

© 2022 Atul Sandur

AN ADAPTIVE PLACEMENT FRAMEWORK FOR EFFICIENT NEAR-DATA
STREAM PROCESSING OVER DATA SOURCE-EDGE-CLOUD SYSTEMS

BY

ATUL SANDUR

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2022

Urbana, Illinois

Doctoral Committee:

Professor Gul Agha, Chair
Professor Klara Nahrstedt
Assistant Professor Tianyin Xu
Associate Professor Myeongjae Jeon, UNIST, South Korea

ABSTRACT

Large amounts of data are being generated across many different domains including datacenters, surveillance cameras, mobile devices and other Internet-of-things (IoT) systems. This data is generated as streams, i.e., an unbounded sequence of data items which needs to be processed in near real-time. Large-scale datacenters report generating up to 10s of PBs of monitoring data per day from hundreds of thousands of server nodes. High bandwidth video data at rates of 1.4 TB/hr or more, will be generated from autonomous vehicles and other IoT systems. Processing this data is challenging due to limited network and compute resources in the cloud. Utilizing compute resources closer to the data source helps reduce network transfer costs and alleviates the compute load on cloud nodes. However, near-data compute resources are limited. E.g. data source nodes may have a small fraction of compute resources (e.g., 1-2 CPU cores) for near-data processing compared to cloud servers. Analytics applications which process streaming data are also becoming complex and increasingly resource-intensive. Thus, all the analytics processing cannot be done on nodes near the data source, while using remote resources incurs network transfer costs.

To alleviate the resulting compute-communication bottlenecks, applications are *partitioned* to reduce network transfer costs, by making placement decisions to split the execution of its operations between cloud and near-data resource nodes. Resource conditions are frequently changing in such systems, thus requiring the placement decisions to also adapt. As an example, monitoring queries on server nodes (i.e., data sources in monitoring pipelines) are co-located with foreground applications such as web services which have different compute resource needs based on incoming traffic requests. Thus, resources available to the monitoring queries change over time. Furthermore, compute resource needs of applications on edge and cloud nodes are also affected by changing input rates. E.g., surveillance camera applications consist of dynamic camera networks, with cameras being added or removed from the network to handle tasks such as object detection and target tracking.

In this dissertation, we argue for the need to make highly efficient placement decisions to meet the application resource demands under fast changing resource conditions. Existing systems place application components between near-data resources and cloud nodes, with the goal to better utilize the available compute resources and improve application performance. However, they do not scale well to efficiently handle the increasing number of application operations/resource nodes in modern stream processing systems. Furthermore, placement techniques which optimize application power consumption on battery-driven data source

nodes depend on accurate power prediction tools. Current tools typically rely on analytical power models based on hardware resource parameters, which are challenging to develop for modern hardware with complex and heterogeneous compute resources (e.g., multi-core CPUs, GPUs, TPUs). Moreover, modern application operations have complex power characteristics which depend on the input data stream.

We address the above challenges by building a framework to make scalable placement decisions for near-data stream processing applications. We develop algorithms that identify effective placement decisions for application partitioning between near-data resources and the cloud. We build proof-of-concept systems to implement the proposed algorithms and validate their effectiveness in scaling to large workloads. The first algorithm focuses on optimizing the application’s end-to-end processing time, which is designed for making partitioning decisions between edges and cloud nodes. We map application operations to resource nodes on which they need to execute, using dynamic programming techniques to scale to large application graphs. The second system investigates the use of a fine-grained placement strategy on data source nodes, which can effectively utilize the limited compute resources on these nodes. An algorithm to quickly adapt the placement plan (in the order of seconds) is designed to handle fast changing resource conditions on data source nodes. Fast adaptation results from combining a query cost model-based and model agnostic heuristic to make placement decisions. Novel system-level abstractions are introduced in the stream processing pipeline to implement and evaluate the proposed placement algorithm. The third system is a first step towards extending our framework to support energy-aware placement decisions. Power cost of application operators is predicted without using complex hardware or software-based analytical power models, which enables our system to be easily applied to devices with heterogeneous hardware resources. The power predictions can then be used to make energy-aware placement decisions.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Gul Agha, for his constant support, encouragement and guidance throughout my PhD program. He has always encouraged me to think big and taught me the skills required to excel in research. I have benefited enormously from his knowledge and motivation during the course of my Ph.D. study. He has been a mentor and guide, both professionally and personally, and has given me opportunities that I could not have ever had without his help. I am incredibly grateful for the privilege to have been his student. Additionally, I would like to thank my committee members, Prof. Klara Nahrstedt, Prof. Tianyin Xu and Prof. Myeongjae Jeon (MJ), for providing insightful feedback that helped shape my thesis. Klara's work ethic and her dedication to her students' success has always been inspiring to me. Tianyin is the systems researcher I aspire to be, prolific with an infectious passion for his work, yet very humble, empathetic and relatable to his students. I am extremely grateful to have had the opportunity to work with MJ. His dedication to research and desire to help me succeed, has pushed me during my tough times. He taught me the critical thinking and execution skills required to do great systems research, and they are invaluable lessons I will carry with me.

I have been fortunate to work with great collaborators during my journey. Prof. Edward Lee, Ben Zhang and Marten Lohstroh introduced me to exciting problems in cyber-physical systems at the beginning of my Ph.D. journey. My internship at Microsoft Research (MSR) was a turning point during my journey. I would like to express my sincere gratitude to Stavros Volos, my internship mentor at MSR. He gave me the opportunity to work on exciting research, continued to guide me after the internship and helped me through several roadblocks along the way. I will always be inspired by his high standards for systems research and meticulous approach to solving problems. I am also thankful to Rahul Potharaju from Microsoft for guiding and supporting me throughout my internship. At the University of Illinois, I have enjoyed working with and learning from seasoned researchers Kirill Mechitov and Karl Palmskog.

I was also fortunate to work with several great student collaborators. I would like to dedicate special thanks to Tarek Elgamal, who has been both a great friend and collaborator throughout my journey. I would always look forward to the long and highly stimulating brainstorming sessions we would have in his office. I admire Tarek's passion for his work and practical approach to system building. His knack for identifying and executing on exciting systems work is inspiring. An important portion of the work in this thesis was the outcome

of my collaboration with Tarek. I was also fortunate to have long term collaborations with Dipayan Mukherjee and ChanHo Park, both of whom I have thoroughly enjoyed working with. Pratik Lahiri has been an amazing collaborator whose insights in machine learning have helped me understand and apply them to practical system building challenges. Si Liu helped me gain insights into exciting challenges on the theoretical side of building large scale systems. I would also like to thank Dan Plyukhin, Minas Charalambides, Sihan Li, Qi Wang for the fun and stimulating conversations on various research topics over the years.

