Towards Efficient Big Data Processing in Data Centres

Luo Mai

Department of Computing
Imperial College London

This dissertation is submitted for the degree of
Doctor of Philosophy

September 2018
Abstract

Large data processing systems require a high degree of coordination, and exhibit network bottlenecks due to massive communication data. This motivates my PhD study to propose system control mechanisms that improve monitoring and coordination, and efficient communication methods by bridging applications and networks.

The first result is CHI, a new control plane for stateful streaming systems. CHI has a control loop that embeds control messages in data channels to seamlessly monitor and coordinate a streaming pipeline. This design helps monitor system and application-specific metrics in a scalable manner, and perform complex modification with on-the-fly data. The behaviours of control messages are customisable, thus enabling various control algorithms. CHI has been deployed into production systems, and exhibits high performance and scalability in test-bed experiments.

With effective coordination, data-intensive systems need to remove network bottlenecks. This is important in data centres as their networks are usually over-subscribed. Hence, my study explores an idea that bridges applications and networks for accelerating communication. This idea can be realised (i) in the network core through a middlebox platform called NetAgg that can efficiently execute application-specific aggregation functions along busy network paths, and (ii) at network edges through a server network stack that provides powerful communication primitives and traffic management services. Test-bed experiments show that these methods can improve the communication of important analytics systems.

A tight integration of applications and networks, however, requires an intuitive network programming model. My study thus proposes a network programming framework named Flick. Flick has a high-level programming language for application-specific network services. The services are compiled to dataflows and executed by a high-performance runtime. To be production-friendly, this runtime can run in commodity network elements and guarantee fair resource sharing among services. Flick has been used for developing popular network services, and its performance is shown in real-world benchmarks.
The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work.
I would like to dedicate this thesis to my beloved mother Suzhen Shu, my honourable father Shaoxiong Mai, my loving wife Nan Mo, and my adorable daughter Yuhan Mai.
Acknowledgements

First of all, I would like to express my sincere gratitude to my advisor Dr. Paolo Costa for his continuous support, patience, encouragement, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor.

Also, I would like to thank the faculty members in the LSDS research group: Prof. Peter Pietzuch and Prof. Alexander Wolf, for their support and insightful advice of related research.

My sincere thanks also go to Dr. Rahul Potharaju, Dr. Kai Zeng, and Dr. Yongqiang Xiong, who offered me an opportunity to join their teams as an intern. Without these experiences, it would not be possible to evaluate my research ideas in production environments.

I thank my fellow lab-mates, in particular, Dr. Lukas Rupprecht, Dr. Xi Chen and Dr. Chin Pang Ho, for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last five years. Also, I thank my collaborators in the following institutions: Imperial College London, University of Cambridge, Brown University, and Microsoft Research.

Last but not the least, I would like to thank the funding bodies of my research including Google Doctoral Fellowship in Cloud Computing, UK EPSRC Project - Network-as-a-Service, and Microsoft Azure Sponsorship Award.
This thesis presents my work in the Department of Computing at Imperial College London between October 2012 and December 2017.

Parts of the work were done in collaboration with other researchers:

- **Chapter 3**: I proposed, implemented and evaluated the design of the CHI system. The analysis of production workloads is conducted by Dr. Rahul Rotharaju. Le Xu ran the test-bed experiments of Flink and Drizzle, as baselines to CHI.

- **Chapter 4**: The implementation of the NetAGG system is a joint effort with Dr. Lukas Rupprecht. I completed the simulation experiments, feasibility study, and the test-bed experiments for Solr and multi-tenant experiment. Dr. Abdul Alim contributed to the optimisation of the performance of the local aggregation tree.

- **Chapter 6**: The implementation of Flick runtime is a joint effort with Dr. Abdul Alim and Dr. Richard Clegg where I was leading the development of the task scheduler runtime and the network layer. I also implemented the test-bed experiments for Memcached. The design of the Flick language was led by Dr. Nik Sultana.

I declare that the work in this thesis is my own, except where declared above.

Luo Mai
September 2018
# Table of contents

List of figures  

List of tables  

1 Introduction  

1.1 Motivation  

1.2 Contributions  

1.2.1 Novel control plane design  

1.2.2 Novel methods for improving data plane performance  

1.2.3 Novel programming model for data centre networks  

1.3 Related Publications  

1.4 Outline  

2 Background  

2.1 Data Centre Computation  

2.2 System Control Plane  

2.3 System Data Plane  

2.4 Network Programmability  

2.5 Summary  

3 Chi: A Novel Control Plane for Distributed Streaming Systems  

3.1 Introduction  

3.2 Background  

3.3 Motivation  

3.4 Design  

3.4.1 Overview  

3.4.2 Control mechanism  

3.4.3 Advanced functionalities  

3.4.4 Compare Chi with existing models

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Contributions</td>
<td>2</td>
</tr>
<tr>
<td>1.2.1</td>
<td>Novel control plane design</td>
<td>2</td>
</tr>
<tr>
<td>1.2.2</td>
<td>Novel methods for improving data plane performance</td>
<td>3</td>
</tr>
<tr>
<td>1.2.3</td>
<td>Novel programming model for data centre networks</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>Related Publications</td>
<td>4</td>
</tr>
<tr>
<td>1.4</td>
<td>Outline</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Background</td>
<td>7</td>
</tr>
<tr>
<td>2.1</td>
<td>Data Centre Computation</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>System Control Plane</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>System Data Plane</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>Network Programmability</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>Summary</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>Chi: A Novel Control Plane for Distributed Streaming Systems</td>
<td>15</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>Background</td>
<td>17</td>
</tr>
<tr>
<td>3.3</td>
<td>Motivation</td>
<td>19</td>
</tr>
<tr>
<td>3.4</td>
<td>Design</td>
<td>23</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Overview</td>
<td>24</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Control mechanism</td>
<td>26</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Advanced functionalities</td>
<td>34</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Compare Chi with existing models</td>
<td>35</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>3.5</td>
<td>Application Examples</td>
<td>38</td>
</tr>
<tr>
<td>3.6</td>
<td>Implementation and Discussion</td>
<td>41</td>
</tr>
<tr>
<td>3.7</td>
<td>Evaluation</td>
<td>44</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Experimental setup</td>
<td>45</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Micro-benchmarks</td>
<td>45</td>
</tr>
<tr>
<td>3.7.3</td>
<td>Adaptivity and fault-tolerance</td>
<td>48</td>
</tr>
<tr>
<td>3.7.4</td>
<td>Chi in action</td>
<td>51</td>
</tr>
<tr>
<td>3.8</td>
<td>Summary</td>
<td>53</td>
</tr>
<tr>
<td>4</td>
<td>NetAgg: Application-specific In-network Data Aggregation</td>
<td>55</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>55</td>
</tr>
<tr>
<td>4.2</td>
<td>Distributed Data Aggregation</td>
<td>57</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Partition/aggregation applications</td>
<td>57</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Edge-based aggregation</td>
<td>58</td>
</tr>
<tr>
<td>4.2.3</td>
<td>On-path aggregation with middleboxes</td>
<td>60</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Feasibility study</td>
<td>61</td>
</tr>
<tr>
<td>4.3</td>
<td>Design and Implementation</td>
<td>63</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Overview</td>
<td>63</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Implementation</td>
<td>66</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Application deployments</td>
<td>70</td>
</tr>
<tr>
<td>4.4</td>
<td>Evaluation</td>
<td>71</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Simulation results</td>
<td>71</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Testbed results</td>
<td>77</td>
</tr>
<tr>
<td>4.5</td>
<td>Summary</td>
<td>83</td>
</tr>
<tr>
<td>5</td>
<td>MLNet: A Novel Application-aware Communication Layer</td>
<td>85</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>85</td>
</tr>
<tr>
<td>5.2</td>
<td>Background</td>
<td>86</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Deep learning</td>
<td>87</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Distributed deep learning</td>
<td>87</td>
</tr>
<tr>
<td>5.3</td>
<td>MLNet Design</td>
<td>88</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Architecture</td>
<td>88</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Distributed aggregation and multicast</td>
<td>89</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Network prioritisation</td>
<td>90</td>
</tr>
<tr>
<td>5.4</td>
<td>Simulation Results</td>
<td>91</td>
</tr>
<tr>
<td>5.5</td>
<td>Summary</td>
<td>95</td>
</tr>
</tbody>
</table>
# Table of contents

6 Flick: Developing and Running Application-specific Network Services  
6.1 Introduction .................................................. 97  
6.2 Application-specific Network Services  
  6.2.1 Use cases ............................................. 100  
  6.2.2 Existing solution space ............................... 101  
6.3 Flick Framework  
  6.3.1 Requirements .......................................... 102  
  6.3.2 Overview ............................................... 104  
6.4 FLICK Programming Model  
  6.4.1 Language design goals ............................... 105  
  6.4.2 Sample program ...................................... 106  
  6.4.3 Application data types ............................... 107  
  6.4.4 Primitives and compilation ......................... 109  
6.5 Flick Platform ............................................. 111  
6.6 Evaluation  
  6.6.1 Use case implementation ............................ 113  
  6.6.2 Experimental set-up ................................ 115  
  6.6.3 Performance .......................................... 116  
  6.6.4 Resource sharing ................................... 120  
6.7 Summary .................................................. 121  

7 Conclusions .................................................. 123  
7.1 Summary .................................................. 123  
7.2 Future Directions ......................................... 124  

References .................................................. 127
List of figures

3.1 Scaling-out control in action where the user is interested in changing the number of reducers in a word-count example. .................. 18
3.2 Heat-map of a subset of data streams in our ingestion workload. Each horizontal band signifies a unique data stream being ingested. ...... 20
3.3 Variability of delta count (order of millions) per minute (Y-axis) for a subset of incoming data streams (X-axis). ......................... 21
3.4 Selectivity of various grouping keys across time. ..................... 22
3.5 Selectivity variability across the grouping key space. ................ 22
3.6 Optimizing the meta topology through pruning. The final stage here is equivalent to Stage (IV) in Fig 3.1 ......................... 27
3.7 State machine transitions for controller and operator .......... 28
3.8 The relation between events and global states. ..................... 32
3.9 Executing a two-stage dataflow using the BSP model implemented by Chi. 36
3.10 A dataflow that needs to modify filters at once. .................. 37
3.11 Handling Skew ................................................. 39
3.12 A CHI-enabled FLARE operator architecture. .................. 42
3.13 Control program structure in CHI ................................. 43
3.15 Control-plane overhead for YSB (left) and IPQ1 (right) .......... 46
3.16 Completion time of the control program for YSB (left) and IPQ1 (right) 48
3.17 Completion time for different dataflow sizes ....................... 48
3.18 Dynamic elasticity ................................................. 49
3.19 Failure recovery ................................................... 49
3.20 Impact of batch size on latency and optimization phases ......... 50
3.21 Work distribution across tasks of the streaming-join operator in IPQ2 before (top) and after workload skew adaptation (bottom) .... 51
3.22 Temporal processing latency/throughput of shuffle join ........... 52
4.1 Distributed data aggregation strategies .............................. 59
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>Flow Completion Time (FCT) for different aggregation processing rates $R$</td>
</tr>
<tr>
<td>4.3</td>
<td>Performance and cost of different DC configurations</td>
</tr>
<tr>
<td>4.4</td>
<td>\texttt{NetAgg} middlebox deployment with sample workflow (numbers correspond to steps in workflow)</td>
</tr>
<tr>
<td>4.5</td>
<td>Architecture of a \texttt{NetAgg} agg box with two deployed applications (shading represents different aggregation functions)</td>
</tr>
<tr>
<td>4.6</td>
<td>CDF of flow completion time of all traffic</td>
</tr>
<tr>
<td>4.7</td>
<td>CDF of flow completion time of non-aggregatable traffic</td>
</tr>
<tr>
<td>4.8</td>
<td>Flow completion time relative to baseline with varying output ratio $\alpha$</td>
</tr>
<tr>
<td>4.9</td>
<td>CDF of link traffic ($\alpha=10%$)</td>
</tr>
<tr>
<td>4.10</td>
<td>Flow completion time relative to baseline with varying fraction of aggregatable traffic</td>
</tr>
<tr>
<td>4.11</td>
<td>Flow completion time relative to baseline with different over-subscription ($\alpha=10%$)</td>
</tr>
<tr>
<td>4.12</td>
<td>Flow completion time relative to baseline with different partial \texttt{NetAgg} deployments</td>
</tr>
<tr>
<td>4.13</td>
<td>Flow completion time relative to baseline in 10 Gbps network with varying over-subscription</td>
</tr>
<tr>
<td>4.14</td>
<td>Flow completion time relative to baseline with varying stragglers</td>
</tr>
<tr>
<td>4.15</td>
<td>Processing rate of an in-memory local aggregation tree on a 16-core server</td>
</tr>
<tr>
<td>4.16</td>
<td>Network throughput against number of clients (Solr)</td>
</tr>
<tr>
<td>4.17</td>
<td>Response latency against number of clients (Solr)</td>
</tr>
<tr>
<td>4.18</td>
<td>Network throughput against outgoing ratio (Solr)</td>
</tr>
<tr>
<td>4.19</td>
<td>Throughput against number of backend servers per rack (Solr)</td>
</tr>
<tr>
<td>4.20</td>
<td>\texttt{Agg} box scale out for CPU-intensive aggregation (Solr)</td>
</tr>
<tr>
<td>4.21</td>
<td>Throughput against number of CPU cores (Solr)</td>
</tr>
<tr>
<td>4.22</td>
<td>Performance of Hadoop benchmarks</td>
</tr>
<tr>
<td>4.23</td>
<td>Hadoop shuffle and reduce time against output ratio.</td>
</tr>
<tr>
<td>4.24</td>
<td>Hadoop shuffle and reduce time against intermediate data sizes.</td>
</tr>
<tr>
<td>4.25</td>
<td>CPU resource fair sharing on \texttt{NetAgg} (Solr and Hadoop) achieved by a no-adaptive scheduler.</td>
</tr>
<tr>
<td>4.26</td>
<td>CPU resource fair sharing on \texttt{NetAgg} (Solr and Hadoop) achieved by an adaptive scheduler.</td>
</tr>
<tr>
<td>5.1</td>
<td>Training makespan against the number of parameter servers.</td>
</tr>
<tr>
<td>5.2</td>
<td>CDF of link traffic for different configurations.</td>
</tr>
<tr>
<td>5.3</td>
<td>CDF of model makespans for different configurations.</td>
</tr>
</tbody>
</table>
List of figures

6.1 Overview of the Flick framework ............................................. 103
6.2 Main components of the Flick platform ................................. 111
6.3 Task graphs for different application-specific network services .... 114
6.4 HTTP load balancer throughput and latency for an increasing number of concurrent persistent connections ......................... 117
6.5 HTTP load balancer throughput and latency for an increasing number of concurrent non-persistent connections ..................... 118
6.6 Memcached proxy throughput and latency versus number of CPU cores used ................................................................. 119
6.7 Median throughput for Hadoop data aggregator versus number of CPU cores ................................................................. 120
6.8 Completion time for “light” and “heavy” tasks with three scheduling policies ............................................................... 121
List of tables

3.1 Comparing Chi with the Synchronous Global Control (SGC) and Asynchronous Local Control (ALC) models. ........................................... 36

4.1 Lines of application-specific code in NETAGG ................................... 70
Chapter 1

Introduction

1.1 Motivation

In the last decade, we have observed the emergence of many large-scale data analytics systems such as batch processing [29, 47, 66], stream processing [72], data ingestion [96, 123, 9], graph analytics [119] and machine learning [65, 58, 167, 105, 11, 152]. These systems are deployed in data centres which accommodate hundreds of thousands of servers. To support computation beyond a server, they usually adopt a scale-out architecture that has a control plane to coordinates servers, and a data plane to process and transfer data.

Due to their commercial nature, data centre networks are usually over-subscribed, making communication the key bottleneck in many important systems such as MapReduce [66, 47], graph processing [119], and machine learning [106, 65, 58]. Today, optimal bi-section bandwidth is still expensive to provide using commodity devices. Practitioners have been developing techniques to accelerate the data transport stacks including data-centre-optimised transport protocols such as DCTCP and MPTCP, and high-speed communication devices such as RoCE and InfiniBand. All these techniques, in principle, rely on new network hardware features, and do not reduce network traffic, making them hard to serve as universal solutions for all data centre networks. Recent advances in software-defined-networks have made the network increasingly programmable, enlightening new ways to address the deluge of communication data. However, this would require new techniques that can effectively integrate aggregation-related application knowledge into resource-constrained network hardware in order to enable the reduction of network traffic.

In addition to communication, server coordination is also increasingly challenging. Today it is not uncommon to observe a data system that spreads across thousands
of servers. These servers are built using commodity hardware and use best-effort communication channels. During coordination, it is necessary to handle hardware failures and communication delays in order to avoid the occurrence of inconsistent server configurations. To avoid this, the de-factor approach performs coordination using system-wide execution barriers where all servers stop computation and can be reconfigured safely. The heavy uses of barriers, however, reduce system availability and resource utilisation. It also limits the types of coordination can be applied. Given these issues, it is important to design new mechanisms that can effectively monitor and coordinate large-scale data systems at a low cost.

Given the increasing number of communication-intensive applications, data centres have been deployed with programmable network elements, thus enabling application-specific network services such as custom packet routing, filtering and firewalls. However, these network elements usually have low-level packet-based programming interfaces, thus making them difficult to be leveraged by developers who are often familiar with applications but lack of knowledge about hardware. To bridge this programming gap, it is necessary to develop a new network programming model that can be easily used by application developers. This model shall not compromise performance and must come with a high-performance runtime that can run on commodity network elements.

To optimise the overall efficiency of Big data computation, my PhD study aims to develop a novel control plane that can effectively monitor and coordinate large-scale data analytics systems, and proposed new networking techniques and programming models that can effectively bridge applications and networks in order to improve network performance.

1.2 Contributions

In the following, I describe the contributions made by my study, respectively.

1.2.1 Novel control plane design

To understand the key requirements for system coordination, I worked with practitioners to analyse real-world data ingestion workloads. These workloads exhibit high variability and unpredictability, and require constant tuning towards performance parameters according to online monitoring data. To fulfil this requirement, I propose a new system control plane, named Chi, that enables flexible and high-performance system monitoring and coordination. Chi proposes to embed control messages within data
channels. It is thus able to reuse the scalable data planes to efficiently monitor system and application-specific performance metrics, and more importantly, apply coordination even among a large number of servers without the use of expensive system barriers. CHI is also able to accommodate various custom system adaptation algorithms though providing an easy-to-learn reactive programming interface. The effectiveness and performance of CHI has been shown in large-scale test-bed experiments that use production workloads. This has motivated practitioners to adopt CHI in production systems, and implement critical real-world adaptation applications.

1.2.2 Novel methods for improving data plane performance

In addition to the coordination challenge that has been addressed by CHI communication is another key challenge especially in data centres where networks are usually over-subscribed. In complementary with conventional efforts that optimise data transport stacks and network hardware, my study explores for a new idea that bridges applications and networks in order to improve communication efficiency.

This idea is first realised in the network core through developing a high-performance middlebox named NetAgg to execute application-specific data aggregation functions along busy network paths. This middlebox has a scaling-out aggregation tree that spreads across all available CPU cores, and process network traffic ingested at a high rate. It also has a light-weight shim-layer that can be seamlessly enabled at application servers in order to transparently route traffic and ultimately build an aggregation tree across the network. Extensive cluster experiments show that NetAgg can remove network bottlenecks in important Big data systems including a distributed search engine and a batch processing engine.

Bridging applications and networks can also be realised at the edges of a network where the access to the core is restricted. To this end, I proposed an application-aware server communication layer, named MLNet. In contrast with existing communication libraries like Message-Passing-Interface (MPI) and ZeroMQ, this layer has novel high-level communication primitives that go beyond low-level socket interfaces. It also provides communication services that can execute application-level data aggregation and prioritisation functions for network traffic. This layer is first demonstrated in distributed machine systems that often report severe network bottlenecks [106, 65, 58]. Using MLNet, these systems can significantly reduce model training times and increase cluster resource utilisation.
1.2.3 Novel programming model for data centre networks

The tight integration between applications and networks, however, poses the need for a new programming model that can make the development of network hardware friendly to application developers. To meet this need, I proposed a novel network programming framework, named Flick. Flick has a high-level functional programming language that can be used for implementing application-specific network services. These network service programs are automatically compiled into a parallel dataflow implementation, and then executed by a high-performance dataflow runtime. To support multiple data centre users, this runtime has a novel service scheduler that can support hundreds of concurrent network services while offering a fair resource sharing guarantee. Flick is shown useful for simplifying the development of numerous key network services. It also exhibits high performance in running challenging benchmarks.

1.3 Related Publications

This thesis is based on the content in my following publications [115, 118, 17, 113]:


- **Optimising Network Performance in Distributed Machine Learning.** Luo Mai, Chuntao Hong, Paolo Costa. The 7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud’15).

- **NetAgg: Using Middleboxes for Application-specific On-path Aggregation in Data Centres.** Luo Mai, Lukas Rupprecht, Abdul Alim, Paolo Costa, Matteo Migli-

Additionally the following publications [158, 69, 171, 114, 117] are over the course of this thesis, and have been impacted but not directly contributed to this thesis.


- **Towards a Network Marketplace in a Cloud.** Da Yu, **Luo Mai**, Somaya Arianfar, Rodrigo Fonseca, Orran Krieger, David Oran. The 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud’16).


### 1.4 Outline

In the following, this thesis presents **Chi**, **NetAGG**, **MLNET** and **Flick** in Chapter 3, Chapter 4, Chapter 5, and Chapter 6, respectively. It presents extensive discussion about related works in Chapter 2. This thesis concludes and discusses future directions in Chapter 7.
Chapter 2

Background

In this chapter, I present the background and basic concepts of this thesis. I start from describing how Big data processing is implemented in data centres, and then discuss recent studies towards the system control plane and the data plane. In the end, I describe the need for adding programmability on top of data centre hardware, and survey recent advances made by researchers and practitioners.

2.1 Data Centre Computation

Data centres are probably the most important data infrastructure today. A data centre can host several hundreds of thousands of servers interconnected by a multi-root tree network. Due to its commercial nature, a data centre is built with commodity devices and shared by a large number of users, thus making it radically different from traditional high-performance clusters that are usually built using specialised hardware and shared by only a small group of experts.

Scaling-out architecture. To utilise data centres, we have developed different kinds of large-scale data analytics systems. Given the limited resource available on servers, Big data systems use a scaling-out architecture to implement parallel computation. Specifically, computation is partitioned and distributed onto multiple servers, and local computation results are collected to compute the final result. The effective implementation of this architecture relies on two system components:

- Control plane that coordinates servers to implement parallel computation.
- Data plane that performs data processing and transfers on servers.

These two components have a big impact towards data processing efficiency. To achieve high efficiency, the control plane must effectively handle computation stragglers,
intermediate failures, data imbalance and many other system coordination issues. The data plane, at the same time, must ensure a fast completion of computation and communication using the locally available resource.

**Online and offline computation.** The computation in data centres can be roughly classified according to the availability of data. Offline computation work with persisted data and thus adopt batch-based engines, e.g., map-reduce, graph processing, and machine learning model training. A batch processing job often adopt a Bulk Synchronisation Processing (BSP) model to coordinate the servers. A typical BSP workflow is as follows: it starts with partitioning inputs into small batches, and let the servers batches in sequential stages. Between stages, all servers stop computation and synchronise with other (i.e., such a timing is called a computation barrier), thus achieving collective, synchronised progress among servers. Optionally, the servers can also checkpoint intermediate results on barriers so that computation can recover from a checkpoint later in case of failures.

On the other hand, online computation work with continuous streams, which are usually infinite, and process streams on a record base, i.e., referred to a record-at-a-time model. Typical streaming systems include complex event processing, online dataflow, and temporal graph. A streaming job can often run for a long time, if not forever, and constantly process events that can be ingested at any time. Only in rare cases, such as failures and checkpoints, the system is suspended, and does necessary maintenance tasks, and resumes later. Recently, the popularity of streaming systems, e.g., Spark Streaming and Flink, is quickly emerging. This is mainly due to their unique and attractive capability that can unify online and offline workloads. Unifying data processing pipelines have numerous benefits in production. The most mentioned benefits include the reduction in infrastructure management cost, the avoidance of resource partitioning, the removal of cross-system overhead, and the consolidation of many system optimisation efforts in a single place. Though promising, accommodating both online and offline pipelines together also raise the requirement for streaming systems, in particular, their control plane designs.

## 2.2 System Control Plane

The importance of a control plane is increasingly recognised due to the need of coordinating more and more servers and working with more and more dynamic workloads. In this section, we discuss related works about the system control plane, with a focus on streaming systems. Compared to their batching counterparts which can easily
apply coordination on synchronisation barriers, streaming systems lack of barriers, and thus must have a sophisticated control plane that can apply coordination with a running data plane. While there have been significant efforts in improving the data plane in streaming systems [49, 37, 31, 45], there has been limited work that thoroughly discusses the design of the control plane [43, 45, 44]. In these days, the control plane is still required to deal with the common issues faced in distributed system implementation such as fault tolerance, scalability, and adaptivity. At the high level, an effective control plane would require rigorous interface and runtime designs. The interface is key to support various control operations that could run in parallel.

**Control plane interface.** Though critical, the design of control plane interfaces has limited discussion by far. The existing interfaces are limited to change the parameters of specific system reconfiguration policies such as the checkpoint interval of asynchronous checkpointing algorithms [43], the parallelism of stages in a dataflow [44], and the hashing scheme in a data re-balance operation [160]. For more sophisticated control operations such as managing and migrating states, the developers have to implement manually. This makes it hard for users to correctly control a system given the big gap between tuneable options and the vast possible number of adoption scenarios. To the best of our knowledge, SEEP [45] is the first study that discusses the interface design for a streaming system. Its interface has a particular focus to integrate the implementation of dynamic scaling and failure recovery. Hence, it has a limited usage for managing parallelism, and is hard to be generalised to support a large collection of control plane operations such as distributed checkpoint and reordering query stages. These operations often have a strong requirement towards the order of data to be coordinated, thus requiring synchronisation support built within the interface.

