - \epsilon = U_h > U_l$, is used to increase the number of threads when the utilization is close to 1. I.e., if $U > U_l$, then the number of threads is increased. C-Stream updates the thread counts by 1 at each adaptation period.

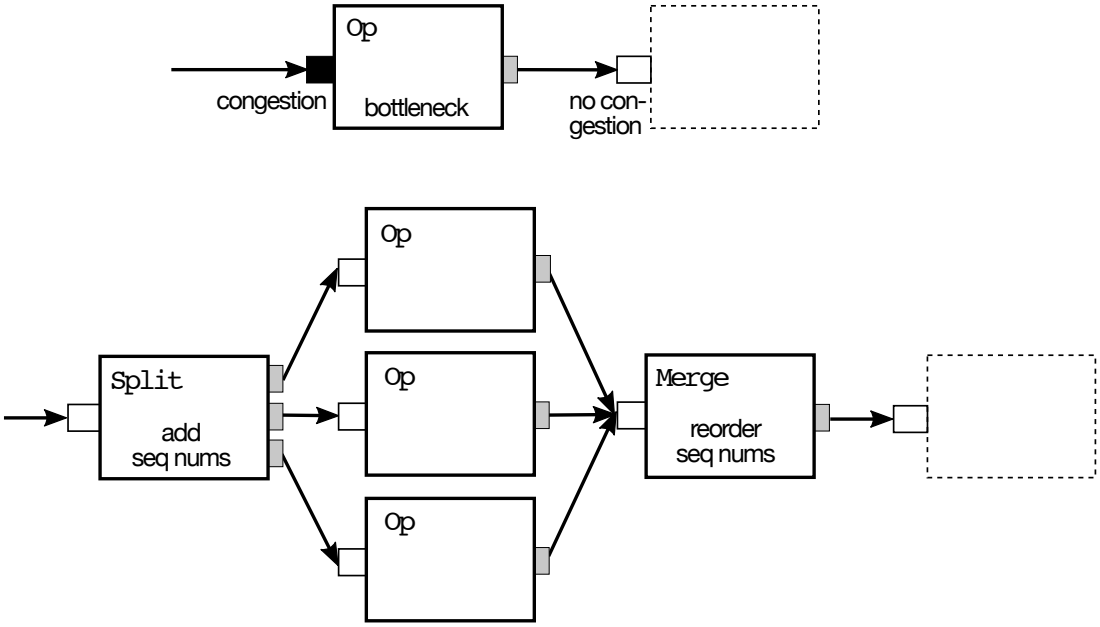


Figure 4.1: Elastic data parallelism.

4.2 Elastic Data Parallelism

C-Stream applies elastic data parallelism, where 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 operators to be executed by more than one thread at the same time, as well as enable the original bottleneck processing task to receive more scheduling time.

Figure 4.1 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 tuple, so that these tuples can be ordered later by the merge operator. If the bottleneck operator is a partitioned stateful one,

Algorithm 9: DetectBottleneck(*candidates*)

Param: *candidates*, list of single input/output operators
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
      $avgIPortWriteBlockedRatio = \frac{\sum_{op' \in op.replicas} op'.iport.writeBlockedRatio}{|op.replicas|}$
      if $avgIPortWriteBlockedRatio \geq \tau$
      and $op.oport.writeBlockedRatio < \tau$ **then**
        return op
  return *null*

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 its input tuples fast enough, thus is a bottleneck.

To define congestion, we use a metric called *write blocked ratio*. For an input port, it is the fraction of time 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, thus bottleneck operators are selected from candidate operators

with this property. Among the candidates, if an operator is not replicated then it is a bottleneck iff its input port's write blocked ratio is above the congestion threshold τ; and its output port's write blocked ratio is below the congestion threshold. 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 the replicas of the operator. There is no change for the output port write blocked ratio, as there is only a single downstream input port subscribing to the output port of all the replicas, which is the input 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 blow the congestion threshold τ), then it does not take any action. If there is a bottleneck operator, then its number of replicas are incremented by one.