I would not have been able to get through graduate school without the support of my family and friends at Illinois. My sister Amitha Sandur was also studying at Illinois during the major part of my stay in Champaign. I am grateful for having close family by my side during my graduate studies. Her constant love, care and support helped me during my tough times and made my stay in Champaign a lot easier and fun. It would have been incredibly hard to get through graduate school, if not for the company of amazing friends that I was fortunate to make during my stay in Champaign. Pratik Joshi, Prashant Jayannavar, Vipul Harsh, Rachneet Kaur, Shreya Arya, Nitin Bhat, Vishaal Mohan, Ashwini Raina, Stephen Skeirek, Li Liyi, Unnat Jain and Anshika Gupta have all given me memories that I will cherish for a long time. A special mention to Sarah Luebke and Pratik Lahiri for all the wonderful times spent together. They were right by my side through my good and bad days and supported me throughout my journey. They taught me how to work hard and also have fun along the way. I will treasure their friendship for the rest of my life.

I am grateful to the Computer Science Department at University of Illinois, Urbana-Champaign, for giving me an opportunity to pursue my Ph.D. degree. Many thanks to the staff of the department, including but not limited to: Viveka Perera Kudaligama, Kathy Runck, Kara MacGregor, Mary Beth Kelley, Maggie Metzger Chappell, and Jennifer Comstock for their help and support during my program.

Finally and most importantly, I want to thank my family for their unconditional love and support. The faith that my parents (Ajith Kumar Sandur and Sreedevi Sandur) have in me gives me the strength and courage to pursue my dreams. They are my pillar of strength and I am extremely fortunate to have their guidance, warmth, encouragement and support in my life. I will forever be indebted to them for the sacrifices they made to raise their children and the hardships they endured, and I would not be able to achieve my goals without their blessings. I dedicate this dissertation to them.

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	1
1.1	Scope	8
1.2	Thesis Contributions	11
1.3	Thesis Organization	13
CHAPTER 2	NEAR-DATA STREAM PROCESSING: OPPORTUNITIES & CHALLENGES	14
2.1	Real-World Motivating Scenarios	18
2.2	Resource Topologies	21
2.3	Summary	23
CHAPTER 3	RELATED WORK	25
3.1	Low-Level Orchestration Frameworks	25
3.2	Performance Optimization	26
3.3	Energy-Aware Placement	35
3.4	Summary	37
CHAPTER 4	MODELS AND PROBLEM DEFINITIONS	40
4.1	Data Model	40
4.2	Resource Model	40
4.3	Application Model	44
4.4	Adaptation	47
4.5	Operator Profile	49
4.6	Operator Energy Cost Model	50
4.7	Execution Model	51
4.8	Problem Definitions	52
CHAPTER 5	ALGORITHMS AND ANALYSIS	59
5.1	General Operator Placement for Type I Resource Topologies	59
5.2	General Operator Placement Algorithm for Type II Resource Topologies: Challenges	74
5.3	Near-Data Operator Placement Algorithm for Type II Resource Topologies .	76
5.4	Energy Estimation on the Data Source for Type II Resource Topologies . . .	85
CHAPTER 6	OPERATOR PLACEMENT BETWEEN EDGE AND CLOUD RESOURCES	90
6.1	Introduction	90
6.2	Motivating Scenarios	92

6.3	System Architecture	93
6.4	Evaluation	97
6.5	Conclusion	102
CHAPTER 7 OPERATOR PLACEMENT BETWEEN DATA SOURCE, EDGE AND CLOUD RESOURCES		
7.1	Introduction	103
7.2	Background & Motivation	105
7.3	Server Monitoring Insights for Partitioning on Data Source Nodes	109
7.4	Jarvis System	110
7.5	Implementation	115
7.6	Evaluation	117
7.7	Conclusion	125
CHAPTER 8 RESOURCE MODEL-AGNOSTIC POWER PREDICTION FOR MOBILE VIDEO ANALYTICS APPLICATIONS		
8.1	Introduction	126
8.2	Motivating Scenarios	128
8.3	Proposed Solution	133
8.4	System Overview	139
8.5	Implementation	139
8.6	Discussion	141
8.7	Evaluation	143
8.8	Conclusion	154
CHAPTER 9 CONCLUSIONS AND FUTURE WORK		
9.1	Summary and Conclusions	156
9.2	Discussion and Future Work	157
REFERENCES		164

CHAPTER 1: INTRODUCTION

Computer systems have been evolving at an incredibly fast-pace, ever since the modern computer era began in 1945. For the next 40 years, computers continued to become increasingly powerful but they were still bulky and expensive. Furthermore, technology had not evolved for enabling efficient communication between different computers, so they operated independently from one another. The two limitations didn't last long as in the mid-1980s, powerful microprocessors were developed. Initially, these were 8-bit machines, but soon 16, 32 and 64-bit machines were developed. Most computing systems today have multi-core CPUs, which enable applications to run in parallel for better performance. The current generation of machines have the computing power of mainframes deployed 30 or 40 years ago, but for 1/1000th of the price or less [1]. Furthermore, computer systems have become miniaturized and today, a very popular example of such systems are smartphones which have impressive computing capabilities available for users at their finger tips. They are packed with sensors, lots of memory and compute resources.

Concurrently, development of high speed computer networks have enabled thousands of machines to be connected for exchanging information within a few microseconds. Wide-area networks (WANs) allow computing systems all over the world to communicate at speeds varying from tens of thousands to hundreds of millions of bits per second. Advancements in compute and networking technologies have enabled distributed computing systems to be composed of many networked machines, whether they are of a small or large form factor. Such networked machines can be geographically distributed, and they may vary in size from a handful of devices to millions of machines. Figure 1.1 shows examples of real world distributed systems. They may range from tens of thousands of powerful server nodes interconnected via high speed cables within a datacenter to smartphones and other internet-of-things (IoT) devices connected in a hierarchical topology. In such a hierarchical topology, edge servers are connected to data sources via radio networks and the servers are connected over wide-area networks (WAN) to the cloud datacenters at the root node. Such distributed systems provide powerful compute capabilities, however, harnessing them effectively to benefit applications is still challenging.

Figure 1.2 shows the high level architecture for modern applications running on distributed resources. At the lowest layer, are the physical compute nodes interconnected via a given network topology. This is the execution plane and it represents the network of physical node agents and interconnects, which enables orchestrating resources for applications to execute. The distributed runtime is the control plane and it interfaces with the execution plane to