**Control plane runtime.** In addition to the interface, the control plane also needs to have a runtime that can perform control operation in a distributed environment. To ensure synchronised server updates, existing control planes often freeze the entire system during coordination (i.e., named freeze-the-world). A typical freeze-the-world operation works as follows: the system master node asks all computation servers to stop pulling new input, finish the processing of on-the-fly data, checkpoint computation states, and reload states with new configurations, and finally resume pulling. This operation, apparently, can block the data plane for a long time, avoiding the timely consumption of data and hurting system availability.

Other approaches adopt punctuations to apply control operations. Punctuations were currently used [161] for directing pass-through servers to flush current computation results, and thus avoid buffering unbound computation states that are common in
streaming query operators, such as group-by and join. The key advantage of using punctuations for controlling is that punctuations propagate with normal data events, and hence the system does not need to be frozen during coordination. However, compared to the freeze-the-world approach, punctuations are not synchronised and thus can support very limited control operations. Borealis [40] uses a special kind of punctuation (called control lines) to modify stream queries but their synchronisation needs to be carefully handled by developers, which is a non-trivial task in a distributed setting. Also, control lines are limited to managing query parameters. Esmaili et al. [149] used punctuations to modify a single-input-and-single-output stream only. Recent streaming systems, such as Flink [44], MillWheel [13] and GigaScope [62], mainly adopt punctuations for limited maintenance tasks such as checkpoint barriers, flushing early results and checking heartbeats.

Given the limitation of the continuous processing model, other people tried to adopt a micro-batch model (i.e., a micro-batch is significantly smaller than a typical batch and can be processed in seconds) to process streaming data [164]. As barriers revive among micro-batches, control operations can be applied now. However, repeatedly scheduling micro-batches often incurs non-negligible performance overhead. Most importantly, the dependency to the barriers constrains when and where a control operation can be applied, making it hard to develop a general control plane on top of a micro-batch model.

The studies regarding control plane runtimes are also conducted in database and machine learning systems. Chandramouli et al. [48] studied the optimal execution plans for checkpointing and recovering a continuous database query. Recently, S-Store [46] added the streaming semantics on top of a transactional database. Controls can be applied between transactions; however, the system suffers from similar synchronisation overhead as in BSP systems. To support emerging reinforcement learning algorithms, Project Ray [127] developed a task scheduling framework for large dynamic dataflow. However, this project does not provide programmability on top of its control plane, which is a key for supporting and consolidating a large collection of control operations tailored for various adaptation use cases.

2.3 System Data Plane

In the previous section, I have extensively discussed the control plane from the perspective of interface and runtime designs. To build an efficient Big data system, it
requires an effective data plane which can achieve high-performance in processing and transferring data.

**Data centre networks and over-subscription.** Compared to the rapid performance progress made in a single server, the progress in a distributed cluster is relatively slow due to the limited bandwidth that can be provided by the Data Centre Network (DCN). One of the most performance properties of a DCN is its *bisection bandwidth*. It is defined as the sum of the capacities of links between the two equally-sized sets of servers in a data centre. Intuitively, the bisection bandwidth of a DCN determines the maximum aggregate network throughput that can be achieved when all servers communicate with each other.

Providing *full-bisection bandwidth* in a data centre is very expensive [15] as it often incurs exponential growth in network cost when increasing the number of servers. As a result, data centre providers often *over-subscribe* their networks. The amount of over-subscription is expressed by the *over-subscription ratio* $1 : x$ where $x$ denotes the factor by which the full bisection bandwidth is reduced. While over-subscription enables cost savings, it decreases the bandwidth available to support all-to-all communication which is, unfortunately, a frequent operation (i.e., data shuffling) in almost every data parallel system. As a result, DCNs are often cited as critical bottlenecks in many important data parallel systems. In the following, I will discuss different kinds of network optimisation techniques that have been explored so far.

**Data aggregation.** When confronting with network bottlenecks, reducing application traffic is often the first option to consider. In data parallel computation [66], aggregation of single data processing tasks through local aggregation functions (e.g., the combiner function in a map-reduce job) is a common technique to help reduce network load and job execution time. To reduce the traffic at the network core, additional aggregation steps at one of the rack servers can be performed [124, 108, 172]. At scale, however, rack-level aggregation provides only limited opportunities for data reduction and is often bottlenecked by the lower edge bandwidth available at the servers.

Tyson et al. [59] describe Hadoop Online, an extension to map/reduce that allows reducers to start reducing intermediate results as soon as they become available. Camdoop [60] proposes an extreme solution to improve the performance of map/reduce by adopting a direct-connect DC topology. All traffic is forwarded between servers without switches, which can aggregate the traffic. This approach, however, requires a custom network topology and redeveloped DC applications.

Aggregation was successfully applied in other domains, including wireless sensor networks and Internet systems. In sensor networks, packet-based aggregation is
used to improve the efficiency of data-collection protocols, primarily targeting energy savings [89, 111]. Internet-scale aggregation systems use overlay networks to collect data in a scalable fashion [170, 42, 91]. Neither use of aggregation requires high throughput, a key design requirement of NetAGG. For example, the design proposed by Yalagandula and Dahlin [170] uses a DHT overlay in a WAN, which is a structure optimised for scale and churn, not throughput.

Traffic compression. When aggregation is not available, compressing traffic becomes an attractive option. Typical data compression techniques include network coding [10], traffic redundancy elimination [21], and even multicast [67, 165] are prime examples. However, compression is often a double sword and spend precious CPU cycles in compressing/decompressing traffic, thus often shift the bottlenecks from the network to the computation. Most importantly, traffic compression introduce extra traffic buffering and processing latency, which limits its application in many latency-sensitive applications.

Network function virtualisation. Researchers have proposed the high-performance software middlebox platforms to process network flow using commodity hardware [146, 150]. ClickOS [120], for example, improves the network performance of Xen using netmap [142]. These platforms operate at the packet level, which makes them unsuitable to aggregate application payload.

A flow-based middlebox platform is provided by xOMB [25], which can process application protocols such as HTTP. Yet its focus remains on a small set of network-level services. It is thus hard to be leveraged as application-specific aggregation functions which are particular useful for improving application performance. FlowOS [36] runs in kernel space and provides zero-copy flow construction for multiple, independent flows. However, FlowOS does not support aggregation across multiple, dependent flows.

Recently, SDN-based techniques were used to manage the traffic to and from middleboxes [141]. However, almost all NFV-based approaches focus on only processing the headers of packets. They are thus only able to implement custom routing, packet filtering, and firewall applications; instead of those network services which need to operate on the application data wrapped in the packet payload.

2.4 Network Programmability

Building an effective data plane often requires comprehensive network and system knowledge, and can take months to develop an optimisation tailored to a new application. To resolve this, a promising approach is to provide a network platform that provides
application-level programming abstractions and useful network and system components, thus reducing the duration of developing application-specific optimisations from months to days.

Network programming languages are essential to the usability and scalability of software-defined networking, allowing high-level configuration logic to be translated to low-level network operations. Inspired by Frenetic [78], NetKAT [24] is a high-level network programming language based on Kleene algebra [102], in which network policies are compiled into a low-level programming abstraction such as OpenFlow [122] flow tables. Similarly, the Policy Graph Abstraction (PGA) [140] expresses network policies as a coherent, conflict-free policy set and supports an automated, correct and independent composition of middlebox policies. These systems focus on network management and configuration and not on the more expressive programs for application-specific network services.

There are proposed extensions to the packet processing done by OpenFlow. P4 [38] is a platform- and protocol-independent language for packet processors, which allows the definition of new header fields and protocols for use in match/action tables. Protocol Oblivious Forwarding (POF) [154] also provides a flexible means to match against and rewrite packet header fields. Packet Language for Active Networks (PLAN) [85] is a stateless and strongly-typed functional language for active networking in which packets carry programs to network nodes for execution. In general, these approaches are limited to expressing control-plane processing of packets, which is insufficient to deal with application layer data.

Recently network services have been deployed on commodity hardware to reduce costs and increase flexibility. Click [101] processes packets through a chain of installed elements, and it supports a wide variety of predefined elements. Programmers, however, must write new elements in C++, which can be error-prone. ClickOS [121] combines Click with MiniOS and focuses on the consolidation of multiple software middlebox VMs onto a single server. It overcomes current hypervisor limitations through a redesigned I/O system and by replacing Open vSwitch [5] with a new software switch based on VALE [144]. ClickOS targets packet level processing, e.g. manipulating header fields or filtering packets. It is thus challenging for ClickOS to parse and process HTTP data when a single data item may span multiple packets or Memcached data when a packet may contain multiple data items.

Merlin [155] is a language that safely translates policies, expressed as regular expressions for encoding paths, into Click scripts. Similarly, IN-NET [156] is an architecture for the deployment of custom in-network processing on ClickOS with an emphasis on
static checking for policy safety. In a similar vein, xOMB [25] provides a modular processing pipeline with user-defined logic for flow-oriented packet processing. FlowOS [36] is a flow-oriented programmable platform for middleboxes using a C API similar to the traditional socket interface. It uses kernel threads to execute flow-processing modules without terminating TCP connections. Similar to ClickOS, these platforms focus on packet processing rather than the application level. SmartSwitch [175] is a platform for high-performance middlebox applications built on top of NetVM [88], but it only supports UDP applications, and it does not offer a high-level programming model.

Eden is a platform to execute application-aware network services at the end hosts [32]. It uses a domain-specific language, similar to F#, and enables users to implement different services ranging from load balancing to flow prioritisation. By operating at the end hosts, it limits the set of network services that can be supported. For example, it would be impossible to implement in-network aggregation or in-network caching.

2.5 Summary

In this chapter, I discussed the basic concepts and related studies that are required for this thesis. I started with an overview of data centre computation which usually adopts a scaling-out architecture to implement online and offline data-parallel computation. To make this architecture effective, it is important to have a control plane that is programmable and runs efficiently in distributed servers. Having an effective control plane alone is not enough; achieving high efficiency in data processing requires further developing a performant data plane that can effectively support data computation and communication. Compared to the significant progress made in lifting the computation capability of a single server, the progress made for raising the communication capability is relatively small, mainly due to the over-subscribed networks which cannot provide sufficient cross-server bandwidth. Improving the communication capability requires a good utilisation of traffic reduction techniques, e.g., data aggregation and compression, combined with emerging network hardware led by software-defined networks. To enable fast development of application-specific network optimisation, it is important to enable programmability on top of networking infrastructure. The programmable network platform shall be easy to application developers; while providing a bunch of shared components that can easily execute custom optimisation programs.
Chapter 3

Chi: A Novel Control Plane for Distributed Streaming Systems

In this chapter, I focus on addressing the key coordination challenges in today’s data parallel systems. In particular, I study distributed stream processing systems. These systems have a rapidly increasing adoption in production clusters as they can unify online and offline computation. They often need to work with long-running, unpredictable data workload, and thus anticipate a control plane that can flexibly monitor and adjust various system aspects, leading to high data processing efficiency over a long period of time.

3.1 Introduction

In these days, real-world streaming workloads are hard to be handled by the streaming systems. First, these workloads often exhibit high temporal and spatial variability, up to an order of magnitude compared to the average load [103, 130]. Second, large shared clusters exhibit high hardware heterogeneity and unpredictable concurrent usage. Third, modern streaming systems expose a large parameter space, tuning which is demanding even for the most experienced engineers. Further, different users and jobs have a diverse set of Service Level Objectives (SLOs), leading to divergent configuration settings. Together these issues necessitate introducing continuous monitoring and feedback, as well as dynamic configuration into all aspects of streaming systems, ranging from query planning, resource allocation/scheduling, to parameter tuning.

Through our interaction with product teams and cloud operators, we identified the following requirements that a control plane should satisfy. First, it should be possible to define new custom control operations, tailored to different scenarios [77]. Second, this
should be achieved with only minimal effort by the developers and through a simple and intuitive API to minimise the chance of bugs, which are particularly challenging to detect in a distributed environment. Finally, the control overhead should be kept at minimum, even in the presence of high event throughput and large computation graphs. This is particularly critical in today’s landscape with different cloud providers competing hard to offer the best SLOs to their customers. Ideally the control plane should match or beat the data-plane SLO (usually in the order of seconds or less).

Unfortunately, to the best of our knowledge, none of the existing streaming systems are able to resolve the above control plane challenge. Heron [103] and Flink [44] have a monolithic control plane, i.e., one that supports only a limited set of predefined control policies (e.g., dynamic scaling and back pressure), and lacks a clean control-plane API, which makes it hard for users to define custom policies. Spark Streaming [174], adopts a Bulk-Synchronous Parallel (BSP) model [163] in which a set of events is buffered and processed as a batch. While this allows the system to modify a dataflow between batches, it has limited flexibility due to the hard batch boundaries and incurs high overhead due to the synchronisation and scheduling operations required.

To overcome the shortcomings of today’s systems and meet the aforementioned requirements, we investigate a novel control plane design for stream processing systems. Inspired by the idea of punctuations [161] being used for data operators, we propose introducing control plane messages into the data stream. By leveraging the existing data pipeline, control messages can be streamed at low latency and in a scalable fashion, without requiring any ad-hoc mechanism (subsection 3.7.2). However, this seemingly simple approach requires support from the underlying streaming infrastructure, to allow executing distributed control operations with consistency requirements, minimising synchronisation overhead and enabling an extensible control programming model.

In this paper, we describe Chi, a control plane built on the idea of embedding control messages in the dataflow to execute low latency control operations. We introduce a reactive programming model for handling control messages that allows users to encode a number of complex control policies. This hides the complex distributed nature of the underlying system, and provides developers an intuitive model to understand the boundaries of data events to which the control events are applied. We then design a mechanism to execute these control policies in an asynchronous manner at each operator, avoiding synchronisation and high runtime overhead. We show that using our programming model and execution mechanism we can efficiently implement a number of control policies ranging from basic functionalities such as checkpointing and replay, to advanced ones even with strong global consistency requirements, such as continuous
monitoring, plan re-optimisation, and parameter tuning (section 3.5). Finally, we also discuss how our control plane infrastructure can be easily parallelised by having a separate control loop per operation, making the control plane scalable.

To validate our claims and evaluate the runtime performance of our new control-plane design, we implement Chi on top of Flare, one of the internal stream processing systems in our enterprise. Flare is built on top of Orleans [39], a highly efficient distributed actor framework, and uses Trill [49] as the underlying stream processing engine. While we choose to showcase our approach on top of Flare for ease of implementation and deployment on our internal clusters, our design is not tied to a specific platform, and with some additional engineering effort, can be applied to existing systems, including Heron, Flink. Our evaluations use production workloads and industry standard benchmarks. Experimental results show that Chi is able to perform reconfiguration for a large stateful dataflow in 5.8 seconds on a 32-server cluster. Real-world use cases further show the effectiveness of Chi in helping developers automatically approach performance-critical system parameters and reduce latency by 61% resolving workload skew during runtime.

In summary, this paper makes the following contributions:

• An extensive study of today’s stream-processing workloads through the logs collected from more than 200,000 production servers.
• A scalable and efficient control-plane that allows a streaming system to efficiently adapt to changes in workloads or environment.
• A flexible control-plane API that enables developers to easily implement custom control operations.
• Evaluation of our prototype implementation on a 32-server Azure VM cluster using production workloads from a cloud service provider and across a large set of control operations, including dynamic scaling, failure recovery, auto-tuning, and data skew management.

3.2 Background

Chi is primarily designed for streaming systems that are based on a streaming dataflow computation model. In this section we provide the reader with the necessary background on the streaming dataflow computation model.

Many existing streaming systems, such as Naiad [129], StreamScope [107] and Apache Flink [44] adopt the streaming dataflow computation model. In this model, a computation job is represented as a directed acyclic graph (DAG) of stateful operators,
Fig. 3.1 Scaling-out control in action where the user is interested in changing the number of reducers in a word-count example.

where each operator sends and receives logically timestamped events along directed edges. Each operator maintains mutable local state. Upon receiving events, an operator updates its local state, generates new events, and sends them to downstream operators. Operators without incoming channels are source operators; those without outgoing channels are sink operators.

For instance, consider an example where the user is interested in tokenising sentences into words, and count the windowed accumulated count (e.g., per hour) of each word across all sentences in a data parallel way. This query can be expressed in a LINQ-style language as below:

**Example 1 (Word Count Example).**

```csharp
stream.SelectMany(line => Tokenize(line))
    .GroupByKey(word => word)
    .TumblingWindow(OneHour).Count()
```

An instance of a dataflow graph for Example 1 is shown in Stage I of Fig 3.1. Note that operators \( \{R_1, R_2\} \) maintain the accumulated counts for the words as their mutable local states. These states cover a disjoint set of keys, and jointly represent the entire key subspace, e.g., \( R_1 \) maintains the accumulated count for all words with the starting letter in the range \([`a`-`l`]\), while \( R_2 \) maintains those for the range \([`m`-`z`]\).

**Dataflow Computation Model:** Formally, a dataflow computation is represented as a DAG \( G(V,E) \), where \( V \) represents the set of operators, and \( E \), the set of edges connecting the operators, \( u \rightarrow v \) represents a directed edge from operator \( u \) to \( v \). We use \( \{ \cdot \rightarrow v \} \) to denote the input edges to \( v \), and \( \{ v \rightarrow \cdot \} \) the output edges from \( v \). An operator \( v \in V \) is described by a triple \((s_v, f_v, p_v)\), where \( s_v \) is the state of \( v \); \( f_v \) defines a function that captures the computation run on \( v \), i.e., \( f: s_v, m_{e_i \in \{ \cdot \rightarrow v \}} \rightarrow s'_v, \{ m'_{e_o \in \{ v \rightarrow \cdot \}} \} \),
meaning a function takes a single input message $m$, from an input edge $e_i$, and based on the current state $s_v$, it modifies the state of $v$ to a new state $s'_v$, and generates one or more messages on a set of output edges $\{m'_{e\in\{v\rightarrow\}}\}$. Operators without input edges are called sources, and operators without output edges are called sinks. For generality, we represent the properties associated with $v$ that are not part of state as $p_v$, e.g., $p_v$ can define the maximum memory used by $v$. An edge $e$ does not have any state but can hold properties $p_e$, e.g., the token size of windows before triggering back-pressure.

3.3 Motivation

In production, streams are mostly the logs constantly produced by servers deployed in cloud data centres. In the following, I collaborated with practitioners to perform an extensive measurement study by analysing the logs generated by more than 200,000 servers of a data-analytics cluster of a popular cloud provider. These clusters generate a significant amount of log data (10s PB/day) and queries on this data are executed by developers for debugging, monitoring etc. Given the data size, we require a large cluster with adequate network and compute resources to process data in real time. The ingested logs are then persisted in a hyper-scale distributed data store, and we perform batch processing jobs to analyse their statistical characteristics. Our setup is consistent with a recent analysis of Google production logs [53].

We begin by summarising the main results of our analysis and discuss the opportunities that arise.

Workload Unpredictability: The load on servers ingesting log events is highly variable. Figure 3.2 shows the heat-map of the normalised number of tuples produced per minute across a random subset of streams in our cluster. This shows two important phenomena. First, as evidenced by the different color patterns across horizontal lines, each stream exhibits a unique workload characteristic (spatial variability). Second, while some streams are dark (high volume) for most time, several streams exhibit busines (temporal variability), as shown by the different colours found on the same horizontal line.

There are three main reasons for such variability in the log events:

- **Heterogenous Workloads**: The clusters generating these logs handle a variety of workloads ranging from typical big data jobs (e.g., filter, transform, join) to iterative machine learning workloads on behalf of hundreds of teams.

- **Failures & Debugging**: Failures are more of a norm than an exception at large scale. These failures (e.g., networking issues, power issues etc.) generate a large amount
of error logs that lead to traffic bursts. In addition, when the data being ingested is not sufficient, developers temporarily activate more verbose logs to perform in-depth debugging, which results in a higher volume stream.

- **Diverse Stream Semantics**: The logs we ingest have diverse semantics (e.g., info, verbose, debug, error etc.) with various components in the service emitting with different characteristics. Therefore, a service at the lowest level (e.g., storage layer) naturally produces the most logs since most requests involve store input and output transactions.

To further understand the degree of variability and to better quantify the degree of temporal variability, in Figure 3.3 we show a box-and-whiskers plot with the Y-axis representing the delta in terms of event count per minute on a subset of incoming data streams. The height of the box shows the difference in counts between the $25^{th}$ and $75^{th}$ percentiles. Beside the significant difference in the behaviour of each stream, it is worth noting the high range of change observed within the same stream, denoted by the large number of outliers (blue dots in the figure). For example, for certain event streams, the event count can increase up to tens of millions in just one minute, indicating that a timely response is critical for sustaining load spikes.

**Data Diversity** Another key element in determining the optimal query plan and resource allocation is the data distribution. To analyse its dynamics, we focus on the *key selectivity*, defined as the number of tuples that fall into a particular bucket. To analyse the dynamics of this parameter, in Figure 3.4 we plot the selectivity of various grouping keys across time while in Figure 3.5 we plot the selectivity over time.

---

*Fig. 3.2 Heat-map of a subset of data streams in our ingestion workload. Each horizontal band signifies a unique data stream being ingested.*
3.3 Motivation

![Variability of delta count (order of millions) per minute (Y-axis) for a subset of incoming data streams (X-axis).](image)

Fig. 3.3 Variability of delta count (order of millions) per minute (Y-axis) for a subset of incoming data streams (X-axis).

across multiple grouping keys. The wide skew in the selectivity observed in both plots indicates that one-shot query planning (e.g., traditional cardinality-based optimiser) either globally or on a per-key basis will likely be sub-optimal over time.

**Multi-tenant control policies**: In our production environment, the same streaming system is used by many teams. Thus there is a big diversity of SLOs across queries from different teams, or even across different queries from the same team, leading to the need for multiple control policies to be active at the same time. For instance, we have many customers who are interested in consuming verbose, info and error logs. While verbose and error logs are used for debugging, info logs are often used to compute important metrics (e.g., billing metrics). One popular request made by our developers has been to provide stronger delivery semantics (e.g., exactly-once) for info logs and weaker delivery semantics (e.g., best-effort, at-least-once) for verbose/error logs. This highlights the importance of supporting multiple control policies either on a per-stream level of per-tenant level.

Multi-tenancy also introduces new opportunities in system management and optimisation. In our production traces we observe queries that share a significant overlap in terms of data sources and operators (e.g. users parsing logs in an identical way). Thus control policies could look at sharing computation across users or materialising intermediate data for later reuse. Further, with insight into the data streams and workloads, control policies could be used to automatically and transparently choose new data layouts (e.g., with partitioning) and storage engines (e.g., in-memory, database) that can improve execution efficiency. While all these optimisations are well understood...
Fig. 3.4 Selectivity of various grouping keys across time.

Fig. 3.5 Selectivity variability across the grouping key space.
in isolation [145, 147, 166, 134], applying them in an integrated manner to optimise the streaming workloads requires flexibility and scalability in the control plane.

**Takeaway:** Our trace analysis uncovered several interesting characteristics of our workloads – high volume (10s of PB/day), low latency requirement (SLO of seconds), widely diverse (100s of data streams) and dynamic (due to the nature of services producing these logs) with clear need for an efficient and extensible feedback-loop controls. Based on this, we can derive the following list of requirements for the control plane:

1. **Efficient and extensible feedback-loop controls:** Because of the diversity of our workloads, it is important to allow users or applications to make flexible late-binding decisions on their data processing logic for optimising performance. From an extensibility standpoint, it should be possible to integrate with dedicated components (e.g., policy controllers such as Dhalion [77]).

2. **Easy control interface:** Since we intend the control plane to be used by application developers, having an easier programming interface is critical to its adoption.

3. **Minimal impact on the data plane:** The control plane have limited or no impact on the latency and throughput of the data plane and it should be able to seamlessly cope with high control frequency and large dataflow graphs.

### 3.4 Design

The intuition behind embedding the control plane into the data plane is that this enables re-using the existing, efficient data-plane infrastructure and offers developers a familiar API to write control operations, i.e., the same used to handle data events. Further, having control messages directly trailing data events provides a natural way to create custom boundaries between sequences of events. This makes it easy to implement asynchronous control operations because control messages can be used to capture causal dependencies without requiring expensive global synchronisation operations.

In this section, we show how we incorporate these principles into our control-plane design and provide several examples to describe how we can easily build different control operations on top. We conclude the section by describing some of the more advanced features of our design and discussing how Chi can be adapted to support BSP-style computations.
3.4.1 Overview

Chi relies on the following three functionalities. First, channels between operators support exactly-once and FIFO delivery of messages. Back-pressure is used to stop the message propagation when the buffer of the downstream operator fills up. Second, operators process messages one at a time and in the order that they have been received. Finally, the underlying engine provides basic operator lifecycle management capabilities. Specifically it allows us to start, stop and kill an operator. These functionalities are already supported by Flare, our internal streaming system, but they can also be found in other existing systems [44, 103, 164, 174].

Our system design uses dataflow controllers that are responsible for monitoring dataflow behaviour and external environmental changes, and triggering control operations whenever needed. Users can define control operations, and submit them to the controllers.

A control operation is carried out through a control loop that is comprised of a dataflow controller and the dataflow topology itself. In general, a control loop consists of three phases: (Phase-I) The controller makes a control decision and instantiates a control operation with a unique identifier. A control operation is defined by implementing a reactive API (section 3.4.2), and has control configurations for each operator (e.g., the new topology to scale out the dataflow stream). The control operation is serialised into a control message (section 3.6). (Phase-II) The control message is broadcasted by the controller to all source operators of the dataflow. The control messages then propagate through the dataflow, interleaved with normal data messages. During the propagation, upon receiving a control message, each operator triggers the corresponding control actions—which can optionally attach additional data (e.g., the repartitioned state) to the control message—and broadcast the control message to all downstream operators. See section 3.4.2 and section 3.6 for more implementation details. (Phase-III) In the end, the sink operators propagate the control messages back to the controller, and the controller carries out post-processing.

We use Figure 3.1 to illustrate this process. We consider a case where the controller wants to increase the throughput by modifying the topology and adding a new reducer.

(I) At the beginning, there are two map operators \{M_1, M_2\} and two reduce operators \{R_1, R_2\} that compute word counting for ingested sentences. These operators are stateful. For example, \{R_1, R_2\} hold the accumulated count for all words, where \(R_1\) maintains the counts for all words starting with \[a'-'l']\], and \(R_2\) maintains those for \[m'-'z']\]. The controller \(C\) is responsible for monitoring
the memory usage of all operators and reconfiguring parallelism if needed. To simplify we omit the control messages that collect memory usage and focus on the parallelism reconfiguration process in the following discussions.