Chapter 5

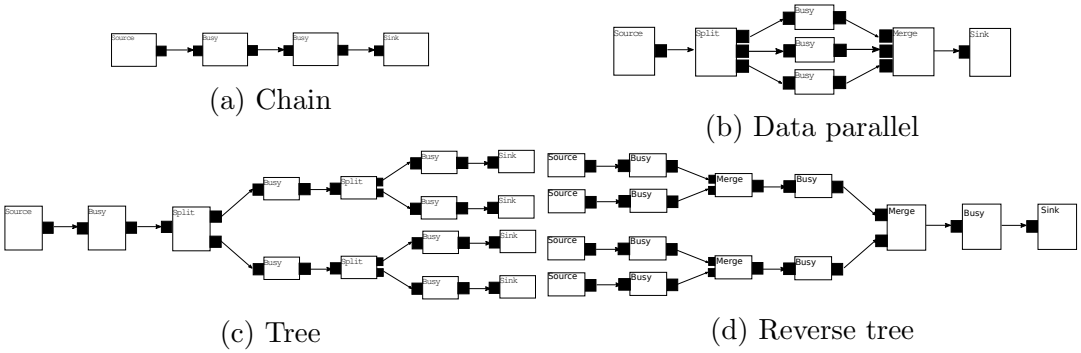
Experiments

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

All of our experiments were performed on a host with two 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. For this purpose, we generate parameterized topologies, which include *chain*, *data parallel*, *tree* and



(a) Chain (b) Data parallel

(c) Tree (d) Reverse tree

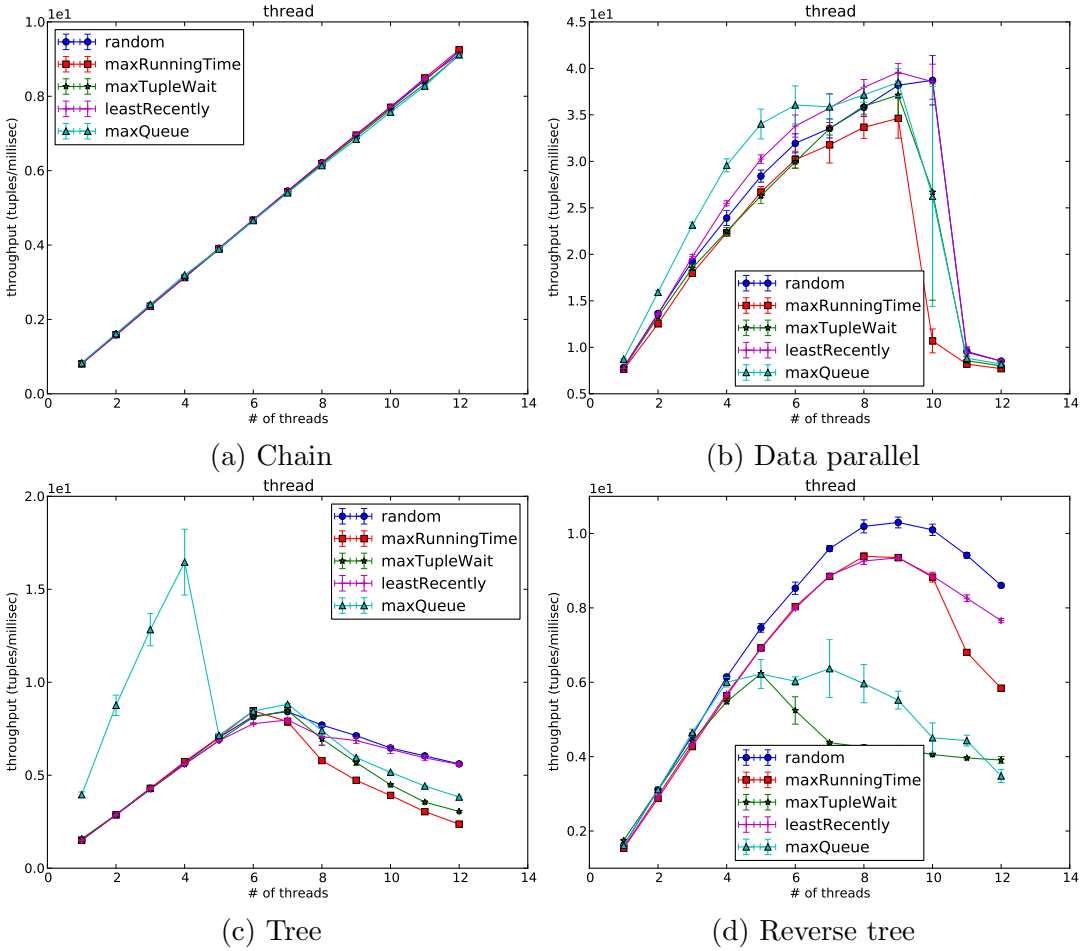
Figure 5.1: Application topologies.

reverse tree topologies. Structures of these topologies are shown in Figure 5.1. 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.

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 and 100 microseconds per tuple.