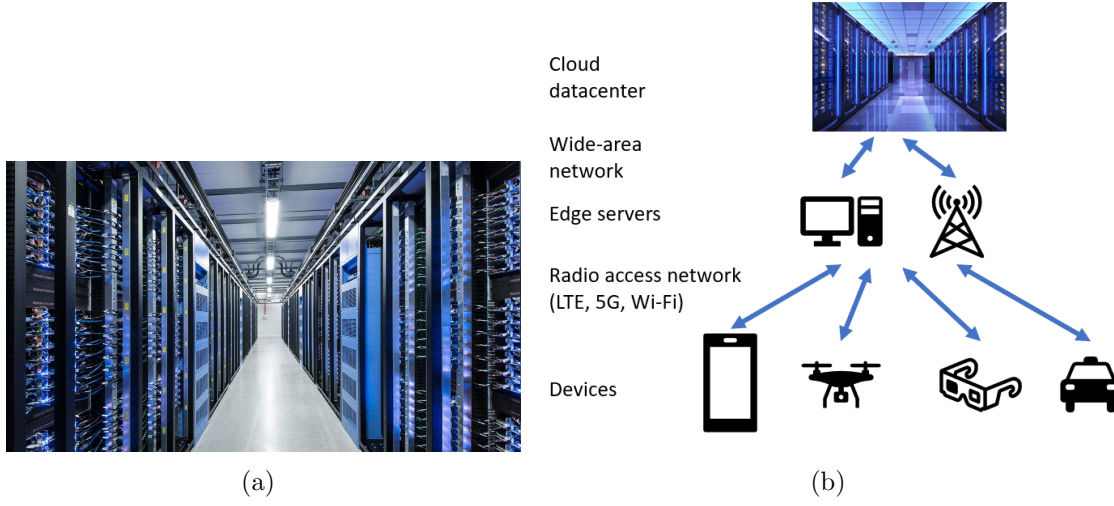


Figure 1.1: Examples of modern distributed systems. (a) Facebook datacenter showing tens of thousands of interconnected servers, which need to coordinate to efficiently process user requests, and (b) Hierarchical architectures with mobile and other IoT devices connecting to edge servers and micro datacenters near cellular towers. Cloud datacenters available to the edge devices via wide-area network (WAN) for compute processing.

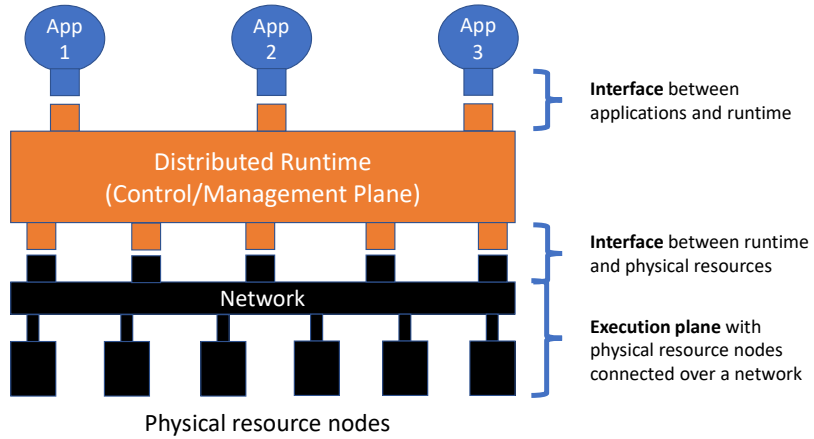


Figure 1.2: Architecture for a modern distributed system. Physical nodes are interconnected in the execution plane to run computation and exchange data over the network, The resources are exposed to the Distributed Runtime via an interface. Resource management for applications is also done by the Distributed Runtime.

manage the underlying physical resources. The runtime decides how to distribute application components across available physical nodes in the execution plane, makes scheduling decisions and provides fault tolerance capabilities. Applications execute on top of the distributed runtime, and rely on it to exploit the available networked-physical resources and scale to handle large input load while maximizing performance metrics such as throughput, latency, etc. We now discuss characteristics of distributed applications and the requirements of distributed runtime to handle the applications.

Modern computing applications are finding utility in diverse domains ranging from health-care, wearables, transportation, social networking, manufacturing, etc. This has given rise to a variety of applications being developed to handle different types of input data and the associated processing logic to be executed on the data. The applications can range from web services and gaming applications to video processing pipelines and query analytics engines. Reduced costs of computing devices including portable mobile phones and desktops, means that more users have easy access to these applications and are able to leverage their benefits in their daily lives. The increasing popularity and ubiquity of computing makes it critical for applications to be designed so they can **scale** well and handle the processing needs of requests from a large number of users/clients. We already see the usage trends indicating how critical it is for applications to be built with scalability as a first class design consideration. E.g. popular web search engines need to process tens of millions of search requests per second globally [2] while web applications like Wikipedia need to handle tens of thousands of client requests per second [3]. More recently, modern video analytics services need to handle incoming data from cameras ranging in number from thousands for traffic cameras [4] to millions of surveillance cameras [5]. Each of these cameras may generate between hundreds of Kbps to 10 Mbps or more depending on the resolution requirements for the application. Furthermore, monitoring analytics services within datacenters need to process up to 10s of PBs of data per day of logs [6]. Current trends indicate that the processing needs of computing applications will continue to grow. Thus, distributing the applications to fully utilize compute resources in a distributed environment becomes a necessity to meet the user processing requirements at large scale. We consider applications consisting of user-defined components such as tasks or operations (referred to as *operators*) which communicate data between them and thus have dataflow dependencies. Operators can be distributed across the available resource nodes. We refer to *application partitioning* as the placement of operators across a distributed set of resource nodes, with the goal to optimize an application service-level agreement (SLA) such as request latency, throughput or energy cost. Solving the application partitioning problem is not trivial. Executing all the operators on a single node is not feasible as we quickly run into compute bottlenecks; a single node cannot handle the processing

needs of applications at large scale of user requests. However, distributing the operators that have dataflow dependencies between them, means that we may introduce communication bottlenecks when processing the input. Thus, minimizing the compute cost per-node can increase the communication bottlenecks between nodes, while minimizing the communication cost requires processing all the data on a fewer number of nodes which can result in compute bottlenecks. The distributed runtime needs to consider this compute-communication tradeoff during application partitioning.

The input data and distributed execution environment in which applications run are also becoming increasingly complex and dynamic. Changes in web page content require search engines to quickly update their indexes so as to reflect the latest content when serving user search requests. Some web pages may require more processing vs. others, depending on their content and frequency of updates. E.g., Wikipedia pages may be less frequently updated compared to content on social media sites. Web applications also serve requests based on their execution context, which includes factors such as network conditions (i.e., broadband vs. not), basic HTML vs. standard view, simplified vs. rich graphics environment and JavaScript-enabled vs. disabled modes [7]. Compute needs of video analytics services are sensitive to input parameters such as incoming frame rate and resolution while query analytics in server monitoring scenarios require more compute resources with increased input rate, such as a spike in the rate of error logs generated during anomalous server behavior. Furthermore, these applications may execute in a multi-tenant environment (e.g., smartphones), i.e., they typically share hardware resources with other applications for better utilization of available compute resources at the platform level. This causes resources available for one application to become dependent on the resource usage of other applications running in the environment. Different partitioning decisions also affect energy use due to changing compute costs on energy-constrained devices such as smartphones. Such resource dynamics have an impact on the compute-communication tradeoff curve that results from distributing application operators across resource nodes. So the distributed runtime needs to **adapt** its application partitioning plan, when the execution environment changes.

