Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

# Operator latency in a Complex Event Processing application

Simon Hagenmayer

**Course of Study:** Informatik

**Examiner:** Prof. Dr. Dr. Kurt Rothermel

**Supervisor:** Henriette Röger, M.Sc.

**Commenced:** 2017-08-15

**Completed:** 2018-02-15

**CR-Classification:** I.7.2

## Abstract

Complex Event Processing often comes with an enormous amount of event data that needs to be processed. Hence, parallelization plays a significant role in handling high workload situations. The cost of an application however is often defined by the amount of used resources, like in Cloud computing, where the pay-as-you-go model applies. Still, one wants to have a working system that can handle traffic peaks within a given latency bound, so the resources-to-latency-proportion needs to be optimized. Previous work mostly focused on studying complex operator types in specific environments. In this thesis however, we want to get a general view, how parallelization degrees and types influence our CEP system, to be able to estimate what costs could arise. Therefore, a CEP application was created that simulates different system conditions with respect to workload, operator processing time and others, in order to test and analyze the latency properties of a wait operator. This work provides an overview over latency behavior of operators in an example Complex Event Processing application, which can provide a basis for future work in creating an optimized system, that not only keeps a certain latency threshold but also minimizes the costs and resources needed to achieve this goal.

# Contents

# List of Figures

# 1 Introduction

Complex Event Processing is used to extract information from continuous streams of data. The vision of the Internet of Things contributed to the problem, that the amount of input information to handle can be quite overwhelming. Furthermore, we often want a specific delay threshhold at which our data needs to be processed. The solution to this is parallelization. If we duplicate our working operator, we are able to treat more information at the same time. However, more instances automatically come together with more ressource utilization, representing additional costs. Minimizing these costs can be quite beneficial, as cloud ecosystems like Amazon Relational Database Services [1] or Google Cloud [5] often come with the pay-as-you-go model. It is therefore important to investigate, how one can reach this objective while keeping the system under a certain latency threshhold.

## 1.1 Overview

To be able to reduce the cost of our system while in the mean time maintaining a certain threshhold, we need to examine, how our system reacts on different types and degrees of parallelization. The problem we tackled in this thesis is exactly that: we wanted to know how well our CEP system can deal with the two most important parallelization types in general, with respect to varying input parameters. Furthermore, we desired a model, that could predict our operator latency values, which would give us the possibility to minimize parallelization costs.
We started by studying a simple operator under different conditions using data parallelization. For accessing our test results, we compared them to an ideal model derived from queueing theory formulas. We also took a look at the stability points of the system. Afterwards we changed the properties of the input parameters to have increased variance. Using our measurements, we then created an approximation model by combining estimations for the individual parts of our system. These were realised using linear regression linked with linear or exponential interpolation. The same steps were repeated for pipelining, which we implemented into our system.
Results show that our system realizes data parallelization really well and comes close to an ideal model. Moreover, our approximation model reaches a good level of fit of our test data. In contrast to that, pipelining shows large latency values for high parallelization degrees, leading to a great discrepancy to an ideal model. The same holds for our pipelining approximation model.

## 1.2 Related Work

Our work can be placed in the section of system profiling. Profiling is a wide term and can be realised in a lot of different ways. We therefore want to present examples that adopt a similar approach as our work or fall into the region of Complex Event Processing in general. Mulitple works deal with the automation of pipelining and data-parallelization with respect to surrounding factors. In their work, Kombi et al. [15] let a system adjust their throughput to the data arrival rate. This is done by estimating the operator workload for the near future and scaling the system automatically on basis of its prediction. Afterwards they implemented their model into apache storm, leading to less CPU and memory ressources used in the test cases than without it. Liu et al. [18] pursued a similar goal. In their paper, they provide a profiler for streaming applications. By comparing distinct operators, they come up with a "stepwise approach" for twitter kestrel. This is done by evaluating three operator configurations and making a conclusion which one works best for certain situations. Two of these configurations are pipelining and data parallelization, while a third one represents a star formation. In this star formation, events flow from multiple operator instances to a single one, which then distributes its outputs again to several nodes. Gad et al. [7] present a work in progress open source software for local parallelization. In their paper they scale the problem of parallelization down from distributed systems to a single core machine. They show that given three distinct scenarios, their profiling approach is able to enhance the throughput of the system by $\approx 40\%$ in comparison to a contemporary profiling method.
Kiefer et al. [13] followed another interesting approach. They did not investigate how cloud computing can increases the cost of an application, but rather how ressource sharing in cloud computing can also lead to a difference in performance. In their tests, they compared two "commercial cloud databases" and measured the time needed for executing four different queries. They get the result, that the type of ressource distribution correlates to the query response time.

## 1.3 Contributions

The contributions of this thesis can be split up into four parts:

- Our first contribution is the derivation of an ideal data parallelization model arising from basic queueing theory formulas. This can be used to test the performance of our system. It can further serve as reference point for the comparison of different systems.
- Our second contribution is the derviation of an approximation model for the data-paralellized system, regarding poisson distributed mean inter-arrival times and processing times.

- Our third contribution is the implementation of pipelining into our CEP application. We further derive an ideal pipelining model for evaluating our implementation. This can also act as tool for system correlations.
- Our fourth contribution is the approach of an approximation model for (M/D/c) pipelining, given our underlying CEP application. We do not present a finished end product here, due to unexpected results of our pipelining tests.

## 1.4 Outline

In chapter 2, we explain Complex Event Processing in general, how it evolved and which components define it. These are introduced in their own subsections. We further present the two most important types of parallelization. Chapter 3 devotes itself to the background subjects used in this work, seperated by the definiton of variables, processes and theoretical models. At the start of chapter 4, we present our setup for the operator tests, split up into a hardware and a software part. The latency experiments themselves are written attached to that. Afterwards we discuss the results and what concludes from them (5). Chapter 6 provides an outlook for possible future improvements. In the end, we draw a whole conclusion that summarizes the experiences of the thesis.

# 2 Complex Event Processing

In our thesis we do not discuss the creation of data in Complex Event Processing in detail, but rather take our given CEP application as test object. Therefore, a lot of our work founds on queueing theory and not on CEP specific formulas or arguments. Still, we want to introduce in this chapter the framework of our thesis, and give a short overview over the development of CEP.

### 2.0.1 Introduction

In the early stages of *database management systems*, data stored in tables or other available data structures was considered static and structured. Tables could be created or dropped and new entries could be inserted or deleted via command queries. These systems were therefore well suited for the management of huge ressources, but also had some disadvantages to them: Data needed to be stored before it could be further treated, and the time of storage of an event was unconnected with its true arrival time [25, p. 9]. The latter applied, because data was only processed when the user explicitly told the system to do so. When regarding fitting examples, these disadvantages would not matter that much: An account management system would just need the admin to create and delete accounts, while changes in the account settings could be made by the employees themselves.
As nonstatic data tended to occur in more and more use cases, research in this field yielded to the developement of multiple systems, that approached the task differently. The *datastream management systems* [3] could be seen as an extension of the DBMS model. Carney et al. [4] saw these systems as “Human-passive, DSMS active”, as the models got their input from data sources and the only human participation was inserting queries, that would once started filter and process the incoming data flow. In contrast to that the DBMS model was “Human-active, DBMS passive”, as it received its queries from human input, and did not “work” at its own.
The other important model that evolved was the *complex event processing* model [19]. While DSMSs usually try to adapt to their changing available data, retrieving more abstract correlations between the information is often not realizable. This is where CEP models come in handy.

## 2.1 Event

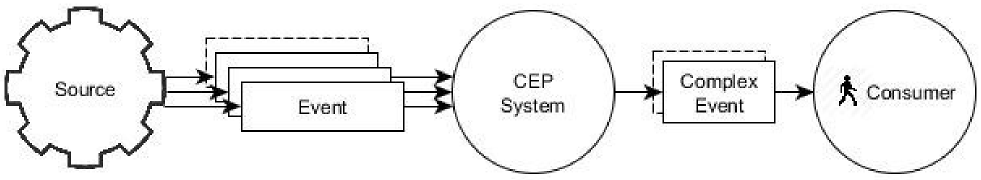

**Figure 2.1:** Generation of complex events

CEP models are based on so called *events*. These can be classified into two categories: The “low-level events” (Cugola) or also *simple events* present incidents in the world, that need to be put together and processed to lead to more abstract information, the so called *complex events* or also “high-level events”. The simple events are produced by the sources of the system and can vary in their type (stock data, sensor feedback, tweets [16]). The subesequently from the system processed complex events are then consumed by sinks or consumers, which can react to the higher-level information.
Each event has the same recurring attributes [20] [6]: Its content is distributed into tupels of two entries (attribute, value), it has a certain type and a timestamp. An *event stream* $\mathbb{E}$ is a perhaps infinite (linear) sequence of events [9] [6], denoted as a tupel $(e_0, e_1, ...)$. Usually, events are ordered by time. This is done by marking every event with a sequence number (id), that differentiates it from other events in a stream, and by the corresponding source stream id to distinguish between event streams from different sources.
An example could be a safety CEP system in a car: Here, data sources like the speedometer and the distance sensor would measure atomic data in small time intervals and send them to the system. When your brake path gets nearly as long as your distance to the object your aiming at, a complex event “collision incoming” is produced, that is then directed to the consumer, in this case the car brake system, which will automatically reduce the velocity of the automobile.

## 2.2 CEP System

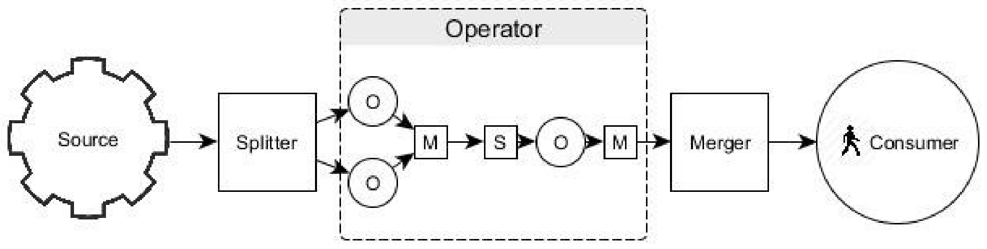

**Figure 2.2:** Example of a CEP system containing both data parallelization and pipelining.

In our thesis we used a Complex Event Processing framework developed by Ruben Mayer. Among his last works with the system include a stream partitioning method for low latency event detection [20] and the creation of a batch scheduling controller, tackling the problem of overlapping partitions between parallelized operator instances [22]. Furthermore he investigated the influence of communication overhead on a stateful operator using overlapping windows [21].

The system consists of five different components, that can be connected through communication ports.

### 2.2.1 Source

The generation of events is done in so called sources. Here, the events' type and content is set, as well as the time between successive events. Examples of complex event processing sources are including but not limited to physical sensors or statistical data (stock data, tweets). There will usually be more than one source in a bigger complex event processing system, but for abstraction reasons we will focus on a single source in this thesis.