(II) Once the controller $C$ detects that the aggregated memory usage of the reducers goes beyond a threshold, it makes a reconfiguration decision to start a new reducer $R_3$ to increase memory provisioning. In the new topology, the states of $\{R_1, R_2, R_3\}$ should be repartitioned so that $R_1$ holds the word counts for the range ['a’-’h’], $R_2$ for ['i’-’p’], and $R_3$ for ['q’-’z’]. This reconfiguration needs to maintain the consistency of the states. It starts by broadcasting a control message with the new topology configuration to all the source nodes ($\to \{M_1, M_2\}$).

(III) When the source (mapper $M_1$ or $M_2$) this control message, it immediately blocks the input channel while processing the message, updates its routing table with the new topology, checkpoints it local state if needed, and broadcasts the message downstream ($\to \{R_1, R_2, R_3\}$).

(IV) When the reducer $R_1$ (or $R_2$) receives the control message, it blocks the input channel on which the message has been received. When the control messages from all input channels have been received, it updates its routing table and checkpoints its state. Next, it splits the state into two parts: the accumulated word counts for the range ['a’-’h’] and the range ['i’-’l’] (or for the ranges ['m’-’p’] and ['q’-’z’]) and attaches the state that needs to be handled by $R_3$, i.e., the word counts for ['i’-’l’] (or for ['m’-’p’]) to the control message and broadcasts along all output channels ($\to \{R_3, C\}$).

(V) When $R_3$ receives a control message, it blocks that input channel. If the control message originates from $R_1$ (or $R_2$), it records the state from the control message. When it receives control messages from all input channels, it proceeds to merge all the states received, generate the new state of the accumulated word counts for the range ['i’-’p’], and install a new function (from the control message) using the new state. Finally, it broadcasts on the output channel ($\to C$).

(VI) When $C$ receives control messages from all the expected sink nodes $\{R_1, R_2, R_3\}$, the scale-out operation is completed. The controller then keeps monitoring the memory usage of these operators in the new topology and can decide to scale out/in if needed. This forms a feedback-loop control.
3.4.2 Control mechanism

Next, we describe the core mechanisms underpinning Chi. We start by formally defining the dataflow computation model and explain how graph transformations occur. Then, we discuss the controller and operator APIs and provide an example control operation implementation. We provide a proof of correctness in section 3.4.2.

Graph Transitions through Meta Topology

Formally, a user control operation $C$ can be modelled as a transformation that converts a dataflow execution graph $G(V, E)$ to a new graph $G^*(V^*, E^*)$. For an operator $v$, such a transformation can change one or more entries in the triple $(S, f, P)$. For an edge $e$, such a transformation can optionally change $p_e$. In particular, since the operator state $S$ can capture state accumulated over a long time period, special care is needed to capture the transformation of states during reconfiguration (i.e., $G \rightarrow G^*$). That is, for $v^* \in V^*$, $S_{v^*}$ is defined by a transformation function $T$ on one or more nodes $\{v\} \subseteq V$, i.e., $T(\{S_v\}) = S_{v^*}$. In cases without ambiguity, we relax the notation and use $T^{-1}(v^*)$ to represent the set $\{v\} \subseteq V$ whose states $S_{v^*}$ depends on.

Most existing systems (e.g., [44, 164]) adopt a freeze-the-world approach to perform the transformation i.e., stop $G$ by appropriate checkpointing mechanisms, start $G^*$, migrate the old checkpointed state on $G$ to $G^*$ and resume the dataflow. However, this would likely trigger back-pressure, causing increased latency and loss of throughput, and in turn limits the frequency of execution and expressivity of dataflow reconfigurations. Therefore, in the design of Chi we opted for an asynchronous alternative: instead of affecting the transformation directly (i.e., $G \rightarrow G^*$), we introduce an intermediate meta topology $G'$, which the control operation can temporarily utilize in order to complete the transformation asynchronously. That is, during propagation of the the control messages, each operator broadcasts messages to its downstream according to $G'$. The operators in $G' - G^*$, after processing the control messages, will shut down; while the operators in $G' - G$ will only start processing data messages after finishing the processing of control messages. When the control message propagation finishes, the resulting topology will be equivalent to $G^*$.

We derive the meta-topology $G'$ for a control operation $C$ as follows. In the most general case, we set $G' = G \cup G^* \cup E_{V,V^*}$, where $E_{V,V^*} = \{(v,v^*)|v^* \in V^*, \forall v \in T^{-1}(v^*)\}$. In other words, the propagation graph for $C$ consists of all the operators and channels from both the old and new execution graph, and channels that capture the dependency relationship between states of the old and new operators. While
this approach can lead to doubling the size of the dataflow execution graph during reconfiguration, in practice, we can significantly reduce the propagation topology through appropriate pruning. For instance:

- **State invariance.** If a control operation does not change a node $v$’s state $S_v$, we can collapse the corresponding new node $v^* \in G^*$ with $v \in G$, and merge the input and output channels adjacent to $v$. For example, in Fig 3.6, $M^*_1$ ($M^*_2$) can be merged with $M_1$ ($M_2$) respectively.

- **Acyclic invariance.** Aggressively merge the old and new topology as long as we can guarantee the graph acyclicity. For instance, in Fig 3.6, we can further collapse $R^*_1$ ($R^*_2$) with $R_1$ ($R_2$) without breaking the acyclicity. This is guaranteed by (i) the functional query interface which ensures initial dataflow topology is acyclic as well as (ii) the pruning algorithm which ensures that no cycles is introduced during optimising a meta topology. For example, for scale-out/in reconfiguration, the pruning algorithm uses consistent hashing as the state allocation scheme to avoid introducing cycles when re-partitioning states.

By applying the above pruning rules repeatedly in Fig 3.6, we obtain the graph shown in Stage (IV) in Fig 3.1.

**Control API**

We next describe features of our control API that enable developers to implement complex control operations. CHI’s API allows expressing different behaviour across the following dimensions: (1) *spatial*, e.g., behaviour of $\{M_1, M_2\}$ being different than $\{R_1, R_2, R_3\}$, and (2) *temporal*, e.g., behaviour of $R_3$ when receiving the first control message vs. the last in Fig 3.1.

To enable such a flexibility, we abstract control operations and provide the following capabilities to users:
• **Configuration injection**: We allow the same control operation to carry different configurations for different operators. The configurations instruct operators to take different control actions. All configurations are injected into a control message (see Figure 3.13 for implementation details). The runtime transparently instantiates the control operation appropriately with the correct configuration at each operator. In the scaling-out example shown in Figure 3.1, the injected configurations need to instruct (1) mappers to reconnect output channels, (2) the reducers $R_1$ and $R_2$ to migrate states, and (3) the new reducer $R_3$ to accept migrated states. Given the implementation of the configuration injection (L1-11) in the Algorithm 1, the configuration of $R_1$ has $\text{SplitState}$ (L6) and $\text{LoadFunc}$ (L9) instructions, and the configuration of $R_3$ has $\text{MergeState}$ (L8) and $\text{LoadFunc}$ instructions.

• **Reactive execution**: CHi exposes a reactive (event-driven) programming interface that users can leverage to define control operations. A control operation comprises two sets of event handlers: those executed at the controller {`OnInitAtController`, `OnBeginAtController`, `OnNextAtController`, `OnCompleteAtController`, `OnDisposeAtController`}, and those executed at the operators {`OnBeginAtOperator`, `OnNextAtOperator`, `OnCompleteAtOperator`, `OnDisposeAtOperator`}. These event handlers offer users great flexibilities to collect metrics or modify configurations when the controller and operators receive the first, next and last control messages. The `OnInitAtController` is called when initialising the control operation and allows users to inject configurations into the control message. The `OnDisposeAtController` and `OnDisposeAtOperator` are called when the control operations are disposed. They are usually used for releasing resources. The runtime will handle the correct invocation of these handlers and the appropriate
state transitions as shown in Figure 3.7, thus allowing the users to express complex control logic in a safe manner.

**Blocking behavior:** In the example in Figure 3.1, operators always block the input channel upon receiving a control message from it. We find that this is a fairly common pattern in many control scenarios e.g., checkpointing and scale-in/out operations. To simplify implementation of complex control, we provide a layer of abstraction that allows the users to implement their control operations in both blocking and non-blocking ways. We do this by classifying control messages into two categories: (1) **blocking:** where the operator blocks the corresponding channel on which the control message is received and subsequently unblocks it only when all the control actions are finished on that operator, and (2) **non-blocking:** where the operator does not block the input channel and continues to receive other data/control messages on that channel. We believe such abstraction is useful for users to express more advanced control operations. For instance, blocking control messages are usually useful for control that affects states, while non-blocking control messages are useful for the other cases, e.g., monitoring.

**Example:** We demonstrate usage of the control API using the example shown in Figure 3.1 where we want to scale-out from $G$ into $G^*$ through $G'$ (shown in Figure 3.7). Algorithm 1 shows a pseudo implementation of the dataflow reconfiguration control operation. Once the reconfiguration decision is made, the developer creates a blocking control message. In `OnInitAtController`, a blocking control message is created and injected with configurations for triggering control actions at different operators. Specifically, as explained in section 3.4.2, an edge $(v, v^*) \in E_{V;V^*}$ describes a dependency between the states of $v^*$ and $v$, operator $v$ needs to be configured to split its states and ship the corresponding part of the state to $v^*$, while operator $v^*$ needs to be configured to load and merge the received states (L5-8). For example, as in Figure 3.1, $R_1$ needs to split the state and ship the accumulated counts for the range $['i'-'l']$ to $R_3$. Once the topology or state is changed, the operators need to reset associated computation functions (L9). Such a control message is broadcast to source operators. It first initialises *session variables* that hold migrated states and control actions (L15-16) in the `OnBeginAtOperator` function. The migrated state is gradually accumulated until all parts of the state are received by `OnNextAtOperator` (L18-19). Once receiving all messages, the operator (shown in the `OnCompleteAtOperator` function) performs control actions including move away states that do not hold any more according to the new state key range (L23-25), merges states given by others (L26-27), and resets the function (L28-29). Once the controller receives the control messages from sink
Algorithm 1 Dataflow Reconfiguration Operation

**Assumption:** Each operator has a context (ctx). In the controller, control operation can access $G'$ and $G^*$ as well as $E_{V,V^*}$ (See section 3.4.2). Users can create session variables (names start with the $ mark) that live through function scopes. A graph vertex has properties including stage (stg), state key range (keyrange), and function (func). A control message has properties including source (src), destination (dest), configuration dictionary (confs) and control payload (payload). A configuration is assigned with a list of instructions.

1: function OnInitAtController  
2: msg := new BlockingControlMessage()  
3: for v in $G'$ do  
4:  instructions = []  
5:  if v in Src($E_{V,V^*}$) then  
6:     instructions.Add(new SplitState(v.keyrange))  
7:  if v in Dest($E_{V,V^*}$) then  
8:     instructions.Add(new MergeState(v.keyrange))  
9:     instructions.Add(new LoadFunc(v.func))  
10:    msg.confs[v] := instructions  
11: return msg  
12: function OnCompleteAtController  
13: ctx.CompleteOperation(this)  
14: function OnBeginAtOperator(msg)  
15: $inState := new Dict<Key, State>()  
16: $instructions := msg.confs[msg.dest]  
17: function OnNextAtOperator(msg)  
18: for key, state in ParseState(msg.payload) do  
19:    $inState[key] := state  
20: function OnCompleteAtOperator  
21: outState := new Dict<Key, State>()  
22: for i in $instructions do  
23:    if i is SplitState then  
24:       for key, state in ctx.state.Split(i.keyrange) do  
25:          outState[key] := state  
26:    if i is MergeState then  
27:        ctx.state.Merge($inState, i.keyrange)  
28:    if i is LoadFunc then  
29:        ctx.func.Load(i.func)  
30: return outState
operators, the control operation is marked completed in `OnCompleteAtController` (L12).

**Correctness Properties**

CHI provides correctness properties that can help users prove the correctness of their control operations.

**Theorem 1.** Consider a control operation that changes a graph from $G$ to $G^*$ using a control message and a state transformation function $T$. The control operation has the following properties:

1. The control operation will terminate in finite time.
2. If a pair of operators $v,v'$ satisfies (a) $v \rightarrow v'$ is an edge in $G$ or $G^*$, or (b) $v \in T^{-1}(S_{v'})$, then $v$ will always invoke `OnCompleteAtOperator` before $v'$.

**Proof.** 1° Proof of Property 1. Assume $G$ is the topology before the control operation starts, and $G^*$ is the topology after a control operation finishes. Both $G$ and $G^*$ are directed acyclic graphs. Furthermore, given that $E(V,V^*)$ always directed from $V$ to $V^*$, $G \cup G^* \cup E(V,V^*)$ is a directed acyclic graph.

Termination is guaranteed by the FIFO exactly-once delivery of channels and the in-order generation of control messages. As channels are reliable, messages will eventually be received as long as the operators are alive. Furthermore, an operator in $G \cup G^* \cup E(V,V^*)$ is always reachable from some source operator, or it is a source operator itself.

2° Proof of Property 2. If (a) $v \rightarrow v'$ is an edge in $G$ or $G^*$, or (b) $v \in T^{-1}(S_{v'})$, then there is an edge $v \rightarrow v'$ in the meta topology. $v'$ will invoke `OnCompleteAtOperator` only after it receives control messages from all input channels, including $v \rightarrow v'$. And $v$ will only broadcast control messages along $v \rightarrow v'$ after the invocation of `OnCompleteAtOperator`. Hence, $v$ will always invoke `OnCompleteAtOperator` before $v'$.

Furthermore, we introduce safe blocking control operations—a special type of blocking control operations whose control actions at each operator only read/write the corresponding operator state in `OnCompleteAtOperator`. Safe blocking control operations have stronger properties—the semantics of safe blocking control operations is equivalent to the freeze-the-world approach—which can facilitate users to understand the semantics and correctness of their customised control operations.

In the following, we prove the correctness of safe blocking control operations. We use a similar model as [50] to describe the distributed system that executes a streaming
dataflow: The execution process of a dataflow system is a sequence of events. An event $e$ is a five-element tuple $(v, s_v, s'_v, m_{e_i}, \{m'_{e_o}\})$, describing an atomic action that an operator $v$ receives a message $m_{e_i}$ from an input channel $e_i$, updates its internal state from $s_v$ to $s'_v$, and sends zero or more messages $\{m'_{e_o}\}$ along some output channels $\{e_o\}$ respectively. Note that the sequence of messages received along a channel is an initial sequence of the sequence of messages sent along the channel. Therefore, we do not explicitly model the channel state, as the state of a channel is always the sequence of messages sent along the channel excluding the messages received along the channel.

A global state of the dataflow, denoted by $S$, is the set of all operator and channel states. The occurrence of an event can change the global state. An event $e$ can occur in a global state $S$ if and only if (1) the state of $v$ in $S$ is $s$ in $e$, and (2) the head of the state of $e_i$ in $S$ is $m_{e_i}$.

Let $seq = (e_i, 0 \leq i < n)$ be a sequence of events, and the global state before $e_i$ be $S$. We say $seq$ is a computation of a system if and only if event $e_i$ can occur in $S$.

Next we show that for safe blocking control operations, we can have stronger properties that can facilitate users to understand the semantics and correctness of customised control operations. Consider a control operation in a dataflow system that starts at state $S_0$, takes a computation sequence $seq = (e_i, 0 \leq i < n)$ and ends at state $S_n$. To facilitate our discussion, we group $\{e_i\}$ in $seq$ into three types: (1) control events—the events where an operator when the operator receives control messages, takes control actions and optionally broadcasts control messages, (2) pre-control events—the events that occur at an operator before the operator receives any control message along the same channel, and (3) post-control events—the events that occur at an operator after the operator receives a control message along the same channel.

Specifically, we want to prove a safe blocking control operation can be permuted to an equivalent computation sequence where all the pre-control events happen before all the control events, which in turn happen before all the post-control events, as shown in Figure 3.8. Intuitively, the control events can be treated as if they happened all at the global synchronisation barrier as in a freeze-the-world control approach. Formally,
Theorem 2. Consider a safe blocking control operation of Chi in a dataflow system that starts at state $S_0$, takes a computation sequence $\text{seq} = (e_i, 0 \leq i < n)$ and ends at state $S_n$. There exists a computation $\text{seq}'$ where

1. all pre-control events precede all control events, and
2. all control events precede all post-control events, and
3. all pre-control events precede all post-control events.

Proof. Assume that there is a control event $e_{j-1}$ before a pre-control event $e_j$ in $\text{seq}$. We shall show that the sequence obtained by interchanging $e_{j-1}$ and $e_j$ must also be a computation. Let $S_j$ be the global state immediately before $e_j$, $0 \leq i < n$ in $\text{seq}$.

Case 1$^o$ where $e_j$ and $e_{j-1}$ occur on different operators. Let $v$ be the operator at which $e_{j-1}$ occurs, and let $v'$ be the operator at which $e_j$ occurs. First, there cannot be a message sent at $e_{j-1}$ which is received at $e_j$ because $e_{j-1}$ sends out control messages, but $e_j$ receives a data message. Second, $e_{j-1}$ can occur in global state $S_{j-1}$. This is because (a) the state of $v'v$ is not altered by the occurrence of $e_{j-1}$ because $e_{j-1}$ is in a different operator; (b) if $e_j$ is an event in which $v'$ receives a message $m$ along a channel $e$, then $m$ must have been the message at the head of $e$ before $e_{j-1}$ since $e_{j-1}$ and $e_j$ receive messages from different channels. Third, $e_{j-1}$ can occur after $e_j$ since the state of $v$ is not altered by the occurrence of $e_j$. Because of the above 3 arguments, the sequence obtained by interchanging the $e_j$ and $e_{j-1}$ is a computation.

Case 2$^o$ where $e_j$ and $e_{j-1}$ occur on the same operator. First, $e_{j-1}$ can occur in global state $S_{j-1}$. This is because (a) $e_{j-1}$ does not alter the operator due to the safe blocking control operation constraint; (b) $e_{j-1}$ and $e_j$ receive messages from different channels. Second, $e_{j-1}$ can occur after $e_j$ since $e_{j-1}$ does not read the operator state due to the safe blocking control operation constraint. Because of the above 2 arguments, the sequence obtained by interchanging $e_j$ and $e_{j-1}$ is a computation.

In conclusion, the sequence obtained by interchanging $e_{j-1}$ and $e_j$ must also be a computation.

Similarly, one can show that if there is a post-control event $e_{j-1}$ before a pre-control/control event $e_j$, the sequence obtained by interchanging $e_{j-1}$ and $e_j$ must also be a computation. The only difference is that $e_j$ and $e_{j-1}$ cannot occur on the same operator due to the fact that an operator can start processing post-control messages only after it has received control messages from all input channels. By repeatedly applying the above interchanges, one can easily see that we can generate a computation $\text{seq}'$ that satisfies the 3 requirements as listed in Theorem 2.

In the end, we show how Theorem 1 and Theorem 2 can help Chi users prove the correctness and semantics of their control operations. Take Algorithm 1 as an
example. This algorithm is a safe blocking control operation. According to Theorem 1, Algorithm 1 will always terminate in finite time. According to Theorem 2, Algorithm 1 is equivalent to the freeze-the-world control approach: (1) suspends the dataflow, (2) finishes processing all the messages on the fly, (3) starts the control operation and waits for its completion, and (4) resumes the dataflow. It is easy to see the correctness of the freeze-the-world approach, and thus the correctness of Algorithm 1 follows.

3.4.3 Advanced functionalities

We conclude our description of Chi’s design by listing some of the functionality that we added based on our experience with production workloads.

Multiple Controllers Thus far our description assumes a single dataflow controller that enforces a control operation. Our design is able to naturally scale out the controller by allowing multiple concurrent controllers for a single dataflow. For instance, users can have one controller per desired functionality, e.g., one controller takes care of monitoring the health of the dataflow execution, another one periodically checkpoints operator states, while yet another takes charge of reconfiguring dataflow execution graphs in case of workload spikes. Multiple controllers can function at the same time as long as the serialisability of control messages for different control operations are guaranteed. In order to perform cross-dataflow control operations, e.g., coordinating resource allocation across multiple dataflows, we can introduce a global controller that can interact with each dataflow’s controller.

Broadcast/aggregation trees. In practice, a dataflow graph usually has a large number of source operators (and sometimes, sink operators). In such a topology, the controller can quickly become a bottleneck due to the large fan-in/out. To mitigate this, we leverage a simple technique such as inserting a spanning broadcast (aggregation) tree before (after) the source (sink) operators.

Dealing with congestion/deadlock. When congestion arises, e.g. due to network or CPU bottlenecks, our back-pressure mechanism is triggered and all messages, including control messages, are delayed. To enable back-pressure, each dataflow operator constantly check the sizes of its input channels, and notifies a sender to suspend sending if its input channel is full. By repeatedly notifying senders to suspend, the dataflow can propagate the congestion signals back to all relevant sources, and cap the memory usage.

The triggering of back-pressure could be particularly critical if these messages are part of a control operation to alleviate congestion. One option might be to have two separate queues and give control messages higher priority, so that in case of congestion
they are delivered first. This, however, would break the ordering of control and data messages, thus making it hard to maintain consistency. Therefore, we wait for finishing processing the message. This is similar to the approach taken by other systems such as Flink [44] and Spark Streaming [174].

**Various Control Messages**. In CHI, users are offered four types of control messages. They can select the appropriate type based on the purposes of control operations. In addition to the blocking and non-blocking control messages, there are two extra kinds of control messages that are also useful. One is the traditional control message that is be inserted into dataflow operators using side-channels. For example, this message is typically used for shutting down non-responding operators. The other is the prioritised control messages that can by-pass messages in a shared channel. This message is, for example, used for checking the liveness of a channel.

**Fault tolerance**. One of the main benefits of integrating control and data plane is that failures in the control plane are handled in the same way as failures in the data plane. More specifically, if control messages are lost, the underlying data channel is responsible to retransmit them until they are acknowledged by the other end. In case of network partition, the controller will eventually time out and will restart the control operation.

CHI allows developers to implement various policies to handle operator failures. For ease of adoption, we implement a checkpoint-replay policy on top of CHI for handling operator failures by default. This policy will first rollback the data flow stream to the last checkpoint and then it will re-insert the lost control messages. Failures in the controllers are handled through checkpointing its own state into a durable store and restoring the state upon launch of a new instance of the failed controller (typically handled by a watchdog).

### 3.4.4 Compare CHI with existing models

We next compare CHI with existing control models for streaming systems (Table 3.1 summaries the comparison). There is a variety of control models being developed, tailored for different stream computation paradigms, i.e., BSP-based micro-batching versus record-at-a-time. We compare the different models according to the consistency, ease-of-use, overheads and scalability.

**Consistency**. Many useful control, e.g., scaling-out, state repartitioning and checkpointing, demand consistency. Such a demand can be easily satisfied in BSP systems because the blocking barrier between parallel computation allow all nodes to synchronise with each other (named Synchronous Global Control). The same consistency can
be satisfied in Chi through blocking control messages embedded in the data plane. They can be viewed as an asynchronous global barrier moving inside the dataflow.

If required, Chi can implement the synchronous global barriers as in BSP by letting the dataflow controller repeatedly act as the barriers, as shown in Figure 3.9. When a stage starts, the controller generates (1) a blocking control message (denoted by $S$) that installs tasks at operators, followed by (2) a data message (denoted by $D$) that describes input data, and end with (3) a blocking control message (denoted by $E$) that marks the completion of a stage. When receiving all stage completion messages, the controller can proceed to the next stage by repeating the same sequence of messages.

The barrier semantics can also be implemented in record-at-a-time systems by freezing the entire dataflow before reconfiguration. However, freeze-the-world requires the system to halt like the synchronous global barrier. To avoid this, other record-at-a-time systems, e.g., SEEP [45], reconfigure workers without synchronisation (named Asynchronous Local Control) at the cost of the barrier semantics. Such relatively weak consistency can be achieved in Chi as well by using non-blocking control messages.

However, the lack of barriers makes it hard to implement many useful control operations. Consider a dataflow that applies filters to a stream that has $x$ and $y$ fields
(shown in Figure 3.10), where the filter parameters are stored in two data stores due to privacy regulation. Hence, the stream must be replicated to be filtered in parallel. The filter results are joined in the end. In a system that provides only asynchronous local controls, it is unable to support control requests that request simple concurrent reconfigurations, such as changing filter selectivity from $x < A$ and $y < B$ to $x < C$ and $y < D$. This is because each replicated tuple must be processed with all new configurations at once, rather than a mixture of them. To ensure consistency, as shown by the Algorithm 3 in SEEP [45], the dataflow has to block ingestion, recover operator checkpoints, apply new configurations, replay the output of the replicate operator since its last checkpoint, and unblock ingestion.

**Ease-of-Use.** In addition to consistency guarantees, **CHI** also provides a flexible control plane interface with comprehensive automation support. **CHI** users can declare custom control operations following the familiar reactive paradigm. The **CHI** runtime is responsible for managing the states of control operations, performing operations asynchronously, and handling failures in both control and data planes.

In contrast, existing control models lack programmability and runtime support. For example, Flink implements a distributed checkpoint algorithm which is unable to support general control operations that, for example, require changing state. More recently, Dhalian studied the high-level representation of a control policy, with a particular focus on identifying symptoms and therapies of detected anomalies. It relies on the underlying system, i.e., Heron, to provide actual reconfiguration capability. Hence, Dhalian does not have a control plane runtime as **CHI** which can be applied onto general record-at-a-time streaming systems.

**Overhead.** When reconfiguring a BSP system, the entire dataflow is halted. This is particularly detrimental for online streaming systems and affects both latency and throughput. Furthermore, the BSP barriers severely constrain the frequency and timing
of control operations. To amortise the scheduling and communication cost, a BSP barrier interval is often set to be no smaller than seconds [164].

As mentioned above, reconfiguring a record-at-a-time system requires freeze-the-world as in Flink (section 3.7), or re-computation through data replays as in SEEP. On the one hand, freeze-the-world incurs a similar overhead than the synchronous global barrier. On the other hand, replaying data is expensive in production, despite reconfigured operators being often already short of resources. Our traces show that buffering all intermediate outputs in preparation for replays require a significant amount of memory, several orders of magnitude larger than the one consumed by the operator computation state. Replaying such a large buffer state not only consumes significant bandwidth, but also blocks the operator for a long time.