Adapting the application partitioning plan when the execution environment changes, requires that the current environment be profiled to understand the new compute and communication costs for different application partitioning strategies. The profiled data is then used to update the partitioning plan if needed. And finally, the new plan is implemented by migrating application operators from the nodes on which they are currently executing, to destination nodes based on the target partitioning strategy. Our vision is to design a distributed runtime which enables efficiently determining new application partitioning strategies and fluidly migrating operators to seamlessly schedule the applications across

compute nodes distributed across the network. Realizing this vision comes with several challenges. Profiling the application in the new execution environment can be expensive.

- If a costly operation needs to be profiled for its compute cost at runtime, it may not be possible to accurately do so if the available compute resources are not enough to fully process all the input. E.g., consider a continuous stream of monitoring logs generated by a web application running on a cluster of server nodes in a cloud platform. An analytics query can process the logs in 5 minute windows to monitor the health of the web service and detect service reliability issues [8]. A potential query partitioning strategy may execute the entire query on the server node itself, so we need to profile the query compute cost in such a configuration. In such a case, if the query takes 2 cores to fully process the input, but the node has only 1 core available for the analytics query (since server node resources are shared with the web application), then the query can only process 50% of the streaming window data. Profiling will not be able to estimate the compute cost of processing the entire window of data. Linear extrapolation may not accurately determine the compute cost since analytics query operators exhibit non-linearity in the compute cost with increasing input records [9]. Similar observations with regard to profiling cost, are made for video analytics applications and corroborated by previous works [10].
- Obtaining a new execution plan based on profiled data can be expensive. The cost of determining a mapping between each operator and the resource node on which it needs to be executed, can be exponential in the number of operators.
- Partitioning frameworks need to consider and optimize the use of multiple resources such as compute, network and energy resources on battery-operated devices.
- Finally, migrating the operators from their source to destination nodes when executing a new partitioning plan requires minimizing the migration overheads for the application. Such overhead comes from the need to transfer intermediate processing state of operations from source to destination nodes and minimizing migration time to reduce the backlog of data that accumulates during the migration period.

Several works have addressed various aspects of the application partitioning problem. In stream processing, DS2 [11] and Sonata [12] completely rely on accurate profiling of the streaming operators to develop a query cost model and make partitioning decisions using the model. But profiling can be expensive and inaccurate with limited computing capacity available, thereby reducing the accuracy of a cost model-based solution. In video analytics

applications, there are no analytical models to compute the resource cost of application operators based on the input and application configuration parameters [10], thus making it infeasible to analytically compute a new partitioning plan based on resource conditions. Profiling such applications at runtime is expensive. For web search workloads, execution costs depend on the input which can change over time, thus the profiling of compute cost cannot be done in advance [13]. Online profiling can be expensive in resource-constrained environments. Thus, there is limited applicability of execution cost-model based approaches for application partitioning, which depends on accurate profiling data. Moreover, on battery-operated devices such as smartphones, energy cost is an important consideration when making partitioning decisions. As we will see in Chapter 8, profiling the energy cost of applications is becoming challenging due to complex energy usage characteristics of modern mobile applications and increase in the heterogeneous compute resources such as multi-core CPU, GPUs and TPUs on which they run.

Computing a partitioning of application operators is the next challenge. Po-Hao Chang, et.al. [7] proposed an adaptation framework for web applications that views them as a composition of actors [14], which are autonomous, concurrent objects executing asynchronously and communicate by asynchronous message-passing. Developers are allowed to annotate the programs using specification rules. The rules specify the execution context in which an action is to be triggered, and the action may depend on setting an attribute such as the location to execute the actor. The execution context can be specified based on resource parameters such as the threshold CPU cost of executing an actor within the web application. To implement a partitioning strategy, the action can fix the attribute value which indicates the location where the actor is to be executed. Such approaches rely on developer knowledge to make local partitioning decisions, one actor/component at a time. It doesn't permit specifying complex partitioning policies which jointly consider the partitioning decisions across all the operators, to optimize for a global application metric such as end-to-end latency. Moreover, such manual partitioning strategies cannot adapt to dynamic resource conditions. Crawlets [15] parallelize the crawling operation for web search engines and reduce network transfer costs by crawling pages at a site locally on its hosting server node, and only sending a compact summary of the pages back to the search engine. The decentralized approach avoids the network bottlenecks that result from crawling strategies using centralized architectures, which require each page to be downloaded to a cluster hosting the search engine and the downloaded data sizes can grow up to tens of TBs or more. However, doing local processing on the nodes hosting the web service is a greedy approach whose effectiveness depends on the available compute resources at the node. It does not account for dynamic compute conditions on the web server nodes. Furthermore, this work is focused on just the crawling operation to be executed on the site

node, unlike other applications which require handling a richer set of application operators (e.g., grouping and reduction in server monitoring analytics and feature extraction in video analytics) whose resource costs can change depending on execution context and input data.

Applications which are not web services-based, also face similar challenges with regard to efficient application partitioning. Analytics applications rely on cost-model based approaches to solve an expensive optimization problem [12, 16]. Solving an expensive optimization does not scale for our systems which need to adapt partitioning strategies frequently due to changing resource conditions, in the order of minutes or tens of seconds as we will see in subsequent chapters. Other works tradeoff accuracy for improving performance [10]. Trading off accuracy may not be desirable in our target scenarios such as video surveillance systems, or server monitoring pipelines which detect anomalous behavior to generate alerts. Loss of accuracy can result in high false negative rates and missed alerts. Furthermore, approximation techniques can be complementary to the techniques explored in this thesis, in scenarios where loss in output accuracy can be tolerated. Approximation operations can be integrated into the analytics pipeline and compute resource management can be done for these operations, just like the remaining analytics operators. Server monitoring pipelines have also explored other techniques which perform static partitioning [17] and cannot adapt to changing resource conditions.

Once a new partitioning configuration is computed, the components need to implement the new partitioning plan by migrating operators from the current to target node configurations at runtime. Programming and runtime frameworks based on actors have been explored for this purpose [18, 19]. They enable developers to offload application components by leveraging actors' code migration capabilities based on location transparency combined with message passing interface. Since messages are processed sequentially with no shared state across actors, the end of processing a message in the invoked message handler provides a natural boundary to continue processing on a different target node. The runtime provides a global naming system for actors along with remote communication and coordination for asynchronous actors to interact with each other via continuations, which makes efficient actor migration feasible. Several other solutions also exist in the domains of mobile code offloading [16, 20, 21, 22] and stream processing [23]. They provide the mechanisms to consider migration overheads incurred at runtime, when making partitioning decisions. They also provide system-level capabilities to package operator state for migration, disconnect input and output communication links for migrating operators, stop operator instances on current nodes, instantiate new operator instances on target nodes, reconnect communication links to resume processing. In this thesis, our focus is not on the runtime code migration capabilities. Addressing code migration related challenges is left for future work. Instead, we exploit

several optimizations which enable the distributed runtime to efficiently make partitioning decisions such that they can scale to large number of data sources. Code migration engines described earlier can then leverage our scalable partitioning decision-making engine, to execute the new partitioning plan.

