Master’s Thesis

Analysis, Benchmarking and Performance Improvement of a FAST Implementation

by

Çağrı Balkesen
bcagri@student.ethz.ch

Supervisors:

Prof. Dr. Nesime Tatbul
Prof. Dr. Timothy Roscoe
Dr. Andrew Baumann

May, 2009
Abstract

Financial Market Data has a trend of ever increasing growth in volumes. Increased volume of data put a burden on processing capabilities of market data processing systems. Financial companies and other market participants need low latency access to recent market data to be very responsive and high throughput to have more processing capability. The only way to cope with increasing volume of data and processing complexity is through improving their systems in the direction of high-throughput and low-latency. In this thesis, we worked on such a real-world production level market data processing system. We thoroughly analyzed and benchmarked this system by setting up a replica of the real-world scenario in our lab. We investigated the problems and bottlenecks in this system in-depth and came with solutions to improve the overall performance of such a system. We experimentally evaluated our improvement idea and compared the results with the current solution. We have shown up to %26 improvement in the overall latency of the system.
Acknowledgements

I was one of those lucky persons who had a chance to work with three great supervisors at the same time. I would like to express profound gratitude to my supervisor Prof. Nesime Tatbul for her helpful and invaluable support, encouragement, supervision and useful suggestions throughout my Master’s studies. Her moral support and continuous guidance enabled me to complete my work successfully. I am also highly thankful to Prof. Timothy Roscoe and Dr. Andrew Baumann for their valuable suggestions and guidance throughout my thesis. It was really my pleasure to work with them.

I would like to specially thank to Dr. Peter Fischer for his support in setting up the systems in our cluster. He was very helpful and patient to support my requests. I also would like to thank Robin Nicholson and Yimin Shen from Credit Suisse for their feedback and help on our progress.

There is also one more person without whom this list would not be complete. I would like to thank Stefan Weber for his extremely helpful guidance in the beginning of my thesis.

Last but not least, I am as ever, especially indebted to my family for their love and support throughout my life.
Contents

1 Introduction ................................................. 5
  1.1 Contributions ........................................ 6
  1.2 Outline of Thesis .................................... 7

2 Background on Data Stream Processing and Financial Markets 8
  2.1 Data Stream Processing ................................ 8
  2.2 How the financial market works ....................... 9
  2.3 FAST Protocol .......................................... 9

3 Analysis and Benchmarking Set-up ........................... 11
  3.1 Description of the Set-up .............................. 11
  3.2 Data Dumps .............................................. 12
  3.3 Load Generator ......................................... 13
  3.4 Other tools .............................................. 15

4 Reproduction of the Previous Measurements ................. 17

5 Overview of Credit Suisse System ......................... 20
  5.1 Data Source Thread .................................... 20
  5.2 Lock-free Queue ....................................... 22
  5.3 Decoding and Filtering Thread ....................... 23
  5.4 Application Thread .................................... 24

6 Scope of Analysis, Benchmarking and Goals ................ 25

7 Complete Analysis and Benchmarking of Queue Performance 26
  7.1 Methodology ............................................ 26
  7.2 Problems ............................................... 27
  7.3 Results ................................................ 30
  7.4 Discussion ............................................. 32

8 Complete Analysis of Latencies ............................. 34
  8.1 Kernel-to-Userspace Latency Measurements .......... 34
    8.1.1 Methodology ...................................... 34
    8.1.2 Results ........................................... 34
    8.1.3 Interrupt Coalescing ............................ 36
  8.2 End-to-End Latency Measurements ..................... 38
    8.2.1 Methodology ...................................... 38
    8.2.2 Overhead of Measurements ...................... 38
    8.2.3 Results ........................................... 39
    8.2.4 Discussion ........................................ 39

9 Improving the Performance of FAST Decoding ............... 42
  9.1 Changes in the implementation ....................... 42
  9.2 Analysis of the Performance of Single Threaded Version .......... 43

10 Conclusions and Future Work ................................ 47
List of Figures

1 Sample FAST Template .................................. 10
2 FAST Encoding Example ................................. 10
3 Load generator problem .................................. 13
4 Load generator corrected ................................. 14
5 Architecture of Credit Suisse FAST Implementation ... 21
6 Producer-consumer scenario with the lock-free queue ... 22
7 Latency Metrics ........................................... 25
8 Queue Iteration Statistics ............................... 27
9 Queue experiments showing patterns and problems ....... 28
10 Breakdown of queue problem ............................ 29
11 Queue length during the run-time of experiment with IAR 25... 30
12 Queue length during the run-time of experiment with IAR 50... 31
13 Queue length during the run-time of experiment with IAR 100... 31
14 Queue length during the run-time of CS dump replay experiment... 32
15 Queue length statistics for the complete runs of experiments... 32
16 CDF of OS Latencies ..................................... 35
17 Statistics for OS Latencies ............................... 35
18 Experiments showing the effect of rx-usecs directly ....... 36
19 CDF of OS Latencies with coalescion set to 0 ............. 37
20 Statistics for OS Latencies with coalescion set to 0 ....... 37
21 End-to-end latency metric for messages .................. 38
22 CDF of End-to-end latencies ............................. 40
23 Statistics for End-to-end latencies ....................... 40
24 CDF of End-to-end latencies with coalescion set to 0 .. 41
25 Statistics for End-to-end latencies with coalescion set to 0 ... 41
26 CDF of End-to-end latencies for single threaded implementation... 44
27 CPU Usage for different IARs ............................ 45
28 Effect of snapshot stream on end-to-end latencies ........ 46
29 End-to-end latency for replay in Credit Suisse version ... 47
30 End-to-end latency for replay in single threaded version ... 48
1 Introduction

Financial Market Data has a trend of ever increasing growth in volumes. More and more instruments are being traded in increasing number of markets and number of market participants are increasing as well. In such a trend of growth in three dimensions, increased volume of transferred data put a burden on processing capabilities of market data processing systems. Financial companies and other market participants want to be both responsive to market changes as soon as possible and be able to handle more data in accordance to the projected data volume growth. As a result, to be very responsive they need low latency access to recent market data and to have more processing capability they need high throughput in their market data processing systems. From individual perspective of market participants, the only way to cope with increasing volume of data and processing complexity is through improving their systems in the direction of high-throughput and low-latency. On the other hand, there is also some collaborative work somewhat at an upper-level to optimize market data processing systems.

Industry participants have already tried to foster standardization of market data exchange. One concrete example is Financial Information Exchange (FIX) Protocol [13], which is industry driven messaging specification for the electronic communication of trade-related messages. It has been developed through the collaboration of banks, broker-dealers, exchanges, industry utilities and associations, institutional investors, and information technology providers from around the world. Although FIX is a standardized way of exchanging trade-related messages, it has shortcomings for increasing volume of data rates. Consequently, FAST (FIX Adapted for Streaming) Protocol [12] is proposed by Market Data Optimization Working Group of FIX Protocol Ltd. in order to decrease size of transferred messages by applying delta encoding compression techniques. At an upper-level, this helps to decrease the amount of exchanged data significantly, but on the other hand it brings some more processing complexity. Because everything is encoded now, they have to be decoded before feeding them into the systems that take care of market data processing such as automated trading systems. This pre-processing is decoding of FAST encoded messages from the data stream. All the specifications of how encoding/decoding is done are described in the FAST Protocol and companies try to have an implementation of FAST Protocol as efficient as possible. Today, the most important bottleneck in front of FAST decoding seems to be the latency problem, although throughput problem may arise in a near future as well. Meanwhile, even today, data peaks may constitute a serious problem from throughput perspective.

On the technical side, achieving low latency and high throughput processing on data streams seem to be conflicting goals at first sight. Usually in data stream processing one can trade off latency for throughput or vice versa. However in case of market data processing, this is not true any more. A financial institution would want to keep up on both directions. At the technical ground, optimizing the software systems as much as possible is the first reasonable choice to try. Common practices with system profiling and iteratively optimizing the source code of the system is the first thing to do. At this point, the system can be improved at a significant level if it did not hit the limits already. Apart from the optimizations to the existing system after it is very mature, we can also
think about different design choices. In case of FAST Protocol, the protocol itself does not tell whether the decoding application has to be multi-threaded or single-threaded for instance. Other than pure software implementation, there are obviously other implementation alternatives as well. Whenever the software implementation hits the limits, one can always think about hardware acceleration according to the context of the application. For instance in case of market data processing, a dominant operation is network processing and this could possibly be accelerated by utilizing special network processors or offloading some part of network processing to programmable network interface card (NIC).

From research point of view, financial market data processing constitutes a very good use case for Data Stream Processing. Main focus of Data Stream Processing is low latency processing of continuously arriving unbounded streams of data. The intersection of two has very interesting problems underneath. Working on a real use case for our research helps us to reveal these problems and to suggest solutions for them from research perspective. Also having an implementation of a production level market data decoding system enables us to investigate characteristics of financial market data streams and their latency requirements in very detail. As such, we understand the problems faced very well and we propose research ideas and directions in order to further improve these systems.