In CHI, changes are applied on the asynchronous global barriers. There is no need for freeze-the-world or data replay. Blocking behaviours are local to each operator. There is no global blocking that freezes the entire dataflow, thus reducing data plane overheads.

**Scalability.** Existing streaming systems usually rely on a separate control plane. This duplicates resources for the control plane to deal with typical distributed system issues such as fault tolerance and scalability, especially when dealing with large execution graphs. CHI, instead, embeds the control plane into the data plane. As the data plane is designed to evolve to handle high volume of data, CHI benefits from the same optimisation techniques introduced for the data plane, e.g., high-scale state storage backend, and broadcast/aggregation trees for large fan-in/out in dataflows.

Furthermore, existing systems mostly adopt centralised controllers. Reconfiguring the dataflow requires delivering a large number of control messages, which makes the controller a single-point of failure and performance bottleneck. On the contrary, the workload on a CHI controller can be easily distributed. Dataflows are managed by parallel dataflow controllers. Each dataflow controller is further partitioned by allowing each control operation to be managed by an operation controller. All these controllers are executed in parallel on multiple servers, thus removing the single-point of failure and performance bottleneck.

### 3.5 Application Examples

To demonstrate the flexibility of our approach and show how its API is used, we illustrate three control applications that we implemented using CHI.
Continuous Monitoring. Due to unpredictability of our workloads, unlike traditional batch processing systems where jobs can be tuned offline to achieve optimal performance, streaming pipelines have to be continuously monitored and optimised on-demand in order to achieve high performance and robustness. We show an example of using Chi to continually collect measurement data of all operators, a foundational block for detecting and acting upon interesting characteristics of the system such as overload/underload, skew/drift in data distribution, intermittent environment-related bottlenecks and mitigating stragglers.

We show an implementation of the monitoring control operation in Algorithm 2. The monitor operation creates a non-blocking control message that is injected with a CPU metric collection collection action (L1-5). When this message pass through an operator, it first create a summary dictionary that contains metrics per operator (L15-17). It is then augmented it with upstream measurement data (L18-21), and finally with local metrics (L23-25). The local summary is then returned to the runtime and packed into the control payload, and keep propagating until reaching the controller (L26). The controller repeats the similar steps to combine measurement data from all sink operators (L6-11) and finally print the global summary at the screen (L13). In fact, there are many more metrics that can be collected, for example, per-path latency or per-operator workload.

Notice that the controller no longer needs to ping each individual operator separately to collect statistics. Instead, the metrics are collected and aggregated along in a scalable tree fashion. section 3.7 shows the evaluation of using the monitoring control for collecting per-join-key cardinality that helps in identifying skewness in the join space (which can then be used for straggler-mitigation).

Dataflow Reconfiguration. Dataflow reconfiguration is important for several interesting use-cases such as adding/removing operator from a given query, increasing degree of parallelism of an operator and exploiting computational reuse.

Besides scale-in/out, one can also carry out more complex reconfigurations including changing the query plan. For instance, Figure 3.11 demonstrates how we can change the
Algorithm 2 Continuous Monitoring Operation

1: function OnInitAtController
2:    msg := new NonBlockingControlMessage()
3:    for v in G' do
4:        msg.confs[v] := new CollectMetric('cpu')
5:    return msg
6: function OnBeginAtController(msg)
7:    $summary := new Dict<Key, Metrics>()
8: function OnNextAtController(msg)
9:    for key, metrics in ParseMetrics(msg.payload) do
10:       if key not in $summary.keys then
11:          $summary[key] := metrics
12: function OnCompleteAtController
13:    print $summary
14:    ctx.CompleteOperation(this)
15: function OnBeginAtOperator(msg)
16:    $summary := new Dict<Key, Metrics>()
17:    $action := msg.confs[msg.dest].Single()
18: function OnNextAtOperator(msg)
19:    for key, metrics in ParseMetrics(msg.payload) do
20:       if key not in $summary.keys then
21:          $summary[key] := metrics
22: function OnCompleteAtOperator
23:    if $action.metrics not in ctx.metrics then
24:        ctx.StartMeasure($action.metrics)
26:    return $summary
3.6 Implementation and Discussion

query plan to alleviate stragglers when we find skew in a streaming join query. Assume originally streams $A$ and $B$ are joined using a shuffle join $[100]$, where mappers read data from $A$ and $B$ respectively, and partition and route the data to the corresponding reducer based on the join key; reducers on receiving the data, join the tuples with the same key together. Due to skewed key space, reducer $R_1$ receives much more data than $R_2$. At this point, we can change the query plan by adding reducers $\{R_3, R_4\}$ to share the load for $R_1$. The idea is to let $A$ (assume $A$ has a significantly higher workload than $B$) partition $R_1$’s load into $\{R_1, R_3, R_4\}$; $B$ broadcasts $R_1$’s load to $\{R_1, R_3, R_4\}$. This reconfiguration requires $R_1$ to replicate its internally maintained hash table of data from $B$ to $R_3$ and $R_4$, while partitioning and redistributing the hash table of data from $A$ to $R_3$ and $R_4$.

**Auto Parameter Tuning.** Big data systems have many parameters that are very hard to tune even for very experienced engineers. CHI can be used for automatic parameter tuning by leveraging both monitoring and dataflow reconfiguration in a tight control loop to simulate A/B testing of multiple instances of a single query. For instance, many existing streaming systems use micro-batching to trade off between latency and throughput. Whereas a large batch size provides good throughput, it does so at an increased latency. Tuning the right batch size is a tricky problem. One solution through CHI would be to continuously monitor latency of the data plane and adjust the batch size when we see considerable fluctuations in the observed latency until we obtain the maximum throughput with the desired latency.

### 3.6 Implementation and Discussion

**Distributed runtime.** To showcase the performance and flexibility of CHI, we implemented it on top of **FLARE**, a streaming system used internally by our team. **FLARE** is built on top of Orleans $[39]$—a virtual actor framework—as a runtime and Trill $[49]$ as the operator-level stream processing engine. By leveraging Orleans, **FLARE** achieves decentralised scalability, specifically: (1) nodes can join/leave the cluster without notifying master nodes, and (2) the lifecycle of an actor is automatically managed by the platform, which transcends the lifetime of in-memory instantiation and particular servers.

An operator in CHI has a stacked architecture embedded into the FLARE operator which is a single-threaded environment, as shown in Figure 3.12. (i) The communication layer provides FIFO exactly-once data communication channels with back-pressure that mimics the TCP. (ii) The message dispatcher/multiplexer invokes the corresponding
Chi: A Novel Control Plane for Distributed Streaming Systems

Fig. 3.12 A Chi-enabled Flare operator architecture.

processing module based on the types of messages, and multiplexes their outputs down to the communication layer. (iii) The data processor applies a Trill pipeline onto data messages. (iv) The control processor invokes a series of local control actions based on received control messages. It loads the corresponding control configuration, manages state machines, and invokes user-defined control actions accordingly.

Flare further provides the following functionalities to simplify control operations: (i) a user-friendly state management functionality which models operator states as a key-value map and can automatically split and merge states using a user-defined partition scheme on the key, (ii) a query compiler (extensible with custom optimisation rules) similar to Spark’s catalyst [29] that converts LINQ queries into a distributed dataflow, and (iii) file server that allows runtime extensibility, where users can upload new control operation dependencies and have them loaded at runtime.

Custom serialisation. Control messages may carry/propagate large payloads including configurations and states/metrics/etc. from each operator. Since serialisation and deserialisation at each operator may introduce unnecessary overhead, we implemented a zero-copy operation that allows us to extract the necessary pieces (e.g., configuration, payload of interest) from the byte buffers without deserialising the entire message.

Fig 3.13 shows the structure of a control message. Each message includes three basic components: 1. Metadata field that is used to ensure FIFO exactly-once delivery; 2. Configuration payload field to store configurations for different operators; and 3. Data payload for an operator to insert any control-specific data e.g., re-partitioned state while scaling-out. Control messages are generated either by a controller (before control instructions are applied to any operators) or by an operator (after a control operation has triggered but before control message has been propagated to succeeding operators).
3.6 Implementation and Discussion

![Diagram of Control Program Structure](image)

Fig. 3.13 Control program structure in CHI
Portability. While we implemented our approach on top of Flare for ease of implementation/deployment in our internal clusters, our design is not tied to a specific platform. Chi can be applied to other systems as long as the systems provides FIFO at-least-once or exactly-once delivery semantics, and in-order message processing. These modest requirements are offered by most contemporary streaming systems such as Flink [44] and SEEP [45]. A typical porting plan includes (1) porting the communication layer if the underlying system does not provide FIFO exactly-once message delivery, (2) porting the message dispatcher/multiplexer and the control processor, and (3) reusing the existing data processor.

Limitations. Currently, the deployment of Chi has two limitations. First, Chi assumes a multi-user environment owed by a single administration. Users are thus unlikely to write malicious/attack code in control programs. However, this limitation rarely exists in the systems running targeted by Chi as, in these days, these systems are mostly operated within dedicated private clusters. To extend its deployment into a multi-tenant environment, that would require extending the Chi control program compiler to be capable of checking malicious/unsafe behaviours, such as the long-running loop or leaky resources.

The second limitation of Chi is the control termination property guaranteed by acyclic dataflow. When serving streaming workloads, acyclic dataflow is the dominating computation paradigm due to the prevailing use of one-pass (i.e., streaming) algorithms which are the keys of generating near-real-time, if not real-time, computation results. Extending the use of the Chi into cyclic dataflow (e.g., graph processing and machine learning training) would require developing asynchronous mechanisms that can let message propagation stop propagating inside a cycle. This could be potentially achieved by adding an extension to the meta-graph pruning algorithm that removes certain edges on a cycle while preserving the data/state migration order.

3.7 Evaluation

In this section, we evaluate the performance of Chi using a number of micro-benchmarks and two real-world benchmarks. The first real-world benchmark focuses on dynamically scaling in/out resources while the second assesses Chi ability to handle control and data failures. These results demonstrate that Chi incurs negligible overhead, even under high data/control load and large dataflow graphs, and it is able to quickly react to changes in the workload or failures. To show the flexibility of our approach, we
also report on the experiments with two more advanced case studies, i.e., handling a skewed key distribution and auto-tuning for meeting SLOs.

### 3.7.1 Experimental setup

Our experimental cluster comprises 32 DS12v2 instances in Azure. Each virtual machine has 4 vCPUs, 28 GB RAM and a 10 Gbps network connection. We consider one public workload, the Yahoo! Streaming Benchmark (YSB) [8], and one private workload based on production traces, IPQ1, which consists of multi-stage queries with complex window aggregations and streaming joins. YSB is rather light-weight in terms of data handling (bytes/event) while IPQ1 is computationally heavier (KB/event). As explained in section 3.6, we implemented Chi on top of Flare, a streaming engine built on .NET CLR and is used internally by our team. When required, we compare against Drizzle [164], a fork of Apache Spark v2.0.0 (a BSP-style engine) with an optimised scheduler for streaming scenarios, and Apache Flink v1.3.2 [44] (a continuous dataflow engine). For all our experiments, we warm up the JVM and CLR before taking measurements to discount bootstrapping effects.

**Flare performance** To help understand whether the underlying engine that we used for Chi is competitive with existing systems, we compare the base performance of Flare against Flink and Drizzle. We show the results in terms of throughput and latency for the three systems when using the YSB and IPQ1. For the throughput experiments (Figure 3.14a), we set the latency SLO to 350 ms and maximize the throughput while for the latency experiment we fix the ingestion rate at 20 million tuples/s and minimise the latency (Figure 3.14b). Following a common practice [164], we define latency as the time it takes all events in the window to be processed after the window has ended. For instance, if a window ended at time $a$ and the last event from the window was processed at time $b$, the processing latency for this window is said to be $b - a$. The results obtained for Flink and Drizzle are consistent with what previously reported in literature [164, 81] and confirm that Flare’s performance is comparable with these systems.

### 3.7.2 Micro-benchmarks

In this sub section, we study the interplay between the control and data planes. Specifically, we are interested in the operational overhead and scalability aspects of Chi. To this end, we vary the data-plane load under three different CPU regimes (resp. 50%, 75%, and 90%) and set the ingestion rates for the two workloads YSB and
IPQ1 accordingly. For the computation-heavy IPQ1 this resulted into 20, 40, and 60 millions events/s (corresponding to 56%, 73% and 94% average CPU in our 32-server cluster) while for the communication-heavy YSB, this led to 140, 180, and 220 millions events/s (resp. 51%, 74%, and 89% average CPU). For the control load, we consider two instances of the control program, using blocking (B-Control) and non-blocking (NB-Control) messages. To accurately isolate the impact of Chi and avoid biasing the result with custom control logic (e.g., CPU-intensive user code), we use only NoOp control messages in our experiments.

**Does control plane affect the data plane?** We now assess the overhead introduced by Chi. In Figure 3.15, we show the relative increase in latency for the two workloads when varying the control-plane load from one control program/s (representative of dataflow management tasks, e.g., checkpointing, and reconfiguration ones, e.g., scale in/out) to 100 control program/s (representative of monitoring tasks). This range
covers all control scenarios that we have observed in production and we believe that
the same should hold for the vast majority.

These results show that both non-blocking and blocking have low overhead. This
is due to the fact that neither requires global synchronisation — control events are
propagated and processed asynchronously just like data events. For example, at the
highest load of the computation-intensive IPQ1 workload (60 million events/s), Ch examines lower than 20% latency penalty to the data plane, even for high control load
(100 messages/s). This is important because the average CPU utilisation was already
at 94% (the latency could be further reduced by scaling out). As expected, blocking
control programs are more expensive than non-blocking ones. This is due to the fact
that blocking control programs require local synchronisation which block the input
channel temporarily.

**Does a busy data plane limit the control plane?** Next, we study the impact
of the data-plane load on the completion time of control programs. The concern is
that by merging control and data events, high data-plane load might negatively affect
the performance of the control programs. To verify this, we repeat the experiment
in Figure 3.15 and we measure the completion time of the blocking and non-blocking
programs (see Figure 3.16). As a reference point, we also show the latency of the
data plane (“data latency” bar in the graph). The message latency is defined as the
difference between the timestamps of the message \( M \) entering the system and the
timestamp of when the last message triggered by \( M \) leaves the system. In addition to
queuing delays, the completion time of a data (resp. control) message also depends on
the complexity of the actual data processing (resp. control logics). Hence, a control
message can complete faster than data messages if there is no backlog and messages
are processed immediately upon receiving.

In most cases control programs complete relatively quickly in comparison to the data
message delivery. Also, when the control load is low (one or ten control program/s), the
completion time is relatively unaffected by the increasing data-plane load. We observe,
however, that at the highest load (resp. 220 million events/s for YSB and 60 million
events/s for IPQ1), the completion time for a control program is similar or higher than
the data latency. This is particularly evident for the blocking programs. The reason is
because the latter introduce a local synchronisation at each operator and, at high load,
there is a higher variability in latencies along different paths in the dataflow (e.g., due
to work imbalance). As already discussed in section 3.4.2, however, we observe that
blocking and non-blocking control programs are semantically equivalent, although they
differ in both their implementation and execution model. Thus, if needed to reduce
the completion time, developers can always convert the blocking control program to a non-blocking version.

**Is the control plane scalable?** As discussed in section 3.6, **Chi** can scale to large dataflows with a large number of sources (sinks) by using a broadcast (aggregation) tree to exchange control events between the operator and the nodes. To evaluate its effect, in Figure 3.17 we show the completion time of the control program as we increase the number of sources. The results show that the completion time increases logarithmically and it remains well below 100 ms even for very large dataflow graphs (8,192 sources).

### 3.7.3 Adaptivity and fault-tolerance

The previous sections have shown that **Chi** incurs low overhead and completion time, and can scale to large data-flow graphs. Hereafter, we study how **Chi** leverages these
properties to improve the adaptivity to workload changes (dynamic elasticity) and to failures (fault recovery) of the data plane using the YSB workload.

**Dynamic Elasticity.** We set the ingestion rate of our 32-server cluster to 30M tuples/sec. At time $t = 40s$, we double the ingestion rate to emulate a workload spike. At the same time, we start the scale-out process on all three streaming engines, using their respective strategies. For instance, in Apache Flink, when the scale-out process begins, we immediately invoke Save-points [76], which are externally stored self-contained checkpoints useful for stop-and-resume of Flink programs, and restart the dataflow with a larger topology. We plot the results in Figure 3.18:

**Chi.** At the scale-out time ($t = 40s$), there is no noticeable drop in throughput while latency temporarily spikes to 5.8s. However, the system quickly recovers within 6s to a stable state.

**Apache Flink.** At $t = 40s$ throughput drops to zero (due to freeze-the-world initiated by Save-points) for five seconds. Processing latency, instead, spikes up to 35.6s and it takes 31s after restart to return to a steady state.

**Drizzle.** At $t = 40s$, there is no visible drop in throughput because, since Drizzle uses a BSP model, it was not feasible to measure throughput continuously. Similar
Chi: A Novel Control Plane for Distributed Streaming Systems

Fig. 3.20 Impact of batch size on latency and optimization phases

(a) Latency vs. batch size  (b) Batch-size optimization phases

to Flink, the processing latency spikes to 6.1 s and it took an additional 10 s before stabilising. Notably, Drizzle starts reacting to the workload spike after 5 s. This is because the workload spike happened in the middle of a scheduling group [164], where the system cannot adjust the dataflow topology — the larger the scheduling group size is, the slower it reacts to workload changes.

Fault Tolerance: Next, we examine how Chi reacts to failure. As in the previous experiments, we set the ingestion rate to 30 M tuples/sec. We also enable checkpointing for all three streaming systems every 10 s. At time $t = 40$ s, 5 s after the last checkpoint, we kill a virtual machine to simulate a failure in the dataflow and we start the failure recovery process on all three streaming engines (see Fig 3.19):

Chi At failure time ($t = 40$ s), there is no noticeable drop in throughput while latency temporarily spikes to 4.3 s. However, the system quickly recovers to a stable state within 5 s.

Apache Flink At $t = 40$ s, throughput drops to zero for five seconds (system downtime), as Flink’s recovery mechanism redeploy the entire distributed dataflow with the last completed checkpoint [75]. The processing latency spikes up to 17.7 s. It then takes 13 s after restart to return to a steady state.

Drizzle At $t = 40$ s, we observe no drop in throughput due to the reason described in the scale-out experiment. The processing latency spikes to 9.1 s and it takes 11 s to restore the stable state. There is a throughput drop (during recovery) around $t = 50$ s due to Spark’s recovery mechanism as Spark needs to re-run the lineage of the failed batch.
3.7 Evaluation

(a) Latency vs. batch size

(b) Batch-size optimization phases

Fig. 3.21 Work distribution across tasks of the streaming-join operator in IPQ2 before (top) and after workload skew adaptation (bottom)

3.7.4 Chi in action

We conclude our evaluation by showing the performance of Chi using two real-world complex control program, one focusing on parameter auto-tuning to meet SLOs and the other addressing workload skew.

**Auto-Tuning for SLOs:** Streaming systems include a number of parameters that are used to tune the system behaviour. This provides great flexibility to the user but at the same time greatly complicate her task due to the large size of the parameter space. As a representative example, we focus our attention on the batch size. Batching is used by streaming system to amortise the per-tuple processing cost. The batch size has a direct impact on system performance and identifying a priori the optimal value is often a non-trivial task as we show in Figure 3.20a in which we plot the relationship between latency and batch size for the IPQ1 workload.

To show how Chi can help to correctly tune the batch size, we implement a control program consisting of a monitoring task, which collects latency information, coupled with a reconfiguration task, which updates the batch size to meet the desired trade-off between latency. To illustrate how this works, we show an example run in 3.20b in which we set up the control program to optimise the batch size given an ingestion rate of 60 million events/s and an upper bound on latency of 500 ms. Every 30 s the
controller collect latency samples, updates the moving average of the processing latency, and, if needed, executes an optimisation step.

Initially (Phase-I) the controller opts a conservative batch size of 40K events while measuring the latency and throughput. It quickly realises that this batch size is not sufficient for meeting the ingestion rate — it overwhelsms the system causing frequent back-pressure, which is clear from the throughput fluctuations in the figure. Then, starting at the 30-second mark (Phase-II), the controller doubles the batch size to 80K which leads to a more stable throughput reducing the processing latency to $\approx 500 \text{ms}$. At the 60-second mark (Phase-III), the controller attempts a second optimisation step by doubling the batch size to 160K but soon it detects that by doing so it will not be able to meet the latency SLO (500 ms) so finally it reverts the batch size back to 80K (Phase-IV).

Detecting and Adapting to Workload Skew: As shown by the analysis of our production workloads in Figure 3.5 and Figure 3.4, join keys exhibit high temporal and spatial skewness in data distribution. This variance can cause imbalance, lead to stragglers, and ultimately violate SLOs. To show the impact of this, we use another production workload, IPQ2, which exhibits high degree of skewness. This workload is a streaming-join query that does a self-join on an extremely large and high throughput live user-activity log. The presence of a self-join magnifies the effects due to key skewness.

As in the previous example, we implemented a control program consisting of a monitoring task and a reconfiguration one, which behaves as described in Figure 3.11. Figure 3.21a and Figure 3.21b show the distribution of the key groups before and after the reconfiguration respectively. Before the reconfiguration, a few tasks handle most of the key space (as indicated by the peaks in the distribution) while after the reconfiguration, the key space is evenly distributed across task. This has a direct
beneficial consequence on latency and throughput. This can be observed in Figure 3.22 after the reconfiguration the throughput increases by 26% while latency drops by 61%.

3.8 Summary

CHI takes a principled approach to control in data streaming—rather than using separate control plane channels, CHI propagates control messages along with data message on the data plane. This enables supporting important streaming requirements such as zero system downtime and frequent reconfigurations without compromising on the ease of use. CHI’s implementation and verification on production workloads not only provide the insights that validate the design choices, but also offer valuable engineering experiences that are key to the success of such a cloud-scale control plane.
Chapter 4

NetAgg: Application-specific In-network Data Aggregation

Once making the control plane capable of supporting large-scale monitoring and coordination, I then focus on resolving the other frequently cited cause of bottlenecks: data centre networks. In this chapter, I explored a novel idea that enables the networks to take application logic for improving communication.

4.1 Introduction

Many applications in data centres (DCs) achieve horizontal scalability by adopting a partition/aggregation pattern [18]. These include search [92] and query processing [124], dataflow computing [66, 47], graph [119] and stream processing [72], and deep learning frameworks [65]. In the partition step, a job or request is divided into independent subtasks, which are executed in parallel by different work servers (“workers”). Each worker operates on a subset of the data and locally generates partial results. In the aggregation step, partial results are collected and aggregated to obtain a final result.

These applications are challenging from a network perspective because they rely on an aggregation step in which a large number of workers cause many-to-few traffic patterns among servers. For example, traces from Facebook’s DCs show that 46% of the overall network traffic is generated in the aggregation phase [54]. This creates network bottlenecks for the following two reasons: (i) the scarce inbound bandwidth available at the edge servers (e.g. 1 Gbps or 10 Gbps) caps the maximum transfer rate; and (ii) DC networks typically exhibit some degree of bandwidth over-subscription [80, 35], limiting available inter-rack bandwidth.
As a result, the network is often cited as one of the main performance bottlenecks for partition/aggregation applications [55, 18, 92]. Interactive applications such as search are significantly impacted by network activity: the network, on average, contributes 12% of latency in Microsoft’s Bing search engine, accounting for 34% of outliers and 21% of timeouts [92]. In Facebook map/reduce jobs, network transfers on average are responsible for 33% of the execution time of the jobs with a reduce phase, and in 16% of these jobs network transfers account for more than 70% of the execution time [55]; in Microsoft Scope jobs, the network is responsible for a 62% median increase in the reduce and shuffle time [23].

Existing approaches attempt to counter this problem by over-provisioning DC networks using full-bisection bandwidth topologies [80, 82, 151, 14], using specialised network technologies [139, 176, 83], or carefully scheduling data movements to avoid network hotspots [55, 16]. Fundamentally, however, all these approaches do not reduce the network traffic and, hence, ultimately their performance is limited by the scarce bandwidth at the end hosts. While many DCs are in the process of upgrading to 10 Gbps networks, cost-effective 40 Gbps networks without an increased over-subscription ratio are still years away.

In contrast, we propose to reduce network traffic by aggregating data along the network paths in a distributed fashion. We describe NetAgg, a software middlebox platform that provides an on-path aggregation service. Middleboxes have been used extensively in DCs to enhance network functionality [97, 141]. A middlebox is a network appliance attached to a switch that provides services such as firewalls, web proxies, SSL offloading, and load balancing [148]. To maximise performance, middleboxes are often implemented in hardware [6], and adopt a vertically integrated architecture focusing on a narrow function, e.g. processing standard packet headers or performing relatively simple payload inspection.

The need to reduce costs, shorten update cycles, and allow more rapid innovation have instead motivated several proposals for software middlebox platforms [25, 146, 150, 36], typically based on commodity server hardware. Advances in leveraging multi-core parallelism and kernel engineering allow such middlebox platforms to run competitively at or near line rate [120].

We leverage this trend by using middleboxes to execute application-specific aggregation functions at each hop. This exploits the observation that often the aggregation phase exhibits high data reduction [51, 66]. By virtue of this aggregation performed by middleboxes, the amount of network traffic is reduced at each hop, thereby loosening network bottlenecks and, ultimately, increasing application performance. This differs
from traditional approaches for traffic redundancy elimination [21], which operate only at the network level and have only limited visibility (if any) into application semantics.

To process data at the highest possible rate, NetAGG aggregation middleboxes (or *agg boxes*) decompose aggregation computation into cooperatively scheduled aggregation tasks, which are executed in parallel across many CPU cores. Multiple agg boxes can exist in a topology to cooperatively form an *aggregation tree*. NetAGG uses *shim layers* at edge servers to intercept application traffic and redirect it transparently to agg boxes. This minimises the changes required to existing applications for benefitting from on-path aggregation.

Multiple applications with different requirements can share a NetAGG deployment because agg boxes manage the scheduling of aggregation tasks: e.g., they can give priority to aggregation computation on behalf of latency-sensitive online applications over throughput-oriented batch applications.

We evaluate NetAGG both at scale, using simulation, and on a 34-server testbed with two deployed applications: Apache Solr [27], a distributed search engine, and the Apache Hadoop map/reduce framework [1]. In simulation, we show that NetAGG reduces the flow completion time up to 88% compared to existing solutions. We also show that our NetAGG deployment improves the performance of Solr search queries by up to 9.3× and Hadoop jobs by up to 5.2×.