Number of threads: In our first experiment, we show the effect of the number of threads on 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 5.2a and 5.3a. Throughput we obtain from different scheduler plug-in is 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 5.2b and 5.3b 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, depending on the scheduler plug-in used. The reason



(a) Chain

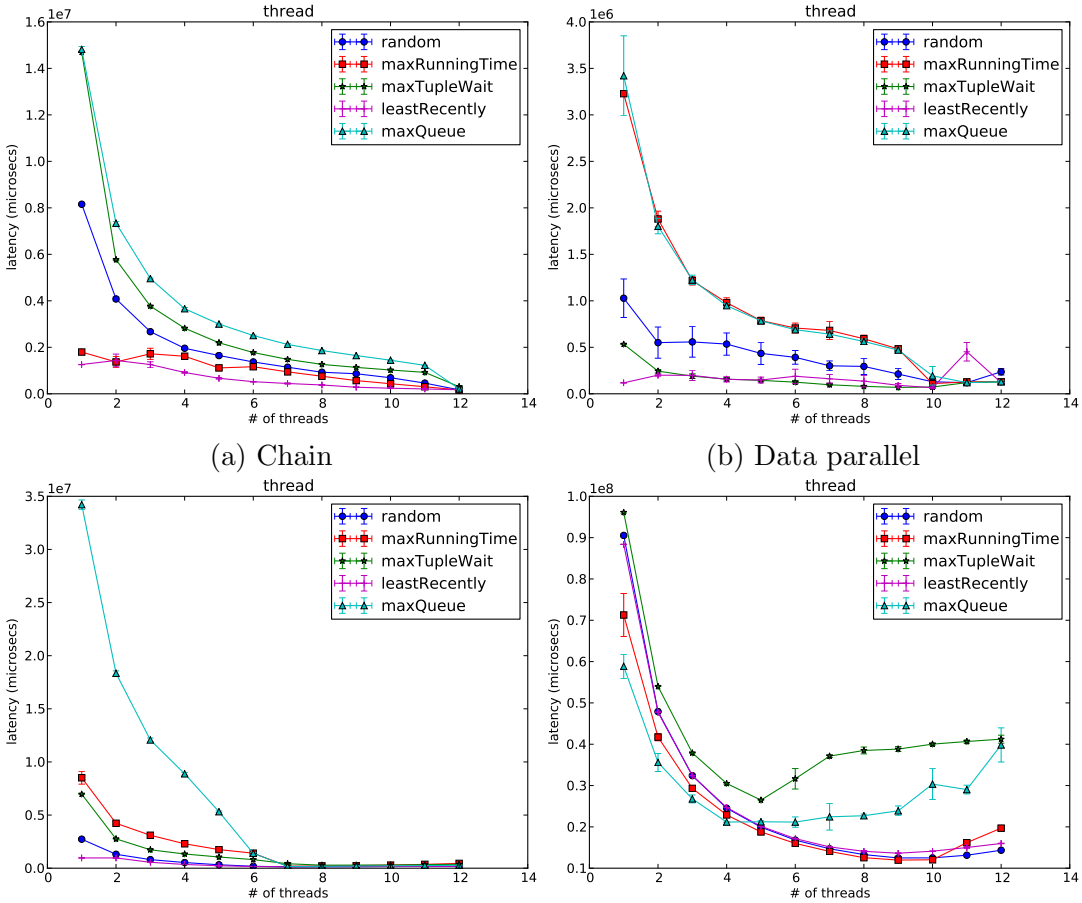
(b) Data parallel

(c) Tree

(d) Reverse tree

Figure 5.2: # of threads vs. throughput

is that the merge operator eventually becomes a bottleneck, since it is sequential. Having more threads than actually needed makes the problem worse, due to the scheduling overhead. In particular, 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 ready state after little space opens up in their downstream ports. However, once these operators resume execution, they would quickly move into 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 the effectiveness of our adaptation module in Section 5.2. From these experiments, we also observe that *MaxQueue* provides slightly higher performance compared to other alternatives,



(a) Chain (b) Data parallel

(c) Tree (d) Reverse tree

Figure 5.3: # of threads vs. latency

but at the cost of significantly increased latency, especially for small number of threads. *MaxTupleWait* and *leastRecently* plug-ins provide the lowest latencies.

Figures 5.2c and 5.3c 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 throughout 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 randomly across the downstream ports. Concretely, if the input rate for a 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