1.1 Contributions

Our main contribution in this thesis has been throughout investigation of the FAST decoding system implementation of Credit Suisse and improving its performance. In [31], the Credit Suisse system has been replicated in our labs at ETH. As a follow up work, we have concentrated on the future work ideas and perceived problems. First we began with improving the load generator that was previously implemented. We assured that we were able to produce the Credit Suisse scenario at our lab by comparing the data dumps from Credit Suisse and our load generator. Afterwards believing that software implementation optimization and improvement are the first things to do, we started working on the software implementation of Credit Suisse. From [31] it was observed that there were certain bottlenecks in the processing pipeline especially in the queue. We ran throughout analysis and benchmarks for the queue performance in the system and revealed the problems. Also for understanding the overall performance of the system we defined an end-to-end latency metric for the whole system, which was missing in [31]. One of the significant contribution was changing the design of the Credit Suisse implementation. The application had two threads and a queue formerly. We changed it to have only one thread without a queue. We integrated the already existing network queueing support from the Operating System to our system. By using our end-to-end latency metric we showed that there is a latency gain on the order of 4-5 $\mu$s. Finally we also revealed some problems in the statistics collection and after correcting them we reproduced experimental measurements of [31]. Last but not least, we believe that this work has been an opener for more problems in the area of financial market data processing and Data Stream Processing for our research group. We anticipate that there will be more collaboration between ETH and Credit Suisse in this direction with more focus on pure Data Stream Processing problems.
1.2 Outline of Thesis

We structured our thesis as follows: In section 2, we give a background on Data Stream Processing and Financial Markets and try to make the reader familiar with the focused topics in the Thesis. In section 3, we describe our set-up. We explain our scenario with system specifications and we briefly mention tools and load generator that we use and how we conduct experiments in our lab. In section 4, we give somewhat more reliable measurements of [31]. We first describe the problems that we revealed in the statistics collection briefly and than we show our results for break-down metrics through the pipeline which are re-production of the measurements of [31] with less variation in numbers. In section 5, we describe the Credit Suisse implementation and give an overview of the system and the source code. In section 7, we show our analysis of the queue performance in the Credit Suisse implementation along with some interesting results. In section 8, we define our end-to-end metric and analyze the full end-to-end latency for each message in the system. In section 9, we present our idea for improving the performance of the FAST decoding system. Lastly we conclude and present some ideas for future work.
2 Background on Data Stream Processing and Financial Markets

2.1 Data Stream Processing

Data Stream Processing is a novel approach in data management. It is proposed as a solution to querying continuously arriving, rapid and possibly unbounded streams of data elements with real-time response requirements. A detailed survey on Data Stream Processing is given in [14]. General purpose Data Stream Management Systems (DSMS) are designed for managing data streams. Up to now, some university prototypes [1, 2, 6, 5] as well as commercial products [28, 29, 8] have emerged. Although Data Stream Processing take its roots from Database community [4], they are radically different from Databases. In Data Stream Processing, volume of data is steeply increasing and continuous queries are commonplace. Also the data is not static anymore as in most Database scenarios. The paradigm becomes no more store-then-query and it shifts to query on the fly. The queries become continuous and data becomes transient. Now the queries are stored and executed as data arrives.

In Data Stream Processing, there is a new type which is “Stream”. A Stream is an infinite sequence of (tuple, timestamp) pairs continuously arriving from a source in real-time. Each Data Stream may have a different schema as in tables in Databases. A Data Stream Management System may have several input data streams on which tuples arrive to the DSMS. Users submit their queries, either continuous or ad-hoc to the DSMS. As tuples arrive on the Data Stream, queries are evaluated and the results are given back to the user from the DSMS. User queries typically contain Data Streaming operators. These operators have similarities to their relational counterparts. Some of the important operators are filter, union, aggregate and join which respectively have similarities to relational select, union, aggregate and join operators of Databases.

As application area of Data Stream Processing, we can easily talk about financial analysis, algorithmic trading and monitoring applications where the volume of data is gigantic and the latency is the key issue. Queries typically involve moving averages, comparisons between different groups and aggregations among different sources. Financial markets constitute one of the important use case area of Data Stream Processing applications. For instance financial market feed processing is one of the prominent applications where the data rates are very high and the response time should be as small as possible. Capital markets companies that use high volume real-time exchange data has to make sure that their automated trading systems as well as other consumers receive the data with lowest latency and without losing any important data. Today, latency is more in the focus. However with the projected rate of data growth, processing throughput will also be a concern for financial companies. On the other hand, these companies usually receive market data from several exchanges and they have a lot of traders. Traders are usually interested in certain subset of the ticker feeds. Receiving data from several exchanges with variety of interests from traders, market data processing systems should employ smart routing strategies to traders and filter non-relevant data as early as possible. Meanwhile a trader or a trading system is usually interested in data feeds from several exchanges at the same time with the focus of correlation of a certain ticker on
both of them. All these things highlight the functionalities that operators of a Data Stream Processing system provide. For instance filter operator can filter out non-relevant data, join can correlate data from different streams, union can combine data from different streams and so on. Implementing a trading algorithm or processing pipeline becomes no harder than combining set of operators to achieve their goal. By using a high-performant Data Stream Management System, capital markets companies can easily build flexible applications which can cope with the increasing data rates and meet low latency requirements. The flexibility provides them with the ability to change their applications or algorithms quickly and have the advantage in processing market data.

2.2 How the financial market works

We would like to give a brief description of how the markets work. Usually a market participant sends an order to the exchange. This order may either be a sell or a buy request. The price that the participant wants to buy is called bid and the price that he or she wants to sell is called ask. The exchange is mainly responsible for keeping state of the buy and sell requests sent so far. It keeps an OrderBook which stores the orders given for each instrument. OrderBook keeps general information about the instrument such as trade volume so far. In an order-driven market, when bid and ask orders match at the same price, then the orders are automatically executed by the exchange and trade messages are sent to both parties. On the other hand, in quote-driven markets, a person who is the market maker sets the prices. What users subscribed to market feeds receive has two variations: In normal operation at a periodic interval exchange sends a snapshot of the OrderBook. It contains limited number of entries, lets say top 10 bids and asks for the instrument. In second variation, whenever an order is placed an update message is sent out to subscribers, which is called delta messages. This has a lower latency in comparison to the snapshot stream since the update is propagated to the clients as soon as it takes place. More detailed information about how the exchanges work and order books are managed are given in [31] and [24].

2.3 FAST Protocol

We would like to give some background on FIX Adapted for STreaming (FAST) protocol in this subsection. The FAST Protocol tries to decrease the size of transferred FIX [13] messages in order to cope with the increasing data volumes. One of the main things it exploits is eliminating message description from the message itself. Templates which describe the message contents are exchanged by other means and this enables both parties to know the layout of the messages in advance given their template ids. Along with this idea, field encoding and transfer encoding compression techniques allows FAST to decrease size of the sent messages and address the ever increasing growth trend in market data volumes.

In Field Encoding removes some redundant information by utilizing the template concept. Multiple messages are put into the same logical frame (which is a 1500B UDP datagram in case of Eurex) and the template defines the data
format, field instructions and their optimizations. As a result of being in the same frame, there happens to be similarities among consecutive messages. For instance there is a field instruction “Copy”, which designates that the value is an exact copy of the data from the previous message. Similarly, “Increment” specifies that the previous value is just incremented. A special field instruction “Delta” specifies that only the difference from the previous one should be sent. “Default” specifies the default value described in the template and “None” specifies all the data on each message should be sent. Message Templates describe the layout of each message type and define which field instruction is used on each field. The following shows a sample message template in XML format:

```
<template name="ExampleOrder">
  <messageRef name="NewOrderSingle">
    <string name="BeginStr" constant value="FIX.4.4"/>
    <u32 name="SeqNum" increment/>
    <string name="SenderId" copy/>
    <string name="SendingTime" delta/>
    <decimal name="Price" delta/>
    <string name="Symbol" copy/>
  </messageRef>
</template>
```

Figure 1: Sample FAST Template

Figure 2 show the application of field instructions specified in the template to some example data. Constant operation puts nothing since the value is defined in the template already. Increment operator only puts the first value. Delta computes the difference from the previous value and puts only this difference. Lastly, Copy operator puts nothing if the value is same as the previous one. Further detailed information about the FAST Protocol could be found in [12].

![FIGURE 2: FAST Encoding Example](image)

```
<table>
<thead>
<tr>
<th>BeginStr</th>
<th>SeqNum</th>
<th>SenderID</th>
<th>SendingTime</th>
<th>Price</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>8=FIX.4.4</td>
<td>34=10000</td>
<td>49=CLIENT1</td>
<td>52=20060126-3:06:58</td>
<td>100</td>
<td>1200</td>
</tr>
<tr>
<td>8=FIX.4.4</td>
<td>34=10001</td>
<td>49=CLIENT1</td>
<td>52=20060126-3:06:58</td>
<td>200</td>
<td>1210</td>
</tr>
</tbody>
</table>

Original size 71 bytes

<table>
<thead>
<tr>
<th>Constant</th>
<th>Increment</th>
<th>Copy</th>
<th>Delta</th>
<th>Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>[10000]</td>
<td>[CLIENT1]</td>
<td>[20060126-13:06:58]</td>
<td>100</td>
<td>1200</td>
</tr>
<tr>
<td>[200]</td>
<td>[10]</td>
<td>[300]</td>
<td>[20]</td>
<td>[BAR2]</td>
</tr>
</tbody>
</table>
```

Figure 2: FAST Encoding Example
3 Analysis and Benchmarking Set-up

In [31], the Credit Suisse infrastructure was replicated in our lab at ETH to be able to conduct similar experiments to the real world scenario as close as possible. In this thesis, there have been some differences to the previous set-up. In this section, we briefly mention them as well as describing the already existing set-up. We also describe the tools that we have used while conducting the benchmarks.

3.1 Description of the Set-up

In the real scenario, stock exchange normally sends data to subscribers. Since we do not have a direct access to the exchange, we thought of simulating the exchange with a stand-alone machine which only sends data to the subscribers. We call this machine the sender. On the other hand, we had a second machine which simulates the Credit Suisse application. This is the machine on which FAST decoding software runs. We call this machine the receiver.

The sender basically runs our data load generator application which we describe in the next subsection. Its main activity is reading the data dumps provided by Credit Suisse and sending them according to the parameters given to it. It can either send the data dump by replaying it or speeding it up. It is also capable of sending the same data dump with constant inter arrival times.