## 4.2 Distributed Data Aggregation

We now describe how application performance can become network-bound due to a partition/aggregation communication pattern in DCs (§4.2.1) and discuss previous attempts to solve this problem (§4.2.2). We then sketch how we make use of middleboxes to perform on-path aggregation (§4.2.3) and present the results of a simulation-based feasibility study that supports our claim of the effectiveness of the approach, both in terms of performance and cost (§4.2.4).

### 4.2.1 Partition/aggregation applications

The partition/aggregation pattern is at the core of many distributed DC applications. A partition/aggregation application has a set of *worker nodes* (“workers”), deployed on edge servers. In the *partition step*, a request or job is divided into independent subtasks, which are executed in parallel by different workers. Each worker operates on a subset of the data and locally generates partial results. In the *aggregation step*,
partial results are collected by a master node, such as a frontend server, and aggregated into the final result.

For example, online search queries are sent to multiple index servers in parallel, each hosting a small portion of the web index. Each index server processes the query locally and returns the top $k$ results best matching the query. These partial results are aggregated to select the final set of results returned to the user.

Similarly, in map/reduce, the input data is partitioned into small chunks (with a typical size of 64–128 MB) and processed by a number of parallel map tasks (the map phase). The intermediate results are then sent to one or many reduce tasks (the shuffle phase), which perform the final step of aggregation (the reduce phase). Other frameworks, such as graph [119] or stream processing systems [72], adopt a similar approach for scaling.

Typically, the aggregation functions used in these applications exhibit two properties. First, they are associative and commutative [172], which implies that the aggregation step can be performed through a sequence of partial aggregations without affecting the correctness of the final result. Examples of aggregation functions exhibiting this property include max, sum and top-k. While there are some functions that do not satisfy this property, e.g. median, it does hold for many aggregations, especially those used for analytic queries [124] and graph processing [172].

Second, in most cases, the size of the aggregated results is a small fraction of the intermediate data generated. For example, the average final output data size in Google jobs is 40% of the intermediate data sizes [66]. In Facebook and Yahoo jobs, the reduction in size is even more pronounced: in 82% of the Facebook jobs with a reduce phase, the final output size is only 5% of the intermediate size, while for Yahoo jobs the number is 8% in 91% of the jobs [51]. Similar trends also hold for traces collected in Microsoft Scope, which show a reduction factor of up to two orders of magnitude between the intermediate and final data [28].

These two properties are important because they show that, by performing partial aggregation on-path, it is possible to reduce the traffic at each hop significantly, thus alleviating network bottlenecks as described below.

### 4.2.2 Edge-based aggregation

We are not the first to notice the benefits of partial aggregation of intermediate data to reduce network traffic. Prior work on interactive services [124] and dataflow frameworks [172, 108] proposes strategies to reduce inter-rack traffic through rack-level aggregation: one server per rack acts as an aggregator and receives all intermediate
4.2 Distributed Data Aggregation

Data from the workers in the same rack. The chosen server aggregates the data and sends it to another server for final aggregation (e.g. a map/reduce reducer).

The main drawback of rack-level aggregation is that its performance is limited by the inbound bandwidth of the aggregator (Fig. 4.1a). For example, assuming 40 servers per rack and 1 Gbps edge network links, the maximum transmission rate per worker is only approximately 25 Mbps.

Rack-level aggregation can be extended to generalised edge-based aggregation by forming a $d$-ary tree of servers that aggregate data within a rack first and then progressively across racks. Intuitively, a lower $d$ (including the degenerate case of $d=1$, when the tree becomes a chain), leads to a higher maximum transmission rate per worker.

While $d$-ary trees eliminate some of the shortcomings of rack-level aggregation, they introduce new challenges: (i) small values of $d$ increase the depth of the tree, affecting the performance of latency-sensitive applications; (ii) small values of $d$ also increase intra-rack bandwidth usage because the incoming links of workers, as opposed to only outgoing links, are used to move data (Fig. 4.1b). As we show in §4.4, this can drastically reduce the performance of other flows in the network that cannot be...
aggregated. For example, in a map/reduce job, only the shuffle flows can be aggregated, while other flows, e.g. used to read from the distributed file system, cannot. By using more links, the bandwidth available to these other flows is reduced.

More generally, a fundamental drawback of any edge-based aggregation approach is that it only applies to *intra-rack* traffic and not traffic in the core of the network. As shown in Fig. 4.1c, if aggregation computation spans multiple racks, the links between aggregation switches can become a bottleneck, especially in the presence of over-subscription.

### 4.2.3 On-path aggregation with middleboxes

In contrast to edge-based aggregation approaches, we propose to use software middleboxes (“agg boxes”) to perform aggregation along the network path. This allows us to minimise edge server bandwidth usage as well as congestion in the core of the network. While aggregation has been shown to increase efficiency in wireless sensor networks [111] and in overlay networks [170, 91, 42], its potential to improve the performance of partition/aggregation applications in DC networks remains unexplored. Existing work on data aggregation has either focused on per-packet aggregation, as in sensor networks, which is not appropriate for application-layer data flows in DCs, or on scalability issues for Internet system monitoring rather than on throughput requirements.

We assume a set-up similar to the one depicted in Fig. 4.1d, in which middleboxes are directly attached to network switches and perform on-path aggregation. We create a spanning tree (hereafter referred to as the *aggregation tree*), in which the root is the final master node, the leaves are the workers and the internal nodes are the agg boxes. Each agg box aggregates the data coming from its children and sends it downstream. To fully exploit the path diversity available in most DC topologies, we use *multiple* aggregation trees that balance the traffic between them. We detail the design of our NetAgg middlebox platform in §4.3.

Due to its compatibility with existing middlebox deployments in DCs, our approach based on application-specific middleboxes can be deployed with low effort. It also allows for an incremental roll-out in which the agg boxes are attached to only a subset of the switches.
4.2 Distributed Data Aggregation

4.2.4 Feasibility study

A potential issue in using middleboxes based on commodity servers for on-path aggregation is the lower processing rate that they can achieve as compared to custom hardware solutions. To understand the feasibility of a software-only approach, we conduct a number of simulation experiments to understand (i) the minimum processing rate $R$ of an agg box required to achieve noticeable benefits and (ii) the performance/cost trade-off involved. We describe the simulation set-up and the full set of results in §4.4.1. Here we highlight the main results, which led to the NetAgg design described in the next section.

Simulation set-up. We consider a three-tier, multi-rooted network topology, modelled after recently proposed scalable DC architectures [80, 14]. Each server is connected through a 1 Gbps link to a top-of-the-rack (ToR) switch. In this experiment, we assume an over-subscription ratio of 1:4, which is consistent with values reported in the literature [80]. We also show the results for the non-oversubscribed case. We consider a mid-sized DC with 1,024 servers and a synthetic workload modelled after Facebook network traces [54], in which 46% of traffic is aggregatable (see §4.4.1 for more details).

Performance requirements. As our baseline, we consider rack-level aggregation, as described above, and we assume ECMP [86] as the routing protocol. We use flow completion time (FCT), i.e. the time elapsed between the start and the end of a flow, as our metric [71] and vary the maximum processing rate that can be sustained by the agg boxes.

Fig. 4.2 shows that relatively modest processing rates are sufficient to achieve significant benefits. Interestingly, even a rate of 1 Gbps per agg box reduces the total
completion time by more than 74% for the 1:4 oversubscribed scenario and 63% for the non-oversubscribed one (88% and 90% for the rate of 8 Gbps, respectively). This shows that, although the bandwidth per agg box is identical to the one used by the rack-level aggregator, performing aggregation at all network tiers (as opposed to just within a rack) reduces the pressure placed on the over-subscribed network core. Note that these results include all flows, not just the aggregation flows, which means that even flows that cannot be aggregated benefit from more efficient bandwidth usage.

In §4.4.2, we show that an agg box in our NetAgg prototype is able to aggregate data at a rate of 9.2 Gbps. Based on the results presented here, this is sufficient to obtain a significant reduction in flow completion time.

Cost analysis. Next we perform a simple cost analysis to understand the trade-off between deploying agg boxes versus increasing the network bandwidth. We consider three alternative options to NetAgg: a full-bisection network topology with 1 Gbps edge links (FullBisec-1G); a 1:4 over-subscribed network with 10 Gbps edge links (Oversub-10G); and a full-bisection network topology with 10 Gbps edge links (FullBisec-10G).

We compare the performance and cost when using rack-level aggregation in the three configurations above to deploying agg boxes using our approach in the base set-up (1:4 over-subscription and 1 Gbps network). We use the same workload as in the previous experiment, and we conservatively assume a configuration with a processing rate \( R = 9.2 \) Gbps because this is similar to the rate achieved by our NetAgg prototype (§4.4.2). For NetAgg, we consider two deployment options: NetAgg, in which we assume that each switch is connected to an agg box, and Incremental-NetAgg, in which only the middle tier of switches (corresponding to 20% of all switches) is connected to agg boxes. We adopt the prices for network equipment from a recently published study [138]. We assume hardware specifications for servers and agg boxes as used in our evaluation testbed (see §4.4.2).

Fig. 4.3 shows the performance improvement and upgrade costs with respect to the base set-up. As expected, upgrading to a 10 Gbps full-bisection network (FullBisec-10G) provides the largest benefit (92% reduction of FCT), but it also incurs the highest upgrade cost. FullBisec-1G has a much lower deployment cost but does not result in the same benefit (only 24% reduction).

In contrast, deploying NetAgg achieves almost the same performance improvement (88%) as FullBisec-10G and outperforms Oversub-10G (87%), with only a fraction of the cost (18% and 22%, respectively). Incremental-NetAgg is also a practical deployment option: it only incurs 4% of the cost of Oversub-10G but reduces FCT by 75%.
Discussion. While we compare the performance of on-path agg boxes against the performance of an upgraded network infrastructure, our solution remains complementary: even with more available network bandwidth, on-path aggregation can reduce bandwidth consumption and provide more bandwidth to other, non-partition/aggregation traffic flows.

In an upgraded network infrastructure (e.g., with 10 Gbps edge links), however, the performance of agg boxes should increase accordingly. As we describe in the next section, the aggregation on agg boxes is embarrassingly parallelisable, which means that agg boxes can exploit increasing numbers of CPU cores for higher performance. In addition, given the nature of aggregation, it is possible to scale out agg boxes by deploying multiple agg boxes connected to one network switch. We evaluate the performance implications of scaling out agg boxes in §4.4.

4.3 Design and Implementation

Next we describe the design of the NetAgg middlebox platform. We explain how it performs aggregation (§4.3.1), and give details on the implementation of aggregation boxes and shim layers (§4.3.2). We also report on two application case studies (§4.3.3).

4.3.1 Overview

As shown in Fig. 4.4, a deployment of NetAgg in a DC consists of two main components, agg boxes and shim layers: (i) agg boxes are connected to a subset of the switches in the DC via high-bandwidth network links. They perform the on-path aggregation of application data according to aggregation functions; (ii) edge servers
The agg boxes cooperate to establish an on-path aggregation tree that aggregates partial results from worker nodes before sending the final result to the master node. As discussed in §4.2, an aggregation tree permits NetAgg to exploit the larger core network bandwidth for data aggregation. Its construction relies on the associativity and commutativity of aggregation functions.

The workflow between agg boxes and shim layers for on-path aggregation follows the steps shown in Figs. 4.4a–4.4c:
1. A client submits a request to a master node, which partitions it into multiple sub-requests (Fig. 4.4a).
2. The master node sends the sub-requests, which pass through the shim layer of the master node without modification, to a set of worker nodes.
3. The partial results generated by the workers are intercepted by the shim layers of the worker nodes. The shim layers redirect the data to the first agg box along the network path from the worker to the master node (Fig. 4.4b). For example, agg box 2, connected to an aggregation switch, aggregates partial results from workers in other parts of the DC.
4. Each agg box aggregates the partial results according to the aggregation function (Fig. 4.4c). It sends its partially aggregated results to the next agg box that is along the network path to the master node.
5. The agg box nearest to the master node (i.e. agg box 1 in the example) sends the fully aggregated results to the master node. The shim layer of the master node passes the results to the application.

Multiple applications. Multiple applications may be executing aggregation trees concurrently on agg boxes. Based on the knowledge of its child and parent nodes, an agg box therefore always forwards partial results to a chosen next-hop towards the master node along a given aggregation tree belonging to that application. The next agg box on-path is determined by hashing an application/request identifier. This ensures that partial data for a given application request traverses the same agg boxes.

Multiple aggregation trees per application. With a single per-application aggregation tree, it is not possible to exploit DC network topologies that support multiple routing paths and thus have higher core network bandwidth (see §4.2.3). NetAgg therefore supports multiple aggregation trees that are used concurrently for a given application, partitioning the aggregation load among the trees. Each aggregation tree
uses a disjoint set of agg boxes (except for the agg box that is in the same rack as the master and the workers), exploiting different routing paths in a multi-path topology.

With multiple aggregation trees per application, the shim layers at the worker nodes partition partial results across the trees. Typically this can be achieved by hashing request identifiers (as in the case of online services) or keys in the data (as in the case of batch processing applications). When an application has multiple aggregation trees, the master node must perform a final aggregation step of the data returned by the roots of the trees.

**Multiple agg boxes per switch.** To increase the throughput of an agg box connected to a switch, it is possible to scale out processing by load-balancing aggregation computation across multiple agg boxes connected to the same switch. In this case, aggregation trees are assigned to agg boxes in a way that balances the load between them.

**Handling failures.** The NetAGG design uses a lightweight failure detection service, running at both the agg boxes and the master shim layer, that monitors the status of downstream agg boxes in the distributed aggregation tree. When a node $N$ (either an agg box or the master node) detects that a downstream agg box $F$ has failed, it contacts the child nodes (either agg boxes or the worker nodes) of $F$ and instructs them to redirect future partial results to $N$. 

![NetAGG middlebox deployment with sample workflow](image)
To avoid duplicate results, when $N$ contacts the downstream nodes of $F$, it also sends the last result that has been correctly processed, e.g. the last partial result received, so that already-processed results are not resent.

**Handling stragglers.** NetAGG is designed to be compatible with existing mechanisms used by applications to handle straggling worker nodes. For example, Hadoop uses speculative execution of backup tasks and reducers can start fetching data from completed mappers while waiting for stragglers or backup tasks to finish [66]. In this case, the agg box just aggregates available results, while the rest is sent directly to the reducer.

Since an aggregation tree involves multiple agg boxes, each agg box itself can potentially become a straggler, delaying the computation of the final result. To handle this scenario, NetAGG uses a similar mechanism to the one for failures: if a node detects that the downstream agg box $S$ is too slow (based on an application-specific threshold), it contacts the downstream nodes of $S$ to redirect future results. The difference with the failure protocol is that the redirection is only applied to results of the same request because the cause of straggling may be specific to it. However, if low performance is observed repeatedly across different requests, the agg box is considered permanently failed, and the failure recovery procedure described above is employed.

### 4.3.2 Implementation

We implement a prototype version of NetAGG in Java, which permits the agg boxes to execute unmodified aggregation functions of partition/aggregation applications written in Java. Agg boxes can host aggregation functions of multiple applications.

An important goal is for agg boxes to use available hardware resources efficiently in order to process data with high throughput. Their implementation is therefore data-parallel: they decompose aggregation functions and parallelise their execution across CPU cores using *cooperative scheduling*. In addition, we minimise the application-specific functions that agg boxes execute: they only receive and process data traffic and do not participate in control interactions of applications.

For multiple applications to share a NetAGG deployment, it is necessary to schedule access to the limited CPU and bandwidth resources of agg boxes. Agg boxes implement prioritised access for certain classes of applications, such as latency-sensitive ones. They schedule the execution of aggregation computation belonging to different applications using *adaptive weighted fair scheduling*, taking different priorities into account.
To make it easy for applications to benefit from on-path aggregation, NetAgg transparently intercepts data flows and redirects them to agg boxes. Traffic is intercepted at the level of Java network sockets due to its well-defined interface across applications.

**Agg boxes**

We show the architecture of an agg box in Fig. 4.5: (i) it executes aggregation tasks that wrap the applications’ aggregation computations; (ii) the tasks are organised into a local aggregation tree that parallelises the aggregation function; (iii) tasks are scheduled cooperatively across CPU cores by a task scheduler; and (iv) a network layer serialises and deserialises data.

**Aggregation tasks.** An agg box represents computation as aggregation tasks, which are fine-grained compute units that can be scheduled in parallel on CPU cores.

Different applications require different interfaces for aggregation functions. An aggregation task therefore wraps aggregation computation using an aggregation wrapper. For example, a Hadoop aggregation wrapper exposes the standard interface of combiner functions, `Combiner.reduce(Key k, List<Value> v)`, which enables agg boxes to run such functions without modification.
Local aggregation trees. Similar to how aggregation computation is decomposed across multiple agg boxes, the computation within a single agg box forms a local aggregation tree of tasks. Local aggregation trees have a large fan-in and are executed in a pipelined fashion by streaming data across the aggregation tasks. This allows for efficient and scalable aggregation, as aggregation executes in parallel and little data is buffered.

As shown in Fig. 4.5, the intermediate tree nodes are aggregation tasks, acting as producers and consumer of data. Leaf nodes receive partial results from worker nodes or downstream agg boxes and send them to their parent aggregation tasks. Tasks are scheduled by the task scheduler (see below) when new input data is available and there is sufficient buffer space at the parent task. They execute the aggregation function on the data from their child nodes. Partially aggregated results propagate up the tree until the root node obtains the final result for that tree. If the computation in the local aggregation tree is not fast enough to sustain the rate at which data arrives from the network, a back-pressure mechanism ensures that the workers reduce the rate at which they produce partial results.

Task scheduler. To reduce the overhead of thread synchronisation, aggregation tasks are executed by a task scheduler using cooperative scheduling: when aggregation is possible, a task is submitted to a task queue and waits to be scheduled. The scheduler assigns tasks to threads from a fixed-sized thread pool and runs the tasks to completion. We assume that aggregation functions are well-behaved and terminate—we leave mechanisms for isolating faulty or malicious aggregation tasks to future work.

If there are multiple applications using an agg box, resources must be shared to achieve acceptable processing times for aggregation computation. For this purpose, an agg box maintains a separate task queue for each application and adopts a weighted fair queuing policy over these queues. This enforces weighted fair shares among applications, similar to other cluster schedulers [173]. When a thread becomes available, the scheduler offers that thread to a task of application $i$ with probability $w_i / \sum_{i=1}^{n} w_i$ that is proportional to its target allocation, where $w_i$ is application $i$’s weight and $n$ is the number of applications on the agg box.

Resource sharing must take the heterogeneity of aggregation tasks into account: the computation and bandwidth requirements of tasks vary depending on the application. To handle this, the scheduler periodically adapts the weights of application $i$ according to the task execution time $\bar{t}_i$ measured at runtime. The intuition is that if an application usually spends twice the time finishing tasks compared to another, the scheduler needs to halve the weight of that application to achieve the targeted proportional share.
More formally, given a target resource share $s_i$ for application $i$, $w_i$ is set to $\frac{s_i}{t_i} / \sum_{i=1}^{n} \frac{s_i}{t_i}$. Our implementation uses a moving average to represent the measured task execution time—the experiments in §4.4 show that this is sufficient in practice.

**Network layer.** Instead of relying on a potentially wasteful application-specific network protocol, such as HTTP or XML, agg boxes transfer data with an efficient binary network protocol using KryoNet [2], an NIO-based network communication and serialisation library for Java. The shim layers also maintain persistent TCP connections to the agg box and parallelise deserialisation using a thread pool.

Since the network data must be deserialised before the aggregation wrapper can call the aggregation function, the network layer of the agg box includes a serialiser/deserialiser taken from the application. For example, to support the aggregation of Hadoop key/value pairs, the agg box uses Hadoop’s `SequenceFile` reader and writer classes for serialisation/deserialisation.

### Shim layers

The shim layers control the redirection of application data and manage the collection of partial results.

**Network interception.** A shim layer intercepts network traffic at the level of network sockets by wrapping the actual Java network socket class in a `NetAgg` socket. By using Java’s ability to change the default socket implementation via the `SocketImplFactory`, applications transparently generate an instance of the custom `NetAgg` socket class when a new socket is created.

**Partial result collection.** A challenge is that agg boxes must know when all partial results were received before executing the aggregation function but, for many applications including Apache Solr, it is not known ahead of time how many partial results will be returned by the worker nodes. We solve this problem by having the shim layer of the master node maintain state about ongoing requests. After intercepting an incoming request, the shim layer records information about the request, typically found in headers, such as the number of partial results, and sends this information to agg boxes.

**Empty partial results.** The master node expects to receive partial results from all worker nodes, but with `NetAgg` this is no longer the case: since the partial results have already been aggregated, only a single fully aggregated result is returned. This means that the shim layer at the master node must emulate empty results from all but one worker, which will include the fully aggregated results. The aggregation logic
Table 4.1 Lines of application-specific code in NetAGG

<table>
<thead>
<tr>
<th>App.-specific NetAGG code</th>
<th>Solr</th>
<th>Hadoop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agg box (serialisation)</td>
<td>8</td>
<td>93</td>
</tr>
<tr>
<td>Agg box (aggregation wrapper)</td>
<td>156</td>
<td>142</td>
</tr>
<tr>
<td>Shim layer</td>
<td>449</td>
<td>528</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>613</td>
<td>763</td>
</tr>
<tr>
<td>Relative to NetAGG code base</td>
<td>13%</td>
<td>16%</td>
</tr>
<tr>
<td>Relative to application code base</td>
<td>0.19%</td>
<td>0.03%</td>
</tr>
</tbody>
</table>

in the master node is not be affected by the empty results because we assume that aggregation functions are commutative and associative.

4.3.3 Application deployments

To demonstrate the generality of NetAGG, we describe its deployment with two partition/aggregation applications: Apache Solr, a distributed full-text search engine [27], and Apache Hadoop [1], a data-parallel map/reduce processing framework. NetAGG can support both applications after providing application-specific serialiser/deserialiser, aggregation wrapper and shim layer code. The required implementation effort is summarised in Table 4.1.

**Apache Solr** performs full-text search across multiple backend server nodes, acting as workers. Clients send small requests to a frontend node, i.e. the master node, which dispatches sub-requests to the backend nodes. The backends return partial search results to the frontend node, which combines them into a ranked result using an aggregation function.

To support Solr, NetAGG requires around 600 lines of code. The aggregation wrapper wraps the custom `QueryComponent` class, which allows user-defined aggregation of the result data. The deserialiser buffers all partial results before invoking the local aggregation tree: this is feasible because results are only of the order of hundreds of kilobytes.

**Apache Hadoop** uses worker nodes to execute mappers, which produce partial results for partitions of the input data. Partial results are aggregated by reducers.

There are fewer than 800 lines of code needed to support Hadoop. The aggregation wrapper implements the Hadoop interface for combiner functions. The deserialiser for Hadoop is slightly more complex than for Solr: as chunks of key/value pairs
are processed by agg boxes in a streaming fashion, the deserialiser must account for incomplete pairs at the end of each received chunk.

4.4 Evaluation

We begin our evaluation by extending the simulation study from §4.2.4. In §4.4.2, we show the experimental results obtained by deploying our NetAgg middlebox prototype with different applications on a 34-server testbed.

4.4.1 Simulation results

We simulate a three-tier, multi-rooted network topology based on recently proposed scalable DC architectures [80, 14]. The network consists of 320 20-port switches that connect 1,024 servers using 1 Gbps links, unless noted otherwise. As a default, we assume a 1:4 over-subscription ratio at the ToR tier, varying it from 1:1 to 1:10 in one
of the experiments. We use a packet-level simulator (OMNeT++), which implements TCP max-min flow fairness and uses standard Equal Cost Multi Path (ECMP) for routing.

Unless noted otherwise, we assume that each agg box is connected to a switch through a 10 Gbps link, and we assume that it can process network flows at 9.2 Gbps. As we show in §4.4.2, this is representative of the performance of our current NetAgg prototype implementation.

We use a synthetic traffic workload, modelled after published network traces from a cluster running large data mining jobs [80]. The sizes of flows follow a Pareto distribution with a mean of 100 KB and a shape parameter of 1.05 in order to simulate a long-tail flow size distribution observed in production [80]. We cap the flow size at 10 GB to exclude chunky network flows as they are often not affected by short-time congestion due to their long lasting periods. The number of flows is chosen so that the load at the edge links is 15% [19]. Except when simulating straggler nodes, all flows start at the same time, which is a worst case for network contention. We also ran experiments using dynamic workloads with various arrival patterns, obtaining comparable results (between 1%–4% of the reported FCT values).

A typical partition/aggregation application generates a mix of flows, of which only a fraction can be aggregated. For example, some of the network flows in Hadoop contain HDFS data, which cannot be aggregated. To account for this, we generate a mixed workload in which only 46% of flows are aggregatable, while the rest constitutes non-aggregatable background traffic. This split is consistent with evidence reported by Facebook [54].
4.4 Evaluation

Fig. 4.12 Flow completion time relative to baseline with different partial NetAgg deployments

The number of workers generating flows follows a power-law distribution where 61% of requests or jobs have fewer than 10 workers, consistent with a recent study on Microsoft and Facebook production clusters [22]. Workers are deployed using a locality-aware allocation algorithm that greedily assigns workers to servers as close to each other as possible. Unless stated otherwise, we use an aggregation output ratio $\alpha$, defined as the ratio between the output and the input data sizes, of 0.1 (in line with the workloads presented in §4.2.1).

**Baselines.** We use the flow completion time (FCT) as our evaluation metric because our goal is to improve network performance. Since our focus is on network bottlenecks that increase FCTs for individual flows, we report the 99th percentile of FCT, unless stated otherwise. We compare the performance of NetAgg against the aggregation strategies described in §4.2.2: rack-level aggregation (rack), a binary aggregation tree ($d=2$; Binary) and chain aggregation ($d=1$; chain). Unless reporting absolute FCTs, we normalise the performance of the other strategies against rack.

**Distribution of flow completion times.** We report the CDF of the absolute FCTs for all flows in Fig. 4.6. As explained in §4.2.2, Binary and chain increase link utilisation, which in turn reduces the bandwidth available for other flows. This explains why they reduce the FCT tail but worsen the median. In contrast, NetAgg improves the performance for all flows by reducing the traffic of the aggregatable flows.