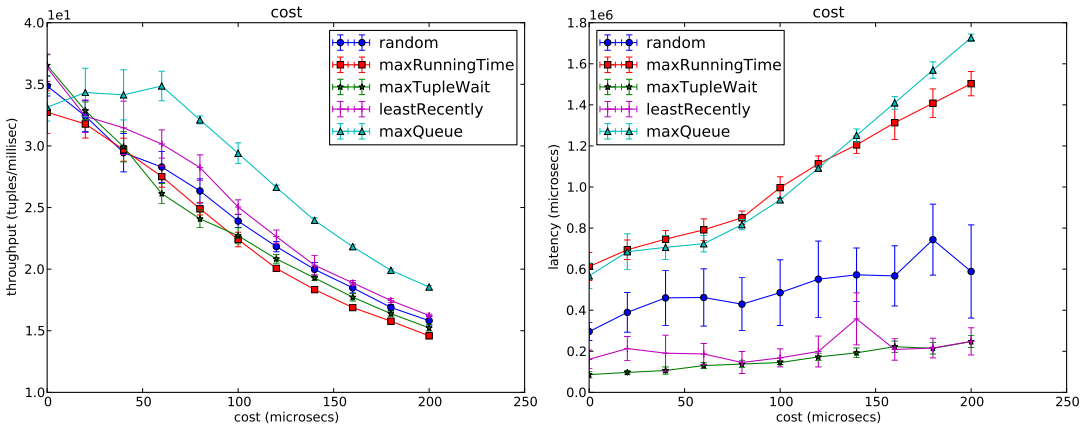
bottlenecks. This experiment again shows that to obtain high thread utilization and high throughput, bottleneck operators should be detected and resolved. The *MaxQueue* plug-in provides higher throughput compared to other alternatives, but only up to 4 threads, reaching as high as 3 times. However, this comes at the cost of increased latency, as high as 3.5 times. Lowest latency is again provided by the *LeastRecently* plug-in.

Figures 5.2d and 5.3d show the effect of the number of threads on throughput and latency, respectively, for the reverse tree topology. Results are similar to data parallel and tree, in which throughput increases up to a certain value of number of threads and then start decreasing. It is surprising that *Random* plug-in provides the best throughput (10% higher than other alternatives). While *MaxQueue* has shows solid performance for all other topologies with respect to throughput, it performs poorly for the reverse topology. In particular, the highest throughout it could reach 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.

We summarize our observations 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, 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, 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.



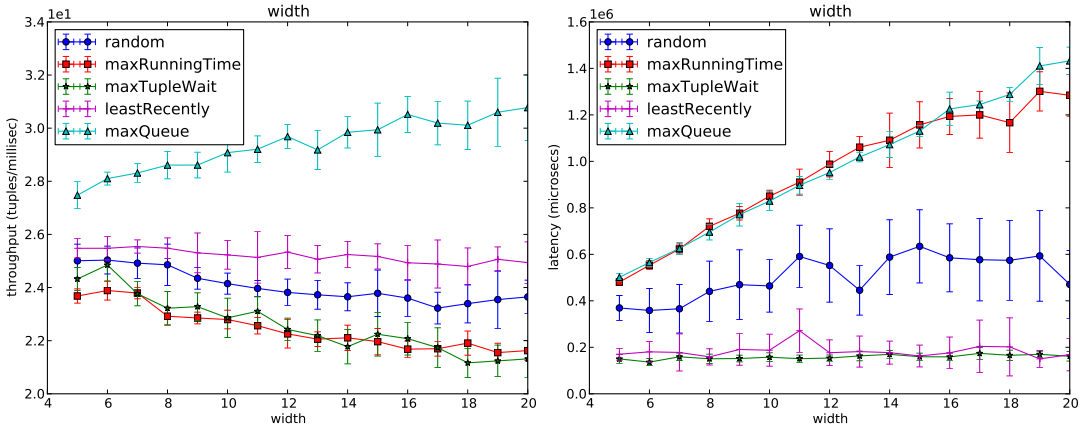
(a) Cost vs. throughput

(b) Cost vs. latency

Figure 5.4: Data parallel cost experiments

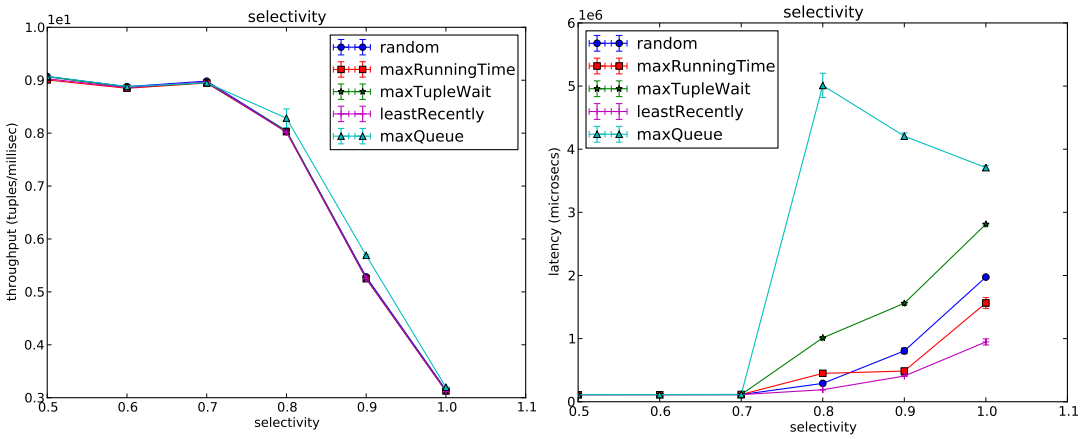
We fix the number of threads to 4. Figure 5.4a shows that throughput decreases as we increase the cost of the busy operators and the decrease is throughput is linear in the increase in operator cost. Figure 5.4b shows that latency increases as we increase the busy operator cost and again we see a mostly linear trend. The only exception is *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, it 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 the throughput and latency, for the data parallel topology. Figure 5.5a shows that for most of the scheduler plug-ins throughput does not change significantly, since the number of 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 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: increased number of data parallel operators result in smaller input queue sizes for them and this in turn increases the amount of scheduling time that the merger gets. Figure 5.5b shows the effect of the number of data parallel operators on latency. We observe that *MaxQueue* and *MaxRunningTime* has linearly increasing latencies, whereas