The receiver is a bit more interesting for us. It runs the main application that we are benchmarking. All the decoding of FAST streams are done on this machine. It takes product symbols which we are interested in from the command line. During the runtime, the application collects various statistics that we will describe in the following sections. At the end of the runtime, the statistics are written to files to be analyzed afterwards.

The sender and receiver are both from our Systems Group cluster at ETH. Although they are connected to the same network, it was previously observed that there had been datagram losses when using the existing network connection. So another exclusive network connection between these machines over their eth1 interface was configured and the multicast group settings were also configured for these two machines exclusively. After these settings, we did not experience any datagram loss.

Specifications of Machines Previously, the used machines were shrek10 and incredibles02 from our cluster. They had the following configurations:

- **shrek10:**
  - 2 x CPU - AMD Opteron(tm) Processor 248, 2.2Ghz
  - 6GB memory, 1024KB cache
  - NetXtreme Gigabit Ethernet, Broadcom Corporation, driver: tg3

- **incredibles02:**
  - Same configuration except memory which is 8GB
Later in the experiments, we experienced very weird situations with the load generator and time measurements in the receiver. We revealed this problem to be due to the AMD Opteron Processors that we were using and decided to change the machines that we were using. Briefly, the problem sourced from the CPU timestamp counter synchronization problem of AMD processors [15, 3]. So we thought that having a CPU with reliable timings would enable us to have more reliable measurements out of our benchmarks. We will describe this problem later in detail but before that we would like to give the specifications of the machines we used for the benchmarking thereafter.

- **mozart12:**
  - 4 x CPU - Intel(R) Xeon(R), 2.83Ghz
  - 8GB memory, 6MB cache
  - Intel Corporation 82566DM-2 Gigabit Ethernet, driver: e1000e

- **mozart13:**
  - Same configuration

Meanwhile, we want to give specifications of the machine used at Credit Suisse too: It is an Intel Xeon machine with 3.2Ghz CPU and 2GB main memory.

### 3.2 Data Dumps

Credit Suisse provided us with a set of binary data dumps that was taken from live Eurex streams. One of the data dumps we mainly focused on was recorded on October 14, 2008, which was a quite busy day. Meanwhile this stream included big German companies like BMW, Siemens and so on; so it was a very busy stream. The data dumps included binary dump of FAST encoded messages from the exchange stream. We were also provided with some meta-data supporting the data dumps. In the meta-data files, we had information about the contents of the binary dump. For instance, number of messages in a datagram and the timestamp of the datagram are recorded. Also template id of each message along with message size in bytes for all the messages in a datagram are provided through these meta-data files.

Before doing the experiments, we ran some static analysis through these binary dumps and meta-data files. Here are some of the important findings from those static analysis:

- Usually the delta stream has very big variations in number of datagrams per time unit, on the other hand the snapshot stream has a very constant frequency over the run-time.
- If we average per minute, we can see up to 6000 deltas per second; if we average per second, we can see up to 16000 deltas per second.
- Most dominant messages in the delta stream are Version (%11), Delta(%85) and Reset(%4).
- Average datagram size : 103 Bytes, average data rate: 223 KB/s.
- Observed mean inter arrival rate (IAR) of datagrams : 0.667 ms.
3.3 Load Generator

In this subsection, we would like to elaborate on our load generator. To be able to conduct reliable experiments and to get reliable results out of our benchmarks, we believe that we need to have a very precise load generator. The load generator should replay the given data dump as if it is being sent from the exchange. On the other hand, by having a good load generator we can simulate various situations and run the bottleneck situations again and again to reveal the problems with our system. With this in mind, a simple load generator was previously implemented. It basically looks at timestamps of datagrams from the meta-data dump and waits for the difference between the timestamps and then after this time it reads and sends the next datagram from the binary dump. It also supports speeding up and slowing down of stream data rates by scaling the timestamp differences with a multiplier parameter. Lastly, it also has options for sending the datagrams at a given constant rate regardless of the timestamps.

As a part of analysis of our work, we tried to understand the behavior of our load generator in detail. To achieve this aim, we implemented a datagram capturing program. The special side of this program was that we captured datagram packets with kernel level timestamps and hence we had very precise timings for when the datagrams arrive. Kernel level timestamps are supported in Linux by enabling SO_TIMESTAMP socket option. Here is an excerpt from the man page:

If the SO_TIMESTAMP option is enabled on a SOCK_DGRAM socket, the recvmsg(2) call will return a timestamp corresponding to when the datagram was received. The msg_control field in the msghdr structure points to a buffer that contains a cmsghdr structure followed by a struct timeval. The cmsghdr fields have the following values: cmsg_len = sizeof(struct timeval), cmsg_level = SOL_SOCKET and cmsg_type = SCM_TIMESTAMP (or SO_TIMESTAMP).

![Figure 3: Load generator problem](image-url)
Figure 4: Load generator corrected

Using this feature, we implemented our datagram capturing program to receive the datagrams along with their timestamps from the multicast stream. When the capturing program terminates, it writes all timestamps to a file for analyzing later. Finally, in the scenario described in 3.1, we ran our capturing program on the receiver side instead of the Credit Suisse FAST decoding application. While we were using the previous machines described in 3.1, we observed very strange behaviors. For instance it was taking half the time of the actual trace for the load generator to send the dump data. Clearly, the load generator was running 2 times faster than it should. For reference one can see Figure 3. In this figure, blue represents load generator trace datagram frequencies averaged over 1 second intervals, whereas red represents the static datagram frequencies measured from the Credit Suisse dump. Apparently for the experiment duration, load generator sends same amount of data in around half the time. We delved into the source of the problem and revealed that it was due to the AMD processors that we were using at that time [15, 3]. After changing the machines, we also changed our load generator to use processor cycle counter for its internal timings which was previously neither used and nor reliable on AMD processors. Afterwards we repeated our experiments and confirmed the correct functioning of our load generator which can be seen in Figure 4. On this figure, blue represents datagram frequencies averaged over 1 second for the static Credit Suisse datagram dump whereas red represents datagram frequencies averaged over 1 second for the sent data from our load generator. Although there is a few seconds lagging, we can still claim that our load generator is successful in reproducing the replay of the real Eurex stream.

Findings from studying the behavior of our load generator resulted in changing the machines we were using. In [31], the measurements were taken on the previous machines. Having a problem with the timings on these machines, we thought that these measurements should have been repeated. From this point of view, we repeated these measurements and acquired results with less variations. We will present our results in section 4.
3.4 Other tools

Several other tools, some of which implemented by us, helped us during our analysis and benchmarking studies. For time instrumentation we have chosen to use ACE High Resolution Timer from the ACE Library [9], which was also being used by the Credit Suisse code. It is shortly described as follows:

ACE_High_Res_Timer is a high resolution timer class wrapper that encapsulates OS-specific high-resolution timers, such as those found on Solaris, AIX, Win32/Pentium, and VxWorks.

The high resolution timer class has a method called \texttt{elapsed\_time()} which gets the elapsed time between calls to methods \texttt{start()} and \texttt{stop()}. In order to measure the time spent by a given code snippet, one can place \texttt{start()} and \texttt{stop()} around the piece of code and elapsed time in nanoseconds could be retrieved by a call to \texttt{elapsed\_time()} method afterwards. \texttt{start()} and \texttt{stop()} methods get the processors current cycle count using read time stamp counter instruction \texttt{(rdtsc)}. Although ACE library assures portability issues among different CPU architectures and Operating Systems, it also has an overhead of extra 2 to 3 more function calls. In the beginning, high resolution timer constructor gets the \texttt{scale\_factor} of the CPU from Mhz value found in \texttt{/proc/cpuinfo} in Linux. Also some additional work to calibrate this frequency is done in order to make sure that the found frequency for the CPU is correct. In CPU architectures where the CPU frequency is changed dynamically with respect to the CPU usage and power modes makes this method unreliable. For instance what we experienced with the load generator was exactly due to this issue. The scale factor is determined in the beginning on the AMD machine and afterwards the CPU scaled up its frequency because of the intensive computing requirement. Consequently, the CPU scale factor found before became invalid and hence the following time measurements were wrong.

During our experiments and benchmarks, we also wanted to understand what percent of the overall time is spent in which locations in the code. To achieve this aim, OProfile [23], which is a low overhead system profiler for Linux, was an excellent tool. It enabled us to profile the system even with kernel time spent during the execution. It provided profiling with separate threads and separate CPUs, meaning that we were able to differentiate between the threads and hence were able to understand break-down measurements for each thread and CPU separately. It is designed as a Linux kernel module and runs as a daemon with root privileges and it has very low overhead. It uses hardware performance counters of the CPU to keep various statistics. But usually we used only time-spent based statistics during our profiling sessions.

In order to assure that our load generator is functioning correctly, we needed a mechanism to compare the trace of the Credit Suisse dump with the one generated by our load generator. We implemented a simple datagram capturing program using the kernel level timestamp retrieval method described in section 3.3. Our capturing program listens to the multicast streams that our load generator sends the datagrams and captures the datagrams with the kernel level timestamps. After a certain period of time, we shutdown the sender and the capturing program dumps the received trace to a file. Later we analyzed these dumps for datagram frequencies over certain time intervals and plotted...
them to compare visually. Some instances of these figures can be seen on Figure 3 and 4.

During single threaded version experiments, we needed mechanisms to make sure that there were no datagram losses. Other than looking at packet counters by using commands like `ifconfig` and `netstat`, we needed more introspective methods. Load generator normally takes two static streams which are delta and snapshot. It sends them by looking at the timestamps in the meta-data files. On the receiver side delta and snapshot datagrams are interwoven. Inside the normal decoding/filtering thread, we recorded the sizes of each received datagram to be dumped to a file after shutdown. And we implemented a comparator to see if the dumped datagram sizes are a combination of the static delta and snapshot datagram sizes of the original Credit Suisse dump. By using this comparator program, we made sure that there were certainly no datagram losses during the execution of the single threaded FAST implementation.