1.1 SCOPE

Data streams are a continuous and unbounded sequence of data items being generated from data sources. There is an explosion in data streams generated by various data sources, both within and outside datacenters managed by cloud platforms. E.g., log streams generated by monitoring pipelines to monitor the health of servers and hosted services within the datacenter, high bandwidth video streams generated by smartphones and surveillance cameras sent to the cloud for processing. There is a need to process the data in a timely manner to extract actionable insights in real-time. Stream processing applications are becoming increasingly distributed and run across large fleets of server nodes hosted in datacenters, harnessing their compute resources to support large scale data processing needs. Recently, near-data processing is becoming a popular mechanism to efficiently analyze enormous amounts of data streams, owing to the emergence of Internet-of-things (IoT) devices and edge-cloud systems. Near-data processing enables a new category of streaming applications which require time-critical monitoring and control, such as server anomaly detection and mitigation in large scale datacenters [24], and processing of high bandwidth video data under constrained network conditions [25] to generate alerts in video surveillance systems or interact with users on smartphones.

The space of distributed applications is large and diverse, so we focus our investigations on the newly emerging paradigm of near-data stream processing applications. We note that such applications share the concerns outlined earlier in this section. Since streams arrive continuously, we need to keep up with the data in real-time, by identifying compute resource and power efficient application operators to be executed on near-data resource nodes, instead of sending all the data to the cloud. Near-data resources can consist of edge or data source nodes with compute capabilities. Note that we take a broad view of what constitutes edge and cloud nodes in our resource topology. Cloud nodes are one or more nodes which perform the final data aggregation and generate output results for the stream processing application. Edge nodes are closer to the data source (in terms of network hops) compared to cloud nodes, and are dedicated compute resources for the stream processing application to execute. They provide higher network bandwidths and lower latencies for data sources to send data for remote computation, compared to the cloud. They do intermediate processing and reduce

the data that needs to be processed at the cloud. Thus, edge and cloud nodes are defined for hierarchical resource topologies which occur within datacenters or outside of them (e.g., in a WAN topology). We highlight that the usage of the terms edge and cloud for resource nodes both within and outside datacenters has also been adopted by other works in the literature [24].

We study two types of applications in particular: video analytics pipelines and analytics queries (e.g., queries involving grouping, aggregation and join operators). Both these categories of applications have stringent quality-of-service (QoS) requirements. A video processing pipeline for a large scale surveillance system needs to perform complex analytics operations such as face classification on incoming streams to quickly alert in case of anomalous human behavior. Mobile video applications need to process incoming video streams at user-interactive speeds. Similarly, a monitoring query in a datacenter needs to aggregate data from large number of server nodes to generate alerts within a minute or less, if anomalous/faulty behavior is detected during the operation of hosted web services and machine learning pipelines.

Deep learning operators are a critical component of many stream processing applications to perform complex classification, recognition and anomaly detection tasks [26, 27, 27, 28, 29]. Deep neural networks (DNN) are trained offline with pre-collected data in a datacenter, then deployed (either on the data source or edge server) to perform inference on the incoming data streams. DNNs present a similar tradeoff between compute and network transfer costs when deployed in streaming systems. DNNs are compute-intensive so they may not run entirely on the data source or edge server with limited compute resources. Sending the video data to the cloud for performing inference may incur high network transfer costs. Instead, the DNN layers can be partitioned, so that the streams are partially processed on near-data resource nodes, subject to a limit on compute capacity. The intermediate values can then be transmitted to a cloud node to execute further DNN layers for inference. However, the size of intermediate data in DNNs is far higher than the input size (by up to 4500x [30]). Thus, we consider the DNN to be a black box and modeled as a single operator in the stream processing pipeline. A placement decision executes the DNN entirely on one of data source, edge or cloud nodes subject to compute resource availability. Partitioning the layers within a DNN to distribute their execution across multiple resource nodes has been investigated in literature [30, 31, 32] and we leave the integration of such techniques into the placement framework proposed in this thesis, for future work.

We look at designing fast and scalable application partitioning techniques so that applications can process data streams at large scale in our target systems. Doing so can enable the distributed runtime to fluidly migrate streaming operators, so that applications can effectively

harness available compute resources, and meet their performance targets. In particular, we design algorithms and build systems to address the following concerns related to streaming application partitioning:

- **Scale:** The number of data sources generating data can be in the order of hundreds of thousands or more (e.g. smartphone apps processing video data and communicating with a cloud service or server nodes in a datacenter generating monitoring data for analytics pipelines). Edge servers can be in the order of tens for each application. So, the distributed runtime needs to be able to partition application operators across a large number of resources nodes.
- **Hardware heterogeneity and complexity of data sources:** Data sources such as smartphones have compute capabilities to perform on-device computation, but with varying compute, network and power resource characteristics across devices. The available resources in a network topology may also change over time, due to varying network conditions and device failures. Moreover, device-specific optimizations make it challenging to build generic analytical models for resource usage behavior, such as power consumption of an application across devices. E.g. GPUs, TPUs and multi-core architectures [33] in modern smartphones along with strategies such as DVFS [34], can cause performance and power costs for the same application operators and input data to vary across devices.
- **Multi-tenancy:** Data sources can be hosting multiple applications to improve resource utilization of the available hardware resources. E.g., cloud datacenters use virtualized resources (e.g., containers) to enable resource isolation for applications colocated on the same physical node, and the virtualized resources can be scaled based on application compute needs. Sharing of resources also occurs in smartphones where users may concurrently execute multiple applications on the device. Thus, resources available per-application change over time, and it depends on the compute resource requirements of other co-located applications. Distributed runtime needs to handle application partitioning requirements under dynamic compute conditions.
- **Application complexity:** Modern stream processing applications consist of operators whose compute usage characteristics are highly dependent on the input data (Chapters 7 and 8). Thus, online profiling is needed to understand operator resource usage behavior when the input distribution changes at runtime. However, profiling can be expensive and cannot be done accurately under constrained compute conditions on devices such as smartphones, smart cameras, etc. Moreover, in Chapter 6, we observe the need

to handle the partitioning of application graphs with large number of operators as we increase the number of data sources. Large graphs result from scaling/replicating the operator instances depending on the current input load. Doing so enables the application to better utilize available compute resources and parallelize data processing. Thus, scaling the partitioning strategy to large application graphs is an important problem to address for stream processors.

- **Diverse performance goal requirements:** Depending on the target system under consideration, there may be diverse requirements for optimizing system properties such as request throughput, latency, energy cost, etc. Battery-driven devices such as smartphones need to optimize for energy use when making application partitioning decisions. Compute nodes may also operate under different resource constraints. Datacenter server nodes running analytics queries for monitoring pipelines are compute-constrained due to resource allocation being prioritized for foreground services hosted by customers of a cloud platform. On the other hand, geo-distributed analytics pipelines [35] (e.g., log analytics for content delivery networks or CDNs) are constrained by network transfer costs across wide-area network (WAN). Distributed runtime needs to handle diverse requirements of applications depending on the target resource topology on which they are running.