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

Figure 5.5: Data parallel application size experiments

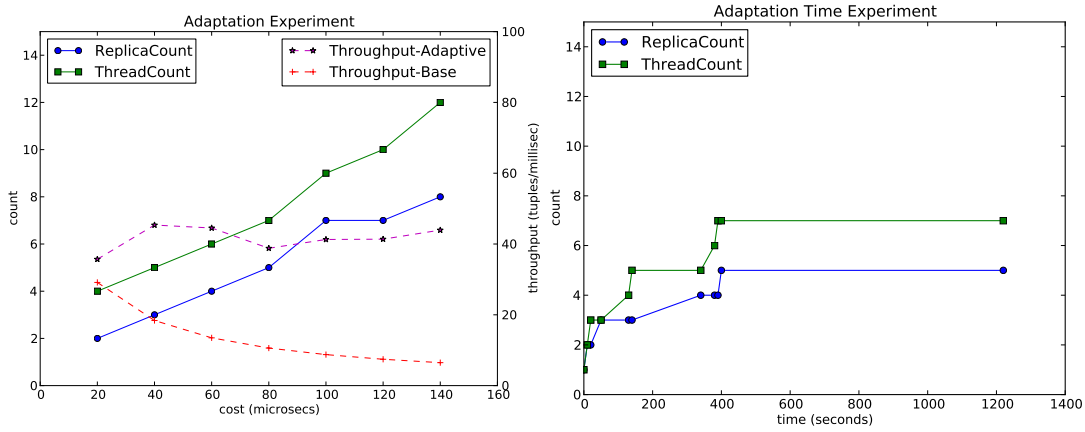


(a) Selectivity vs. throughput (b) Selectivity vs. latency

Figure 5.6: Chain selectivity experiments

other plug-ins show more stable results in terms of latency.

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 tuples will be generated by operator per tuple, thus it plays a role on system workload. As shown in Figure 5.6a, throughput decreases and as shown in Figure 5.6b, latency increases as we increase operator selectivity.