Another tool that emerged was used for understanding the performance of the `gettimeofday()` calls. We call this tool `gettimeofday()` benchmarker. It is a very simple tool which runs `gettimeofday()` several times which is given as a parameter and it separately measures the elapsed time using ACE High Resolution Timer, `gettimeofday()` itself and custom `rdtsc` timer implementation by us. In the end it outputs some statistics for each of the executions.

Last but not least, Matlab [20] has also been a very useful tool during our studies. We used it to efficiently do some statistical computations and to create nicely formatted plots. Also we used CMake [7] build system for the build configurations and usual compilation tasks.
4 Reproduction of the Previous Measurements

In [31], very detailed measurements for the metrics defined through the pipeline were produced. This initial work helped us to understand the Credit Suisse system in detail and the measurements revealed possible bottlenecks in the system. This work has been very motivating for us to pursue this Master’s Thesis on the same project as a follow-up. We began our investigations and analysis with trying to reproduce the same measurements of the previous work. We used the same machines and set-up and eventually we were successful to reproduce exactly the same measurements mentioned in [31]. Later during our further analysis, we revealed some problems with the system and with the instrumentation methods. One of the problems was the AMD processor timestamp cycle counter problem which we already mentioned in section 3.3. Another problem that we observed was due to the statistics collection mechanism. At certain times during the runtime, due to more memory space requirements for statistics collection the system needed to reallocate memory. Later in the runtime, it was taking a lot of time and it caused the measurements to be somehow unreliable. We saw its effects in queue analysis and throughout investigation and the explanation of the problem is given in section 7.2.

After observing problems with the system and the instrumentation, we thought that reproducing the measurements of the previous work would give us more reliable results. As we changed the used machines, we believe that we should not make one-to-one comparison of the measurements because of slightly different numbers observed on different machines. We also believe that we should take these measurements as our base case in this work from now on and in other future works.

<table>
<thead>
<tr>
<th>Previous Measurements</th>
<th>New Measurements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>Mean</td>
</tr>
<tr>
<td>Median</td>
<td>Median</td>
</tr>
<tr>
<td>Std. Dev.</td>
<td>Std. Dev.</td>
</tr>
<tr>
<td>Min</td>
<td>Min</td>
</tr>
</tbody>
</table>

Some commonality emerges in all of the new measurements. For instance the average times are approximately around 100 nanoseconds better than the previous measurements. Certainly there is also the effect of the difference of the machines in these measurements. So we cannot clearly say that this improvement is due to our reliable measurements. On the other hand, a big difference catches our attention in standard deviations. Nearly in all measurements, the standard deviations significantly drop from 66-70 microseconds to 300-2000 nanoseconds.
<table>
<thead>
<tr>
<th>Statistic</th>
<th>Value (ns)</th>
<th>Statistic</th>
<th>Value (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Previous Measurements</td>
<td>New Measurements</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>delta decodeTime(ns)</strong></td>
<td><strong>delta decodeTime(ns)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mean</td>
<td>770.08</td>
<td>Mean</td>
<td>631.41</td>
</tr>
<tr>
<td>Median</td>
<td>497</td>
<td>Median</td>
<td>705</td>
</tr>
<tr>
<td>Std. Dev.</td>
<td>66714.76</td>
<td>Std. Dev.</td>
<td>481.75</td>
</tr>
<tr>
<td>Min</td>
<td>281</td>
<td>Min</td>
<td>150</td>
</tr>
<tr>
<td><strong>filtered delta libTime (ns)</strong></td>
<td><strong>filtered delta libTime (ns)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mean</td>
<td>830.95</td>
<td>Mean</td>
<td>680.95</td>
</tr>
<tr>
<td>Median</td>
<td>593</td>
<td>Median</td>
<td>504</td>
</tr>
<tr>
<td>Std. Dev.</td>
<td>68708.61</td>
<td>Std. Dev.</td>
<td>318.36</td>
</tr>
<tr>
<td>Min</td>
<td>405</td>
<td>Min</td>
<td>222</td>
</tr>
<tr>
<td><strong>filtered delta decodeTime (ns)</strong></td>
<td><strong>filtered delta decodeTime (ns)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mean</td>
<td>649.74</td>
<td>Mean</td>
<td>523.2</td>
</tr>
<tr>
<td>Median</td>
<td>439</td>
<td>Median</td>
<td>372</td>
</tr>
<tr>
<td>Std. Dev.</td>
<td>68718.61</td>
<td>Std. Dev.</td>
<td>279.45</td>
</tr>
<tr>
<td>Min</td>
<td>281</td>
<td>Min</td>
<td>150</td>
</tr>
</tbody>
</table>
This certainly shows us that there was a big variation in measurements in the previous experiments due to several artifacts as mentioned before. Also due to the decrease in the variations and decrease of peak values, the median values drop around 100-200 nanoseconds.

Lastly, for the sake of completeness we would like to present average values from the measurements side-by-side as a summary. Again, it is not very reasonable to compare them one by one but our purpose is to give information on our new measurements.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Previous Measurement</th>
<th>New Measurement</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataSourceTime</td>
<td>1936.36 ns</td>
<td>1844.04 ns</td>
</tr>
<tr>
<td>queueTime</td>
<td>3.5 µs</td>
<td>5 µs</td>
</tr>
<tr>
<td>datagramTime</td>
<td>1936.36 ns</td>
<td>1481.63 ns</td>
</tr>
<tr>
<td>avg. version time</td>
<td>280.91 ns</td>
<td>382.208 ns</td>
</tr>
<tr>
<td>avg. reset time</td>
<td>172.676 ns</td>
<td>142.7 ns</td>
</tr>
<tr>
<td>avg. delta time</td>
<td>975.556 ns</td>
<td>744.49 ns</td>
</tr>
<tr>
<td>avg. snapshot time</td>
<td>681.086 ns</td>
<td>275.876 ns</td>
</tr>
<tr>
<td>avg. app time</td>
<td>20.2458 ns</td>
<td>21.2 ns</td>
</tr>
</tbody>
</table>

Table 1: Summary of average values from the measurements
5 Overview of Credit Suisse System

The Credit Suisse FAST System is implemented as a library to be reused in several locations. Its eventual goal is providing the most recent market data to the trading application logic. In this thesis, we mainly worked on this software implementation. We focused to the major goal of this system which is latency that the application logic observed for each update in the exchange. Since this library is used in many locations in the bigger picture of the Credit Suisse market data processing software systems, optimizing this system is really important for Credit Suisse. Before going on with the details of the components of this system, we would like to present the general architecture of the FAST implementation at Credit Suisse. The Figure 5 shows the architecture as a blueprint of the system. The system is implemented as a multi-threaded application. It mainly consists of 3 different threads running concurrently in the system. The first thread is the so-called Data Source thread, second thread is the Decoding and Filtering thread and the last thread is the Application thread. Other than the threads the Lock-free Queue sitting between the first two threads also plays an important role. Now we would like to describe the functionality of components in more detail.

5.1 Data Source Thread

Data Source thread is the initial point where data from the Eurex Broadcast Solution (EBS) [10] enters the system. Basically network packets are received from the UDP connections provided by Eurex. The EBS runs in a mode called live/live configuration. This is a mode designed to compensate for the unreliability of the underlying UDP protocol. The idea is sending the datagrams over two UDP streams and try to compensate for losses on one of the streams. It also comes with a processing overhead. After receiving the datagrams from different streams, the system should check whether the datagrams are already seen. By the specification [11], every first version message of each datagram contains a sequence number. In Credit Suisse implementation a map data structure is used for book-keeping the already seen sequence numbers. Whenever a datagram is received on one of the streams, its version message is decoded on the Data Source thread and than it is checked from the map data structure whether it already exists. If it exists, it is simply ignored and otherwise it is put into the lock-free queue for further processing. The main task of this thread becomes getting datagrams out of the network, eliminating duplicate datagrams and pushing into the queue if the datagram is not a duplicate. This functionality is simply depicted in the architecture figure. Other than these tasks, Data Source thread also manages subscriptions to products. It has an internal socket for handling these types of requests. It is designed in this way to handle subscriptions from different threads. When a user subscribes to a product, a subscription message is sent to this internal socket and these messages are also processed as market updates. The main function being executed in a tight loop in this thread is DataSource::eurexReaderThread(). It sets up sockets which are command and request sockets and then calls DataSource::read() in a loop. DataSource::read() first instantiates a fast::Decoder(versionTemplate). It also keeps the std::map<uint,uint***> sourceSequences variable for keeping track of seqNums and eventually eliminating the duplicates. After the ini-
Figure 5: Architecture of Credit Suisse FAST Implementation
tialization stuff, this function runs the following inside an infinite loop:

- does a `select()` on sockets
- if from request socket
  - then it is a request to the socket multiplexer to add or remove a socket
- else it is a multicast socket
  - reads the datagram from the socket
  - calls `decoder.decode(decoder, version)`
  - does version checking from the bitmap of the map structure
  - puts the datagram into the lock-free queue
- continues looping

5.2 Lock-free Queue

The situation in Credit Suisse FAST implementation is basically a producer-consumer scenario. A Lock-free queue is placed between the producer - Data Source thread and the consumer - Filtering & Decoding thread. Figure 6 shows the interactions of the two threads with the lock-free queue. Lock-free queue provides concurrent access to the shared elements of the queue without a need for semaphores, mutexes or critical sections protected by operating system managed locks. As a result of unnecessity of these lock-based methods, efficient access to the queue is possible. Given the high frequency access pattern to the queue in the Credit Suisse implementation, there remains a little overhead related to the queue. Some general techniques and ideas for implementing a lock-free queue is also mentioned elsewhere [21, 30, 16]. Lock-free data structures usually employ a low level operation called CAS (Compare-And-Swap) [32]. This is a special instruction which atomically compares a given value with a memory location and if they are equal it changes the memory location with a given value. Syntactically it is similar to a call like the following: `variable.CAS(expectedValue, newValue)`. It can be used as a boolean expression to do some operation on the lock-free data structure, the condition expression becomes something similar to “If I am the person who changes the variable from x to y.” Other than the CAS operation, lock-free queue also involves some memory fencing operations to order the execution of instructions changing the memory locations. But we do not want to go into low-level details of the lock-free queue implementation in this report.