Interestingly, NetAgg also improves the performance of non-aggregatable flows, as shown in Fig. 4.7. The reason is that by reducing the traffic of aggregatable flows, more bandwidth becomes available for other flows.

**Aggregation output ratio.** In Fig. 4.8, we show the impact of the output ratio $\alpha$. We vary $\alpha$ from 0.05 (i.e. high data reduction that emulates $n$-to-1 aggregation
patterns such as top-k, max or count) to 1 (i.e. no aggregation is possible). As expected, the benefits of NetAgg decrease for high values of $\alpha$. When $\alpha$ is small, NetAgg significantly reduces the FCT: for $\alpha=10\%$, NetAgg reduces the 99th FCT compared to rack by 88\% (by 80\% and 52\% for Binary and chain, respectively).

For very high values of $\alpha$, the performance of NetAgg is only marginally better than rack. The reason is that, when there is little or no aggregation, the use of NetAgg penalises throughput because it forces the cross-rack traffic still to be processed by the agg box (at a maximum rate of 9.2 Gbps in our experiments), which constrains the overall throughput, without providing a benefit in terms of data reduction. However, since production traces show that most applications exhibit an output ratio lower than 10\% \cite{51, 28}, a small $\alpha$ is representative of typical workloads, and we adopt $\alpha=10\%$ in the rest of the experiments.

It is also worthwhile to analyse why chain is always outperformed by rack for $\alpha>70\%$. The reason of this counter-intuitive behaviour is that, as explained in §4.2.2, chain utilises more link bandwidth compared to rack. As shown in Fig. 4.9, with $\alpha=10\%$, the median link traffic for chain is 4 times higher than for rack (2.5 times for binary, respectively). When the network is highly loaded and $\alpha$ is large, this aspect dominates, thus lowering throughput.

**Fraction of aggregatable flows.** As described in §4.2.4, a typical DC traffic workload contains a mix of flows, of which only a fraction can be aggregated. We conduct a sensitivity analysis in which we vary the fraction of aggregatable flows from 0.1 to 1 (i.e. all flows can be aggregated).

Fig. 4.10 shows the 99th percentile of FCT relative to rack. With more aggregatable flows, the benefit of all three aggregation strategies increases. With more than 60\%
of aggregatable flows, however, the effectiveness of binary and chain starts to decrease again because their more wasteful usage of network bandwidth introduces network bottlenecks. In contrast, NetAGG maintains the lowest FCTs, all the way to a fully-aggregatable workload.

**Over-subscription.** We quantify the performance impact of the bisection bandwidth on NetAGG and our baselines in Fig. 4.11. We vary over-subscription from 1:1 (i.e. full-bisection bandwidth) to 1:10. As expected, NetAGG performs best when over-subscription is high. By aggregating flows along the network paths, it reduces congestion in the core, alleviating network bottlenecks.

However, the use of NetAGG is beneficial even for networks with full-bisection bandwidth. In a full-bisection network, the inbound bandwidth of the rack and the master becomes the bottleneck. By aggregating the traffic along the network path, NetAGG reduces the congestion at the master, thus decreasing the completion time.

**Partial deployment.** One important question is whether it would be possible to achieve a performance close to NetAGG by simply deploying a 10 Gbps server in each rack and use it for rack-level aggregation. More generally, we want to understand what should be the best deployment configuration if we had only a limited number of agg boxes.

In Fig. 4.12, we compare the performance of a full NetAGG deployment against three configurations: (i) agg boxes only at the ToR switches; (ii) only at the aggregation switches; and (iii) only at the core switches. The results show that the biggest improvement is achieved with agg boxes at the aggregation or core network tiers—deploying them at the rack tier provides limited benefits, improving the FCT by 38% against 83% and 75%, respectively.

This comparison, however, does not take into account that in a DC topology the number of switches at each tier varies significantly. For example, in our simulated topology, we have 128 ToR switches, 128 aggregation and 64 core switches. Therefore, we fix the number of agg boxes to 64, and we measure the performance when deploying them (i) at the core tier only; (ii) uniformly distributed at the aggregation tier; and (iii) uniformly distributed at the two tiers.

Interestingly, the first configuration achieves the largest improvements (75%) while the other two achieve 29% and 43%, respectively. The reason is that the core tier has the best opportunity for aggregating data because it intercepts more flows originating from the workers. This result is important because it shows the benefit of aggregating in the network and not just at the edge. Furthermore it also demonstrates that NetAGG can be deployed incrementally while still yielding most of the benefits.
Upgrading to a 10 Gbps network. In our next simulation experiment, we explore the performance of NetAgg when deployed in a 10 Gbps network. We use the same topology and workload as before but with 10 Gbps links connecting servers to ToR switches.

In Fig. 4.13, we vary the over-subscription from 1:1 to 1:10. For large over-subscription values, NetAgg provides significant benefits compared to rack. Although edge servers have the same bandwidth as agg boxes, the over-subscription creates high contention in the core, which limits the throughput of rack. This shows the benefits of aggregating flows at each network hop as opposed to the edge servers only.

For smaller over-subscription values, the agg box processing rate becomes the bottleneck, reducing benefit. To further increase performance, we explore a scale out configuration in which we attach 2 or 4 agg boxes to each switch. We use a strategy similar to ECMP to assign flows of the same application to the same set of agg boxes. As the results show, two agg boxes are sufficient to reduce the FCTs by up to 85% compared to rack. This demonstrates the value of scaling out, making NetAgg compatible with future network upgrades.

Stragglers. To quantify the impact of straggling workers on our solution, we run an experiment in which we artificially delay the starting time of some of the flows of a given request or job, following the distribution reported in the literature [23]. The results in Fig. 4.14 show that for reasonable ratios of stragglers, NetAgg can still provide significant traffic reduction. As expected, however, with a higher number of stragglers, the benefits of NetAgg decrease as there are fewer opportunities to aggregate data.
4.4 Evaluation

4.4.2 Testbed results

Next we describe the results of our testbed experiments designed to evaluate the effectiveness of our NetAGG middlebox prototype. To demonstrate the flexibility of NetAGG, we experiment with an online application, the Apache Solr distributed text search engine [27], and a batch-processing application, the Apache Hadoop map/reduce framework [1]. The two applications set different objectives for NetAGG, namely increased request throughput (Solr) and reduced job completion time (Hadoop).

We first evaluate the applications in isolation (§4.4.2 and §4.4.2) and then, to demonstrate NetAGG’s ability to host multiple applications, run them simultaneously (§4.4.2).

Testbed set-up. The experiments are performed using a prototype NetAGG implementation deployed on a 34-server testbed across two racks. Each rack contains one 16-core 2.9 Ghz Intel Xeon server with 32 GB of RAM, acting as a master node, and ten 4-core 3.3 Ghz Intel Xeon servers with 8 GB of RAM, acting as the workers. In addition, each rack has five 4-core 1.8 Ghz AMD Opteron servers, generating the client queries in the Solr experiments. All servers have 1 Gbps network links to their ToR switches. Agg boxes have the same hardware as the master nodes and are attached to the ToR switches via 10 Gbps links.

Performance of local aggregation tree. We first conduct a micro-benchmark that evaluates the processing performance of differently-sized in-memory local aggregation trees. The experiments use a 16-core 2.9 Ghz Intel Xeon server with 32 GB of RAM. For an n-input binary tree, n worker threads feed deserialised data to the tree. We use the Hadoop WordCount workload (see below) with an output ratio of $\alpha = 10\%$.

Fig. 4.15 shows the processing throughput for different thread pool sizes when varying the number of leaves in a tree ($L$). We only consider binary trees and, hence, the number of internal nodes, i.e. the aggregation tasks, is equal to $L - 1$. We observe that 8 leaves are sufficient to saturate a 10 Gbps link. When the number of leaves increases, more aggregation tasks can be executed concurrently, and the throughput increases accordingly.

Apache Solr

We begin our testbed evaluation with Apache Solr in a single-rack scenario with 1 frontend, 10 backend and 5 client servers. We load the Wikipedia dataset, a snapshot of all Wikipedia pages in XML format from June 2012 [168], into the backends, and each client continuously submits a query for three random words. To generate different
workloads, we vary the number of concurrent clients generating requests. Unless stated differently, error bars indicate the 5th and 95th percentiles.

We use two aggregation functions with different computational costs, representing extremes for functions commonly used in search engines. The first function, sample, has low computational cost: it returns a randomly chosen subset of the documents to the user according to a specified output ratio $\alpha$, which therefore controls the amount of data reduction: a lower value of $\alpha$ means more aggregation, i.e. less data is returned to the clients.

The second function, categorise, is CPU-intensive: it classifies Wikipedia documents according to their base categories [7] and returns the top-k results per category. Each base category contains several sub-categories, and categorisation is performed by parsing the document content for category strings and determining the base category for the majority of strings.

**Throughput.** Fig. 4.16 shows the median throughput with an increasing number of clients for Solr deployed on NetAGG. To test the capability of NetAGG to process at line rate, we use the sample function for aggregation with a fixed output ratio of $\alpha = 5\%$ to prevent the link to the frontend from becoming a bottleneck. For comparison, we also show the throughput of plain Solr.

For plain Solr, the throughput with 5 clients is limited by the client workload. As the number of clients increases, the throughput also increases until saturation. For 10 clients, packets from the backends start to queue on the 1 Gbps link to the frontend until the system completely saturates for 30 clients (processing at a maximum rate of 987 Mbps). After that, adding clients results in queuing at the frontend without improving the throughput.

With NetAGG, the throughput grows steadily up to 50 clients and then starts to saturate, reaching a median throughput of 9.2 Gbps for 70 clients. This corresponds to a 9.3× increase compared to Solr. After this, the throughput is bottlenecked by the incoming link bandwidth of the agg box.

**Latency.** Besides improving the throughput, NetAGG also reduces the request latency. Fig. 4.17 shows the 99th percentile of the response times when varying the number of clients. For Solr, as the throughput of the frontend is limited to 1 Gbps, response times rise significantly when the workload increases. NetAGG can serve a higher load with low response times: with 70 clients, the response time for Solr is 5.1 s, while it only increases to 0.4 s for NetAGG. This shows that NetAGG achieves low response times because it removes congestion from the network link to the frontend.
4.4 Evaluation

Output ratio. Next we vary the output ratio \( \alpha \) of Solr requests with the sample aggregation for a fixed client population of 70. Fig. 4.18 shows the throughput for Solr and NetAgg. Since the performance of Solr is network-bound, the throughput does not depend on the output ratio. For NetAgg, a higher output ratio also increases the utilisation of the network path between the agg box and the frontend, which remains limited to 1 Gbps. Therefore the effectiveness of NetAgg decreases. As described in §4.2.1, output ratios in production environments typically range from 5% to at most 40%, in which case NetAgg can offer significant benefit.

Two-rack deployment. To investigate the performance of NetAgg with multiple agg boxes, we extend our set-up to two racks, each with one agg box. We also add a second Solr deployment to generate a higher request load. We vary the number of backends per Solr deployment, which are divided equally between the racks to generate cross-rack traffic.
Fig. 4.21 Throughput against number of CPU cores (Solr)

Fig. 4.22 Performance of Hadoop benchmarks

Fig. 4.19 compares the throughput of a single agg box in one rack to the aggregate throughput of two agg boxes in two racks. For both configurations, throughput scales linearly with the number of backends. With two racks, the agg boxes serve twice as many backends compared to one rack, and hence their aggregate throughput doubles. This shows the ability of NetAgg to operate in larger deployments.

**Scale out.** To show the feasibility of increasing the performance of NetAgg by connecting multiple agg boxes to a single switch, we conduct an experiment in which we compare the performance of a single agg box with two agg boxes attached to the same switch. Requests are split equally between the agg boxes by hashing request identifiers. We use the computationally-expensive *categorise* aggregation function to ensure that the agg box is the bottleneck.

The results in Fig. 4.20 confirm our expectation: they show that, by adding a second agg box, the throughput increases until the agg boxes become network-bound.

**Scale up.** To demonstrate the effectiveness of data-parallel processing on agg boxes, we increase the number of active CPU cores on a single agg box from 1 to 16 for both of our aggregation functions. The results in Fig. 4.21 show that, while the performance of the computationally-inexpensive *sample* function is network bound, the performance for the *categorise* function increases linearly with more CPU cores due to increased parallelism.

**Apache Hadoop**

Next we investigate the performance of NetAgg when used with a batch processing application (Apache Hadoop). We deploy Hadoop in one rack with 8 mappers and 1 reducer, with a single aggregation tree. The workload consists of a set of bench-
marks: (i) WordCount (WC), counting unique words in text; (ii) AdPredictor (AP), a machine learning job for generating click-through predictions from search engine web logs [79, 98]; (iii) PageRank (PR), an implementation of the PageRank algorithm [87]; (iv) UserVisits (UV), a job for computing ad revenue per source IP address from web logs [28]; and (v) TeraSort (TS), a sorting benchmark with an identity reduce function [87].

**Job completion time.** We deploy each benchmark job on plain Hadoop and on Hadoop with NetAgg. For each job, we measure (i) the total shuffle and reduce time (SRT) relative to Hadoop, which includes the time spent at the agg box (AGG) and at the reducer; and (ii) the corresponding processing throughput at the agg box. We ignore the map phase because it is not affected by NetAgg.

Figure 4.22 shows the shuffle and reduce times (normalised with respect to plain Hadoop) and the agg box processing rate. NetAgg reduces SRT up to $4.5 \times$ compared
to plain Hadoop. Only for TS, there is no benefit because sorting does not reduce data. AP exhibits a speed-up of only 1.9× because the benchmark is compute-intensive.

Notably, in all cases, the time spent at the agg box (AGG) is a small fraction of the total SRT. This is due to the fact that the reducer is unaware that the results received from the agg box are already final and, regardless, reads them again. This is a conscious design decision because it makes agg boxes transparent to applications—with more invasive modifications of the application, the end-to-end SRT can be reduced to be closer to the AGG time.

The figure also shows that the agg box processes traffic at around 6 Gbps for almost all jobs—the processing rate for TS is lower due to a bottleneck at the reducer.

**Output ratio.** Next we compare the performance of Hadoop and Hadoop deployed with NetAGG for different output ratios α, obtained by varying the repetition of words in the input for the WC job. The size of the input data is 8 GB, and we measure SRT.

Fig. 4.23 shows that relative SRT increases with higher output ratios. This is due to the fact that, with higher ratios, there are more results that are written to disk. A decrease in the output ratio, however, does not alleviate the network bottleneck at the reducer, which has to receive data from all the mappers over a 1 Gbps link.

For all output ratios, NetAGG improves performance over plain Hadoop, up to a factor of 3.7× for α = 10%. The improvement is highest for low output ratios for the same reason as in the case of Solr: performance eventually becomes restricted by the limited edge bandwidth.

**Data size.** We explore how the performance benefit of NetAGG is affected by the amount of data processed by the reducer. For this experiment, we fix the output ratio at 10%. Fig. 4.24 shows the absolute shuffle and reduce times for different intermediate data sizes, ranging from 8 GB to 128 GB.

As we increase the amount of intermediate data, the effect of the shuffle phase on the overall job completion time becomes more pronounced because more data is sent across the network. Hence, the benefit of NetAGG increases with more intermediate data, up to a factor of 5.2× for 128 GB. In addition, the reducer receives only a small fraction of the already reduced intermediate data. This reduces processing time as well as CPU cycles and disk I/O.

**Multiple applications**

In our final experiment, we evaluate the behaviour of NetAGG in the presence of multiple deployed applications, focusing on the fairness that the adaptive task scheduler of an agg box can achieve (§4.3.2). For this, we execute a Hadoop job on NetAGG
while running a Solr deployment at the same time. We route all aggregation traffic through a single agg box and measure the CPU share of each application. As the resource consumption of a single Solr task is significantly higher than that of a Hadoop one, the task scheduler must be able to account for this heterogeneity.

Fig. 4.25 shows the CPU usage of the two applications when the task scheduler uses regular weighted fair queuing with fixed weights, which are set according to application priorities (see §4.3.2). Although each application is assigned a desired utilisation of 50%, this is not reflected in the achieved CPU usage: a Solr task takes, on average, 34 ms to run on the CPU, while a Hadoop task runs only for 2 ms. Fixed weights therefore lead to a starvation of the Hadoop tasks, which exhibit low performance.

In contrast, Fig. 4.26 shows the same set-up using our adaptive task scheduler, which adjusts weights based on the actual CPU consumption by applications. As a result, CPU usage is split fairly between the processing of aggregation tasks for Solr and Hadoop.

### 4.5 Summary

Many applications in today’s DCs operate in a partition/aggregation fashion. We observe that in typical workloads, the aggregation functions are associative and commutative, and exhibit high data reduction. This motivates us to explore a novel point in the network design space in which data are aggregated on the network path before reaching the end hosts.

We describe the design and implementation of NetAgg, an on-path aggregation service that transparently intercepts aggregation flows at edge servers using shim layers and redirects them to aggregation nodes (agg boxes). NetAgg is designed to aggregate data of multiple applications at high rate by decomposing aggregation computation within and across agg boxes to exploit parallelism. We evaluate the effectiveness of NetAgg using a combined approach of simulations and a prototype implementation deployed with real-world applications. Our results show that NetAgg outperforms typical DC set-ups while incurring little extra deployment cost.
Chapter 5

MLNet: A Novel Application-aware Communication Layer

In data centres where the access to the network is restricted, the uses of NetAgg are constrained on the network edges. In this chapter, I removed the dependency towards programmable in-network elements, and explored for an edge-based solution: developing a communication layer that can completely run in the server network stack. This layer aims to leverage offloaded application functions to improve communication. In particular, I demonstrated the usage of this layer in deep learning systems. These systems are widely used today, but by far suffer from communication bottlenecks given the large amount of synchronisation traffic.

5.1 Introduction

Typically, deep learning systems adopt an approach referred to as data parallelism [65]. Rather than training a single model with all the available input data, they replicate the model across many servers and feed each replica with a subset of the input data. Since the model replicas are trained using different input data, their model parameters will typically diverge. To reconcile these parameters and ensure that all model replicas eventually converge, each replica periodically pushes its set of parameter values to a centralised server, called the parameter server [105]. The latter aggregates all the received updates for each parameter (e.g., by averaging them) and then sends back to all replicas the newly computed set of values, which will be used at the beginning of the next iteration. As the total numbers of parameters can be very high (up to $10^{12}$ [41]), multiple parameter servers are used, with each one being responsible for a subset of the parameters.
A major challenge of this approach is the high communication cost. Model replicas must frequently read and write global shared parameters. This generates a large amount of network traffic and, due to the sequential nature of many of the deep learning algorithms used, it may also stall the computation if the synchronisation latency is high. Therefore, the network is often regarded as one of the main bottlenecks for distributed deep learning systems \cite{106, 65, 58}. To alleviate this issue, these systems are often deployed on high-performance network fabrics such as Infiniband or RoCE \cite{58, 104}, while others have proposed to trade-off algorithm training efficiency for system performance by introducing asynchronous communication \cite{105, 52}, thus removing some of the barriers. Unfortunately, neither of these approaches is completely satisfactory as the former significantly increases infrastructure costs while the latter reduces overall training efficiency.

In this chapter, we explore a different yet complementary point of the design space. We argue that network bottlenecks can be greatly reduced through a customised communication layer. To demonstrate this, we designed MLNet, a novel communication layer for distributed deep learning. MLNet uses tree-based overlays to implement distributed aggregation and multicast and reduce network traffic, and relies on traffic control and prioritisation to improve average training time.

A key constraint underlying our design is that we wanted MLNet to be a drop-in solution for existing deep learning deployments. Therefore, we implemented MLNet as a user-space process running on hosts without requiring any changes in the networking hardware, in the OS stack, or in the training algorithm code. Further, by sitting in between workers and parameter servers, it decouples these two classes of servers, enabling scaling each one independently and efficiently masking network and server failures. We evaluate its effectiveness in Section 5.4 by means of large-scale simulations with 800 servers and 50 to 400 parameter servers. The results show that MLNet reduces the training time by a factor of up to 5x.

5.2 Background

In this section, we provide a brief introduction to deep learning, motivate the need for its distributed execution, and discuss the use of parameter servers for scaling distributed deep learning.
5.2 Background

5.2.1 Deep learning

The goal of a deep learning algorithm is to construct a prediction model that extracts useful knowledge from training data, and uses it to make inferences about future arrival data. This can be formalised as an optimisation problem: Given a set of training data $X$, it tries to find a model $W$ that minimises the error of a prediction function $F(X,W)$. Typically, a deep learning algorithm approaches this problem iteratively, starting from a randomly generated $W$ and then refining its solution gradually as more and more data are processed.

Complex models are usually able to capture the knowledge hidden in training data. To an extreme, a sufficiently complex model can “memorise” all the information contained in the data. In this case, it can give the correct prediction for any sample it has seen before, but may perform poorly for unseen samples. This is called over-fitting: a model fits its training data well, but does not generalise to others. This is why a large amount of training data is necessary for deep learning. By using more data, a model can generalise sufficiently, reducing the risk of over-fitting.

5.2.2 Distributed deep learning

As the size of training data can significantly affect prediction accuracy, it has become common practice to train models with large datasets. To speedup these training tasks, they are often distributed across many servers.

In a distributed setting, a server iteratively refines a shared model by learning from a local data partition, and periodically synchronises this model with the other servers. More specifically, after each iteration, it calculates a refinement $\Delta W_i$ to the model $W$. To make sure that all servers eventually converge to the same model, they can synchronise every iteration, every $n$ iterations, or completely asynchronously. When deep learning algorithms are implemented on traditional distributed platforms such as Hadoop or Spark, servers have to synchronise every iteration. This requires placing a barrier at the end of a iteration, incurring increasing overhead as the system scales.

Parameter Server [105] is another approach to implement synchronisation in distributed deep learning. It outperforms the aforementioned platforms thanks to domain-specific engineering and algorithmic optimisations. In this approach, a set of servers act as parameter servers that store the model $W$. The other servers process the training data and act as workers. After $\Delta W_i$ are calculated, workers do not communicate with each other directly, but push $\Delta W_i$ to the parameter servers and then pull a new $W$ to be used in the next iteration. By tuning push/pull frequencies, programmers can
balance the training efficiency and system performance. For example, in the *Stale Synchronous Parallel (SSP)* model [63, 57], workers are allowed to cache \( W \) and use it in the next iteration while sending the \( \Delta W_i \) of the previous iteration to servers, as long as the cached version is within a staleness threshold \( s \). In this way, communication can be overlapped with computation. Nevertheless, as workers still have to periodically synchronise the model, the network can quickly become a bottleneck. In the next section we show how MLNet can alleviate this problem using a customised communication layer.

5.3 MLNet Design

We begin the section by describing the MLNet architecture and then we show how this makes it easy to implement our two techniques to optimise network performance, namely *traffic reduction* and *traffic prioritisation*. While we believe that this is a contribution per se, we also see this a first step towards a deeper rethinking of the network design for distributed deep learning.

5.3.1 Architecture

MLNet is a communication layer, running as a local process on workers and parameter servers, similar to Facebook mcrouter setup [132]. These local processes behave like proxy, intercepting all exchanges between workers and parameter servers. To push training results, a worker initiates a normal TCP connection to a parameter server that is actually emulated by a local MLNet process. Regardless of the actual numbers of parameter servers and workers, MLNet maintains a single parameter server abstraction to workers, and symmetrically a single worker abstraction to parameter servers. This design decouples workers and parameter servers and allows them to scale in and out independently. Further, it makes it easy to change the communication logic, e.g., to which server and when send the traffic to, without requiring modifications in the worker's or parameter server's code.

To achieve transparency w.r.t. both workers and parameter servers, MLNet inherits the standard communication APIs from the Parameter Server [105] where data is sent between nodes using *push* and *pull* operations:

- \( \text{weights} = \text{pull}(\text{modelId}, \text{staleness}) \): Pull the *weights* of a model within a *staleness* gap.
- \( \text{push}(\text{modelId}, \text{gradients}) \): Push the *gradients* of the weights of a model.
• *clock()*: Increment the clock of a worker process.

We give an example of using this interface. The training of a shared model consists of multiple iterations. At the end of an iteration, a worker *pushes* the newly calculated gradients of model weights to a parameter server and increases its local clock $c_{\text{worker}}$ by calling *clock()* . The parameter server aggregates all pushes to update model weights and maintains a vector of the clocks of all workers. A model clock $c_{\text{model}}$ is defined as the minimum worker clock in the vector. To start the next iteration, the worker then *pulls* the model weights that have to be within a staleness threshold $s$ (see Section 5.2.2) by checking if $c_{\text{model}} \geq c_{\text{worker}} - s$.

### 5.3.2 Distributed aggregation and multicast

The push and pull phases are the primary sources of network traffic in distributed deep learning. In the push phase, workers send the gradients of model weights to parameter servers, while in the pull phase they receive the new weights generated after aggregating the gradients from all model replicas. These two operations can generate high congestion at servers and, if the network fabric is over-subscribed, in the core of the network too.

Adding more parameter servers would reduce the edge congestion by spreading the traffic (albeit at the expenses of increasing the overall server count) but it is of little help to reduce congestion in the core. We propose, instead, a different yet complementary approach that aims at reducing, rather than simply re-routing, network traffic by exploiting its domain-specific characteristics.

**Aggregation and multicast tree.** The aggregation functions used by parameter servers are typically *associative* and *commutative*, e.g., an *average* function is often used. This means that gradients can be aggregated incrementally and the final result is still correct. We leverage this property to reduce traffic during the push phase. For each parameter server, MLNet uses it as the root and builds a spanning tree connecting all workers. The workers in the leaves send gradients to their parent nodes. The latter aggregate all received gradients and push the results upstream towards the root where the last step of aggregation is performed.

Assuming that a node has an in-degree $d$. For each parameter, this node receives $d$ values and transmits only one value (i.e., the aggregated result), thus drastically reducing the network traffic at each hop. This not only helps alleviating the load on parameter servers (avoiding the so-called *in-cast* effect), but also reduces the load in the network core too, which is beneficial in case of over-subscribed networks. As shown
in Section 5.4, depending on the network fabric characteristics and the system load, different values of \(d\) may be preferable.

A dual technique is also used during the pull phase. In this case, rather than aggregating values on path, we use the same tree to multicast the weights to all replicas. By using a multicast tree, only \(d\) values per server are transmitted, which reduces the load on the outbound link of a parameter server (we expect \(d \ll W\), where \(W\) is the number of workers in the system).