### 2.2.2 Splitter

The splitter is the connecting link between the source and the operating system. It receives events from the source and sends them to tied instances. The receiver is thereby determined by the scheduling strategy of the splitter.
When using pipelining (2.3) we also work with splitters placed inside the operator. These are put together with mergers around the operator instances, to be able to use both pipelining and data parallelization simultaneously .

### 2.2.3 Operator

The core of the system is the operator. Here, events flowing from the source are processed and transmuted into messages or signals that can be understood by the consumer. We differentiate between the operator itself, which represents the whole complex between the first splitter and the last merger, and operator instances. Second are duplicated or permutated instances of the operator that occur in parallelization. In data parallelization (see 2.3) for example, the operator is duplicated as often as the parallelization degree. In pipelining on the other hand, parts of the operator production process are divided into different operator instances, that enable a faster run through the operator pipeline.

### 2.2.4 Merger

The merger collects the data produced by the operator or multiple instances of it. With increasing data parallelization degree this gets more costly as various operator sources will connect to the merger.
Mergers are, like splitters, also part of the operating system when using pipeling, where they are used to collect intermediate events that are then further processed from other instances. These modified mergers do not send the events to consumers, but to the next connected splitter.

### 2.2.5 Consumer

What is in the end really done with the filtered complex events determine the consumers plugged in at the end of the pipeline. These are not part of the complex event processing production but rather react automized on different occuring events, like ringing a fire alarm or creating a pop up message.

## 2.3 Parallelization

Figure 2.2 shows us an example of an arrangement of our previous introduced components. Within the operator we find the two most common types of parallelization:

- **Data parallelization:** An easy method to improve the latency of the operator is to create multiple operator instances. This way the work can be split and no instance is overloaded or crowded. For a high parallelization degree though, the problem can arise that the splitter has a lot of work to do and will slow down the system. In the figure, the first chain of the operator represents a data parallelization of degree two with two operator instances (O).

- **Pipelining:** The alternative to data parallelization is pipelining. In pipelining, the complex event production process is split and put into several chained operators. That way, events do not need to be fully processed in one step but are converted to complex events until the end of the pipe. We put splitters and mergers around each part of the pipeline to be able to combine it with data parallelization. The figure represents a pipeling degree of two, the first part being data parallelized (S-OO-M) and the second a single operator instance (S-O-M).

One could argue what pipelining has to do with parallelization. In fact, if we speak of pipelining as parallelization method, we refer to the parallel use of CPU ressources rather than their topology context, which is sequential.

# 3 Background

In this section, we explain the basics for our test scenarios. We define variables that we will analyze and measure, as well as equations for the computation of context between the variables. Furthermore we describe our notation used for certain test cases and methods for the analysis and approximation of our end results.

## 3.1 Variables

- The *inter-arrival time* $a$ is the amount of time that passes between the production of events at the source. We use it as our parameter to change in our application tests. Inter-arrival times in our scenarios are either of constant value, i.e. each subsequent event follows at the same time as events before, or exponentially distributed. The *arrival rate* $\lambda = \frac{1}{a}$ is the amount of events flowing from the source to the system at a given time interval, and is just the inverse of the inter-arrival time.

- The *processing time* or *service time* $p$ describes the time a part of the system needs to process a single event. Parts of the system are the splitters, operator instances or mergers. Their service times are marked respectively as $p_s^e$, $p_i^e$ and $p_m^e$, with e being the dedicated event. The *mean processing time* can be computed as average of all event processing times: $\frac{p_s^{e1}+p_s^{e2}+...+p_s^{eN}}{N}$, with $N$ being the amount of events in total. Calculating the (mean) processing time is important for our tests, as it gives information about the time needed for an event to flow through the system. Its inverse, the *service rate* $\mu = \frac{1}{p}$ specifies, how much events can be served at a given time step.

- The *utilization* $\rho$ of a queueing system represents the average amount of time the system is busy processing events. It can be computed as the quotient of the arrival rate of events and the system service rate $\rho = \frac{\lambda}{\mu}$. A queueing system is considered *stable* for $\rho < 1$ and *unstable* for $\rho > 1$. The point where $\rho = 1$ is called *stability point*. In our test section we try to analyze how our measured stability points will differ from ideal computed ones, to recognize for which inter-arrival times our system is stable / unstable.

- The *operator latency* $t_{op}^{e}$ is the delay, that occurs between the arrival of an event $e$ at the system, and its exit, i.e. the arrival at the last merger. Derived from that, we can calculate the *mean operator latency* $T_{op}$ as the average operator latency of all events. In short $T_{op} = \frac{t_{op}^{e1}+t_{op}^{e2}+...+t_{op}^{eN}}{N}$, with N being the amount of events running through our system, in our case always 100. The mean operator latency is the variable we want to analyze and model in this thesis depending on parameters like the inter-arrival rate or the parallelization degree. Another variable is often used in context of latency: the *throughput* is the amount of input the operator or system can handle at once. We will not consider this in our thesis though.

- The *queue length* of a splitter ($l_s^e$) / operator instance ($l_i^e$) / merger ($l_m^e$) is the queue size of the input queue when the event $e$ arrives at one of these parts of the system. The mean queue length $L$ can then be computed by averaging over the measured queue lengths at each event appearance. For the splitter this is for example: $L_s = \frac{l_s^{e1}+l_s^{e1}+...+l_s^{eN}}{N}$, with $N = 100$. We analyze queue lenghts in our tests as they can represent large delay sources for our operator latency.

## 3.2 Distributions / Queuing system

- When running our tests we used externally saved *distribution samples* for the inter-arrival and service times. They were created in an own java class, supporting the java.lang types AbstractRealDistribution and AbstractIntegerDistribution. We wanted samples, that would represent our underlying distribution well given a specific mean. Due to the small amount of events per run though, namely 100, the mean varied a lot when creating individual distribution samples. Indeed one can show that the variance to the original mean can be computed as $\frac{\sigma}{\sqrt{c}}$ [23], with $\sigma$ being the variance of the original distribution. We therefore discarded samples with high variance to our desired mean until we got a good fitting one which we saved on the disk.

- A *queueing system* is a model containing some kind of queueing or waiting. In their book, Gross et al. [8] split systems with respect to six defining characteristics: The arrival time and service time distributions, the “queue discipline” which specifies the type of the used queue (First-in-first-out, First-in-last-out, . . . ) and the “system capacity”, indicating the feasible queue size. They then differentiate between the “number of service channels” which in our case specifies the parallelization degree and the “number of service stages”, that we will describe as pipelining degree. In this thesis, we will confine ourselves on a First-in-first-out queue with infinite system capacity.
  An often used notation for queueing systems is the so called *Kendall notation*. In his paper, Kendall [12] subdivides the processes into three properties: (i) The “input” denotes if the arrival rate of events is deterministic (D) or poisson distributed (M). For the “service-mechanism”, which maps our service rate, the same holds. The

"queue-discipline" specifies the amount of servers that process events.
In our thesis we will use the Kendall notation, looking at two different queueing models in special: A (D/D/c) model with constant inter-arrival and processing times, as well as different stages of parallelization. An example for this scenario could be stock data arriving at certain specific time steps at the market, updating statistics for a certain stock. We explain later on that we can represent pipelining in this case by conjugating multiple (D/D/1) queueing systems.
The second model we will use is the (M/M/c) model with inter-arrival times following a poisson process, exponentially distributed service times and a parallelization degree of c. The most intuitive example for this kind of model is a shopping queue with c cash points being able to serve multiple customers in parallel. For pipelining, we restrict the processing time to be constant (M/D/c), as distributed processing times would be hard to realize (see 4.6).

- A *poisson process* is a widely utilized stochastic process to model the arrival of events at a system [17, p. 19]. One can show, that in a poisson process, the inter-arrival times arising between two events are exponentially distributed. To get an exponentially distributed random variable, we can make use of the algorithm provided by Donald Knuth in his second volume of *The art of computer programming*: [14]

**Algorithmus 3.1** Poisson process algorithm

```
procedure GetRandomPoisson($\mu$)
    $L = e^{-\mu}, k = 0, p = 1$
    repeat
        $p = p \cdot rand()$
        $k$++
    until $p > L$
    return $k - 1$
end procedure
```

We implemented this method in python and saved the required poisson processes for our test runs in log files. Similar to the distributions, we created processes and payed attention that the mean did not diverge too much from our desired mean. Otherwise we would discard the process and try again.

## 3.3 Theoretical methods

- To analyze our results, we make use of three statistical terms (see [11]). The *mean absolute error* (MAE) is the absolute deviation of observed values $\hat{Y}$ to the true values $Y$:

$$MAE(\hat{Y}, Y) = \frac{1}{n} \sum_{i=1}^{N} |y_i - \hat{y}_i|$$

To weight great discrepancies between values of $\hat{Y}$ and $Y$ more, we also regard the *mean squared error* between the data sets given as

$$MSE(\hat{Y}, Y) = \frac{1}{n}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

The influence of a measured difference of the data sets is also dependent on the values of $\hat{Y}$ and $Y$ themselves. A deviation of 5 ms can be huge in nanoseconds range, but negligible in hours range. We therefore set the *mean absolute percentage error* (MAPE) as

$$MAPE(\hat{Y}, Y) = \frac{1}{n}\sum_{i=1}^{N}|\frac{y_i - \hat{y}_i}{y_i}|$$

- *Linear regression* is used to approximate a specific system variable by another parameter or parameter set. Therefore, we represent our estimated output values $\hat{Y}$ as linear model depending on the variables $X = (X_1, X_2, \ldots, X_p)$

  $$\hat{Y} = \hat{\beta}_0 + \sum_{i=i}^{p} X_i\hat{\beta}_i$$

  with $\beta$ being our coefficients for the sum. If we increase the vector $X$ by the value 1 we can shorten this equation to

  $$\hat{Y} = X^T\hat{\beta}$$

- To do linear regression, we need a method to fit the linear model to our real data $Y$. We do this by the method of *least squares*, which means that we want to minimize the mean-squared error (here also called residue) between $Y$ and $\hat{Y}$.

  $$MSE(\hat{Y}, X^T\beta) = \frac{1}{n}\sum_{i=1}^{N}(y_i - x_i^T\beta)^2$$

  We can calculate the minimum by setting the derivative to 0 and solving the equation for our betas. We can then directly calculate them as

  $$\hat{\beta} = (X^TX)^{-1}X^Ty$$

  We will use linear regression and the least squares method in our approximation models of our data-parallelized or pipelined system. To retrieve our optimal $\beta$s, we used an online website supporting linear regression[1]. Beforehand, we compared some sample regression sets by hand to verify its propriety.

[1] http://www.xuru.org/rt/lr.asp

# 4 Application tests

## 4.1 Test setup

In this section we explain how we built up our operator latency tests for data parallelization and pipelining, as well as going into detail about the instantiation of all the parts of the system. We further give an overview over our folder structure that we used for saving and plotting the data.