Figure 6: Producer-consumer scenario with the lock-free queue
Some of the important methods of the lock-free queue class of Credit Suisse implementation are the following:

- `Queue<T>::getSolitaryAll()` : drains the queue for all elements. It blocks if the queue is empty until at least one element is put in the queue. Lastly, it returns a `NodeGroup` structure with a number of elements to iterate through.

- `Queue<T>::getConcurrent()` : gets an element from the queue, can be safely called by many threads at the same time. If there are no elements in the queue, it blocks.

- `Queue<T>::getSolitary()` : gets an element from the queue. It is only allowed to be called by one thread at the same time, but it is faster than `getConcurrent()`.

- `Queue<T>::putConcurrent(T*)` : puts an element on the queue. It can be safely called by many threads at the same time.

- `empty()`, `finish()`, `restart()` and `shutdown()` as well as some variations of the above. These are the utility methods for the queue. Please note that the lock-free queue does not provide a `size()` method.

5.3 Decoding and Filtering Thread

Decoding in Credit Suisse system is mainly done by `decoder.processStream()` method. Credit Suisse decoder is composed of two parts: One of them is the FAST specific implementation which takes care of all FAST decoding stuff. It is very re-usable and does not depend on the exchange. As a result it is used for different exchanges such as SWX. The other part that we have contains Eurex specific stuff. In the main loop of decoding and filtering thread, `processStream()` method is called by giving the contents of the received datagram as input. `processStream()` basically loops for the all messages in the datagram and decodes them message by message with a call to `decode()` method of the corresponding message parser. At the beginning the template id is decoded from the message and with this id corresponding message parser for the message type is determined. This determined message parser knows the details or schema of the message as it is already hard-coded in the implementation. Finally, the message parser decodes the fields of the message by using generic FAST operations of the FAST decoder.

Second phase of the processing on this thread is filtering. Clients usually subscribe to products by their symbols. Upon subscription a message is sent to Data Source’s internal socket and it is processed as a normal datagram. After that time related information about the subscription (matching products and so on) will be passed to specific methods of the `DataClient`. For instance for a delta message `processOrderBookDelta()` is called. This is where the actual filtering is executed. When a client subscribes to a product by symbol, all the IDs of contracts of that product are added to the list of IDs that we watch. But strategies do not have a corresponding ID and for them we need to keep track of the product names. Because of this reason a list of watched products are also kept. In `process()` method of `DataClient`, `getTracking()` method is called to
find whether there exists a tracking for the given instrument. Filtering basically takes place as follows: If the instrument is a contract; then using its ID, list of IDs we are watching is searched. Otherwise if it is a strategy, symbol name is read from the strategy message and it is searched in watched product names. If the search is successful, it means the message is not filtered. Otherwise it is filtered out. This is basically how the filtering takes place in the Credit Suisse FAST implementation. More detailed information is given in [31].

5.4 Application Thread

Normally subscriptions have a callback to be executed after the order book is up to date again. These are entry points for the actual application logic. By calling appropriate application methods or putting its code inside the callbacks, applications get integrated with the decoding system. Furthermore, in the Credit Suisse implementation there is a priority-queue for the application callbacks in order not to spend a lot of time for the application logic. So basically application logic also runs in a different thread than filtering and decoding. In our set-up at ETH, we were not provided with the source code of this part. So in all the measurements that we do, this part is simply neglected.
6 Scope of Analysis, Benchmarking and Goals

This thesis project on Credit Suisse FAST System has been a follow-up project to the previous work done by another Master student Stefan Weber [31]. He did a lot of work to set up a similar environment in our lab which is very close to the situation found in Credit Suisse. In our lab, he was able to run the same scenario as in Credit Suisse. He did a lot of measurements in the system to understand where the bottleneck in the processing pipeline is and he came out with very interesting break-down of latency numbers in the processing pipeline. His results were the motivating points for this follow-up Thesis on the same project. Also the raised questions and proposed future work ideas from his work constituted the starting point of this Thesis. He separated latency metrics through the pipeline as follows in the figure.

![Figure 7: Latency Metrics (courtesy of Stefan Weber)](image)

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataSourceTime</td>
<td>2 μs</td>
</tr>
<tr>
<td>queueTime</td>
<td>5 μs</td>
</tr>
<tr>
<td>datagramTime</td>
<td>2 μs</td>
</tr>
</tbody>
</table>

From these metrics; **queueTime**, **libTime** and **datagramTime** are already used in original Credit Suisse code. They respectively measure how long a datagram spends in the queue, how long decoding-filtering-order book management of each message takes and how long the entire processing of a datagram takes. In addition to these metrics; **dataSourceTime**, **decodeTime** metrics are introduced. They respectively measure time spent in DataSource for receiving the datagram and time spent for actual decoding of a message. Both according to Stefan’s report and our observations, the following are average values for some of the metrics at a very slow data rate (5000 datagrams/sec):

- **dataSourceTime** 2 μs
- **queueTime** 5 μs
- **datagramTime** 2 μs

A significant fact here is that **queueTime** is really huge. This is more than half of the total time required in the processing pipeline of a single datagram. We thought that investigating why the queue is taking that much time would be a good point to start. We studied the queue behavior in detail. We will present our results and findings in the next section about the queue.

On the other hand, having several metrics through the pipeline does not give us the overall performance of the system in terms of latency. Some of the metrics such as **queueTime** is per datagram and some of them such as **decodeTime** is per message. As a result, they are not uniform through the pipeline and one can not sum them up to find the overall latency through the pipeline. As the second step,
we thought that instead of looking at separate metrics through the pipeline we should understand end-to-end latency through the pipeline. But the question mark in Stefan’s metrics, which corresponds to the sum of hardware latency coming from the NIC and Operating System kernel latency was problematic to understand. Going a bit further, we figured out that we could find the latency coming from the OS kernel. As a result, we measured the end-to-end latency from the time that Operating System first sees the datagram to the time when the processing of the last message is complete and the order book is updated. We will show our results along with plots of cumulative distribution of latencies in section 8.2.

We also wanted to know what share the Operating System has in the overall measured end-to-end latency. In section 8.1, we show our measurements of OS kernel latency; the time that a datagram takes from the first time OS kernel sees the datagram to the time that DataSource component in userspace receives the datagram.

Lastly, we propose an idea for improving the overall performance of the FAST decoding system at Credit Suisse. Our motivations for this idea are low utilization of the lock-free queue and the OS featured kernel level packet queues. We investigated this idea and did some changes in the original Credit Suisse code to realize our idea. Afterwards we ran our benchmarks on this implementation. We present our methodology and results in section 9.

7 Complete Analysis and Benchmarking of Queue Performance

One of the obvious bottlenecks is the queue that is sitting between Data Source (datagram reader) thread and datagram decoding/filtering thread. This queue is implemented as a lock-free queue and as a result access to the queue does not require acquisition of expensive OS structures such as locks and mutexes. Lock-free queues are extensively used in producer-consumer scenarios with a high frequency access patterns. To our knowledge, current implementation has a better performance in comparison to the previous version. However, it still seems to be a problem according to both our recent measurements and measurements of Stefan. Although datagrams spend most of their time in the queue, we wanted to understand how the queue utilized in terms of number of elements waiting in the queue at a time. Our basic statistics at this step was number of datagrams in the queue throughout the execution of the software.

7.1 Methodology

Since the queue is a lock-free queue, it is really very hard to understand how many elements there are in the queue at an instant. It does not provide any method to get the number of elements. As a workaround, we are keeping our own counters for the current number of items in the queue. This counter is kept in eurexThread() which is doing datagram decoding and filtering. Basically, this thread runs in a tight loop for extracting datagrams from the queue if there
are any. If there are no items in the queue, this thread blocks on the following call:

```cpp
lockfree::NodeGroup nodes = dataQueue.getSolitaryAll();
```

If there are items, then this call returns a `NodeGroup`, a list of current items from the queue. Afterwards, the thread loops for all `Nodes` from returned `NodeSet`. Each `Node` in the queue represents a datagram enqueued to the queue. Consequently, when this call returns we get all the datagrams found in the queue as a `NodeGroup` and by counting them as we process, we easily understand number of datagrams found in the queue at that time. Meanwhile, before starting the processing of the first `Node(Datagram)` of the `NodeGroup` we record the time using `ACE_High_Res_Timer`. To our understanding, we are only processing the datagram list returned by the last call to `getSolitaryAll()` method. So if there are datagrams appended to the queue during the current processing, they will be processed after the next call of the `getSolitaryAll()` method. Along with queue size statistics, we are also collecting some execution time statistics for each iteration of the loop of processing datagrams popped out of the queue. The code snippet below, although a bit simplified version of the source code will summarize the statistics we are collecting more precisely:

![Figure 8: Queue Iteration Statistics](image)

We will go over the detailed results in the Results subsection, but before that we would like to mention some of the problems and exceptional situations that we came across during the experiments and measurements in the next subsection.

### 7.2 Problems

When we ran the experiments with various datagram rates, we noticed that sometimes queue lengths were going extremely up. Sometimes we observed queue lengths over 4000 with the inter arrival rate of 25 µs, which is around 40K datagrams/second. For reference one can see the Figure 9. Another very interesting fact was that these peak queue lengths were occurring at around the same time regardless of how many times we repeat the experiment. Also with different datagram rates, very large queue sizes still persisted. Since the datagram rates were different, these peaks were occurring at different times but one could observe their similarity by looking at the plots.