(a) Cost vs. thread count and throughput

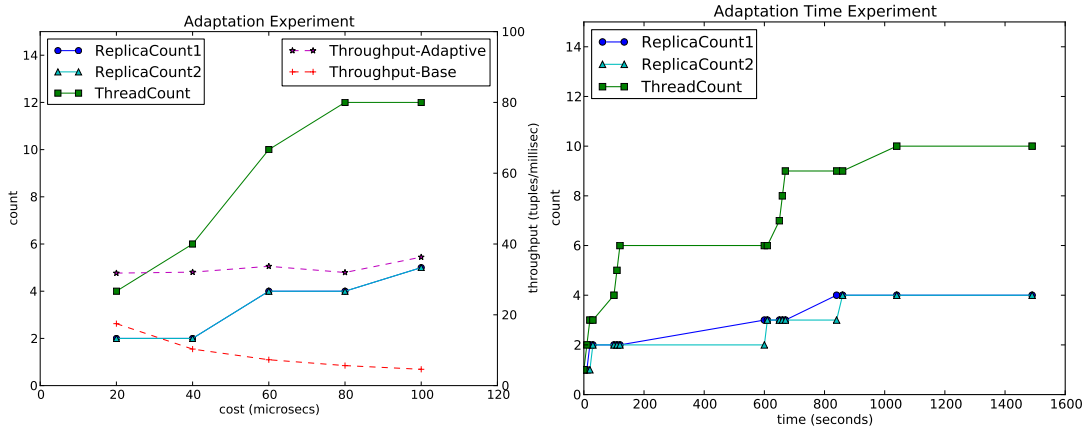
(b) Time vs. thread count

Figure 5.7: Adaptation 1 busy experiment

5.2 Adaptation Experiments

In this section, we look at the performance of elastic parallelization module of C-Stream. First, we perform our experiments using the chain topology, setting maximum number of threads to 12. The chain topology with adaptation is similar to data parallel topology, but the number of replicas for the data parallel operator is adjusted automatically. Furthermore, with adaption, there could be multiple data parallel sections. We compare throughput values obtained from elastic scaling against the throughput values of single thread, single replica scenario. Results show that, while we increase the busy operator costs, throughput remains the same, as adaptation module adjusts the number of active threads and operator replicas accordingly. Second, we perform an experiment on data parallel topology. We compare throughput values obtained from elastic scaling against the case that number of threads is set manually. Results show that, depending on the threshold values, adaptation module adjusts the number of threads and we obtain the maximum throughput.For all the experiments in this section, maximum number of threads is set to 12. Scheduler plug-in is *RandomScheduling*. τ (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. Figure 5.7a shows that,



(a) Cost vs. thread count and throughput

(b) Time vs. thread count

Figure 5.8: Adaptation 2 busy experiment

the number of threads and replicas of busy operator both increase as the cost of the busy operator is increased. Furthermore, while throughput value decreases dramatically in the single thread single replica scenario, adaptation module of C-Stream prevents that, and throughput values remain stable as the operator cost increases. C-Stream achieves this by increasing the number of operator replicas and thread.

Figure 5.7b plots the number of operator replicas and the thread count, as a function of time, for the operator cost of 80 microseconds. It shows that number of threads and replicas increase and eventually stabilize. In this particular case, the stabilization happens at 5 replicas and 7 threads.

In the second experiment, we use 2 busy operators of same cost. Figure 5.8a shows that, together with number of active threads, the number of replicas for each of the busy operators increase as the cost of the busy operators increases. Also, it shows that, while throughput value decreases in the base case, C-Stream adaptation module maintains a stable throughput.

Figure 5.8b plots the number of operator replicas and the thread count, as a function of time, for the operator cost of 60 microseconds. It shows that adaptation module resolves the bottleneck and increments the replica count for one of the busy operators first, and increments the replica count of the other busy

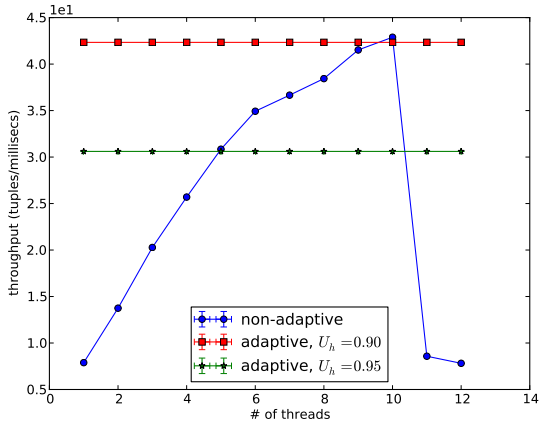


Figure 5.9: Adaptation data parallel experiment

operator after that, and this pattern continues until there is no congestion in the flow. The congestion goes away when both busy operators reach at 4 replicas. The total number of threads stabilizes at 10 treads.

In the third adaptation experiment, we use data parallel topology of 12 busy operators. *High threshold* U_h is set to 0.95 and 0.90 in this experiments, other threshold values remain the same. Figure 5.9 plots the throughput attained as a function of the number of threads for the case when the adaptation module is disabled, and the final throughput achieved via the adaptation module for different high thresholds. Figure 5.9 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.

Chapter 6

Related Work

Stream processing has been an active area of research and development over the last decade. Systems such as STREAM [13], Borealis [14], TelegraphCQ [15], IBM InfoSphere Streams [16], and StreamBase [17] are examples of academic and industrial stream processing middleware, and in many cases these systems provide declarative languages for developing streaming applications as well. StreamIt [18], Aspen [19] and SPL [5] are domain specific programming languages for high-level stream programming, which shield users from the complexity of distributed systems. There are also open-source stream processing platforms such as Storm [9], Apache S4 [10], and Apache Samza [20].

Storm [9], Apache S4 [10], Apache Samza [20], SPL [5], 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 multi-port operators requires additional effort to provide port synchronization and to handle back-pressure resulting from the difference incoming stream rates to the input ports. In C-Stream, operators have their own control loop, and tuple access is orchestrated via our data availability API. To manage operator termination, punctuations are used in InfoSphere Streams [11]. In C-Stream, termination is handled by our base scheduler, separating the termination control from operator's tuple execution control loop. This

feature further simplifies the operator development in C-Stream.