### 4.1.1 Hardware

Tests were executed on three different machines on an Open stack server of the institute:

- RAM: 8 GB, VCPUs: 4, Disk: 10 GB
- RAM: 32 GB, VCPUs: 4, Disk: 10 GB
- RAM: 32 GB, VCPus: 24, Disk: 5 GB

For data parallelization tests, we confined ourselves to the first machine. Pipelining tests were run on all three machines to compare the behavior for high pipelining degrees and recognize the benefit of more RAM or extra VCPUs.

### 4.1.2 Data parallelization

We executed our operator latency tests with different shell scripts, that set up our system depending on our current test case. For data parallelization, we first instantiated the splitter, which created a socket at port 48458 for sources or mergers to connect. Other parameters, that are used in Mayers work [20] [22] were set to have no influence on the splitter. These include the four cases after “48458”, as well as the last 5 parameters of the execution call. The “&” at the end of the line signals that the jar-file is run in parallel with the next lines. The scheduling strategy was set to round-robin, which is marked in the program call as “rr”. The trailing zeros describe scheduling parameters which are in the round-robin case negligible.

**Listing 4.1** Setup for our data parallelization tests, in this example with constant inter-arrival and processing times.

```
java -jar Splitter.jar 48458 -1 tu 0 0 rr 0 0 0 0 0 0 none 100 100 100 100 &
java -jar Merger.jar 48459 null 0 &
for ((d=1;d<=$b;d++)); do
      java -jar Instance.jar localhost 48458 constant_wait_operator 90 10 false 100 100 100
           localhost 48459 &
done
java -jar Source.jar localhost 48458 false 5 Distributions/distribution_const_$a.txt
     Distributions/distribution_const_90.txt $a 100 true

#Create own directory for the experiment, move log files there
mkdir -p master_wait_operator_data_sh/D90n_test/stage_$c/pdegree_$b"_iatime_"$a

#Move all log files
mv *.txt master_wait_operator_data_sh/D90n_test/stage_$c/pdegree_$b"_iatime_"$a
```

Afterwards the merger was started at port $48459$, also listening for operator instance connections. We implemented two additional arguments for the merger, to be able to send elements to another splitter for pipelining. The first argument represents hereby the ip of the next splitter, the second argument its port. As we did not need this for data parallelization, we set it to “null” and “$0$”. With these parameters defined, the merger can also realise if it is the last merger in a chain. That is why we bound the operator latency measurement to the splitter ip being zero, as we only wanted to measure it at our last merger of the system.

The program now instantiates operator instances depending on the given parallelization degree $p$. These are connecting to the splitter with ip “localhost” and port “$48458$” (first two arguments) and the merger with ip “localhost” and port “$48459$”. Our used operator type is a simple wait operator which description is given as “constant_wait_operator”. The behavior of the operator is easy to describe: for each event $e$ arriving, it extracts the processing time $p_i^e$ and waits via “Thread.sleep($p_i^e$)”, simulating the processing of the event. The next two arguments describe operator specific variables. They can be ignored in our case as the processing time (or wait time) of the operator is given by the events that arrive at its port. The remaining parameters are needed for CPU measurement and monitoring, which are not used in our case.

In the end the source is started, passing the events to the system. The first two parameters of the function call are the splitter ip (“localhost”) and port (“$48458$”) it connects to. The third argument set to “false” indicates that we want to produce events, with the fourth parameter indicating which type of events we want to construct. In our case we want events of type “SimpleValue”, which constitutes events that are given a simple double value: the processing time at the operator instance(s). For the event inter-arrival times and processing times we load our external sample distributions. In the (D/D/n) case these are just files of constant values. As our processing time is of constant $90$ ms, we always load the same file, but for the inter-arrival time we load different ones depending on the value of “$a”, which indicates the inter-arrival times used in the current run. Our amount of events produced is given as penultimate parameter, which is consistently $100$ for all tests. As we are playing a log file, the last argument of the source is not important.

### 4.1.3 Pipelining

**Listing 4.2** Setup for our pipelining tests, in this example with constant inter-arrival and processing times.

```
for (( b=$s_start; b<=$s_end; b+=2 )); do
      java -jar Splitter.jar $b -1 tu 0 0 rr 0 0 0 0 0 0 none 100 100 100 100 &
done
for (( c=$m_start; c<=$m_end; c+=2 )); do
      let next=$c+1
      java -jar Merger.jar $c localhost $next &
done
java -jar Merger.jar $last_m null 0 &
for (( d=$s_start; d<=$s_end; d+=2 )); do
      let merger=$d+1
      java -jar Instance.jar localhost $d constant_wait_operator 90 10 false 100 100 100
          localhost $merger &
done
java -jar Source.jar localhost 48458 false 5 Distributions/distribution_const_$a.txt
    Distributions/distribution_const_$ptime.txt $a 100 true

#Create own directory for the experiment, move log files there
mkdir -p master_wait_operator_pipe_sh/D90n_test/stage_$f/pdegree_$e"_iatime_"$a

#Move all log files
mv *.txt master_wait_operator_pipe_sh/D90n_test/stage_$f/pdegree_$e"_iatime_"$a
```

For the pipelining tests, a little more work had to be done when setting up the system, as we now also had more mergers and splitters. We introduced different variables for calculating the ports: $s_{start}$ represented the first splitter port $(48458)$, $s_{end}$ the last one of the chain, $m_{start}$ and $m_{end}$ respectively for the mergers. We then iterated from $s_{start}$ to $s_{end}$ and started splitter instances, surpassing every second port, as these represented the affiliated merger ports. An example of the port ips for pipelining degree $c = 3$ can be seen in figure 4.1

The same process was done for the mergers. Iterating from $m_{start}$ to $m_{end}$, every merger connected to the next splitter, which could be computed as the current merger port increased by $1$. The last merger was set up exclusively, as it did not connect to any further splitter.

Afterwards we instantiated the operator instances with the start port being the start port for the first splitter, incrementing by $2$ until the last. The port of the next merger could then be calculated by just adding 1 to the splitter port. All other arguments were kept the same as for data parallelization.

In the end, the source connected to the first splitter ("48445") and sent the events to the system.

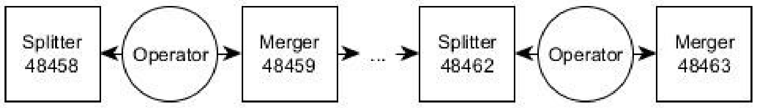

**Figure 4.1:** Port connections (with ips) for pipelining with degree three.

### 4.1.4 Folder structure

The test results were saved in a hierarchical structure like seen in 4.2. The highest layer forms the parallelization type differentiation, afterwards the used inter-arrival time and processing time model. Runs are then divided by stage. Afterwards we subdivide them into different parallelization degrees and inter-arrival times. In each subfolder, the following log files were saved, representing the results of a run:

- splitter queue size
- splitter processing time
- instance queue size
- operator latency

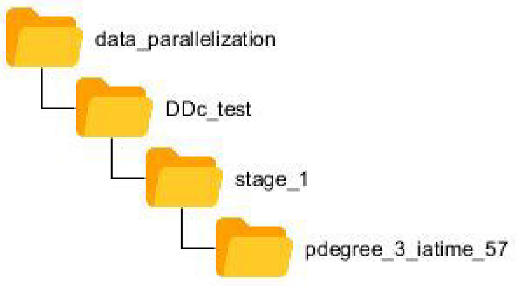

**Figure 4.2:** Example folder structure for our tests.

This was done for every part in the system except for the operator latency which was measured at the last merger of the chain. The splitter queue size / instance queue size were measured at the arrival of each event. This lead to a queue size of $0$ if the event arrived and no events were in the queue. The splitter processing time measured the delay that occured between picking an event and passing it on.

## 4.2 Wait Operator

In this section we analyze a simple wait operator in different data parallelization and pipelining situations. The results are then compared and interpreted to evaluate how well both techniques perform with different parallelization degrees. A short extension afterwards looks for sources of deviant operator behavior. We are assuming that we have a task that can be data parallelized, which means that the events are independent from each other, or the operator is stateless. We further determine, that the task done by the operator can be split up into $45$ atomar steps that can be executed in serial order using pipelining.

## 4.3 Data parallelization (D/D/c)

We set up an experiment consisting of three stages. In each individual stage, thirteen runs were executed. The runs were built up the following way: we initialised our system with the source producing $100$ events with specific constant inter-arrival times. Next to that, we connected the splitter, the wait operator with constant $90$ ms event processing time and the merger (4.1.2). In each run, a different inter-arrival time was chosen, varying from $1$ ms to $9 \cdot 10^{1.4}$ ms in logarithmic time steps. Furthermore we tested for parallelization degrees 1,2,3,30 and $45$. Repeating these runs three times resulted in the three stages. The outcome can be seen in figure 4.3.

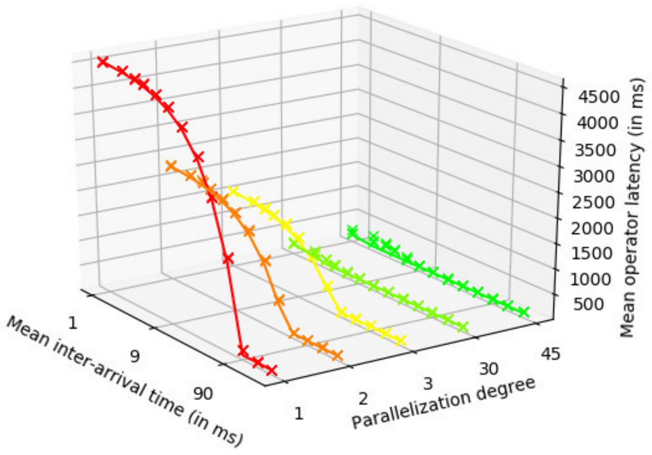

**Figure 4.3:** Measured mean operator latency (crosses) and our approximation model for the data parallelization (D/D/c) case.

While the mean latency $T_{op}$ is assuming really high values for low inter-arrival times and parallelization degrees, the curve flattens out for higher parallelization degrees. The slope of the curves are linearly decreasing up to the stability point where $\lambda = \mu$. From there on the operator latency is just the service time, as no event needs to wait in the queue but can be processed immediately.