Consider Figure 9. On top, we see two plots showing the queue length over the run-time of the experiment with inter arrival rate of 25 µs. The experiment
Figure 9: Queue experiments showing patterns and problems. Experiments with IAR 25 and 50 $\mu$s. Each experiment is repeated to see the similarity of occurring patterns of the problem.
with the same inter arrival rate is repeated twice to show the similarity of occurring maximum queue lengths. At the bottom, similar experiments for inter arrival rate of 50 \( \mu s \) are shown. Having a look at the plots one can see the similarity in the beginning of the timeline between 0 and 50 seconds. Moreover, on top, there are peaks at around 50-60, 80, 100, 140-150 on both figures. And at the bottom, we see these similar peaks at around 50, 100, 120, 150-160 seconds.

In brief, after repeating the same experiments several times, these similarities claimed that either there are some datagrams taking a lot to process in the system or there is something wrong with the system. An alternative explanation for these defects was having another process in the system, which may be waking up and hogging the CPU at those instants. In order to reveal the real reason of the problem, we considered investigating the queue length iteration statistics shown on Figure 8. From those statistics, we aimed to understand whether there are some datagrams taking longer to process in comparison to others. We chose inter arrival rate of 100 \( \mu s \), since it is rather a slow data rate that we do not expect any queueing to occur in the system. We ran this experiment and collected the statistics shown on 8. On the collected data, we recorded maximum queue lengths occurred during the run of the experiment. In the collected data we looked at how these large queue lengths are occurring. On Figure 10, we listed a few lines of the statistics data around a very large queue length of 366.

**Details of occurrence of queue length 366: (very interesting)**

```
<table>
<thead>
<tr>
<th>DevTime</th>
<th>Len</th>
<th>getSolitaryAll()</th>
<th>NodeGroup all</th>
</tr>
</thead>
<tbody>
<tr>
<td>183559203386 1</td>
<td>96710</td>
<td>2563</td>
<td></td>
</tr>
<tr>
<td>183559302122 1</td>
<td>95873</td>
<td>1671</td>
<td></td>
</tr>
<tr>
<td>183559405160 1</td>
<td>101056</td>
<td>6400</td>
<td></td>
</tr>
<tr>
<td>183559405160 5</td>
<td>124444</td>
<td>31882206</td>
<td></td>
</tr>
<tr>
<td>183584024839 366</td>
<td>292</td>
<td>2128509</td>
<td>366 datagrams to arrive with IAR 100( \mu )sec</td>
</tr>
<tr>
<td>183586157206 21</td>
<td>68</td>
<td>121966</td>
<td>5.8$/sec/datagram</td>
</tr>
<tr>
<td>183596279499 2</td>
<td>31</td>
<td>4750</td>
<td>15 ( \mu )sec/datagram</td>
</tr>
<tr>
<td>183596380634 1</td>
<td>95680</td>
<td>3834</td>
<td>247</td>
</tr>
<tr>
<td>183596481402 1</td>
<td>96250</td>
<td>4873</td>
<td>234</td>
</tr>
</tbody>
</table>
```

Figure 10: Breakdown of queue problem. A few samples are shown from the queue length breakdown statistics around queue length of 366.

Interestingly, on the line marked with red we see a very abrupt behavior. Normally, `getSolitaryAll()` method blocks for around 95-100 \( \mu s \) which is what we expect from the inter arrival rate of 100 \( \mu s \) and which also means that the queue is almost empty for most of the time. But suddenly on this red line, a datagram is taking around 35 milliseconds to be processed, which is really huge. This time is also equal to the time for 366 datagrams to arrive at constant inter arrival rate of 100 \( \mu s \). As a result, after next call of `getSolitaryAll()` we see that there are 366 datagrams in the queue. This information obviously showed that certain datagrams are taking longer, so we traced down to those datagrams. But only then we saw that those datagrams had no difference to the others. At this point, the source of the problem was really hard to understand. We tried all methods to identify the real source of the problem. Chancefully, at sometime when we are going over the source code, we noticed that there is something
wrong with the statistics collection mechanism. For datagram statistics (queueTime, datagramTime, libTime, ...) collection, the statistics component uses \texttt{std::vector} from the C++ Standard Template Library (STL) \[27, 33\]. Whenever a new statistics sample is to be added to this list, \texttt{vector.push_back();} method is used. \texttt{std::vector} begins with an initial capacity allocated when it is defined. So it has a fixed capacity, as you \texttt{push_back()} items it dynamically resizes itself. But this reallocation of memory occurs by doubling the current capacity. Considering that we are keeping around 2,5M datagram statistics in our experiments, occurrence of this memory reallocation is very frequent. At certain number of datagrams, lets say at 500,000th datagram vector’s capacity is doubled to 1M by memory reallocation. This memory re-allocation brings to much overhead to the system at those reallocation instances \[26\]. And that is why we are suffering in datagram processing at those times. As a workaround, we tried to fix the number of statistics that we are collecting at 4M and pre-allocated the vectors. Basically, we used \texttt{vector.reserve(num_items)} in the beginning to pre-allocate the necessary memory for statistics collection. Afterwards, we ran the experiments again and we saw that those abrupt behaviors had gone. What we learned from this experience is that one should be very careful with using standard vector with ever growing number of data in a very low latency requiring application.

7.3 Results

In this subsection, we would like to present our results about the queue behavior analysis after solving the problem with memory re-allocation as mentioned in the previous subsection. Now we have more reliable results for queue length statistics during the all run time of the experiments. Change of the queue length during the run-time could be observed in detail by looking at the figures below. One important finding is that, maximum queue lengths are no more above thousands. In the wildest case, it is less than 70.

Figure 11: Queue length during the run-time of experiment with IAR 25.

At this point, we also would like to present some statistics about the queue
Figure 12: Queue length during the run-time of experiment with IAR 50.

Figure 13: Queue length during the run-time of experiment with IAR 100.
Figure 14: Queue length during the run-time of CS dump replay experiment.

Figure 15: Queue length statistics for the complete runs of experiments.

7.4 Discussion

Results from the previous section show that queue size never goes beyond 70 or lets say 100 even with the inter arrival rate of 25 \( \mu s \). This corresponds to a datagram rate of 40K/second which is quite higher than the Credit Suisse real dump. In the replay experiment, maximum queue length is 15. Beyond maximum queue lengths, another striking fact is that median queue length is 1. This means for most of the time there is only one datagram in the queue. As a consequence, we have to question in the first place whether having a queue in the system is a good choice. Operating systems already have queuing mechanism at the kernel level for network packets. This means that we have a second level of queuing in the Credit Suisse implementation above the OS. Linux provides option for setting the kernel datagram receive queue size. Kernel parameter `net.core.netdev_max_backlog` specifies number of unprocessed input packets.
before kernel starts dropping them. By setting an appropriate value to this parameter, kernel can take care of the queueing of packets for us. Considering this fact, we can get rid of the queue sitting between datagram receiving thread and decoding/filtering thread in Credit Suisse implementation. But this requires a sharp design shift. It requires us to combine datagram reading thread and decoding/filtering thread in one thread. As a result the application will be single threaded (ignoring the error thread and application thread). We already investigated this idea. First we did some minor changes in the implementation by preserving the functionality same as the current version. Afterwards, we tried to see the latency measurements as we did in the following sections and we compared them. We will go over the improvement idea in section 9.
8 Complete Analysis of Latencies

8.1 Kernel-to-Userspace Latency Measurements

Before going on with the important end-to-end latency measurements, we would like to present the latency measurements for the time spent in the kernel for each datagram. In the next section we will see that these latencies are already included in the full end-to-end latency measurements; so seeing them before the end-to-end ones will give an insight on what share has the OS kernel on end-to-end latencies.

8.1.1 Methodology

Linux provides an opportunity to receive kernel level timestamps with the reception of datagrams. To enable this feature, the SO_TIMESTAMP option on the datagram socket should be set [18]. Afterwards, call to recvmsg() returns a timestamp corresponding to when the datagram was received by the kernel along with the datagram itself. When this option is enabled, Linux Kernel timestamps when the datagram is received by using gettimeofday(). Since it is done at kernel level, it has little overhead. recvmsg() returns this struct timeval object as an ancillary data to the userspace process. Just after recvmsg() call returns we do another call to gettimeofday() to understand the time passed in the kernel for each datagram. Knowing that gettimeofday() is a costly function call on most systems (i.e. system call), we also wanted to understand its overhead. Previously, we were using AMD 64 Opteron machines and gettimeofday() was not using processor clock cycle counter due to its unreliability on AMD machines. As a result, it was taking around 2 µs per gettimeofday() call. Due to this reason we had to change our machines. We switched to Intel Xeon 4-core machines. Now gettimeofday() is implemented as a virtual system call (less overhead than an ordinary system call) and it is using process clock cycle counter as the timer source. We benchmarked gettimeofday() on these machines and observed that it is taking less than 0.2 µs. So from overhead point of view, gettimeofday() overhead is now insignificant.

8.1.2 Results

In this section we would like to present our results for OS kernel latency. We ran experiments again with varying datagram inter arrival rates (25, 50, 100 µs and replay of CS dump). Number of datagrams for statistics were limited at 2.5M. On Figure 16, you can see cumulative distributions of latencies for OS Kernel latency. On the figure, it can be seen that kernel latency for inter arrival rate 100 µs almost stays constant. Kernel latency for dump replay is also constant up to % 70, then it begins to increase but still under 20 µs for 90th percentile. Interestingly, latency starts increasing very early for inter arrival rate of 25 µs. 90th percentile for IAR 25µs is around 28 µs. Early increase in latency for IAR 25 µs shows that there is a certain amount of queueing occurring already in the kernel. For dump replay, kernel queueing begins after 70th percentile. Last but not least, median latency for IAR 25 µs is around 18 µs; by far greater than the latency of other inter arrival rates.
Figure 16: CDF plots for OS Kernel latency of datagrams with different inter arrival rates.