Scheduling relies on operator grouping in SPADE [21], in which 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 flexible. In their on Aurora, Carney et al. form superboxes fpr scheduling, which is a sequence of operators that are executed as an atomic group [12]. However, no results are given on throughput scalability with increasing number of worker threads. In C-Stream, our scheduling relies on assigning 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.

StreamIt compiler auto-parallelizes operators using round robin split to guarantee ordering, but only for stateless operators with static selectivity. In [7] and [8], stateful operators are parallelized by partitioning the state by keys. The same method is can also be found in distributed database systems [22] [23], and it is also the main technique behind the success of batch processing systems such as Map/Reduce [24] and [25]. Brito et al. describes how to apply auto-parallelization using software transactional memory [26], but only if selectivity is 1 and memory is shared.

To exploit the parallelization opportunities on data flow graph, auto-pipelining solution is proposed in [27] for multi-core processors to improve throughput of streaming applications. In the area of auto-parallelization, dynamic multi-threaded concurrency platforms such as Cilk++ [28], OpenMP [29] and x10 [30], decouple expressing a programs 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 system takes advantage of it at run-time.

Kremlin [31] 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.

Elastic parallelization should be supported to achieve higher throughput in stream processing. In Storm [9], data parallelism can be achieved by requesting multiple copies of operators [9]. However, preserving order should be handled by the developer. In S4 [10], creating processing element replicas enables data parallelism. Again, safety is left to developer. In C-Stream's elastic parallelization module, using split/merge operators before/after operator replicas tuple order is preserved.

Elasticity is more involved in distributed streaming environments or in the Cloud. It requires machine to operator mapping (placement), and operator migration after scaling in/out decisions. FUGU [32] is an allocation component for distributed complex event processing systems, and it is able to elastically scale in and out with a varying system load. In [33], auto-scaling techniques are presented on top of FUGU, including local thresholds, global thresholds, and reinforcement learning. StreamMine3G [34] is another elastic stream processing system, supporting both vertical and horizontal scalability. Stormy [35], on the other hand, is an elastic stream processing engine running on the Cloud. As part of the elasticity, migration protocols are proposed for operators in [36] [8]. In [8], Gedik et al. proposes incremental migration protocol relying on consistent hashing. In [36]. Heinze et al. presents their migration algorithm trying to reduce the migration latency, which is based on the operator movement cost estimation. In contrast, C-Stream is a single-node multi-core stream processing system, with a focus on flexible scheduling and elastic streaming execution.

Chapter 7

Conclusion

In this thesis, we present C-Stream — a coroutine based scalable, highly customizable, and elastic stream processing engine. Unlike traditional event based stream processing operators, in C-Stream each operator is implemented as coroutine having its own control loop and each can decide when to perform its computations using data availability API. This property of C-Stream simplifies the development of operator that requires multiport synchronization. Second, we present a customizable scheduler in C-Stream. Base scheduler handles back pressure, provides data availability API, and manages preemption and termination handling. It can also be configured via plug-ins for various demands, such as to obtain high throughput or low latency. Third, our adaptation module adjusts the level of parallelism by detecting bottleneck operators. It increments the replica count until bottleneck is resolved. Adaptation module also measures the thread utilization so that it decides when to increase/decrease number of threads as workload on the system changes. In the experiments, we show the efficacy of our system.

Bibliography

[1] X. J. Zhang, H. Andrade, B. Gedik, R. King, J. Morar, S. Nathan, Y. Park, R. Pavuluri, E. Pring, R. Schnier, P. Selo, M. Spicer, C. Venkatramani, A. Frenkiel, W. D. Pauw, M. Pfiefer, P. Allen, N. Cohen, and K.-L. Wu, "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*, 2009.

[2] P. Zerfos, M. Srivatsa, D. Dennerline, H. Franke, and D. Agrawal, "Platform and applications for massive-scale streaming network analytics," *IBM Journal of Research and Development*, vol. 57, no. 3/4, pp. 11:1 – 11:13, 2013.

[3] D. L. Schales, M. Christodorescu, J. R. Rao, R. Sailer, M. P. Stoecklin, W. Venema, and T. Wang, "Stream computing for large-scale, multi-channel cyber threat analytics," in *IEEE International Conference on Information Reuse and Integration (IRI)*, pp. 8–15, 2014.

[4] M. K. Garg, D.-J. Kim, D. S. Turaga, and B. Prabhakaran, "Multimodal analysis of body sensor network data streams for real-time healthcare," in *Multimedia Information Retrieval*, pp. 469–478, 2010.

[5] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soule, and K.-L. Wu, "Streams processing language: Analyzing big data in motion," *IBM Journal of Research and Development*, vol. 57, no. 3/4, pp. 7:1–7:11, 2013.