### 4.3.1 Ideal model

We now want to compare how similar our CEP application is to an ideal sytem when using data parallelization. Hence, we first have to derive an ideal model. For the moment, we do not consider our splitter, to be able to determine its influence on the overall operator latency after our comparison. We therefore just take into acccount the operator instances in our simple model.
The mean operator latency can be calculated as

$$T_{op} = T_Q^i + s = s \cdot L_i + s = s(L_i + 1) \tag{4.1}$$

which results from Gross' book [8, p. 11]. $T_Q^i$ is here the average time an event has to wait in the queue. We further know that the average queue size for $a = \frac{s}{c}$ should be $0$ as the system is stable at this point. We can also assume that when the events arrive at an inter-arrival time $a \to 0$ the queue will fill and no event is released before every event arrived at the queue, which gives us a mean queue length of $\frac{e}{2}$. As we have a round-robin splitter, higher parallelization degrees split up the events evenly across the operator instances, which results in a maximal mean queue length of $\frac{e}{2c}$ for degree $c$. With the aid of these conclusions we get as our final equation for $L_i(a, c)$ (with inter-arrival time $a$ and parallelization degree $c$):

$$L_i(a,c) = \begin{cases} -\frac{100}{2s}a + \frac{100}{2c} & \text{if } a < \frac{s}{c} \\ 0 & \text{else} \end{cases}$$

and as end result our model

$$T_{op}(a,c) = \begin{cases} s(-\frac{100}{2s}a + \frac{100}{2c} + 1) & \text{if } a < \frac{s}{c} \\ s & \text{else} \end{cases}$$

Figure 4.3 shows the measured mean operator latencies with the approximated ones. The model fits the data relatively well with a mean absolute error of $44.66$ ms, which represents our average absolute approximation deviation. Still, our MSE with $3293.23$ is pretty high (expected would be $44.66^2 = 1936$), indicating that our error varys a lot for different values. This does not have to be a significant deficiency, but rather results from the fact that our range of y values differs from $\approx 90$ ms to $\approx 4500$ ms. We therefore also investigated the mean absolute percentage error, that tells us how far our approximation differs from the ideal model independent from the measured value. We obtained a value of $23.515\%$, which is very good for a naive approach. The results point out, that our application behaves very similar to an ideal data parallelized system.

### 4.3.2 Operator latency difference

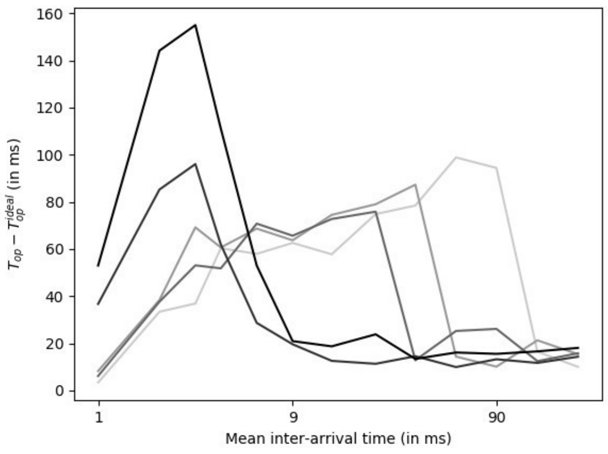

**Figure 4.4:** Difference between measured and approximated mean operator latency for different parallelization degrees and mean inter-arrival times. From light grey to black : degree 1 to degree 45

Plotting the difference between the two data sets directly, we get figure 4.4. There is a recognizable peak in the data for each parallelization degree, which shifts more to the left for higher degrees. A possible explanation for that is the stability point. Here, the exact model thinks that the system is still stable, while our additional latency influences make the system unstable and our operator instance queues begin to fill. We can further recognize that for all parallelization degrees the difference ranges from about $10$ ms to $90$ ms while for degree $45$ we find the peak at $\approx 155$ ms.
What are our subsequent latency resources? Like stated at the start of the chapter, we did not consider our splitter in the ideal model as we wanted to compare our application with a perfect data parallelization system. In figure 4.5 we can see, that for higher parallelization degrees the splitter processing time indeed increases, although our data is too noisy to specify a specific approximation. This growth is a consequence of the extra connections the splitter has to handle when working with a high parallelization degree.
The plot of the splitter queue sizes is more meaningful. One can notice, that the length is at around zero up until an inter-arrival time of $9 \cdot 10^{1.2}$. From there on the queue size starts to grow with increasing values for larger parallelization degrees. The peak value of additional latency caused by the splitter can be found for parallelization degree $c = 45$ and inter-arrival time $a = 9 \cdot 10^{1.4}$. Regarding the splitter as a unique (D/D/1)-system we can compute this as

$$T_s = (1 + L_s)P_s = (1 + 28.25) \cdot 3.23\text{ms} = 94.4775\text{ms}$$

We see that especially for low mean inter-arrival times, the splitter plays a significant role for the mean operator latency. For high mean inter-arrival times on the other hand, the splitter transforms to a nearly constant delay source.

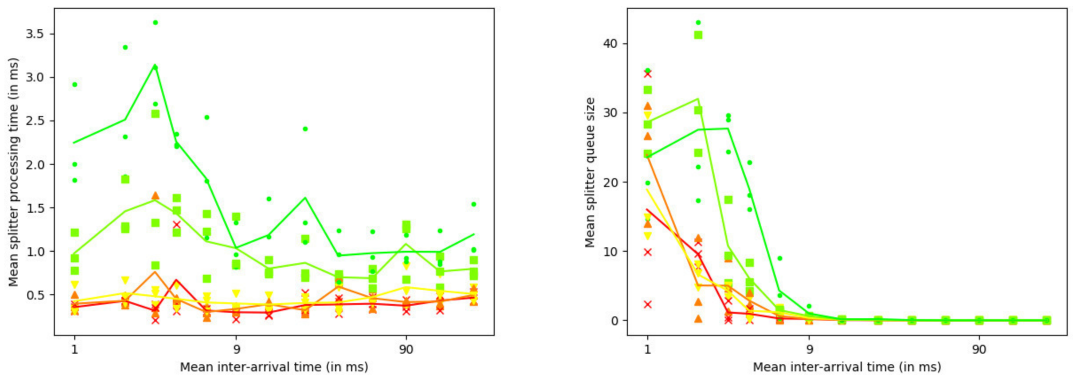

**Figure 4.5:** Splitter processing time (left) and queue size (right) for different parallelization degrees and mean inter-arrival times. From red to green : degree 1 to 45.

### 4.3.3 Stability points

Another interesting information is the stability of our system. In particular, we would like to know how far our stability point of our application differs from the ideal data parallelization model. We therefore had to initially find out the points in our measured data sets. This was done by first calculating the service time $p_{op}$ of the balanced system, i.e. an empty operator queue. The points left over were then linear approximated using linear regression. The intersection point was our calculated stability point for the given parallelization degree. Two examples for parallelization degree $2$ and $3$ can be seen on the left of 4.6.
The right part of the figure shows us the difference of the stability points in terms of their affiliated inter-arrival times. In concrete, this means that for our system the mean inter-arrival time needs to be as much higher as the y-values of the plot present. Regard for example parallelization degree $c = 30$: the ideal model has its stability point at $a = s = 90$ ms. Our system though works slower, resulting in the stability point at $a = s = 93.6$ ms and a stability point difference of $3.6$ ms. We can see that for low parallelization degrees this difference decreases slightly, while for higher parallelization degrees the values starts to grow, leading to $3.49$ ms for $c = 30$ up to $8.5$ ms for $c = 45$. One explanation for that is already given by the increasing processing time and queue length of the splitter.

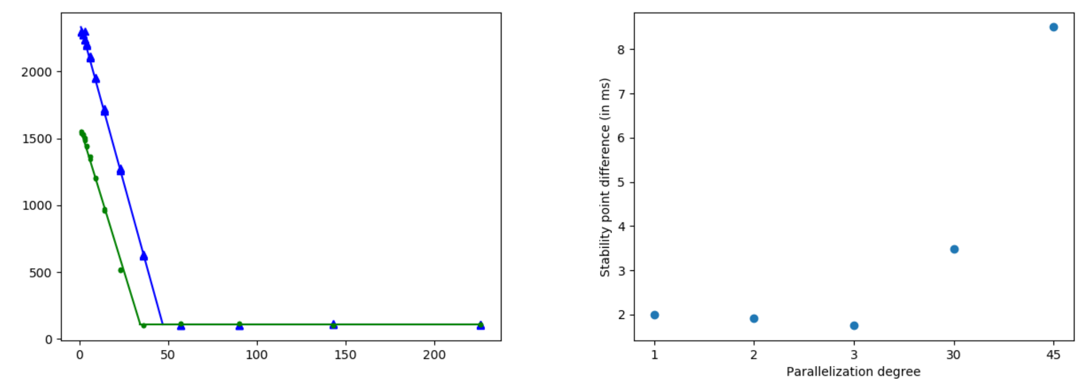

**Figure 4.6:** Left: linear regression of measured operator latency for parallelization degree 2 (blue) and 3 (green). Right: difference between measured and computed balance points for different parallelization degrees.

## 4.4 Data parallelization (M/M/c)

To simulate a more realistic environment, we worked with distributions for the inter-arrival time and processing time. The inter-arrival times were thereby depicted as poisson process and the service time exponentially distributed with given parameter $\lambda$, which equals our arrival rate. We then ran the same tests as for our (D/D/1) environment.

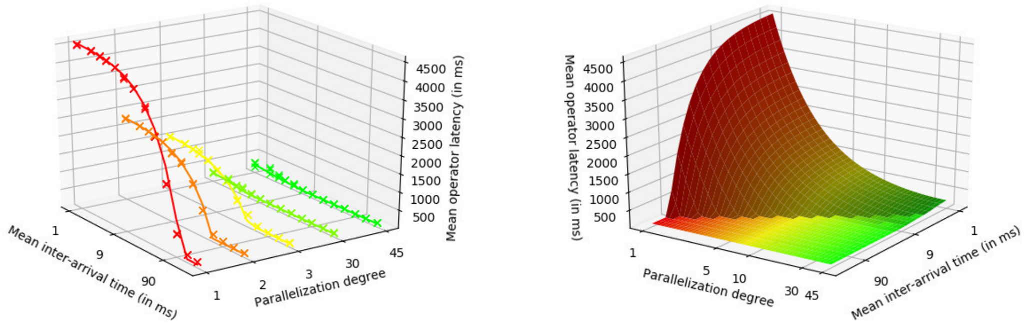

**Figure 4.7:** Measured mean operator latency (crosses) and our approximation model for the data parallelization (M/M/c) case.

In figure 4.7 (left), you can see the operator latencies for different degrees of data parallelization, already combined with our model that we will derive in the next paragraph. The data actually just differs slightly from our (D/D/c) measurements (MAE: $39.11$), due to the distributions for the inter-arrival and the processing time. Like 4.3, the operator latency curve flattens out for larger data parallelization degrees, which can be affiliated to the declining queue sizes in the operator instances and the splitter.