1.2 THESIS CONTRIBUTIONS

The goal of an application partitioning solution is to facilitate the timely processing of incoming data streams under limited compute, network and energy resources, i.e., it finds a placement of the streaming application operators which maximizes application performance within the available resource budget. Stream processors can scale better to handle massive data streams by processing data closer to the data source, thereby reducing network transfer costs and alleviating resource bottlenecks in the processing pipeline. Near-data compute resources can be modeled as data source (e.g., smartphones, datacenter server nodes generating monitoring data for analytics pipelines), edge (e.g., micro-datacenters near cellular towers, aggregation nodes for analytics close to the datacenter servers) and cloud (e.g., servers in cloud platforms such as Amazon AWS [36]) nodes arranged in a hierarchical network topology. The amount of compute resources increases as we go farther from the data source, but sending large data volumes to remote compute nodes result in network transfer bottlenecks.

In this thesis, we focus on two popular types of stream processing applications: video processing and query analytics applications. We highlight their common characteristics

that make them suitable for near-data processing (see Section 2.1). To make placement decisions for operators across a relatively small number (in the order of tens) of compute nodes between the edge and cloud, a centralized placement solution with a global view of the application compute resource needs and available resources can be used. Such a solution can facilitate minimization of end-to-end application processing latency within the available resource budget, to efficiently process massive data volumes. However, if data sources can also perform computation and need to be considered by the placement solution, then we prove that a centralized technique does not scale for the large number (in the order of thousands or more) of nodes available in the data source layer (e.g., hundreds of thousands of smartphones generating video data, server nodes generating monitoring data in large scale datacenters). This is because compute nodes in the edge and cloud layers are finite and shared by data sources. Thus, application partitioning between the data source and remote edge and cloud nodes needs to be considered jointly across all the source nodes, making the search space exponential in the number of data sources. Instead, a decentralized and greedy heuristic for local placement decisions on each node can improve application performance when network transfer costs are high. This is accomplished by improving the utilization of available resources on each node to execute application operators, while reducing network transfer costs. Partitioning needs to be done under dynamic resource conditions on data source nodes (e.g., changes in input data affect compute costs of application operators). Runtime profiling for the operator compute costs is expensive and may not be accurate under constrained compute conditions on data sources. Thus, placement strategies which utilize execution cost model-based approaches and rely on accurate profiling of operator compute costs, are not effective on data source nodes. Moreover, data source nodes have limited energy resources. With the advent of smartphones with multi-core architectures and accelerators such as GPUs, TPUs, etc., there is high variability in hardware power characteristics across data source nodes. Developing accurate analytical models to estimate application energy costs for each new architecture is becoming increasingly complex and infeasible. Thus, model-agnostic techniques which empirically estimate operator compute and energy costs by executing the application and collecting on-device measurements, can play a critical role in enabling fast and accurate estimation of runtime operator resource usage costs on data source nodes. These cost estimates can be used to make effective placement decisions which improve the performance and energy usage characteristics of streaming applications.

The critical insights provided from the work in this thesis are summarized as follows:

- A hybrid placement approach which leverages- (1) a global view of available resource nodes to reduce the end-to-end application processing time between edge and cloud

layers and, (2) a local view of available resources on each data source node, to make local placement decisions independently of other nodes, can provide an efficient and scalable placement strategy for hierarchically arranged resource topologies such as IoT and edge/cloud systems.

- Placement solutions on data source nodes can significantly benefit from resource cost model-agnostic techniques to obtain operator execution costs and make placement decisions. Such techniques use direct empirical measurements during on-device execution and, unlike resource cost model-based approaches, do not require expensive runtime profiling or additional manual work to build custom analytical models for supporting different hardware/software components.

In this thesis, we first define models and assumptions for our target systems (see Chapter 4) and then describe algorithmic contributions based on our models and assumptions in Chapter 5. We then discuss three case studies in which we build proof-of-concept systems to implement our proposed placement algorithms. The case studies investigate the effectiveness of our algorithms across diverse real-world applications. The first case study is on placement of video application operators across edge and cloud nodes. The second case study is on placement of analytics query operators across data source, edge and cloud nodes. And the third case study focuses on energy profiling of video applications running on data sources such as mobile devices. The case studies highlight system-level contributions made by this thesis in the studied domains.

1.3 THESIS ORGANIZATION

In the next chapter, we go into details of the opportunities available for exploiting near-data processing in stream processing applications, with the advent of edge computing and smart data sources with compute capabilities. We then discuss the models and assumptions used in this thesis, followed by formal definitions of different placement related problems that this thesis addresses. In Chapter 5, we discuss the challenges with addressing the placement problems and propose practical algorithms to make placement decisions effectively and efficiently. In the next 3 chapters, we discuss 3 different case studies addressing placement related challenges in real world systems. Each chapter implements the proposed algorithms in a proof-of-concept system and discusses system design contributions to integrate placement solutions in distributed runtimes for stream processing systems. Finally, we conclude the thesis and discuss potential directions for future work.

CHAPTER 2: NEAR-DATA STREAM PROCESSING: OPPORTUNITIES & CHALLENGES

Depending on the type of data source, data is generated in different formats ranging from text or numeric data as we observe in server monitoring systems [6], to video and images generated by smartphones and cameras [37]. Large-scale datacenters report generating up to 10s of PBs of monitoring data per day from hundreds of thousands of server nodes [6]. Autonomous vehicles are projected to generate 1.4 TB/hr of video data [38] and 770 million surveillance cameras [5] are deployed worldwide generating continuous video streams for analysis. It is also estimated that more than 7 billion smartphones globally will be generating 345 PB/hr of data by the end of 2027 [39, 40]. Apart from the high data rates, the rates observed per data source node can also vary over time. If an anomalous event occurs on the datacenter server node (e.g., server node network issues cause client requests to take too long), it increases the volume of generated error logs. Similarly, when the number of cameras generating video streams changes (e.g., when tracking a moving target in surveillance systems), it also changes the aggregate frames which are generated and need to be processed [5, 41, 42] by the analytics applications.

There is an increasing need to harness the generated data streams and perform complex analytics operations to derive valuable insights about the target systems being monitored in real-time. Emerging applications analyze data streams to quickly identify anomalies and generate alerts across large scale datacenters with hundreds of thousands of nodes. Anomalies can range from high CPU utilization or high network latencies [43] experienced by server nodes to bugs in the application code causing hosted services to not meet client request service-level agreements (SLAs) [6]. Moreover, video analytics application perform real-time face detection and classification tasks on camera frames which require extracting the facial regions of interest and performing resource-intensive classification operations using deep neural networks [44]. Such applications are resource-intensive, and require large amounts of compute and memory resources to meet the application SLA. Delays in video processing can affect user-perceived latencies in interactive mobile applications and cause delayed tracking of persons of interest in video surveillance systems. Changes in the incoming data further affect their resource requirements so resource management strategies need to adapt to such changes.