Figure 17: Statistics for OS Kernel latency of datagrams with different inter arrival rates. All metrics are \( \mu \)s.
8.1.3 Interrupt Coalescion

Linux kernel provides a feature to reduce CPU load for network processing. Instead of creating an interrupt for each received packet, only one interrupt for several number of packets could be created by setting related parameters [25]. This is a kind of batching of several datagrams together and notifying the CPU about them together by interrupt coalescion. Under heavy loads, this may be very beneficial for reducing the CPU load. But since notification of already arrived datagrams are delayed for a while, this increases the observed latency for those datagrams in the kernel. This is what is happening in the Figure 16. We noticed that during those experiments, \texttt{rx-usecs} parameter of the NIC was 3 \(\mu s\). \texttt{rx-usecs} on Linux is basically a coalescion parameter for the NIC. It tells the NIC to wait for \texttt{rx-usecs} before creating an interrupt to the CPU. For very low latency requiring applications, it is obvious that this parameter should be 0. To understand its effect on our measurements, we set it to 0 by issuing the following command:

\begin{verbatim}
ethtool -C eth1 rx-usecs 0
\end{verbatim}

Results after this change can be seen in Figure 19. It has interesting results. It is apparent that packets were being processed in batches in the previous configuration and now they are processed one by one. So we are seeing almost constant latency per packet in the kernel. Another observation is that for slow data rates (IAR 100 and CS Replay) the median kernel latency goes up around 2 \(\mu s\). One crucial finding from these measurements is that even with data rates of 40K/second, we have constant kernel latency up to 80th percentile.

![Figure 18: In this figure, we repeated the same experiment with IAR50 \(\mu s\) and only played with the rx-usecs kernel parameter by setting it to 0 and 3 \(\mu s\) respectively. The effect of rx-usecs can be seen clearly on this figure.](image-url)
Figure 19: CDF plots for OS Kernel latency of datagrams with different inter arrival rates - Kernel interrupt coalescing parameter rx-usecs set to 0.

Figure 20: Statistics for OS Kernel latency of datagrams with different inter arrival rates - Kernel interrupt coalescing parameter rx-usecs set to 0. All metrics are µs.
8.2 End-to-End Latency Measurements

In this section we would like to present our measurements for end-to-end latencies through the processing pipeline. Knowing the end-to-end latency for messages gives good insight to how our system is behaving. Also we can judge whether low latency requirements of the system is satisfied or not and if satisfied to which degree it is satisfied. First we would like to explain what we exactly mean by our end-to-end latency metric.

8.2.1 Methodology

During end-to-end measurements, we utilize Kernel level packet timestamps as described in section 8.1.1 as well. We first enabled Kernel level timestamps by setting `SO_TIMESTAMP` option on the socket. Then we added a custom `recvfrom()` method to the `Socket` class. This custom method returns a `struct timeval` object along with bytes of the received packet. At this point, we would like to show our end-to-end metric on a figure.

![Figure 21: End-to-end latency metric for each message of a given datagram.](image)

Every datagram consists of several messages. For each message in the given datagram, we take the Kernel arrival of the corresponding datagram as arrival time. Then when processing of each message is complete, we record another timestamp using `gettimeofday()`. The time difference between this timestamp and the kernel level datagram timestamp gives us the latency of the corresponding message in the system. Previously, we had problems with `gettimeofday()` as mentioned before, but now only 0.2µs per `gettimeofday()` call seems to be negligible. An important point to note here is that, since message n is coming after message n-1 in the datagram its latency will also include the processing time of the previous message. So our measurement method never underestimates the latencies of messages.

8.2.2 Overhead of Measurements

According to Stefan’s measurements, `processStream()` method or in another word datagram time is around 2-3 µs. Also average message count per datagram is around 3-4. Since we have one extra `gettimeofday()` per message on this
thread, the overhead is around $3 \times 0.2 \mu s$ per datagram. Actually, in end-to-end measurements we are keeping end-to-end latencies per message but either way the overhead of extra `gettimeofday()` calls is $0.6\mu s / 3\mu s$ which is equal to nearly %20. But be aware that it is %20 of the `eurexThread()`, the other threads are not included. Meanwhile, this mathematical approximation is also confirmed by conducting a system profiling with OProfile. As we will see in the results section, we suppose that this number should be negligible since we are talking about around 15-20 $\mu s$ latencies per message.

8.2.3 Results

Now it is time to have a look at our end-to-end measurements. As we did in Kernel latency measurements, we will present two sets of results. First set is acquired before changing the interrupt coalescion parameter to 0. Please refer to Figure 22 for these measurements. Second set is after changing interrupt coalescion parameter to 0. These measurements can be seen on Figure 24.

One good result from these measurements is that median latencies for all datagram arrival rates is below 30 $\mu s$. Even further, except IAR 25, median latencies are below 20 $\mu s$. By the way, IAR 25 is really very fast in comparison to the Credit Suisse dump. Up to 70th percentile, latencies are pretty good but after that point they go up to 50 $\mu s$. In any way, we can certainly say that 90th percentile of end-to-end latencies for all inter arrival rates is below 50 $\mu s$. We also need to remember that %20 of this latency is coming from the OS Kernel latency as we have seen in the previous section.

Second set of results show us some more interesting results. On these measurements, median latencies are exceptionally good. All median latencies are below 20 $\mu s$ including inter arrival rate of 25 $\mu s$. As we have seen in OS Kernel latencies, here we also see that latencies tend to stay constant up to 60th percentile. This is mostly due to the fact that we get one interrupt per packet. Only after 60th percentile, or around 20 $\mu s$, queueing begins to show its effects. For inter arrival rate of 25 $\mu s$, we certainly see the effect of interrupt coalescion. Since the data rate is high, latency is highly affected from the coalescion parameter in the previous measurements and they are 5-10 $\mu s$ better in comparison to the previous measurements. But other data rates do not seem to be affected very much. They are either at around the same latencies or 1-2 $\mu s$ worse. Also from Figure 24, we can make such a strong statement that 90th percentile of all latencies are below 50 $\mu s$.

8.2.4 Discussion

In statistics listed on Figures 23 and 25, maximum latency numbers require some explanation as they are sometimes extremely high. At the same time they are very rare and they should not be causing much variation in the measurements. We guess that these extreme spikes are occurring due to scheduling artifacts in the Operating System. At certain times, another process in the system with a higher priority is scheduled for execution and during this time our processing is paused for a while. And these moments are exactly the ones that we see these extreme maxima latencies. Since we do see such spikes in OS Kernel latencies as well, we certainly can not attribute these spikes to Credit Suisse code.
Figure 22: CDF plots for End-to-end latency of datagrams with different inter arrival rates.

Figure 23: Statistics for End-to-end latency of datagrams with different inter arrival rates. All metrics are $\mu$s.
Figure 24: CDF plots for End-to-end latency of datagrams with different inter arrival rates - Kernel interrupt coalescing parameter rx-usecs set to 0.

Figure 25: Statistics for End-to-end latency of datagrams with different inter arrival rates - Kernel interrupt coalescing parameter rx-usecs set to 0. All metrics are µs.
Another point which needs clarification is what is not included in our end-to-end latency measurement. First, as can be seen on Figure 21, the time spent on the hardware is not included in our metric. Along with that time spent in the driver code of the NIC is not included as well.

9 Improving the Performance of FAST Decoding

Our queue analysis was resulted in interesting facts. First of all, we were really surprised to see the median queue size of only 1. Along with this fact another surprising fact was the maximum queue size during the whole execution. Even with the fastest synthetic data rate of 40K datagrams per second, we only saw a maximum queue size of 66. These facts forced us to think about the queue. Whether the pipelined design of FAST decoding is a good choice or not is a debatable topic. What we know already claims that most of the time is spent in between the pipelines of the processing which is the queue. Also the claim is supported by the fact that the utilization of the queue is not at a very high level. These together made us to think about eliminating the queue all together. We began investigating ways of eliminating the queue. Later, we learned that the Linux is already doing packet queuing at the kernel level \[19\]. It is also supported by a kernel parameter (\texttt{net.core.netdev_max_backlog}) which could be changed by root privileged users. Basically this parameter determines the maximum size of the kernel packet queue. If the kernel can not process packets for a while and if the number of packets go above this maximum size, then it begins to drop packets. It is suggested to be set around 2500 for Gigabit Ethernet and around 30000 for 10 Gigabit Ethernet connections \[17\]. This parameter could be changed by a root privileged user using the following command:

\$ \texttt{sysctl \textbackslash net.core.netdev_max_backlog} = 2500

On the other hand since everything is in the kernel space, access to this queue is protected by spinlocks at the OS level and probably more efficient than any lock-free queue implementation in userspace. We also learned that this queuing is done in anyway but probably with a smaller default maximum size. If we think about the Credit Suisse implementation from this aspect again, we are effectively having a second level of queuing on top of OS kernel queuing mechanism. The kernel level queuing was a strong argument for us to think about eliminating the queue from Credit Suisse implementation. Since we are not doing this work towards a production environment implementation, we thought that changing the design to a single threaded version and experimenting with that design would be one of the greatest outcomes of our work. In this respect, we made some changes to the original Credit Suisse code which we will mention in 9.1. Afterwards we analyzed the performance of this version by using our previously defined end-to-end metric. We also analyzed CPU usage for the single threaded version as well as some other analysis. We show our results in section 9.2.