### 4.4.1 Approximation model

To approximate the operator latency curve, we first divided the operator instance and the splitter into two queueing models, which we can do due to their independent behavior in a sequential chain [8, p. 167f]. We then get for our mean operator latency:

$$\begin{aligned} T_{op} &= T_Q^s + P_s + T_Q^i + P_i \\ &= P_s \cdot L_s + P_s + P_i \cdot L_i + P_i \end{aligned}$$

$$T_{op} = P_s(L_s + 1) + P_i(L_i + 1) \tag{4.2}$$

We retrieve four variables that we need to approximate to create a working model. The mean processing time for the instance is already known as the time parameter for the wait operator, which in our case is $90$ ms. The other variables are approximated via linear regression (see 4.6 left plot).
We started with the mean splitter processing time of the system. Regarding the proper graph, one can see that the processing time fluctuates between $0.6$ ms and $0.3$ ms for low parallization degrees. In contrast to that, high use of parallelization lets the splitter processing time rise between inter-arrival times $9$ and $1$. In the further course, the curve stabilizes at around $1$ ms for parallelization degree $45$ and $0.6$ ms for degree $30$. Our approximation just regarded degree $1$ and $45$, using both curves as marginal functions and linearly interpolating immediate parallelization degrees. The functions itselfs were a combination of a linear regression straight line and a constant or for parallelization degree 1 just a constant all the way. Combining these, we got as our final general function for the mean splitter processing time given mean inter-arrival time $a$ and parallelization degree $c$:

$$m = \Big(-0.0001 - 0.1687\frac{c-1}{46}\Big),\ z = \Big(0.3788 + 2.4628\frac{c-1}{46}\Big),\ y = \Big(0.03788 + 0.6894\frac{c-1}{46}\Big)$$

$$P_s(a,c) = \begin{cases} ma + z & \text{if } a < \frac{y-z}{m} \\ y & \text{else} \end{cases}$$

Proceeding with the splitter queue size, the task got a little bit easier here. In the figure 4.8 one can recognize that the mean splitter queue size for all parallellization degrees drops to $0$ at a inter-arrival time of around $9$. We therefore split the curves into two parts, examining both pieces seperatly. The part from $a = 0$ to $a = 9$ was approximated by linear regression, while the right part was just constant 0. Again, we linearly interpolated between parallelization degree $1$ and $45$ for intervening parallelization degrees. The function for the splitter queue length then reads as

$$m = \Big(-3.6322 - 0.3104\frac{c-1}{46}\Big),\ z = \Big(17.5539 + 15.5603\frac{c-1}{46}\Big)$$

$$L_S(a,c) = \begin{cases} ma + z & \text{if } a < \frac{-z}{m} \\ 0 & \text{else} \end{cases}$$

The easist function arised for the instance queue size. Due to the linearity of the instance queue length increment, we calculated one slope for all parallelization degrees and just changed our $z$ of our linear function depending on the current degree. Starting from the root of the straight line, we then set all other values to $0$. This resulted in the composite function

$$m = -0.4401,\ z = \frac{100}{2c}$$

$$L_I(a,c) = \begin{cases} ma + z & \text{if } a < \frac{-z}{m} \\ 0 & \text{else} \end{cases}$$

The result of the complex model is seen in figure 4.7. Here we plotted our approximated mean operator latency values together with our measured results from the test.
To determine the accurancy of our approximation model, we compare it with the naive approach established in the previous section (4.3.2). Analyzing the difference between the measured mean operator latency values and the naive model, we get for the MSE, MAE and MAPE: MSE $= 11932.66$ ms, MAE $= 71.58$ ms, MAPE $= 45.88\%$. One recognizes, that the approximation displays a really large MSE value, which indicates that the model does not fit the data well. Furthermore we see, due to the MAPE value of $45.88\%$, that on average our estimated values lie almost $50\%$ under or over our measured values. We take these results as a benchmark for our slightly more complex model.
Here, we receive an MSE of $9554.63$. Still, this value seems pretty high. As we cover a big interval with our measured mean operator latencies though, this does not say much. One can see that when analyzing the MAE and MAPE values. Our mean absolute error drops from $71.58$ ms to $54.37$ ms, which is an improvement of nearly $25\%$. The highest change can be seen for the mean absolute percentage error. Its value declines from $45.88\%$ to $25.23\%$, resulting in almost $50\%$ less biased results. This indicates a great improvement of approximation accurancy in contrast to our naive model.

### 4.4.2 Evaluation

To test the peformance of the model, we executed three example runs:

- inter-arrival time and parallelization degree in our measured range (1-226, 1-45)
- inter-arrival time in our measured range, parallelization degree outside of our measured range (1-226,>45)
- inter-arrival time outside of our measured range, parallelization degree in our measured range (>226, 1-45)

The exact values can be seen in table 4.1. The value in our approximation range is estimated with a small percentage error, while operator latencies outside our measured range appear to be more biased. Still, the approximated values are well enough to be used for a valuation of the latency delay in the operator, given a certain mean inter-arrival time and parallelization degree. Obviously, these runs do not represent the whole range of our model, and do just serve the purpose of small examples.

| a | n | $T_{op}$ | $T_{op}(a,n)$ | Error (absolut, prozentual) | |
|---|---|---|---|---|---|
| 57 | 20 | 108.8146 | 108.6636 | 0.151 | 0.14% |
| 23 | 50 | 110.9318 | 109.1132 | 1.8186 | 1.64% |
| 300 | 2 | 115.4882 | 108.3938 | 6.0944 | 6.14% |

**Table 4.1:** Results for the performance measurement of our approximation model

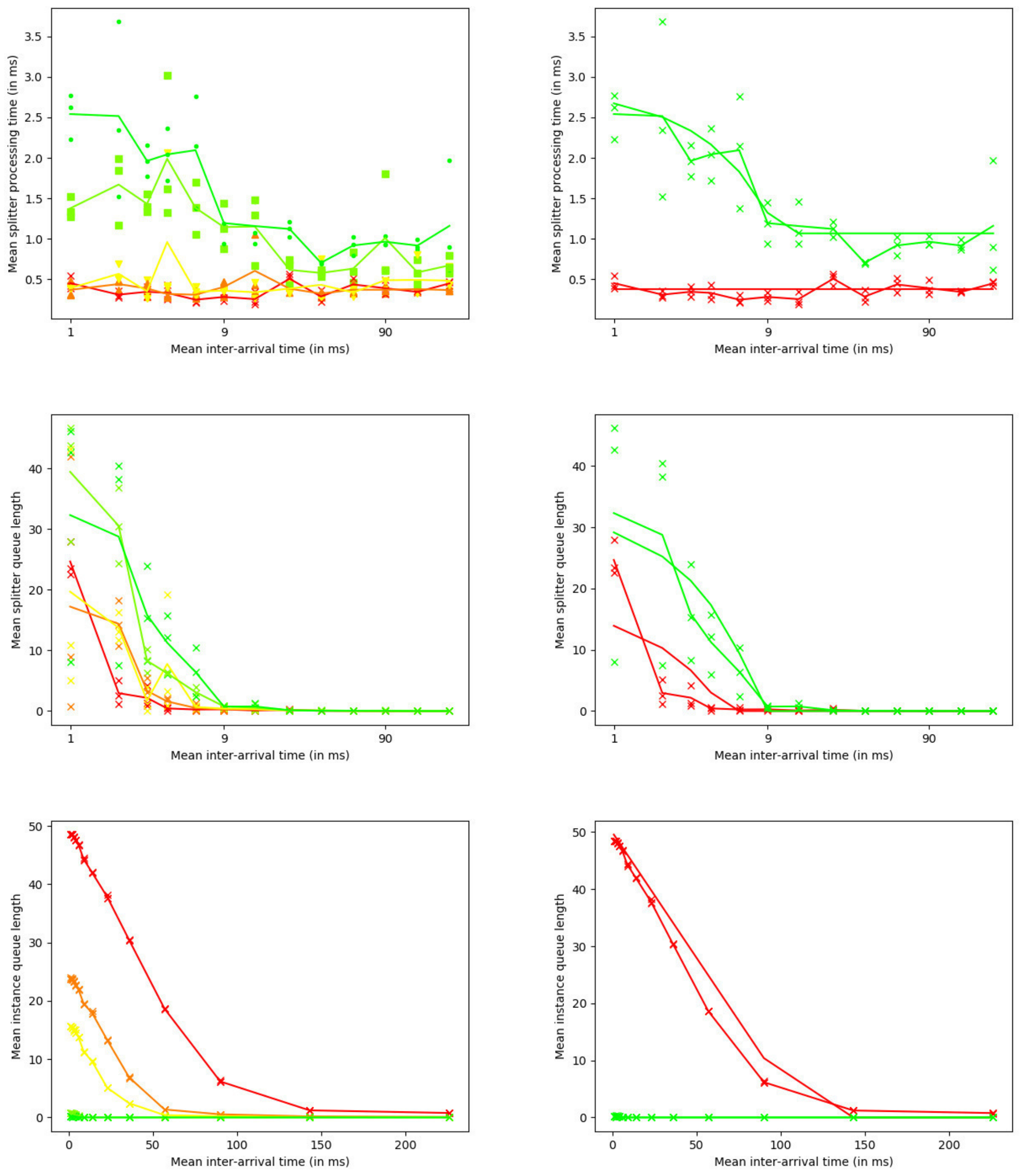

**Figure 4.8:** Using linear regression for the parameter approximation. Top: mean splitter processing time, middle: mean splitter queue length, bottom: mean instance queue length. From red to green : degree $1$ to $45$.

## 4.5 Pipelining (D/D/c)

We now want to analyze the wait operator as (D/D/c) queue using pipeling. The setup for our experiment looked similar to our (D/D/c)-model for data parallelization: three stages containing thirteen runs for five parallelization degrees were executed, with varying inter-arrival times and a constant $90$ ms processing time. The service time was distributed to the amount of operators, i.e. each operator had a processing time of $\frac{s}{n}$ ms.
Figure 4.9 shows the point cloud arising from measuring the mean operator latency $t_{op}$. One can directly see that for high parallelization degrees, the mean operator latency does not decrease but rather increase a lot to almost $10000$ ms. We therefore think that the difference to an ideal pipelining model will be very large. We compare both in the next paragraph.

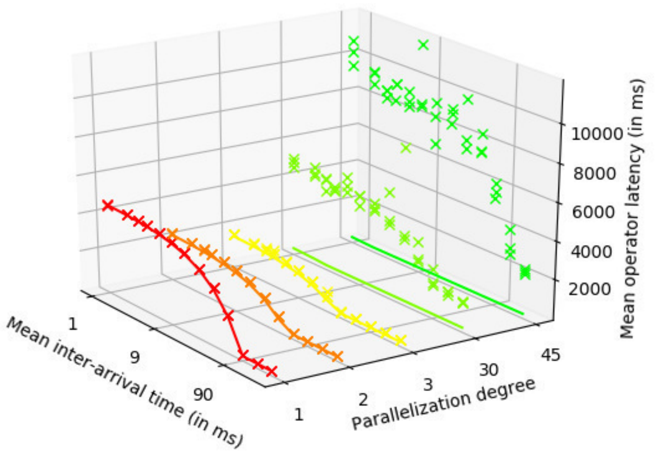

**Figure 4.9:** Measured mean operator latency (crosses) and our approximation model for the pipelining (D/D/c) case.

### 4.5.1 Operator latency difference

Recall the mean operator latency equation (4.1)

$$T_{op} = T_Q^i + s = s \cdot L_i + s = s(L_i + 1)$$

By Gross [8, p. 167f], we know that we can consider each operator instance in a pipeline as individual (D/D/1) queue. Therefore, we initially analyze our first instance in the chain. We know that it has a processing time of $\frac{s}{c}$, as for parallelization degree $c$, the processing time is distributed equally over all $c$ operator instances. Like in the data parallelization case, we further conclude that for inter-arrival times $a \to 0$ the queue will fill, which results in $L_i = 50$. Moreover, the queue stays empty when $\frac{s}{c} \leq a$. In an ideal model, the operator instance takes $\frac{s}{c}$ ms to process an event, yielding to an mean inter-arrival time of the same amount for the next operator. This way, subsequent operators have an empty queue, as they can serve each event before the next one arrives. Taking these assumption into account we get for the queue length of the first operator

$$L_{i_0}(a,c) = \begin{cases} -\frac{100n}{2s}a + \frac{100}{2} & \text{if } a < \frac{s}{c} \\ 0 & \text{else} \end{cases}$$