Despite the scale at which data streams are being generated and the complexity of emerging applications processing such data and their resource demands, hardware advancements are not able to keep up with the increasing resource needs. Figure 2.1 shows traditional system architectures in which generated streams are sent from data sources to a centralized cloud

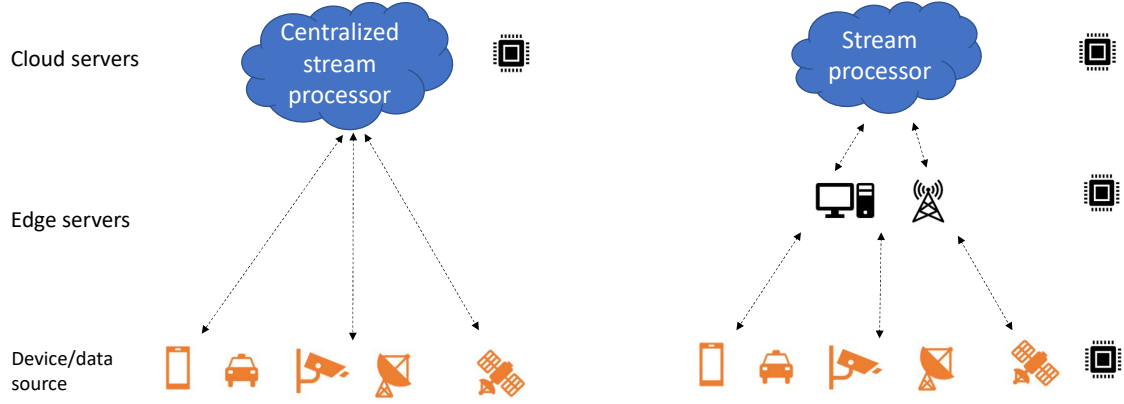


Figure 2.1: Figure on the left shows traditional system architecture with a centralized stream processor running in the cloud. On the right, near-data stream processing system is shown where computation is done across distributed resource nodes between data source and the cloud.

which is running in a remote cluster hosting a stream processing engine such as Apache Flink [45] or Spark Streaming [46]. Such an architecture cannot scale, as we increase the number data sources. This is due to the high network transfer costs in transmitting data to the remote servers. Moreover, the compute costs for processing the data to meet the application SLA is prohibitive. E.g., Chapter 7 shows that a single stream generated to monitor network latencies between server nodes in a large scale datacenter can generate as much as 512.6 Gbps of data to be processed by a centralized stream processor. Other works describe applications running on server nodes generating logs that can go up to 10s PB/day. Video streams are also relatively high bandwidth data generated by mobile devices and traffic camera, and need to be transported over wide-area network (WAN) to the cloud. Such data traffic incurs substantial network bandwidth costs. Furthermore, datacenter operators provision clusters spanning hundreds to thousands of machines to process the incoming monitoring data streams [6, 10, 47, 48]. This is because the per-core compute capacity in terms of the data rates that they can process is 2-3 orders of magnitude less than the input rates [12].

To alleviate the above-mentioned compute and communication bottlenecks, Figure 2.1 also shows on the right, recent advancements in hardware infrastructure, wherein resources on the data source or edge nodes [37] closer to the data source (i.e., near-data nodes), can be utilized for computation. Even within datacenters, edge servers are deployed closer to the data source in monitoring pipelines, to perform intermediate processing and alleviate resource bottlenecks at the stream processor running in a central cluster. Applications are *partitioned* to split the

execution of operators between cloud and near-data resources, such that the application QoS requirements are met. Near-data resource nodes are heterogeneous in the amount of compute, memory as well as energy (in case of battery powered devices) available for processing the incoming data streams. Thus, decisions about which operators can be executed on resource nodes closer to the data source without violating available compute budget, need to be done on a per-node basis. Compute resources are also limited on each node. E.g. a server node may allocate a small fraction of compute (e.g., 1-2 CPU cores and 1 GB RAM) for near-data processing in server monitoring pipelines. Edge servers may have relatively higher compute capabilities (e.g., 4 cores and 32 GB RAM [49]). But they are still a small fraction compared to remote commodity server nodes with large compute resources (e.g., each node with 64 cores and 256-512 GB RAM or more), dedicated for analytics processing (referred to as cloud nodes). Due to the reduced compute resources available closer to the data source, near-data processing nodes need to coordinate with remote cloud to meet the application quality-of-service (QoS) requirements. Furthermore, not all operators can be executed on near-data resource nodes. Certain operators like global aggregation requires processing by the cloud which receives data from all data sources.

Resource conditions are also constantly changing in near-data processing systems. Within datacenter monitoring pipelines, data source nodes can be shared by other foreground applications such as web services. Such services are co-located with analytics queries to process monitoring data generated on the server nodes, and may have different compute needs based on incoming traffic requests. On smartphones, users may have multiple applications co-located with a video stream analytics pipeline such as face detector on the device, and the resource needs of such applications can depend on user activity. Thus, resources available for analytics operators change over time on the data source. Furthermore, input data rates and their distribution change over time, affecting the application resource needs. Examples of such input changes include (1) data sizes for error log streams from datacenter server nodes change within minutes when anomalous/error behavior occurs in the service code or hardware infrastructure, (2) varying number of mobile devices within a transport vehicle such as a bus or train, due to passengers boarding and alighting, result in varying sizes of data streams sent to an on-board edge server for near-data processing within the vehicle [42] and, (3) number of paramedics with wearable cameras using edge servers within a nearby ambulance vehicle for video processing in telehealth scenarios [41]). Changes in input data affects the application compute needs. A monitoring application summarizing error logs for generating alerts, experiences a surge in data volume when there are anomalies in the monitored system. The need to process more data without violating target SLA means the application's compute demands increase. Similarly, video feeds from cameras for traffic monitoring may contain less

temporal variation if the traffic is slow moving and thus the streams can be processed at a lower frame rate, thereby reducing the resource footprint of application operators.

Changing resource conditions on near-data nodes means that we need to update our placement decisions at runtime. Otherwise, we may under/over-subscribe the compute resources, causing them to either be under-utilized or the application QoS to be violated. In this thesis, we argue for the need for highly efficient runtime partitioning techniques. Existing systems do not address several challenges associated with partitioning application operations between near-data and cloud nodes. Such challenges include scaling the placement mechanism to efficiently handle large number of application operators and data source nodes, efficiently utilizing the limited resources on data source nodes with the goal to minimize resource bottlenecks at the stream processor and effective techniques to profile at runtime, the compute and energy costs of application operators with limited resources on data source nodes, under different partitioning strategies.