[6] M. Hirzel, R. Soule, S. Schneider, B. Gedik, and R. Grimm, "A catalog of streaming optimizations," *ACM Computing Surveys (CSUR)*, vol. 4, no. 46, 2014.

[7] S. Schneider, M. Hirzel, and B. G. an Kun-Lung Wu, "Safe data parallelism for general streaming," *IEEE Transactions on Computers (TC)*, vol. 64, no. 2, pp. 504–517, 2015.

[8] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 25, no. 6, pp. 1447–1463, 2014.

[9] Storm, "Storm project." `http://storm-project.net`. Retrieved June., 2015.

[10] "S4 project." `http://incubator.apache.org/s4`. Retrieved June., 2015.

[11] H. Andarade, B. Gedik, and D. Turaga, *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*, ch. 4 Application development – data flow programming, section 4.3.5 Punctuations. Cambridge University Press, 2014.

[12] D. Carney, U. Çetintemel, A. Rasin, and S. Zdonik, "Operator scheduling in a data stream manager," in *Very Large Data Bases Conference*, 2003.

[13] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom, "Stream: The stanford stream data manager," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, (New York, NY, USA), pp. 665–665, ACM, 2003.

[14] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "The design of the borealis stream processing engine," in *In CIDR*, pp. 277–289, 2005.

[15] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah, "Telegraphcq: Continuous dataflow processing," in *Proceedings of the 2003*

ACM SIGMOD International Conference on Management of Data, SIGMOD '03, (New York, NY, USA), pp. 668–668, ACM, 2003.

[16] B. Gedik and H. Andrade, "A model-based framework for building extensible, high performance stream processing middleware and programming language for ibm infosphere streams," *Software: Practice and Experience Journal, Wiley (SP&E)*, vol. 11, no. 42, 2012.

[17] Storm, "Streambase systems." `www.streambase.com`. Retrieved June., 2015.

[18] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "Streamit: A language for streaming applications," in *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, (London, UK, UK), pp. 179–196, Springer-Verlag, 2002.

[19] G. Upadhyaya, V. S. Pai, and S. P. Midkiff, "Expressing and exploiting concurrency in networked applications in aspen," in *In Proc. of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 13–23, ACM Press, 2007.

[20] "Apache Samza project." `http://incubator.apache.org/samza`. Retrieved June., 2015.

[21] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, "Spade: The system s declarative stream processing engine," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, (New York, NY, USA), pp. 1123–1134, ACM, 2008.

[22] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen, "The gamma database machine project," *IEEE Trans. on Knowl. and Data Eng.*, vol. 2, pp. 44–62, Mar. 1990.

[23] G. Graefe, "Encapsulation of parallelism in the volcano query processing system," in *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, (New York, NY, USA), pp. 102–111, ACM, 1990.

[24] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[25] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, (New York, NY, USA), pp. 59–72, ACM, 2007.

[26] A. Brito, C. Fetzer, H. Sturzrehm, and P. Felber, "Speculative out-of-order event processing with software transaction memory," in *Proceedings of the Second International Conference on Distributed Event-based Systems*, DEBS '08, (New York, NY, USA), pp. 265–275, ACM, 2008.

[27] Y. Tang and B. Gedik, "Autopipelining for data stream processing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 12, pp. 2344–2354, 2013.

[28] Cilk++. `https://cilkplus.org/`. Retrieved June., 2015.

[29] Openmp. `http://openmp.org`. Retrieved June., 2015.

[30] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, (New York, NY, USA), pp. 519–538, ACM, 2005.

[31] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor, "Kremlin: Rethinking and rebooting gprof for the multicore age," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, (New York, NY, USA), pp. 458–469, ACM, 2011.

[32] T. Heinze, Y. Ji, Y. Pan, F. Josef, G. Zbigniew, and J. C. Fetzer, "Elastic complex event processing under varying query load."

[33] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer, "Auto-scaling techniques for elastic data stream processing," in *Proceedings of the 8th ACM*

International Conference on Distributed Event-Based Systems, DEBS '14, (New York, NY, USA), pp. 318–321, ACM, 2014.

[34] A. Martin, A. Brito, and C. Fetzer, "Scalable and elastic realtime click stream analysis using streammine3g," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, (New York, NY, USA), pp. 198–205, ACM, 2014.

[35] S. Loesing, M. Hentschel, T. Kraska, and D. Kossmann, "Stormy: An elastic and highly available streaming service in the cloud," in *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, EDBT-ICDT '12, (New York, NY, USA), pp. 55–60, ACM, 2012.

[36] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Latency-aware elastic scaling for distributed data stream processing systems," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, (New York, NY, USA), pp. 13–22, ACM, 2014.