The mean operator latency for the remaining operator instances in the chain is now simply $s - \frac{s}{c}$ (empty queues), and if we plug $L_{i_0}$ into equation 4.1, we retrieve as our model

$$T_{op}(a,c) = \begin{cases} \frac{s}{c}(-\frac{100n}{2s}a + \frac{100}{2} + 1) + s - \frac{s}{c} & \text{if } a < \frac{s}{c} \\ s & \text{else} \end{cases}$$

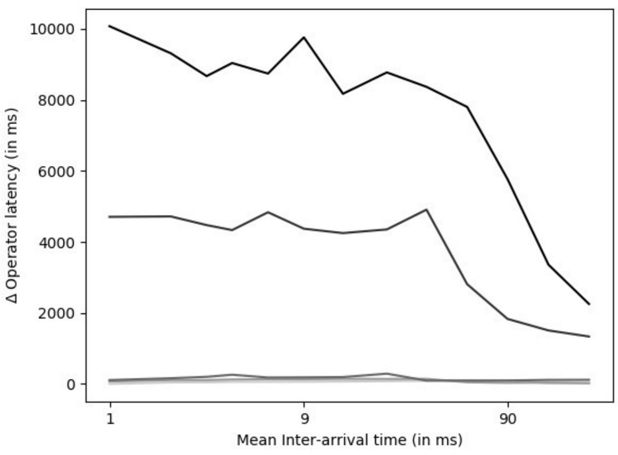

**Figure 4.10:** Difference between measured and approximated mean operator latency for different parallelization degrees and mean inter-arrival times. From light grey to black : degree $1$ to degree $45$

The plot of the mean operator latency difference to the ideal model confirmed our previous assumption of great distinction between the values. The graph shows that for high parallelization degrees the mean operator latency skyrockets, although our ideal model says otherwise. We tried to analyze the mean splitter queue length and processing time to see if we could find any context here (4.11). In the diagrams one can see, that for higher degrees the two variables indeed increase steadily. However they do not generate a delay this high as seen in our test plot, with the maximum delay occuring for parallelization degree $c = 45$ and mean inter-arrival time $a = 1$, resulting in the latency of all splitters: $\approx 45 \cdot (2.7 \cdot 8.2) = 996.3$ ms.

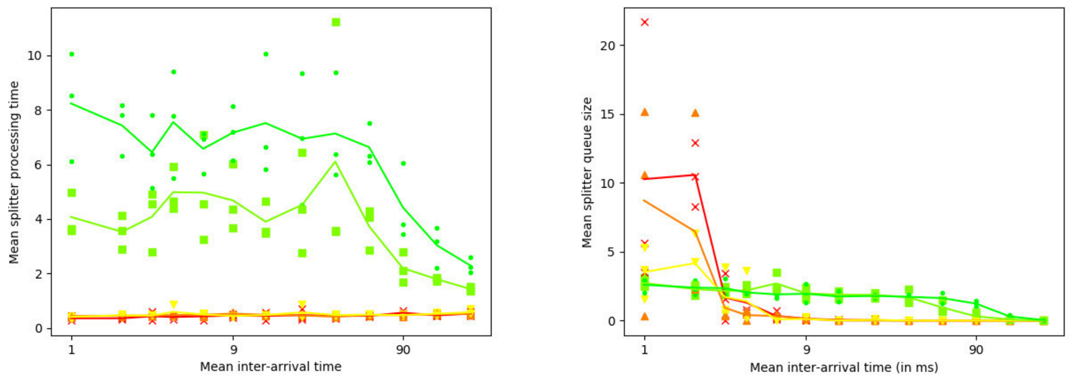

**Figure 4.11:** Splitter processing time (left) and queue size (right) for different parallelization degrees and mean inter-arrival times. From red to green : degree 1 to 45.

To prosecute the surprising results, we ran our tests on two other machines with different specifications (4.1.1). For the first test, we increased our RAM from 8 GB to 32 GB. The resulting plot can be seen on the left of figure 4.12. Analyzing the difference between the measured values for degree 30 and 45 in comparison to our first test, we retrieved the following mean operator latency reductions: for degree 30, the mean operator latency was 78.5% the amount of the original measurements and for degree 45 it even went down to 37.8%. We can derive from this fact, that the difference in RAM has a large influence on the system. We now extended the system specifications even further, stocking up from 4 VCPUs to 24 VCPUs. The results went down to 74.8% the amount of measured latency than our original test results for degree 30, and 27.7% for degree 45. Important to note is that the change in hardware specification did just affect our high parallelization degrees, with low latency values staying almost the same. Furthermore, the additional VCPUs have a large impact on the high parallelization degree 45, dropping the mean operator latency from 37.8% to 27.7%, but less on degree 30 with just 3.7% improvement.

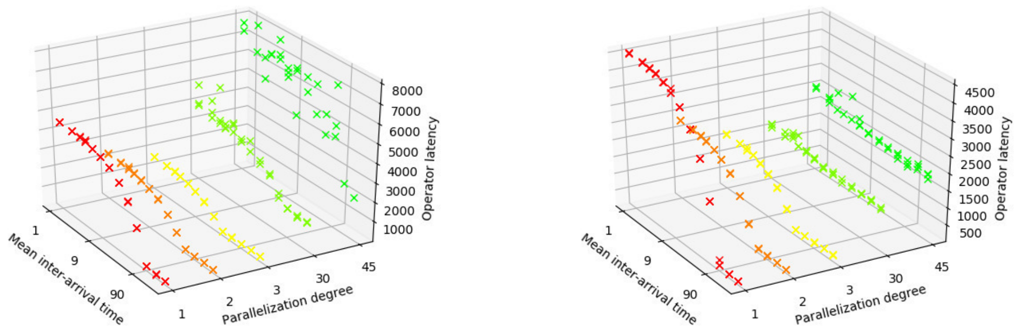

**Figure 4.12:** Plot of the system running on a 32 GB RAM machine (left), right additionally with 24 VCPUs.

## 4.6 Pipelining (M/D/c)

Also for pipelining we were interested in a more realistic scenario by analyzing the wait operator for different pipelining degrees using inter-arrival times depict from a poisson process. To be able to investigate our operator latency phenomen better, we increased our measured parallelization degrees from $5$ to $8$. We kept the processing time constant, to be able to divide it over all operator instances depending on our current parallelization degree. Otherwise, a $2$ millisecond processing time for example would be hard to split up over $45$ instances, especially as we implemented the smallest wait time to be $1$ ms.
The results of the test can be seen in figure 4.13. The graph looks similar to the (D/D/c) case, due to the fact that we just changed the variance of the inter-arrival time. We also included our calculated model, which we derive in the next section. Comparing the two data sets of the (D/D/c) test and the (M/D/c) test, given the five joint parallelization degrees, we retrieve an MAE of $374.51$ ms. We ground this large value on the great variance in the data for the high parallelization degree range. This could signify, that our operator can deal better with varianced inter-arrival times with a smaller pipeline than a bigger one.

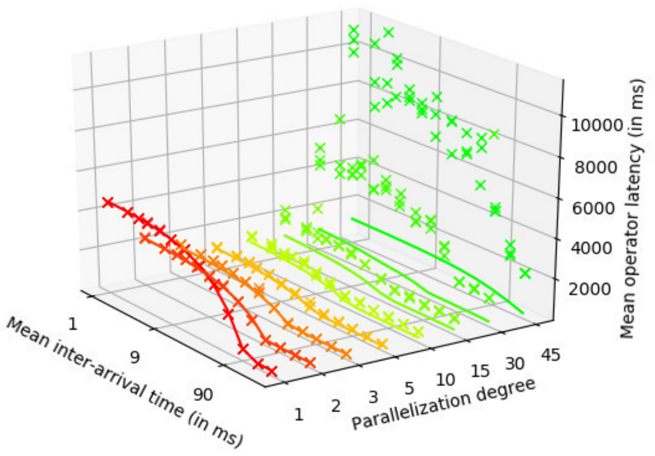

**Figure 4.13:** Measured mean operator latency (crosses) and our approximation model for the pipelining (M/D/c) case.

### 4.6.1 Approximation model

Based on our test results for the pipelined operator, we created a model for the (M/D/c) case. Like for data parallelization, we used the splitter and instance queue lengths as well as the splitter processing time to derive our approximation. We do not consider the mergers properties here, as they just pass events on to the next splitter.

We can see our pipelining configuration as expansion of equation 4.2, resulting in a sequential chain of splitter-instance pairs

$$T_{op} = P_{s_0}(L_{s_0} + 1) + P_{i_0}(L_{i_0} + 1) + \ldots + P_{s_c}(L_{s_c} + 1) + P_{i_c}(L_{i_c} + 1)$$

As each instance has the same processing time $P_{i_1} = P_{i_2} = \ldots = P_{i_c} = \frac{P_i}{c}$ ms, we can sum up the instance clauses to a single clause, executing the following reductions

$$\begin{aligned} P_{i_1}(L_{i_0} + 1) + \ldots + P_{i_c}(L_{i_c} + 1) &= \frac{P_i}{c}(L_{i_0} + \ldots + L_{i_c} + 1 + \ldots 1) \\ &= \frac{P_i}{c}(L_i \cdot c + c) \\ &= P_i(L_i + 1) \end{aligned}$$

For simplicity reasons, we also shorten our splitter clausels. We therefore do not consider every splitter individually but assume that each splitter has the same processing time $P_s = \frac{P_{s0}+\ldots+P_{s_c}}{c}$ and queue length $L_s = \frac{L_{s0}+\ldots+L_{s_c}}{c}$. These can now be plugged into equation 4.1. As we have $c$ splitters in the system we need to multiply the result with $c$. In the end we can write our full approximation model as combination of the two parts

$$T_{op} = c \cdot P_s(L_s + 1) + P_i(L_i + 1) \tag{4.3}$$

which is almost identical with equation 4.2. Like in the data parallelization section, we have four variables that we need to approximate. The instance processing time is already given as $\frac{s}{c}$ for the total service time $s$ and parallelization degree $c$.
Regarding the splitter processing time, one can see that for low parallization degrees up to $c = 15$, the curve is nearly constant, while increasing its y-value for higher degrees. The parallelization degrees $30$ and $45$ act differently: for $c = 30$ the curve seams to be somewhat constant but one can recognize a small decay for higher mean inter-arrival times. This is much more significant for $c = 45$. Due to this observation, we approximated the mean splitter processing time linearly between paralleization degree $1$ and $45$, with the first curve being a nearly constant function averaging all measured y-values for the $c1$ parameter. The second curve just used linear regression to identify its slope and offset. Our final function then reads as

$$m = \Big(0.003011 - 0.01246\frac{c-1}{44}\Big),\ z = \Big(0.4166 + 4.5155\frac{c-1}{44}\Big)$$
$$P_s(a, c) = ma + z$$