**Synchronous vs. asynchronous operations.** MLNet use the staleness threshold \(s\) to determine how push/pull operations are performed. In a synchronous setting, i.e., \(s = 0\), a worker needs to wait for all its child nodes on the spanning tree in each iteration. In this case, a MLNet process uses its position on the tree to figure out the number of pushes in an aggregation window as well as the pull responses to multicast.

When \(s > 0\), a MLNet process needs to decide when to perform push/pull operations. If a pull request can be satisfied with the cached weights \((c_{\text{model}} \geq c_{\text{worker}} - s)\), the process responds it immediately, without incurring extra upstream network traffic. If not, the request is forwarded upstream, until it is satisfied with a cached weight, or it waits on the parameter server. When receiving a push message, the process first updates its own model with this message. It then pushes this message upstream if and only if the gradients carried by this message are not within the staleness thresholds compared to its parent node. This means that the degree of asynchrony is controlled by MLNet and it does not have to be hard-coded in a training algorithm.

**Fault tolerance.** To detect failures, a MLNet process uses heartbeats to check the liveness of its parent node on the spanning tree. If a node fails, the downstream nodes are connected to the parent of this failed node. They then exchange model information to ensure that staleness bounds are not broken. If the root of the tree, i.e., a parameter server, fails, its children wait for it to be replaced and then re-initiates a connection. If a failed node is back, it asks its neighbors to restore its local model and re-enters the tree. In case of multiple concurrent failures, the above mechanism may incur high overhead. MLNet then tears down the tree entirely and reverts to the traditional setup where workers directly communicate with parameter servers.

### 5.3.3 Network prioritisation

The second technique used by MLNet is network prioritisation. Mainstream congestion control protocols such as TCP strive to provide *per-flow fairness*. This can be particularly detrimental in a multi-tenant environment where different models are
trained in parallel because contention on network resources will delay the training time of all models sharing that bottleneck.

To address this issue, we implemented a network prioritisation mechanism in MLNet to arbitrate access to network resources. We do not constrain how priorities are defined. For example, a model with relatively smaller communication cost could be given a high priority in order to complete earlier, leading to a shorter average model training time. Hereafter, we only assume that the MLNet process has a way to extract the priority from a flow (e.g., this can be encoded in the first transmitted byte).

A key challenge of implementing this feature is how to achieve this functionality without requiring any change in existing network infrastructure. Recent proposals, e.g., [20, 68, 169], require custom switch hardware and, hence, do not fulfil our requirements.

In contrast, we opted for a software only solution. If the network fabric provides full bisection bandwidth, e.g., through a fat-tree design [80, 14], contention only occurs at the edge [94], i.e., at either the worker’s or parameter server’s uplink. When the destination machine, either a worker or a parameter server, receives a new TCP connection, it can inspect the relative priority (e.g., by looking at the first byte) in order to decide whether to reject the connection or to accept it by possibly dropping some of the existing ones if they have lower priority than the new one. As we show in the Section 5.4, this simple mechanism is effective in reducing the median training time without hurting the tail performance.

If the network is over-subscribed, the above mechanism is not sufficient as congestion can occur in the network core. If switches support enough priority queues, we can extend the above mechanism to take advantage of them, similar to recently proposed solutions [30, 128]. For the cases in which switch queues are not available, an alternative approach is to extend our previous work on bandwidth guarantees in a multi-tenant data center [33] to allocate bandwidth to flows according to their priority. This, however, would require knowledge of worker and parameter server locations and, hence, it might not be suitable for public cloud deployments. We are currently working on a decentralised solution that does not suffer from this limitation, possibly reusing some ideas from recent work on co-flow scheduling [68, 56].

5.4 Simulation Results

We use simulations to evaluate the performance of MLNet. While preliminary, the results indicate that our design can i) reduce the end-to-end training time (makespan),
by performing distributed aggregation and multicast, and \textit{ii)} shorten the median make-spans of concurrent deep learning tasks by prioritising flows.

We use OmNET++ \cite{4}, a discrete event simulator, to model a mid-size cluster with a three-tier, multi-rooted network topology based on popular scalable data centre architectures \cite{80, 14}. The cluster comprises 1,024 servers connected by 320 16-port switches via 10 Gbps links. We model the TCP max/min fairness model and we use the standard Equal Cost Multi Path (ECMP) routing protocol.

We adopt a synthetic workload modelled after a recently published deep learning algorithm for sparse logic regression \cite{105}. The model has 65 billion parameters (i.e., AlexNet) and is trained with a 141 TB dataset that has 1000 epochs of the ImageNet image classification dataset. The dataset is evenly partitioned and consumed by 800 workers. We vary the number of parameter servers between 50 and 400. The parameter space is equally divided across the parameter servers. Both workers and parameter servers are randomly deployed on the 1,024 servers. Based on our experience in training models with Minerva \cite{167}, we assume that workers process training data at a rate which follows a uniform distribution parameterised with $a$ as 100 MB and $b$ as 200 MB. They use synchronised push and pull, and communicate with the parameter servers every 30 s.

**Distributed Aggregation and Multicast.** We begin our analysis by focusing on the benefits of using our aggregation and multicast mechanism. To show the impact of the node in-degree $d$, we compare the performance of today’s approach (\textsc{Baseline}) against two strategies that use a different value of $d$. The first one, \textsc{Rack}, uses a single aggregator per rack, i.e., $d = 7$, as in our experiments we assumed eight servers per rack. The other one, \textsc{Binary}, instead, uses binary trees, i.e., $d = 2$.

Figure 5.1a shows the makespan of these configurations against the number of parameter servers together with 800 workers in a \textit{non-oversubscribed} network. When there are 50 parameter servers, \textsc{Binary} and \textsc{Rack} outperform \textsc{Baseline}, taking only 22\% and 42\% of the time used by \textsc{Baseline}. This shows the benefits of reducing the inbound and outbound load at the parameter servers. However, with a large number of parameter servers, the load is already sufficiently spread and our solutions become less effective compared to \textsc{Baseline}. Finally, when the number of parameter servers is very high ($\geq 300$), the cost of re-directing traffic through the tree dominates and the performance of both \textsc{Binary} and \textsc{Rack} gets worse than \textsc{Baseline}. This shows the trade-off between the number of parameter servers and traffic reduction. Having more parameter servers helps to spread the load across the fabric and alleviate network
5.4 Simulation Results

Fig. 5.1 Training makespan against the number of parameter servers.

(a) No over-subscription.

(b) 1:4 over-subscription.

bottlenecks. However, this comes at the cost of using more machines and is only effective when the network is not oversubscribed.

We then repeated the same experiment assuming a more realistic network setup with an over-subscription ratio of 1:4. In this case, congestion occurs in the network core too and, hence, unlike in the non-oversubscribed case, just increasing the number of parameter servers does not help in removing the main bottlenecks. In contrast, by reducing the overall network traffic, MLNet is also able to reduce the congestion in the core. This explains why, as shown in Figure 5.1b, RACK and BINARY outperform BASELINE in all configurations. For example, with 50 parameter servers, RACK reduces the makespan compared to baseline by 73% (resp. 66% for BINARY) and with 400 parameter servers, the makespan is reduced by 71% (resp. 40% for BINARY). Interestingly, in this configuration, BINARY achieves a worse performance than RACK. The reason is that since the height of BINARY’s trees is higher, the path length increases accordingly and, hence, BINARY consumes more bandwidth than RACK. While this is
not an issue in a non-oversubscribed network in which the core bandwidth is abundant, this becomes problematic in this setup where the core bandwidth is scarce.

In Figure 5.2, we show the CDFs of traffic across network links. In this experiment, we consider a non-oversubscribed network with 800 workers and 200 parameter servers. As expected, both Rack and Binary exhibit a lower traffic per link, which explains the overall improvement in performance. We observed a qualitatively similar result in the over-subscribed case too (omitted for space reasons).

Flow Prioritisation. Next, we explore the impact of network prioritization. We consider a workload in which 20 different models are trained concurrently. To reduce the computation load of our simulations, we scale down each mode to use only 200 workers and 50 parameter servers, and reduce the input data size proportionally. We randomly assign priority to each model and we start all the models at the same time.

In Figure 5.3, to understand the impact of using prioritisation alone and combined with traffic reduction, we show the CDF of the makespan for these configurations.
a comparison, we also report the CDF achieved when using traffic reduction alone and when using our TCP-based baseline.

Just using network prioritisation reduces the median make span by 25% while only increasing the tail by 2%. Combining traffic reduction and prioritisation together further improves the performance by reducing the median by 60% (resp. 54% for the 95\textsuperscript{th}-percentile) compared to baseline, and reducing the median by 13.9\% compared to using traffic reduction only.

5.5 Summary

In this chapter, we show the effectiveness of using traffic aggregation and network prioritisation for resolving the communication bottlenecks in distributed machine learning clusters. Though preliminary, the experiment results point out the need to rethink the existing network stack design. In addition to providing conventional reliable data delivery, a more advanced network stack can further provide abstracted optimisation techniques for accelerating expensive communication operations that are common within Big data applications.
Chapter 6

Flick: Developing and Running Application-specific Network Services

In the previous two chapters, I have shown the large promises of integrating applications and networks. The tight integration of applications and networks, however, exposed the complexity of programmable network elements or communication stack directly to application developers. In this chapter, I thus explored for developing a new network programming abstraction with the aim of supporting application developers to efficiently develop and deploy network services trailed for different communication requirements.

6.1 Introduction

Distributed applications in data centres increasingly want to adapt networks to their requirements. Application-specific network services, such as application load-balancers [99, 137], request data caches [133], therefore blur the boundary between the network fabric at the core and applications at the edge. For example, a Memcached request router can transparently scale deployments by routing requests using knowledge of the Memcached protocol [133]. In this chapter, we explore how application developers, not network engineers, can be supported when implementing new application-specific network services.

Existing software middlebox platforms, such as ClickOS [121], xOMB [25] and SmartSwitch [175], support only application-independent network services, i.e. IP
routers, firewalls or transport-layer gateways. Using them to interact with payload data in network flows leads to an impedance mismatch due to their byte-oriented, per-packet APIs. Instead, application developers would prefer high-level constructs and data types when expressing processing logic. For example, when defining the dispatching logic of a Memcached request router, a developer would ideally treat key/value pairs as a first-class data type in their program.

Today’s middlebox platforms also force developers to optimise their code carefully to achieve high throughput—implementing a new Click module [101, 121] in C++ that can process data at 10 Gbps line rate is challenging. As a result, many new application-specific network services [137, 116] are built from scratch rather than leveraging the above platforms.

Considerable work went into developing new high-level languages for network control within software-defined networking (SDN) [78, 126, 38, 155]. While these simplify the specification of network management policies, they typically operate on a per-packet basis and support a limited set of per-packet actions once matched, e.g. forwarding, cloning or dropping. In contrast, application-specific network services must refer to payload data, e.g. messages, key/value pairs or deserialised objects, and carry out richer computations, e.g. arbitrary payload transformations, caching or data aggregation.

Our goal is to enable developers to express application-specific network services in a natural high-level programming model, while executing such programs in an efficient and scalable manner. This is challenging for several reasons: (i) in many cases, the cost of data deserialisation and dynamic memory management reduces achievable processing throughput. While high-level programming languages such as Java or Python can manipulate complex application objects, they struggle to provide predictable processing throughput for line-rate processing of network data; (ii) a typical data centre may host hundreds of applications, with each potentially requiring its own network service. Services must thus share resources, e.g. CPU and memory, without interference. Existing middlebox platforms use coarse-grained virtualisation [121], which carries a context-switching overhead of hundreds of microseconds. This is too high for fine-grained resource sharing between many application-specific network services; and (iii) most of the applications use TCP for transport, and an application-specific middlebox needs to terminate TCP connections to access data. Performance and scalability of such middleboxes are often bounded by the high cost of connection termination and frequent socket reads/writes.

We describe Flick, a framework for developers to program and execute application-specific network services. It consists of the Flick language for defining network services,
and the Flick platform for executing compiled programs efficiently on multi-core CPUs.

Programs in the Flick language have bounded resource usage and are guaranteed to terminate. This is possible because most application-specific network services follow a similar pattern: they deserialise and access application data types, iterate over these data types to perform computation, and output the results as network flows. The language is therefore statically typed, and all built-in types (e.g. integer, string, and array) must have a maximum size to avoid dynamic memory allocation. Programs can refer to complex application-defined data types, such as messages or key/value pairs, for which efficient parsers are synthesised from the type definitions in the program. Since functions can only perform finite iteration over fixed-length data types, Flick programs with finite input must terminate.

A compiler translates Flick programs into task graphs implemented in C++. Task graphs are designed to permit the efficient and safe execution of many concurrent network services on a shared platform. A task graph consists of parallel tasks that define the computation of the Flick program, and channels that propagate data between concurrently executing tasks. Input/output tasks perform the serialisation/deserialisation of data to and from application objects. Since Flick programs explicitly specify accesses to application data fields, the compiler can generate custom parsing code, eliminating the overheads of general-purpose parsers.

The Flick platform executes multiple task graphs belonging to different services. To reduce the overhead of frequent connection termination and socket operation, task graphs use a modified version of a highly-scalable user-space TCP stack (mTCP [93]) with Intel’s Data Plane Development Kit (DPDK) [90]. Task graphs are also scheduled cooperatively, avoiding context-switching overhead. They cannot interfere with each other, both in terms of performance and resources, due to their safe construction from Flick programs.

We evaluate a prototype implementation of Flick using both micro-benchmarks and three application-specific network services: an HTTP load balancer, a Memcached proxy and a Hadoop data aggregator. Our results show that Flick can execute these services with throughput and latency that matches that of specialised middlebox implementations. In addition, it scales with a larger number of compute tasks. This chapter focuses on the design, implementation and performance of a single Flick middlebox. However, the wider vision is of a number of such boxes within a data centre [61].
6.2 Application-specific Network Services

Flick focuses on a specific context: data centres in which multiple, complex, distributed applications run concurrently. In this case, to achieve higher performance, flexibility or efficiency, it is advantageous to execute portions of these applications, e.g., related to load-balancing, caching or aggregation, as application-specific network services directly on network elements.

To do this, application developers must add code to network elements such as software middleboxes. Today this typically means that they must implement complicated features of the underlying network protocols (e.g., TCP flow construction, HTTP parsing and application data deserialization). For performance reasons, network services must be highly parallel, which requires considerable developer expertise to achieve. Network resources are also inherently shared: even if hosts can be assigned to single applications, network elements must host many services for different applications.

The goal of Flick is to allow developers to easily and efficiently introduce application-specific processing into network elements. Present approaches are unsatisfactory for three key reasons: (i) they provide only low-level APIs that focus on the manipulation of individual packets, or at best, individual flows; (ii) they do not permit developers to implement services in high-level languages, but typically rely on the use of low-level languages such as C; and (iii) they provide little support for the high degrees of concurrency that are required to make network service implementations perform well.

Next we elaborate on some of these challenges as encountered in our example applications (§6.2.1), and then contrast our approach with existing solutions (§6.2.2).

6.2.1 Use cases

We consider three sample uses for application-specific services: HTTP load balancing, Memcached request routing, and Hadoop data aggregation.

HTTP load balancer. To cope with a large number of concurrent requests, server farms employ load balancers as front ends. These are implemented by special-purpose hardware or highly-optimized software stacks and both sacrifice flexibility for performance. As a result, load balancers must often be reimplemented for each application to tailor them to specific needs. For example, this may be necessary to ensure consistency when multiple TCP connections are served by the same server; to improve the efficiency of clusters running Java code, a load balancer may avoid dispatching requests to servers that are currently performing garbage collection [110]; finally, there is increasing
interest from Internet companies to monitor application-specific request statistics—a task that load balancers are ideally placed to carry out [70].

**Memcached proxy.** Memcached [74] is a popular distributed in-memory key/value store for reducing the number of client reads from external data sources by caching read results in memory. In production environments, a proxy such as twemproxy [162] or mcrouter [133] is situated usually between clients and servers to handle key/value mappings and instance configurations. This decouples clients and servers and allows the servers to scale out or in horizontally.

Past attempts to implement Memcached routers have involved user-space solutions [133], incurring high overheads due to expensive memory copies between kernel- and user-space. More recent proposals, such as MemSwitch [175], have shown that a dedicated single-purpose software switch that intercepts and processes Memcached traffic can be more efficient. To customise MemSwitch, developers, however, must write complex in-network programs that process raw packet payloads. This not only compromises the safety and performance of the network stack, but also complicates development—it requires knowledge about low-level details of networking as well as skills for writing high-performance, parallelisable packet-processing code.

**Hadoop data aggregator.** Hadoop is a popular map/reduce framework for data analysis. In many deployments, job completion times are network-bound due to the shuffle phase [54]. This means that performance can be improved through an in-network data aggregation service, which executes an intermediate in-network reduction within the network topology before data reaches the reducers, thus reducing traffic crossing the network.

Providing an in-network data aggregation for Hadoop serves as a good example of an application-specific service that must carry out complex data serialisation and deserialisation. A developer wishing to implement in-network reduce logic must therefore re-implement the logic necessary to reconstruct Hadoop key/value pairs from TCP flows—a difficult and error-prone task.

### 6.2.2 Existing solution space

There are several proposals for addressing the challenges identified in the use cases above. We observe that existing solutions typically fit into one of four classes:

(i) **Specialised, hand-crafted implementations.** Systems such as netmap [142, 143] provide for efficient user-space implementations of packet-processing applications.
Unfortunately, they offer only low-level abstractions, forcing developers to process individual packets rather than high-level business logic.

(ii) Packet-oriented middleboxes. Frameworks for implementing software middleboxes, such as ClickOS [121] and SmartSwitch [175], enable high-performance processing of network data and can be used to build higher-level abstractions. However, they fail to support useful high-level language features such as strong and static typing, or simple support for data-parallel processing.

(iii) Network programmability. More recently, we see increasing deployment of software-defined networking techniques, usually OpenFlow [122]. More advanced technologies have been proposed such as P4 [38] and Protocol Oblivious Forwarding [154]. These enable efficient in-network processing of traffic, selectively forwarding, rewriting and processing packets. However, they suffer from many of the same issues as (ii) due to their narrow focus on packet-level abstractions.

(iv) Flow-oriented servers. For in-network processing concerned with higher-level flow abstractions, it is common to leverage existing server implementations, such as Nginx [3] or Apache [159], and customise them either at the source level or through extensibility mechanisms such as modules. Another example is Netflix ribbon [131], which provides a number of highly configurable middlebox services along with a Java library to build custom services. While this raises the level of abstraction somewhat, the overheads of using such large, complex pieces of software to perform application-specific network services are substantial.

6.3 Flick Framework

We motivate our design by outlining requirements (§6.3.1), and providing a high-level overview (§6.3.2).

6.3.1 Requirements

Based on the shortcomings of the approaches highlighted in §6.2.2, we identify the following three design requirements for our framework:

R1: Application-level abstractions: developers should be able to express their network services using familiar constructs and abstractions without worrying about the low-level details of per-packet (or per-flow) processing;
**R2: High parallelism:** to achieve line-rate performance, programs for application-specific network services must exploit both data and task parallelism without requiring significant effort from the developers;

**R3: Safe and efficient resource sharing:** middleboxes are shared by multiple applications/users, therefore, we need to ensure that programs do not interfere with one another, both in terms of CPU and memory resources.

To meet these requirements, FLICK follows the scheme shown in Figure 6.1. For the desired level of abstraction (**R1**), it provides a novel high-level language (**1;** §6.4). The language allows developers to focus on the business logic of their network services ignoring low-level details (e.g. serialisation or TCP reassembly).

Compared to general-purpose languages such as C or Java, the FLICK language offers a constrained programming environment. This makes it easier to compile FLICK programs to parallel FLICK task graphs (**2;** §6.5). The division of programs into tasks allows the platform to take advantage of both data and task parallelism, thus exploiting multi-core CPUs (**R2**).

Finally, the FLICK language bounds the resource usage for each invocation of a network service. This allows task graphs to be executed by the FLICK platform according to a cooperative scheduling discipline (**3;** §6.5), permitting a large number of concurrent task graphs to share the same hardware resources with little interference (**R3**). A pool of worker threads execute *tasks* cooperatively, while *channels* move data between tasks.
6.3.2 Overview

We now give a more detailed overview of how a developer uses the FLICK framework (see Figure 6.1). First they write the logic of their application-specific network services in the FLICK language. After compilation by the FLICK compiler, the FLICK platform runs a program as an instance, consisting of a set of task graphs. Each task graph comprises of a directed acyclic graph of tasks connected by task channels. Depending on the program semantics, multiple instances of the task graph can be instantiated for each network request, or a single graph can be used by multiple requests.

A task is a schedulable unit of computation. Each task processes a stream of input values and generates a stream of output values. Initial input to the task graph is handled by one or more input tasks, which consume data from a single input channel, i.e. the byte stream of a TCP connection. An input task then deserialises bytes to values using deserialisation/parsing code generated by the FLICK compiler from the types specified in the FLICK program. Deserialisation splits data into the smallest units appropriate for the task being considered. For example, if the input is from a web client, the byte stream would be deserialised into individual complete HTTP requests; for Hadoop, a key/value pair is more appropriate.

Received data is then processed by one or more compute tasks and, finally, output from the task graph is emitted to the outside world via an output task, representing a single outgoing TCP connection. The output task also executes efficient serialisation code generated from the FLICK program, converting values into a byte stream that is placed onto an output channel for transmission.

6.4 FLICK Programming Model

FLICK provides a domain-specific language (DSL) targeting application-specific middlebox programming. While a difficult task, we decided to design a new language because we found existing general-purpose languages inappropriate for middlebox programming due to their excessive expressive power. Even safe redesigns of widely-used languages, such as Cyclone [95], are too powerful for our needs because, by design, they do not restrict the semantics of programs to terminate and bound the used resources. Existing specialised languages for network services, such as PLAN [85], are typically packet-centric. This makes it hard to implement application-specific traffic logic that is flow-centric. A new domain-specific language presents us with the opportunity to incorporate primitive abstractions that better fit the middlebox domain.
We also considered restricting an existing language to suit our needs, given the numerous systems programming tasks have been simplified by providing DSLs to replace general-purpose programming languages [109, 112, 34, 136, 64]. This, however, presented two difficulties: (i) exposing programmers to a familiar language but with altered semantics would be confusing; and (ii) it would prevent us from including language features for improved safety, such as static type-checking.

### 6.4.1 Language design goals

We thus opt for developing a new language that provides all the essential pieces for enabling application-like programming experience when processing network flows, and yet being able to achieve high-performance and safe execution. This language has the following design goals:

- **Communicating Sequential Processes (CPS) model.** The language shall mask the low-level system execution from developers. This can be achieved by offering a high-level concurrent programming experience using the widely used CPS model. In a CPS model, developers work with logical processes (which represent processing units) and channels (which represent network flows or in-memory data streams) which are automatically mapped to physical resources by the runtime.

- **Functional grammar and primitives:** The language shall provide an application-like data processing experience. The language provides a functional-language-based grammar, and the operations towards a group of channels or tasks are modelled as functional primitives. These primitives are inspired by existing collection processing language (i.e., map-reduce and C# LINQ) and have been shown useful in working constantly ingested data.

- **Strong application-types:** The language shall enforce strong application-types so that it can guarantee the termination and infer the program behaviours during compilation.

- **Malleable network data parser:** The language shall have built-in support for flexibly declaring the parsing logic towards binary network flows, and converting the network data from the byte format into application types.

In the following, we overview the FLICK design using a sample program, and then dive into different aspects respectively.
Listing 6.1 FLICK program for Memcached proxy

```
type cmd: record
  key : string

proc Memcached: (cmd/cmd client, [cmd/cmd] backends)
  | backends => client
  | client => target_backend(backends)

fun target_backend: ([cmd] backends, req:cmd) => ()
  let target = hash(req.key) mod len(backends)
  req => backends[target]
```

6.4.2 Sample program

In the FLICK language, developers describe application-specific network services as a collection of interconnected processes. Each process manipulates values of the application’s data types, in contrast to earlier work which described network services as simple packet processors [101, 36, 121]. Application data is carried over channels, which interconnect processes with one another and with network flows. Processes interact with channels by consuming and processing input data read from them, and by transmitting output over them. Processes, channels and network interactions are handled by the FLICK platform.

The FLICK language can achieve efficient parallel execution on multi-core CPUs using high-level parallel primitives. By default, the language offers parallelism across multiple requests, handling them concurrently. It supports the safe sharing of resources by bounding the resource use of an individual program. Processing of continuous network flows belonging to an application is subdivided into discrete units of work so that each process consumes only a bounded amount of resource. To achieve this, FLICK control structures are restricted to finite iteration only. This is not a significant limitation, however, as application-specific network services typically carry out deterministic transformations of network requests to generate responses. User-defined functions are written in FLICK itself, rather than in a general purpose language as in Click [101] or Pig [135]), which preserves the safety of network services expressed in FLICK.

Listing 6.1 shows a sample FLICK program that implements a Memcached proxy. Programs are composed of three types of declarations: data types (lines 1–2), processes (lines 4–6) and functions (lines 8–10).

Processes have signatures that specify how they connect to the outside world. In this case, a process called Memcached declares a signature containing two channels
the client channel produces and accepts values of type cmd, while backends is an array of channels, each of which produces and accepts values of type cmd.

Processes are instantiated by the FLICK platform, which binds channels to underlying network flows (§6.5). In this example, when a client sends a request, the FLICK platform creates a new Memcached task graph and assigns the client connection to this graph. Giving each client connection a new task graph ensures that responses are routed back to the correct client.

A process body describes how data is transformed and routed between channels connected to a process. The language design ensures that only a finite amount of input from each channel is consumed. The body of the Memcached process describes the application-specific network service: data received from any channel in backends is sent to the client channel (line 5); data received from the client is processed by the target-backend function (line 6), which in turn writes to a suitable channel in the backends array (line 10).

### 6.4.3 Application data types

FLICK programs operate on application data types representing the exchanged messages. After an input task reads such messages from the network, they are parsed into FLICK data types. Similarly, before processed data values are transmitted by an output task, they are serialised into the appropriate wire format representation.

The transformation of messages between wire format and FLICK data types is defined as a message grammar. During compilation, FLICK generates the corresponding parsing and serialisation code from the grammar, which is then used in the input and output tasks of the task graph, respectively. The generated code is optimised for efficiency in three ways: (i) it does not dynamically allocate memory; (ii) it supports the incremental parsing of messages as new data arrives; and (iii) it is adapted automatically to specific use cases.