9.1 Changes in the implementation

We gave an overview of the Credit Suisse FAST implementation in section 5. We mentioned about the 3 most significant components of the system which were
Data Source thread, Lock-free queue and Decoding/filtering thread. In this scenario, basically most of the processing was going on in the decoding/filtering thread. Data source thread was putting the network datagrams into the lock-free queue as soon as possible and continuing with the network processing part in order not to lose any datagrams. From our analysis, we have seen that there was enough time in the Data Source thread to do some extra processing before datagrams are dropped even with the data rate of 40K datagrams per second. Our further analysis also showed that the queue is not very highly utilized in terms of number of datagrams found in the queue at a time. These findings forced us to think about the design decisions. We decided to make a sharp design shift from multi-threaded pipelined design to a single threaded everything in-one place design. Basically what we did was just combining the functionality of Data Source thread and Filtering/decoding thread in one single thread. In this design the lock-free queue is completely eliminated.

On the implementation side, what we did was very simple. We just changed the instantiation locations of important components. For instance the Decoder was previously declared as a local variable in eurexThread() function which was running in the loop of the decoding/filtering thread. As a result, it was only accessible from this function. As datagrams were extracted from the lock-free queue, this function was calling the processStream() method of this decoder instance. We removed this instantiation from this function and moved it to DataSource::read() function. Since the Decoder constructor needs references to a DataClient instance and a Statistics instance, we also needed to move declarations of DataClient and Statistics to the anonymous namespace of the DataSource thread. After having an access to the Decoder instance from the DataSource::read() thread, the rest was easy. The processing up to putting the datagrams into the lock-free queue was the same. Instead of putting datagrams into the queue after duplicate elimination, we called decoder.processStream() method of the Decoder instance that we created. Practically we had a short-cut of calling the processStream() method. Lastly, we did not put the datagram into the lock-free queue and bypassed it.

9.2 Analysis of the Performance of Single Threaded Version

After the implementation changes, we ran the system several times to see whether it is functioning without any problems. These tests did not show any anomalies in the execution. We also wanted to make sure that we are processing all the datagrams received. As mentioned in section 3.4, we implemented a comparator tool to see if the sent dump is same as the received dump after being processed by the FAST decoder. By using this tool, we assured that there was no datagram loss and everything was processed.

Next we wanted to understand the performance of the single threaded implementation. Our basic benchmarking metric at this step was our end-to-end latency metric. We ran the same benchmarks that we ran for understanding the performance of the original Credit Suisse implementation. As a result, we were able to produce side-by-side comparable results with the previous results. Now we would like to show our main result from those benchmarks. Figure 26
shows the cumulative distribution of end-to-end latencies for the single threaded version.

First of all, from the figure we see a slight shift towards left for nearly all the curves. This clearly shows the elimination of queuing effect in the lock-free queue. In section 4, where we reproduced the previous metrics such as \textit{queueTime}, we saw that the \textit{queueTime} was around 3-4 µs. By simple math, we may expect a reduction of 3-4 µs in end-to-end latencies. But this is not exactly the case, because we are still having a queuing at kernel level. However in general we may talk about a 2-5 µs improvement in overall latencies. For instance if we look at median latencies in all inter arrival rates, we can easily see an improvement of 4-5 µs. The latencies are almost constant upto 50th percentile, only then they begin to increase just over 14 µs. The queuing effect in Linux kernel can be easily seen in the case of IAR 25 after 60th percentile and latency of 20 µs. Other observations can be summarized as follows:

- Median latencies drop from around 17-19 us to 13-14 us. (all IAR’s are 4-5 us better at 50th percentile)
- 90th for Replay is 35us which was around 43us in the previous version.
- 90th for IAR 50 is 45us which was around 48us in the previous version.

Figure 26: CDF of End-to-end latencies for single threaded implementation.
- 90th for IAR 100 is 37us which was around 39us in the previous version.
- 90th for IAR 25 is 55us which was around 48us in the previous version.
- Up to 60th percentile all IARs are below 20us and only then kernel queuing begins for IAR25 and slightly for IAR50.
- We almost have better latency numbers varying between 2-5us in all IARs except IAR25, which only begins to get worse after around 70th percentile.

After seeing the end-to-end latency measurements and the general trend of improvement in numbers, we also wanted to see how much of the CPU is used for the FAST decoding with different inter arrival rates. We believed that the improvement in latency with the single threaded version should also be supported by CPU usage statistics before deciding to switch to the single threaded version in the production environment. We ran the same benchmarks of end-to-end latency measurements with the single threaded version and collected CPU usage statistics by using the mpstat [22]. The results of these experiments can be seen in Figure 27. It shows percent of CPU used for different inter arrival rates as well as the Credit Suisse dump replay. %user shows the percentage of CPU used during execution of the user application. %sys shows the CPU used for kernel level code. %irq shows the CPU percent used for servicing interrupts and %soft shows the percentage of CPU used to service softirqs which are software interrupts. Although we expected more overall CPU usage in single threaded version, we did not observe this on our measurements. The total CPU usage is around %35 percent in IAR25 experiment. Also with the increase of inter arrival rate, we see a slight increase in CPU usage but again this is not on the order that we expected. To assure the reliability of these measurements, we repeated them several times and found similar results again. What we can say as a result of these measurements is that there is no CPU-intensive processing going on and from what we see we can easily say that there is enough processing resource to switch to the single threaded version.

![Figure 27: CPU Usage for different IARs](image-url)
Last but not least, we would like to present an interesting finding from our measurements. In order to see the effect of the snapshot stream processing in the end-to-end latency measurements, we made an experiment. In this experiment, we only sent delta stream messages with constant inter arrival rate of 50 $\mu$s and we did not send any snapshot messages on the snapshot stream. With this setup, we repeated our end-to-end latency measurement experiment. The plot of cumulative distribution of end-to-end latencies from this experiment can be seen on Figure 28. It is very interesting to see that almost all of the variation in latencies is coming from the processing of snapshot stream messages. Since the snapshot messages are large in comparison to the delta messages, they are taking a lot to process. On the other hand since they are less frequent than delta messages, their effect can not be seen clearly upto 60th percentile in this figure. Lastly, it is also very interesting to see that we have almost a constant latency of processing delta messages with the single threaded version.

Figure 28: End-to-end latency measurements with and without snapshot stream.
10 Conclusions and Future Work

In this work, we have throughoutly analyzed and benchmarked the Credit Suisse FAST implementation. We have seen that 50 µs latency per message at 90th percentile is a sustainable latency from the system in any case. This brings the important question to one’s mind: We know that this is an arms race, but the question is whether we can judge how good this latency is. Excluding data rate of 40K/second, we see median latency to be less than 20µs. Further subtracting around 10 µs Kernel latency, a median latency of 10 µs sounds really good to us. But we are also aware that even any sub-microsecond improvement in the end-to-end latency is of value to Credit Suisse. As an improvement idea for the FAST implementation, we have proposed a single threaded implementation where the network processing, duplicate elimination, decoding and filtering are done in a single thread. This design enabled us to get rid of the lock-free queue which was in fact one of the performance bottlenecks according to our measurements. Furthermore, by applying our benchmarks and metrics we compared the performance of the single threaded version with the original Credit Suisse implementation. Our experimental results have shown that we can improve the end-to-end latency around 2-5 µs in average if we switch to a single threaded implementation. We also showed that there is enough CPU processing resources for switching to the single threaded version. As a result of CPU usage measurements we can easily say that there is no big CPU bound problem going on during these experiments. For the measurements we also have to note that real latencies could be a few microseconds better than our measurements. We did not exclude gettimeofday() overhead and also the median latency of messages increase due to variations (jitter) in processing time.

We would like to show the processing pipelines and the measurement of end-to-end latencies on these pipelines using the figures below. In Figure 29, the original Credit Suisse code is benchmarked with the replay of the Credit Suisse dump sent from our generator and end-to-end latencies for each message is measured. We measured the median latency to be 19 µs. In Figure 30, we used our single threaded version and again benchmarked it with the replay of the Credit Suisse dump. In this benchmark, we measured the median latency per message to be 14 µs. Consequently, with a straightforward statement, we can claim a 5 µs improvement in the replay scenario.

Figure 29: End-to-end latency for replay in Credit Suisse version.

The Figures 30 and 29 shows two more things that we need to discuss about. They are basically the question marks on the figures. First of them is the NIC part. During our experiments, we were not able to count in the latency coming from the network interface card. So this still seems to be as a future work. Possibly, doing network hardware level timestamping or measuring time in network
card driver software may solve the problem. Nevertheless we anticipate this latency not to be comparable to what we are measuring in overall. We expect latency around a few microseconds. So it should not disrupt our end-to-end measurements. The second question mark in the pipeline comes from the application callback part. Since we did not have the application code in our setup, we were not able to measure the latency coming from there. This also prevents us from making strong arguments for switching to the single threaded version. We have shown that there is enough CPU resource for the single threaded version but these experiements were without the application code. So we are not sure about the processing resource requirements of the application logic. If they require more than the available, then it might not be wise to decide for the single threaded version. Investigating the processing requirements of the overall system with the application logic remains as a future work.

There are still two directions in our mind for further improvement of the performance of the system: we can either try to reduce the latency by trying to cut off the latency coming from the Operating System (possibly around 6-7 µs) or we can try to cut off the processing latency. First option will push the plots on figures to left and the second option will push the plots and median percentages up. One of the interesting results from our study was the share of the operating system in the overall latency. It took almost more than the all other processing in the pipeline. So this really calls for an optimization of network processing in the OS level. Since we only worked on RedHat Linux systems, we are not aware whether this is also the case in other operating systems. Some of the ideas for optimizations which remain as a future work are trying to optimize the Linux network stack for this type of applications, trying to use better network cards taylored to low latency, trying to offload network stack to a smarter NIC hardware and switching from interrupt driven networking to polling driven networking at NIC level.

Figure 30: End-to-end latency for replay in single threaded version.
Bibliography