The two other parameters act differently on their pipelining degree course. Figure 4.14 shows that for degrees $c = 1$ until $c = 10$, the mean splitter queue length shrinks for low inter-arrival times and in the end hits a point where $L_s \approx 0$. Due to that, we simulated $c = 1$ up to $c = 10$ with linear regression of all the points that we thought were still part of the straight line, and then took the mean of the remaining points for an approximation for high paralleization degrees. Intermediate degrees were then linearly interpolated. From parallelization degree $c = 10$ to $c = 45$, the mean splitter queue length now increases though. Looking at the trend of the curve $c = 45$, one can see, that it slowly decreases up to a certain point which is not in our measurement range (possibly). We therefore just used linear regression and took the resulting curve as our approximation for degree 45. Here again, we interpolated linearly between parallelization degree $c = 10$ and $c = 45$. Putting all together we get the function for the mean splitter queue length

$$m_1 = \Big(-0.8702 + 0.2067\frac{c-1}{9}\Big),\ z_1 = \Big(6.8219 - 4.6563\frac{c-1}{9}\Big),\ y_1 = \Big(0 + 0.2063\frac{c-1}{9}\Big)$$

$$L_s(a, 1-10) = \begin{cases} m_1 a + z_1 & \text{if } a < \frac{-z_1}{m_1} \\ y_1 & \text{else} \end{cases}$$

$$m_2 = \Big(-0.6635 + 0.6527\frac{c-11}{34}\Big),\ z_2 = \Big(2.1656 + 0.1437\frac{c-11}{34}\Big)$$

$$y_2 = \Big(0.2063 - 0.2063\frac{c-11}{34}\Big)$$

$$L_s(a, > 10) = \begin{cases} m_2 a + z_2 & \text{if } a < \frac{-z_2}{m_2} \\ y_2 & \text{else} \end{cases}$$

We calculated our instance queue length similar to our splitter queue length case. Figure 4.14 shows, that the mean instance queue length decreases linearly for pipelining degrees 1-10, and then rests at a point where $L_i \approx 0$. For that reason, we took both degrees as our outer curves for an inner interpolation of intermediate degrees. Regarding some of the degrees in between, we noticed that the values dropped exponentially, which made us interpolate exponentially between degree 1 and 10. The functions for $c = 1$ and $c = 10$ were derived by a linear regression of the first points of the data, leading to a straight line. This line was then intersected with the constant resulting from the mean of the remaining data points. The outcome was a partial function and our first part of the approximation. Higher degrees did not behave like we expected, figure 4.13 shows that for example for pipelining degree 45. The curve starts with a lower mean instance queue size, but also descends slower, resulting in higher values for $a > 25$ than the lower degrees $c = 5$ and $c = 15$. Therefore, we did a second linear interpolation between $c = 10$ and $c = 45$, with the curve for $c = 45$ being created from linear regression of all the data points. In the end we retrieved the function for the mean instance queue length

$$m_1 = \Big(-0.5099 + 0.13290(1 - e^{-(c-1)})\Big),\ z_1 = \Big(50.3322 - 39.9922(1 - e^{-(c-1)})\Big)$$

$$y_1 = \Big(0.735 - 0.0704(1 - e^{-(c-1)})\Big)$$

$$L_i(a, 1-10) = \begin{cases} m_1 a + z_1 & \text{if } a < \frac{-z_1}{m_1} \\ y_1 & \text{else} \end{cases}$$

$$m_2 = \Big(-0.3762 + 0.3603\frac{c-11}{34}\Big),\ z_2 = \Big(10.34 - 6.5254\frac{c-11}{34}\Big)$$

$$y_2 = \Big(0.6646 - 0.6646\frac{c-11}{34}\Big)$$

$$L_i(a, > 10) = \begin{cases} m_2 a + z_2 & \text{if } a < \frac{-z_2}{m_2} \\ y_2 & \text{else} \end{cases}$$

### 4.6.2 Evaluation

For pipelining we confine ourselves to two example runs, as we saw in the previous section that high parallelization degrees are not approximated properly:

- inter-arrival time and parallelization degree in our approximation range (1-226, 1-10)
- inter-arrival time outside of our measured range, parallelization degree in our measured range (>226, 1-10)

The results are listed in 4.2. One can recognize that both values are not that well approximated as in the data parallelization case. Still, the error is in a acceptable range, being $2.66\%$ higher for the test outside of our measured range. Of course, these two runs do not reflect the whole model in detail and are just meant as example test cases.

| **a** | **n** | $T_{op}$ | $T_{op}(a,c)$ | **Error (absolut, prozentual)** | |
|---|---|---|---|---|---|
| 57 | 2 | 185.1711 | 165.0282 | 20.1429 | 10.88% |
| 300 | 2 | 132.2508 | 152.9598 | 20.709 | 13.54% |

**Table 4.2:** Results for the performance measurement of our approximation model

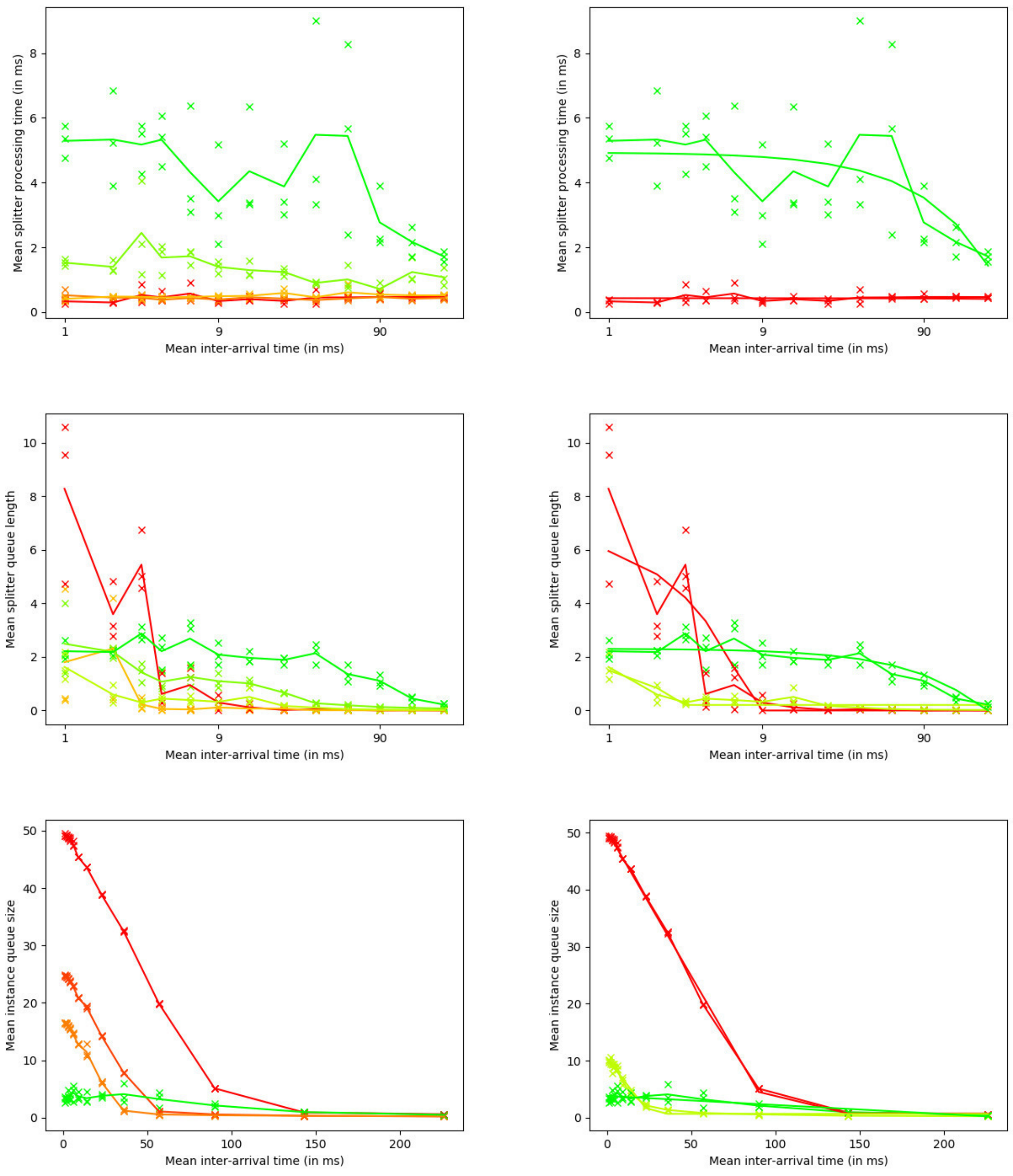

**Figure 4.14:** Using linear regression for the parameter approximation. Top: mean splitter processing time, middle: mean splitter queue length, bottom: mean instance queue length. From red to green : degree $1$ to $45$.

# 5 Results

In the last chapter, we tested our wait operator under multiple different conditions, leading to affirmative but also surprising results.
In the (D/D/c) section for data parallelization, we analyzed the deviation of our measured data to an ideal data parallelization model. Results showed, that the latency difference is very high at the stability points and increases for higher parallelization degrees. Afterwards we compared the stability points directly, with the corresponding figure indicating that for higher degrees the stability point shifts up to 8ms of mean inter-arrival time, while being at around 2ms for low parallelization degrees.
We then investigated the operator as (M/M/c) queueing system. We retrieved a model by approximating different parameters of the system, namely the mean instance and splitter queue lenghts as well as the splitter processing time. In the end, we measured the distinction to the test data, resulting in an MSE of $9554.63$. Hereafter, we executed three runs that gave us an insight on how good the model approximates mean operator latency values (i) in our measured range and (ii) outside our measured range. The result showed that outside our range we had a larger absolut and percentage error than in our borders.
We then switched to pipelining, first regarding it as a (D/D/c) queueing system. Here, we retrieved surprising results: for low parallelization degrees, the mean operator latency fell, but became really large for degrees 30 and 45. Still, we compared the data to an ideal pipelining model and oveserved, that there exists a large distinction of the mean operator latency for high parallelization degrees.
In the (M/D/c) section for pipelining, we investigated our high mean operator latency values for large pipelining degrees by running tests on multiple systems. The results show that both more RAM and VCPUs yield to lower values, but have still a large deviation to our ideal pipelining model. Afterwards, we created an approximation model, investigating the same parameters as in the (M/M/c) section for data parallelization. The model approximated our parallelization degrees well up to $c = 10$. For higher degrees, our model was not capable of approximating the values right. Like in the data parallelization case, we added a little evaluation, comparing approximated values inside and outside of our variable value range. We saw that, despite an higher prozentual and absolute error than the data-parallelization test cases, the data was approximated pretty well. However, we also used a small parallelization degree of $2$ in our runs.