The syntax to define message grammars is based on that of the Spicy (formerly Binpac++) [153] parser generator. The language provides constructs to define messages and their serialised representation through units, fields, and variables, and their composition: units are used to modularise grammars; fields describe the structure of a unit; and variables can compute the value of expressions during parsing or serialisation, e.g. to determine the size of a field. FLICK grammars can express any LL(1)-parsable grammar as well as grammars with dependent fields, in a manner similar to Spicy. The FLICK framework provides reusable grammars for common protocols, such as the
Listing 6.2 Partial grammar for Memcached protocol

```plaintext
type cmd = unit {
  %byteorder = big;

  magic_code : uint8;
  opcode : uint8;
  key_len : uint16;
  extras_len : uint8; # anonymous field
  status_or_v_bucket : uint16;
  total_len : uint32;
  opaque : uint32;
  cas : uint64;

  var value_len : uint32
    &parse = self.total_len -
    (self.extras_len + self.key_len)
    &serialize = self.total_len =
    self.key_len + self.extras_len + $$;
  extras : bytes &length = self.extras_len;
  key : string &length = self.key_len;
  value : bytes &length = self.value_len;
};
```

HTTP [73] and Memcached protocols [157]. Developers can also specify additional message grammars for custom formats, such as application-specific Hadoop data types.

Listing 6.2 shows a simplified grammar for Memcached. The cmd unit for the corresponding Flick data type is a sequence of fixed-size fields (lines 4–12), a variable (lines 14–18), and variable-size fields (lines 19–21). Each field is declared with its wire-format data type, e.g. the opcode field is an 8-bit integer (line 5). The sizes of the extras, key, and value fields are determined by the parsed value of the extras_len and key_len fields as well as the value_len variable, which is computed during parsing according to the expression in lines 15 and 16. During serialisation, the values of extras_len, key_len, and value_len are updated according to the sizes of the values stored in the extras, key, and value fields. Subsequently, the value of total_len is updated according to the variable’s serialisation expression in lines 17 and 18. The %byteorder property declaration in line 2 specifies the wire format encoding of number values—the generated code transforms such values between the specified big-endian encoding and the host byte-order. More advanced features of the grammar language
include choices between alternative field sequences, field repetitions (i.e. lists), and transformations into custom Flick field types (e.g. enumerations).

Flick grammars aim to be reusable and thus include all fields of a given message format, even though application-specific network services often only require a subset of the information encoded in a message. To avoid generated parsers and serialisers handling unnecessary data, Flick programs make accesses to message fields explicit by declaring a Flick data type corresponding to the message (Listing 6.1, lines 1–2). This enables the Flick compiler to generate input and output tasks that only parse and serialise the required fields for these data types and their dependencies. Other fields are aggregated into either simplified or composite fields, and then skipped or simply copied in their wire format representation. Developers can thus reuse complete message grammars to generate parsers and serialisers, while benefiting from efficient execution for their application-specific network service.

The current Flick implementation does not support exceptions, but data type grammars could provide a default behaviour when a message is incomplete or not in an expected form.

6.4.4 Primitives and compilation

The Flick language is strongly-typed for safety. To facilitate middlebox programming, it includes channels, processes, explicit parallelism, and exception handling as native features. For example, events such as broken connections can be caught and handled by Flick functions, which can notify a backend or record to a log. State handling is essential for describing many middleboxes, and the language supports both session-level and long-term state, whose scope extends across sessions. The latter is provided through a key/value abstraction to task graph instances by the Flick platform. To access it, the programmer declares a dictionary and labels it with a global qualifier. Multiple instances of the service share the key/value store.

The language is restricted to allow only computations that are guaranteed to terminate, thus avoiding expensive isolation mechanisms while supporting multiple processes competing for shared resources. This restriction allows static allocation of memory and cooperative task scheduling (see §6.5).

The Flick language offers primitives to support common datatypes such as bytes, lists and records. Iteration may only be carried out on finite structures (e.g. lists). It also provides primitives such as fold, map and filter but it does not offer higher-order functions: functions such as fold are translated into finite for-loops. Datatypes may be annotated with cardinalities to determine statically the required memory. Loops
Listing 6.3 Flick program for Hadoop data aggregator

```java
1 type kv : record
2   key : string
3   value : string
4
5 proc hadoop : ([kv/-] mappers, -/kv reducer):
6   if (all_ready(mappers)):
7     let result = foldt on mappers
8       ordering elem e1, e2 by elem.key as e_key:
9       let v = combine(e1.val, e2.val)
10      kv(e_key, v)
11     result => reducer
12
13 fun combine(v1 : string, v2 : string) => (string) : ...
```

and branching are compiled to their native counterparts in C++. Channel- and process-related code is translated to API calls exposed by the platform (see §6.5). The language relies on the C++ compiler to optimise the target code.

Channels are typed, and at compile time the platform determines that Flick programs only send valid data into channels. Due to the language’s static memory restrictions, additional channels cannot be declared at runtime, though channels may be rebound, e.g. to connect to a different backend server.

The language also provides foldt, a parallel version of fold that operates over a set of channels. This allows the efficient expression of typical data processing operations, such as a k-way merge sort in which sorted streams of keys from k channels are combined by selecting elements with the smallest key. The expression foldt foldt f o cs aggregates elements from an array of channels cs, selecting elements according to a function o and aggregating according to a function f. As f must be commutative and associative, the aggregation can be performed in parallel, combining elements in a pair-wise manner until only the result remains.

As shown in Listing 6.3, the foldt primitive can be used to implement an application-level network service for parallel data aggregation in Hadoop. Whenever key/value pairs become available from the mappers (lines 5–6), foldt is invoked (lines 7–10). Elements elem are ordered based on elem.key (line 8), and values of elements with the same key (e_key) are merged using a combine function (line 9) to create a new key/value pair (line 10). While foldt could be expressed using core language primitives, the Flick platform has a custom implementation for performance reasons.
While designed to achieve higher safety and performance, the constraints introduced in the design of the FLICK language, e.g. the lack of support for unbounded computation or dynamic memory allocation, imply that not all possible computations can be expressed in FLICK. For instance, algorithms requiring loops with unbounded iterations (e.g. while-like loops) cannot be encoded. In a general purpose programming language, this would be a severe constraint but for the middlebox functionality that FLICK targets we have not found this to cause major limitations.

6.5 FLICK Platform

The FLICK platform is designed around a task graph abstraction, composed of tasks that deserialise input data to typed values, compute over those values, and serialise results for onward transmission. The FLICK compiler translates an input FLICK program to C++, which is in turn compiled and linked against the platform runtime for execution. Figure 6.2 shows an overview of the FLICK platform, which handles network connections, the task graph life-cycle, the communication between tasks and the assignment of tasks to worker threads. Task graphs exploit task and data parallelism at runtime as tasks are assigned to worker threads. Even with only one large network flow, serialisation, processing and deserialisation tasks can be scheduled to run on different CPU cores.
(i) The *application dispatcher* manages the life-cycle of TCP connections: first it maps new incoming connections \(^1\) to a specific program instance \(^2\), typically based on the destination port number of the incoming connection. The application dispatcher manages the listening sockets that handle incoming connections, creating a new input channel for each incoming connection and handing off data from that connection to the correct instance. When a client closes an input TCP connection, the application dispatcher indicates this to the instance; when a task graph has no more active input channels, it is shut down. New connections are directly connected to existing task graphs \(^3\).

(ii) The *graph dispatcher* assigns incoming connections to task graphs \(^4\), instantiating a new one if none suitable exists. The platform maintains a pre-allocated pool of task graphs to avoid the overhead of construction \(^5\). The graph dispatcher also creates new output channel connections to forward processed traffic.

(iii) Tasks are cooperatively scheduled by the *scheduler*, which allocates work among a fixed number of worker threads \(^6\). The number of worker threads is determined by the number of CPU cores available, and worker threads are pinned to CPU cores.

Tasks in a task graph become runnable after receiving data in their input queues (either from the network or from another task). A task that is not currently executing or scheduled is added to a worker queue when it becomes runnable. All buffers are drawn from a pre-allocated pool to avoid dynamic memory allocation. Input tasks use non-blocking sockets and *epoll* event handlers to process socket events. When a socket becomes readable, the input task attached to the relevant socket is scheduled to handle the event.

For scheduling, each worker thread is associated with its own FIFO task queue. Each task within a task graph has a unique identifier, and a hash over this identifier determines which worker’s task queue the task should be assigned to. When a task is to be scheduled, it is always added to the same queue to reduce cache misses.

Each worker thread picks a task from its own queue. If its queue is empty, the worker attempts to scavenge work from other queues and, if none is found, it sleeps until new work arrives. A worker thread runs a task until either all its input data is consumed, or it exceeds a system-defined time quantum, the *timeslice threshold* (typically, 10–100\(\mu\)s; see §6.6). If the timeslice threshold is exceeded, the code generated by the Flick compiler guarantees that the task re-enters the scheduler, placing itself at the back of the queue if it has remaining work to do. A task with no work is not added to the task queue, but when new items arrive in its input channels, it is scheduled again.
A disadvantage of allocating tasks belonging to the same task graphs onto different CPU cores is that this would incur several cache invalidations as data move from one core to another. On the other hand, our design enables higher parallelism as different tasks can execute concurrently in a pipelined fashion, leading to higher throughput.

Some middlebox services must handle many concurrent connections, and they frequently write and read small amounts of data. The kernel TCP stack has a high overhead for creating and destroying sockets to support the Linux Virtual File System (VFS) interface [84]. Socket APIs also require switching between user- and kernel-mode, which adds further overhead. As a result, the Flick platform uses mTCP [93], a highly scalable user-space TCP stack, combined with Intel’s DPDK [90] to reduce these overheads. The original mTCP implementation did not support multi-threaded applications, and we modified mTCP so that Flick I/O tasks can access sockets independently. To leverage the efficient DPDK runtime environment, mTCP executes as a DPDK task. All of these optimisations, significantly improve performance for network-bound services (see §6.6.3).

6.6 Evaluation

The goals of our evaluation are to investigate whether the high-level programming abstraction that Flick carries a performance and scalability cost and whether DPDK and mTCP improve performance. We implement Flick programs for the use cases introduced in §6.2.1, i.e. an HTTP load balancer, a Memcached proxy and a Hadoop data aggregator, and compare their performance against baselines from existing implementations.

After describing the implementation of our use cases (§6.6.1) and the experimental set-up (§6.6.2), we explore the performance and scalability of Flick (§6.6.3). After that, we examine how well the Flick platform isolates resource consumption of multiple Flick programs using cooperative scheduling (§6.6.4).

6.6.1 Use case implementation

For our three use cases, Figure 6.3 shows the task graph obtained from the corresponding Flick program.

**HTTP load balancer.** This Flick program implements an HTTP load balancer that forwards each incoming HTTP request to one of a number of backend web servers. Forwarding is based on a naive hash of the source IP and port and destination IP and
Figure 6.3a shows the corresponding task graph. The application dispatcher forwards each new TCP connection received on port 80 to the graph dispatcher. The graph dispatcher creates a new task graph, which is later destroyed when the connection closes. The input task deserialises the incoming data into HTTP requests. For the first request, the compute task calculates a hash value selecting a backend server for the request. Subsequent requests on the same connection are forwarded to the same backend server. On their return path no computation or parsing is needed, and the data is forwarded without change. We also implement a variant of the HTTP load balancer that does not use backend servers but which returns a fixed response to a
given request. This is effectively a static web server, which we use to test the system without backends.

**Memcached proxy.** In this use case, the FLICK program (Listing 6.1) receives Memcached look-up requests for keys. Requests are forwarded based on hash partitioning to a set of Memcached servers, each storing a disjoint section of the key space. Responses received from the Memcached servers are returned to clients.

Figure 6.3b shows the corresponding task graph. As before, a new task graph is created for each new TCP connection. Unlike the HTTP load balancer, requests from the same client can be dispatched to different Memcached servers, which means that the compute task must have a fan-out greater than one.

When a request is received on the input channel, it is deserialised by the input task. The deserialisation code is automatically generated from the type specification in Listing 6.2. The deserialiser task outputs the Memcached request object, containing the request keys and body, which are passed on to the compute task. The compute task implements the dispatching logic. It identifies the Memcached server responsible for that key and forwards the request to it through the serialiser task. When the response is received from the Memcached server, the deserialiser task deserialises it and passes the response object to the compute task, which returns it to the client through the serialiser task.

**Hadoop data aggregator.** The Hadoop data aggregator implements the combiner function of a map/reduce job to perform early data aggregation in the network, as described in §6.2.1. It is implemented in FLICK according to Listing 6.3. We focus on a wordcount job in which the combiner function aggregates word counters produced by mappers over a set of documents.

For each Hadoop job, the platform creates a separate task graph per reducer (Figure 6.3c). The input tasks deserialise the stream of intermediate results (i.e. key/value pairs) from the mappers. Compute tasks combine the data with each compute task taking two input streams and producing one output. The output task converts the data to the byte stream, as per the Hadoop wire format.

### 6.6.2 Experimental set-up

We deploy the prototype implementation of the FLICK platform on servers with two 8-core Xeon E5-2690 CPUs running at 2.9 Ghz with 32 GB of memory. Clients and back-end machines are deployed on a cluster of 16 machines with 4-core Xeon E3-1240 CPUs running at 3.3 Ghz. All machines use Ubuntu Linux version 12.04. The clients
and backend machines have 1 Gbps NICs, and the servers executing the Flick platform have 10 Gbps NICs. The client and backend machines connect to a 1 Gbps switch, and the Flick platform connects to a 10 Gbps switch. The switches have a 20 Gbps connection between them. We examine the performance of Flick with and without mTCP/DPDK.

To evaluate the performance of the HTTP load balancer, we use multiple instances of ApacheBench (ab) [26], a standard tool for measuring web server performance, together with 10 backend servers that run the Apache web server [159]. Throughput is measured in terms of connections per second as well as requests per second for HTTP keep-alive connections. We compare against the standard Apache (mod_proxy_balancer) and the Nginx [3] load balancers.

For the Memcached proxy, we deploy 128 clients running libmemcached [12], a standard client library for interacting with Memcached servers. We use 10 Memcached servers as backends and compare the performance against a production Memcached proxy, Moxi [125]. We measure performance in terms of throughput (i.e. requests served per second) and request latency. Clients send a single request and wait for a response before sending the next request.

For the Hadoop data aggregator, the workload is a wordcount job. It uses a sum as the aggregation computation and an input dataset with a high data reduction ratio. The datasets used in experiments are 8 GB, 12 GB and 16 GB (larger data sets were also used for validation). Here we measure performance in terms of the absolute network throughput.

In all graphs, the plotted points are the mean of five runs with identical parameters. Error bars correspond to a 95% confidence interval.

### 6.6.3 Performance

**HTTP load balancer.** We begin by measuring the performance of the static web server with an increasing load. This exercises the following components of the Flick platform: HTTP parsing, internal and external channel operation and task scheduling. The results are for 100 to 1,600 concurrent connections (above these loads, Apache and Nginx begin to suffer timeouts). Across the entire workload, Flick achieves superior performance. It achieves a peak throughput of 306,000 requests/sec for the kernel version and 380,000 requests/sec with mTCP. The maximum throughput achieved by Apache is 159,000 requests/sec and by Nginx is 217,000 requests/sec. Flick also shows lower latency, particularly at high concurrency when Apache and Nginx
6.6 Evaluation

use large numbers of threads. This confirms that, while Flick provides a general-purpose platform for creating application-specific network functions, it can outperform purpose-written services.

To investigate the per-flow overhead due to TCP set-up/tear-down, we also repeat the same experiment but with each web request establishing a separate TCP connection (i.e. non-persistent HTTP). This reduces the throughput in all deployments: 35,000 requests/sec for Apache; 44,000 requests/sec for Nginx; and 45,000 requests/sec for Flick, which maintains the lowest latency. Here the kernel TCP performance for connection set-up and tear-down is a bottleneck: the mTCP version of Flick handles up to 193,000 requests/sec.

Next, we repeat the experiment using our HTTP load balancer implementation to explore the impact of both receiving and forwarding requests. The set-up is as described in §6.6.2. We use small HTTP payloads (137 bytes each) to ensure that the network and the backends are never the bottleneck. As for the web server experiment, we first consider persistent connections. Figures 6.4a and 6.4b confirm the previous
results: Flick achieves up to $1.4 \times$ higher throughput than Nginx and $2.2 \times$ higher than Apache. Using mTCP, the performance is even better with higher throughput and lower latency: Flick achieves a maximum throughput $2.7 \times$ higher than Nginx and $4.2 \times$ higher than Apache. In all cases, Flick has lower latency.

With non-persistent connections, the kernel version of Flick exhibits a lower throughput than Apache and Nginx (see Figure 6.5a). Both Apache and Nginx keep persistent TCP connections to the backends, but Flick does not, which increases its connection set-up/tear-down cost. When mTCP is used with its lower per connection cost, Flick shows better performance than either with a maximum throughput $2.5 \times$ higher than that of Nginx and $2.1 \times$ higher than that of Apache. In addition, both the kernel and mTCP versions of Flick maintain the lowest latency of the systems, as shown in Figure 6.5b.

**Memcached proxy.** For the Memcached proxy use case, we compare the performance of Flick against Moxi [125], as we increase the number of CPU cores. We chose Moxi because it supports the binary Memcached protocol and is multi-threaded. In our
set-up, 128 clients make concurrent requests using the Memcached binary protocol over persistent connections, which are then multiplexed to the backends.

Figures 6.6a and 6.6b show the throughput, in terms of the number of requests/sec, and the latency, respectively. With more CPU cores, the throughput initially increases for both systems. The kernel version achieves a maximum throughput of 126,000 requests/sec with 8 CPU cores and the mTCP version achieves 198,000 requests/sec with 16 CPU cores. Moxi peaks at 82,000 requests/sec with 4 CPU cores. FLICK’s latency decreases with more CPU cores due to the larger processing capacity available in the system. The latency of Moxi beyond 4 CPU cores and FLICK’s beyond 8 CPU cores increases as threads compete over common data structures.

**Hadoop data aggregator.** The previous use cases had relatively simple task graphs (see Figure 6.3) and considerable overhead comes from the connection set-up and tear-down, with many network requests processed in parallel. In contrast, the **Hadoop data aggregator** use case has a more complex task graph, and we use it to assess the overhead of FLICK’s communication channels and intra-graph scheduling. Here the
tasks are compute bound, and the impact of the network overhead is limited. We only present the kernel results because the mTCP results are similar.

We deploy 8 mappers clients, each with 1 Gbps connections, to connect to the Flick server. The task graph therefore has 16 tasks (8 input, 7 processing and 1 output). The Flick Hadoop data aggregator runs on a server with 16 CPU cores without hyper-threading.

Figure 6.7 shows that Flick scales well with the number of CPU cores, achieving a maximum throughput of 7,513 Mbps with 16 CPU cores. This is the maximum capacity of the 8 network links (once accounted for TCP overhead), and matches measurements from iperf. We conclude that the Flick platform can exploit the high level of parallelism of multi-core servers and efficiently schedule multiple tasks concurrently to maximise network throughput.

The results in Figure 6.7 represent three data sets of 8 GB, 12 GB and 16 GB mentioned in §6.6.2, consisting of words of 8, 12 and 16 characters, respectively. The Flick platform can more efficiently process the longer words because they comprise fewer key value pairs.

6.6.4 Resource sharing

We finish our experiments by examining the ability of the Flick platform to ensure efficient resource sharing, as described in §6.3.1. For this, we use a micro-benchmark running 200 tasks. Each task consumes a finite number of data items, computing a simple addition for each input byte. The tasks are equally split between two classes: light tasks operate on 1 KB data items; and heavy tasks operate on 16 KB data items. We consider three scheduling policies: (i) cooperative is the policy used by Flick, in which each task is given a fixed amount of CPU time before it yields control to another task; (ii) non-cooperative runs a scheduled task to completion, potentially letting the
OS scheduler preempt it; and (iii) \textit{round robin} schedules each task for one data item only.

Figure 6.8 shows the total completion time for light and heavy tasks. Since the light tasks handle less data, they should, given a fair share of resources, finish before the heavy tasks. With the \textit{round robin} policy, this does not happen: the heavy tasks take longer to process one data item. Each time they are scheduled, they occupy the worker thread for longer than a light task. Conversely, with the \textit{non-cooperative} policy, each task runs to completion. The total completion time for the light and heavy tasks is determined by their scheduling order. However, with FLICK’s \textit{cooperative} policy, the light tasks are allowed to complete ahead of the heavy tasks without increasing the overall runtime—each task is given a fair share of the CPU time.

### 6.7 Summary

Existing platforms for in-network processing typically provide a low-level, packet-based API. This makes it hard to implement application-specific network services. In addition, they lack support for low-overhead performance isolation, thus preventing efficient consolidation. To address these challenges, we developed FLICK and Emu, two domain-specific language and supporting platforms that provide developers with high-level primitives to write generic application-specific network services. Extensive experiment results show that these platforms are able to greatly reduce the development effort, while achieving better performance than specialised software and hardware implementations.
Chapter 7

Conclusions

In this chapter, I summarise the findings of my PhD study and point out future research opportunities.

7.1 Summary

Building an efficient data analytics system is a critical but non-trivial task. It requires novel designs in both control and data planes. It also requires rethinking towards the programming interface of commodity hardware, thus spawning more and more application-specific optimisations that can be quickly developed and efficiently deployed.

In this thesis, I introduced Chi, a new control plane for distributed streaming systems. These systems have been suffering from high coordination overhead, which makes them exhibit poor performance in working with real-world ingested data. Using Chi, streaming systems are able to implement various feedback-loop policies towards large-scale data processing pipelines, leading to stable optimal performance with dynamic and unpredictable workloads. This enabled feature has motivated practitioners to adopt Chi in large-scale production clusters.

This thesis also provides two novel methods, NetAGG and MLNet, for improving data plane performance. NetAGG explores the possibility of adopting commodity middlebox to perform complex application-specific data aggregation functions. By developing scaling-out aggregation tree bind with a high-performance network stack, NetAGG is able to process ingested traffic at a high rate, which makes it suitable to be deployed at high-throughput network nodes. MLNet extends the promises of NetAGG into the networks where the access to networking hardware is restricted. By providing high-level communication primitives and application-aware communication services, MLNet is able to largely improve the performance of large-scale machine
learning jobs. In fact, NetAGG and MLNet are complementary. They can serve as a joint solution that provides flexible network optimisation options for Big data systems.

The success of NetAGG and MLNet demonstrates the benefits of bridging applications and networks in data centres; however, their development often require expert knowledge about both application and hardware, implying a high development barrier. To advocate application-network co-designs among developers, I proposed Flick, a high-level declarative network programming language that helps developers quickly implement application-specific network services. The Flick language comes with a high-performance runtime that can efficiently run on commodity network elements. The use of Flick reduces the development duration of application-network co-designs from weeks to days. This does not come with the cost of performance. The Flick runtime has shown satisfactory performance in popular benchmarks.

7.2 Future Directions

Looking forward, I anticipate there would be more and more Big data systems being deployed into the real world, and being increasingly intelligent through the uses of artificial intelligence algorithms. Hence, future Big data systems are likely to have (i) online pipelines that process constantly generated data, leading to timely understanding towards changes in environments, and (ii) offline pipelines, which are usually more expensive in computation compared to the online counterpart, that accumulates the knowledge over time using, for example, knowledge-base and statistical models, and (iii) a malleable dataflow engine that can efficiently run on heterogeneous platforms. Such a system need to efficiently bridge the online and offline worlds. Online pipeline results are served to users and used as feedbacks to refine offline pipelines. Refined knowledge is then used for improving online pipeline performance, establishing a feedback loop that can constantly manage data processing. Towards such a system, we need to explore three research directions that cover the system scheduling, coordination and heterogeneity aspects.

Decentralised, low-latency scheduling. To process real-world data that can be ingested at any time, computation systems need to run a large number of long-running, if not forever, tasks. These tasks can have highly dynamic resource usage as data are produced by loosely connected edge devices. They are different from the existing design of schedulers. The existing tasks are mostly short-lived, scheduled in batches, and connected to reliable data sources, such as Hadoop or Kafka. It thus points out the need for having a new scheduler that can cope with both long-running and
short-lived tasks. In addition, it is also important to support service-level agreements (SLAs), e.g., bounded processing delays, within the scheduler. SLAs are critical for supporting multi-tenancy as competing tenants could over-claim resource, resulting in severe under-utilisation in resources. Also, the commitment of SLAs is important for mission-critical jobs, e.g., monitoring and analysing patient real-time data, as it can avoid catastrophic consequence caused by, for example, unbounded processing delays.

**Generalised coordination for both online and offline computations.** The second direction is to develop coordination algorithms for bridging the online and offline pipelines. In existing systems, online and offline pipelines usually leverage an external data store to communicate. Offline pipeline results are first stored in the store, and waits for the online pipeline to read after a fixed time interval. The online pipeline results, on the other hand, are often batched in the store for amortising processing cost, and later pushed into the offline pipeline as batches. The use of an external hop prevents the systems from achieving low-latency coordination between online and offline pipelines, leading to poor performance in running real-time workloads, such as self-driving and healthcare monitoring. In addition, the coordination algorithms shall be able to adapt itself in managing a large-scale pipeline. Future pipelines can have billions of nodes, due to the rise of Internet-of-things and reinforcement learning applications [96, 123]. According to the trade-off between the computation complexity and user’s consistency requirement, these algorithms need to accomplish different levels of consistency (between the eventual and strong consistency).

**Seamless integration of edge devices.** The third direction is to enable the joint utilisation of the high-end processors in data centre servers and the heterogeneous processors on edge devices. In these days, edge devices are equipped with increasingly powerful processors that can achieve comparable performance as their counterparts on servers. Machine learning systems, such as TensorFlow, have developed dataflow engines that can run on both mobile and server platforms. However, it remains unclear how to migrate computation tasks between the edges and servers. Each migration decision has to carefully attack a trade-off between latency, throughput and consistency; while respecting deployment and energy constraints. An exemplary system can offload parts of computation into edge devices that can spontaneously go offline, and thus help end users to obtain early yet approximated results. It later consolidates data centre servers to compute the accurate results which are sent to all users once possible.
References


References


References