### 5.0.1 Use Cases

The derived (M/M/c) or (M/D/c) models for data parallelization and pipelining can be used in different scenarios:
In queueing systems, stability is a huge factor, as an unstable system gets out of control quickly, leading to huge mean operator latency values. Therefore we can use our model to approximate the stability points of our system. This can be done with linear regression similar to the (D/D/c) case (4.3.3). We can then create systems that get close to the stability points, reducing arising expenses.
Another important usage of our model is the latency reduction prognosis. Consider you need to know, how much latency reduction you gain when you increase your parallelization degree for your data parallelized system with $c = 6$ and $a = 15$ ms by one. This can be easily calculated by just subtracting the values for both parallelization degrees: $T_{op}(15, 6) - T_{op}(15, 7) = 127.127$ ms, leaving you with a value that can be used to estimate between additional costs and latency reduction.
The model is further suitable for cost computation. As an example we can calculate how high our parallelization degree has to be, so that the mean operator latency does not exceed a certain threshhold. Given $a = 15$ ms and $\hat{T}_{op} = 100$ ms this is done by setting $T_{op}(15ms, c) = 100$ ms. We are then able to solve this equation by using a numeric process like the newton method [2, p. 52 - 56], rounding up our found solution. In the end we arrive at our result: $c = 8$.
Of course there exist non-negligible borders to our models: we just regarded 100 events per run, meaning that the system is not able to stabilize itself perfectly. The used operator is a simple wait operator, more complex operators could, by virtue of their composition, affect the mean operator latency in their own way. Furthermore also the machine, we run our system on, makes a difference, like seen in the pipelining (D/D/c) case.

## 5.1 Conclusion

In this thesis we especially wanted to analyze, how well our system can compete with an ideal system regarding different degrees of data parallelization and pipelining. In the (D/D/c) data parallelization section, we had the hypothesis that for higher degrees, the splitter influences the system more and more. Our tests affirm this, as the difference between measured mean operator latency and ideal one grows for high degrees, as well as the splitter queue size and processing time. The same happens with the stability points, that depart from the ideal mean inter-arrival time when we increase the data parallelization degree. We had the same assumptions for our pipelining model, but due to the surprisingly high mean operator latencies for large parallelization degrees, we can not assign our extra delays in the operator to the increasing splitter service time. Rather, our machine properties led to the unexpected results and created huge differences to the ideal model.
The second important part of our thesis was to create a model that is able to approximate our results in a reasonable way. Our assumption was that by deriving an approximation

model from queueing theory formulas, we would be able to reach this goal. Ultimately we can say, that the (M/M/c) model for data parallelization does a suitable job for approximating the mean operator latency of our system, while our (M/D/c) model for pipelining fails to succeed in that due to the machine dependancy that we did not include.

# 6 Future Work / Resume

As this thesis was a basis work for future investigations of the given Complex Event Processing application, there exist a lot of regions further research can expand on: A more in depth analysis of the pipelining method used in this work is required, to be able to explain the poor performance for high parallelization degrees. An example could be the context between the mean operator latency and the hardware specifications, especially the available RAM and VCPUs. Furthermore, we concentrated on a simple wait operator to test the fundamental properties of the system. Based on these results, one can extend the work for an analysis of Complex Event Processing specific operators like time-window or entity-window operators (see [10] or [24] for additional information). Furthermore, there exists the possiblity to explore the wait operator for example for different mean processing times. Another interesting area is regarding the system as stationary process. In our case, we only sent 100 events through our system, due to the high amount of individual test procedures. If you run the system over a longer time period, it will equilibrate, leading to new measurable values, that can be set in relation to our inputs (see [8]). Subsequent research could use this, to create an operator latency prognosis for a running system, that is able to adjust its data parallelization or pipelining degree solely on its historical sensor information.

## 6.1 Resume

In our thesis we analyzed the behavior of an operator of a Complex Event Processing application under different conditions. We investigated, how our system would react on different degrees of data parallelization and pipelining, as well as examining, how the individual parts of the operator would behave like. Furthermore, we compared our results with ideal data parallelization and pipelining models, to be able to measure the performance of the system. Going further, we introduced variance by adding exponential distributions to the inter-arrival times (and processing times). For the resulting (M/M/c) or (M/D/c) queueing sytems, we tried to create approximation models that are capable of estimating the mean operator latency given a certain mean inter-arrival time and parallelization degree.

Complex Event Processing has been explored a lot in the last decades. Still, plenty of research tasks are not completely exhausted yet, and there are arising further thematic areas time and again. It is therefore important to prosecute this field and investigate, especially in the direction of cloud computing, how one can reduce the cost for a Complex Event Processing system, while in the meantime providing low latency.

# Bibliography

[1] *AWS | Amazon Datenbankserver Relational Database Service*. URL: https://aws.amazon.com/de/rds/ (visited on 02/13/2018) (cit. on p. 9).

[2] K. E. Atkinson. *An introduction to numerical analysis*. Wiley, 1989, p. 693. ISBN: 0471624896 (cit. on p. 48).

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom. "Models and Issues in Data Stream Systems." In: (). URL: https://infolab.usc.edu/csci599/Fall2002/paper/DML2{_}streams-issues.pdf (cit. on p. 13).

[4] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, S. Zdonik. "Monitoring Streams – A New Class of Data Management Applications." In: () (cit. on p. 13).

[5] *Cloud-Computing, Hostingdienste und APIs von Google | Google Cloud Platform*. URL: https://cloud.google.com/ (visited on 02/13/2018) (cit. on p. 9).

[6] G. Cugola, A. Margara, P. Milano. "Processing Flows of Information : From Data Stream to Complex Event Processing." In: 44.3 (2012). DOI: 10.1145/2187671.2187677 (cit. on p. 14).

[7] R. Gad, M. Kappes, I. Medina-Bulo. "Local parallelization of pleasingly parallel stream processing on multiple CPU cores." In: *2016 IEEE 7th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. IEEE, 2016, pp. 1–8. ISBN: 978-1-5090-0996-1. DOI: 10.1109/IEMCON.2016.7746285. URL: http://ieeexplore.ieee.org/document/7746285/ (cit. on p. 10).

[8] D. Gross, J. F. Shortie, J. M. Thompson, C. M. Harris. *Fundamentals of Queueing Theory*. Hoboken, New Jersey: John Wiley & Sons, Inc., 2008. ISBN: 9781118625651. DOI: 10.1002/9781118625651. URL: http://doi.wiley.com/10.1002/9781118625651 (cit. on pp. 20, 29, 33, 37, 51).

[9] M. Hirzel, S. Schneider, R. Grimm, R. Soulé, B. Gra Gedik. "A Catalog of Stream Processing Optimizations." In: *ACM Comput. Surv. Article* 46.46 (2014). DOI: 10.1145/2528412. URL: http://dx.doi.org/10.1145/2528412 (cit. on p. 14).

[10] W. Hummer, B. Satzger, S. Dustdar. "Elastic stream processing in the Cloud." In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 3.5 (2013), pp. 333–345. DOI: 10.1002/widm.1100. URL: http://doi.wiley.com/10.1002/widm.1100 (cit. on p. 51).

[11] R. J. Hyndman, A. B. Koehler. "Another look at measures of forecast accuracy." In: (). DOI: 10.1016/j.ijforecast.2006.03.001 (cit. on p. 21).

[12] D. G. Kendall. *Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain*. Oxford University, England and Princeton University, 1958. URL: https://projecteuclid.org/download/pdf{_}1/euclid.aoms/1177728975 (cit. on p. 20).

[13] T. Kiefer, H. Schön, D. Habich, W. Lehner. "A Query, a Minute: Evaluating Performance Isolation in Cloud Databases." In: Springer, Cham, 2015, pp. 173–187. DOI: 10.1007/978-3-319-15350-6_11. URL: http://link.springer.com/10.1007/978-3-319-15350-6{_}11 (cit. on p. 10).

[14] D. E. Knuth. *The art of computer programming*. Addison-Wesley Pub. Co, 1973, p. 132. ISBN: 9780201038224 (cit. on p. 21).

[15] R. K. Kombi, N. Lumineau, P. Lamarre. "A Preventive Auto-Parallelization Approach for Elastic Stream Processing." In: *Proceedings - International Conference on Distributed Computing Systems* (2017), pp. 1532–1542. DOI: 10.1109/ICDCS.2017.253 (cit. on p. 10).

[16] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, S. Taneja. "Twitter Heron." In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15*. New York, New York, USA: ACM Press, 2015, pp. 239–250. ISBN: 9781450327589. DOI: 10.1145/2723372.2742788. URL: http://dl.acm.org/citation.cfm?doid=2723372.2742788 (cit. on p. 14).

[17] G. Last, M. Penrose. *Lectures on the Poisson Process*. Cambridge University Press, 2017. ISBN: 9781107088016. DOI: 10.1017/9781316104477. URL: https://www.cambridge.org/core/product/identifier/9781316104477/type/book (cit. on p. 21).

[18] X. Liu, A. V. Dastjerdi, R. N. Calheiros, C. Qu, R. Buyya. "A Stepwise Auto-Profiling Method for Performance Optimization of Streaming Applications." In: *ACM Transactions on Autonomous and Adaptive Systems* 0.0 (2017), pp. 1–33 (cit. on p. 10).

[19] D. C. Luckham. *The power of events : an introduction to complex event processing in distributed enterprise systems*. Addison-Wesley, 2002, p. 376. ISBN: 0201727897 (cit. on p. 13).

[20] R. Mayer, B. Koldehofe, K. Rothermel. "Predictable Low-Latency Event Detection With Parallel Complex Event Processing." In: 2.4 (2015), pp. 274–286 (cit. on pp. 14, 15, 23).

[21] R. Mayer, M. A. Tariq, K. Rothermel. "Minimizing Communication Overhead in Window-Based Parallel Complex Event Processing *." In: (2017). DOI: 10.1145/3093742.3093914. URL: ftp://ftp.informatik.uni-stuttgart.de/pub/library/ncstrl.ustuttgart{_}fi/INPROC-2017-33/INPROC-2017-33.pdf (cit. on p. 15).

[22] R. Mayer, M. A. Tariq, K. Rothermel. "Real-Time Batch Scheduling in Data-Parallel Complex Event Processing." In: (). DOI: 10.1145/1235. URL: ftp://ftp.informatik.uni-stuttgart.de/pub/library/ncstrl.ustuttgart{_}fi/TR-2016-04/TR-2016-04.pdf (cit. on pp. 15, 23).

[23] S. McKillup. *Statistics explained : an introductory guide for life scientists*. Cambridge University Press, 2006, p. 267. ISBN: 0521543169 (cit. on p. 20).

[24] K. Patroumpas, T. Sellis. “Window Specification over Data Streams.” In: (). URL: http://dl.ifip.org/db/conf/edbtw/edbtw2006/PatroumpasS06.pdf (cit. on p. 51).

[25] R. Ramakrishnan, J. Gehrke. *Database management systems*. McGraw-Hill, 2000, p. 906. ISBN: 0072322063 (cit. on p. 13).

All links were last followed on February 15, 2018.

## Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Stuttgart, 14.02.18, [illegible]

place, date, signature