We address the above challenges by building a framework to make scalable placement decisions for near-data processing, within the stream processing engine which constitutes the distributed runtime (shown in Figure 1.2) for streaming analytics. We develop algorithms that identify effective placement decisions for distributing computation across data source, edge and cloud resource nodes. We design systems for different real-world case studies, that implement the proposed algorithms in a scalable manner with low overhead on the resource nodes. We then build proof-of-concept systems to validate the effectiveness of our proposed algorithms and system design decisions. The **first solution** focuses on optimizing the application’s end-to-end latency, which is designed for making partitioning decisions between edges and cloud nodes. We model the application as a directed acyclic graph (DAG) of operators, with each node representing an operation or task within the application. We then find a mapping for the operators to resource nodes on which they need to execute. We increase the number of application operators to make placement decisions, with the number of data sources (i.e., the original application graph is replicated with a separate instance for each data source, followed by a final merging operator to generate the final output). By doing so, we can scale to handle massive data volumes by exploiting opportunities for parallelizing the processing of data streams from data sources. We then use dynamic programming techniques to make placement decisions for large application graphs. The **second solution** investigates the effectiveness of a fine-grained partitioning mechanism called data-level partitioning, under limited compute conditions on the data source. Current systems use operator-level partitioning: if a data source does not have sufficient resources to execute an operator on all incoming data items in the stream, the operator is executed only on the remote compute node. Instead, we perform data-level partitioning, i.e., we allow an operator to be executed both on the data

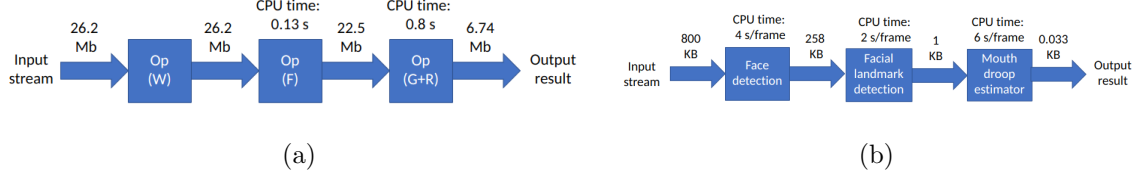


Figure 2.2: (a) Real-world analytics query consisting of Window (W), Filter (F) and Grouping+Reduction (G+R) operators, showing intermediate data sizes and compute costs for operators processing each epoch (defined by a time interval). Data rates are scaled by 10x to analyze the system under stress and for ease of experimentation. (b) Real-world video processing pipeline, showing intermediate data sizes and compute costs for operators processing each video frame. Data is shown for a single 2.4 GHz CPU core.

source and remote nodes. Available compute resources also change on data source nodes. An algorithm to quickly (in the order of seconds) adapt the partitioning plan is designed to handle fast changing resource conditions on data sources. This is done by computing an initial partitioning plan using a cost model-based heuristic, and fine-tuning the obtained plan by combining it with an iterative search strategy which is cost model-agnostic. Novel abstractions are introduced in the stream processing pipeline to implement and evaluate the proposed algorithm as we scale the system to a large number of data source nodes. The **third solution** enables extending our placement framework to support energy-aware placement decisions. The solution focuses on the problem of estimating the energy cost of individual operators to facilitate such energy management. We propose an empirical method for estimating power consumption using device-level energy draw information in conjunction with an application’s execution traces. The method is adaptive, captures the effect of input data, and dynamically changing energy costs based on the operator logic for a given input. The tool built using this methodology can be easily ported to devices with diverse types of compute resources (e.g., multi-core CPUs, GPUs or TPUs on smartphones).

2.1 REAL-WORLD MOTIVATING SCENARIOS

We focus on streaming applications with certain characteristics that make them particularly amenable to near-data processing. Below are two example scenarios which have very different characteristics, and guide the design of approaches proposed in this thesis:

- *Scenario 1:* Consider a real-world analytics query as represented in Figure 2.2(a). It shows the individual operator compute costs and the data reduction at each. It is executed in a server monitoring pipeline in datacenters and is used to summarize server-to-server network latency probing data collected in the Pingmesh [43] system.

Specifically, network engineers deploy Pingmesh agents on datacenter nodes to collect latency data between node pairs. An example scenario where this is useful is a web search team which uses Pingmesh to monitor network health of their latency-sensitive service and generate an alert if more than a predefined proportion of hosting nodes (e.g., 1%) have probe latencies exceeding a threshold such as 5 ms. Our query filters the records of interest (e.g., probing error code is 0) and then aggregates the round trip time (rtt) to generate statistical summaries of rtt for each source and destination node pair. Such aggregation near the data source can lead to reduction in data size before transmitting it over the network.

- *Scenario 2:* Figure 2.2(b) shows an example of a telehealth application [41] in which paramedics wear body cameras to analyze video feed of patients to detect strokes. Analysis is done using an edge server in the ambulance and a cloud node in the hospital. The application consists of three operators, and we use video streams with epochs defined to consist of 10 frames.

Target applications. In both of the example scenarios above, there are opportunities for data reduction after partially executing the operators in the data stream. If filtering in Figure 2.2(a) query is executed on the data source, we can possibly drop a significant fraction of the records and not need for them to be transmitted over the network. The video pipeline in Figure 2.2(b) can reduce the data size by 36% by executing the face detection operator on the edge node. In both cases, more the operators we execute on the near-data resource node, greater the reduction in intermediate data size. This insight forms the basis for designing our algorithms for doing near-data processing for streaming applications and efficiently identifying operator workloads that can be executed on compute resources close to the data source. Depending on the input data size and characteristics, data reduction at the end of processing each operator can vary and the resource nodes available to run the operators can affect their compute costs. Considering that we make partitioning decisions at the granularity of an operator, when the input data is such that data reduction occurs going from one operator to the next, it gives us an opportunity to partition the application at that operator and obtain savings in network transfer costs. However, the potential savings are constrained by the available compute resources to execute the operators mapped to the data source or edge. E.g., the filtering operation takes 0.13 seconds and grouping+reduction (i.e., G+R) operator takes 0.8 seconds to process an entire epoch on the data source. If the incoming data rate is such that a new epoch is generated every 0.5 seconds on the data source node, then we are over-subscribing the compute and cannot execute both F and G+R operators at the data source. Similarly, if a new epoch consisting of 10 frames is

generated every 10 seconds, then executing face detection operator on the edge node would not be possible as we take 40 seconds to fully process an epoch of 10 frames. There are several differences between video processing and query analytics workloads. Video processing pipelines operate on video frames while analytics workloads operate on text or numeric data such as performance counters or service logs collected from servers in datacenters or IoT devices [50]. The type of data has implications on network/compute resource requirements for running the applications, nature of operations/application logic defined to process the streams, and the characteristics of resource topologies that generate the two types of streams.

- Bandwidth requirements at the data source: High bandwidth video data may require anywhere between 5-25 Mbps for transmitting from data source to the cloud, depending on the video resolution and frame rate [25]. However, text or numeric data is much lower bandwidth, typically in the range of 10s-100s of Kbps [6], with server logs possibly exceeding 1 Mbps during peak traffic (e.g., server experiencing failed requests) [43]. Partitioning strategies need to consider the differences in bandwidth requirements per data source, when making operator placement decisions.
- Application processing logic: video processing pipelines typically contain operations for filtering of frames, reducing the dimensionality of the input data to extract features and perform tasks such as classification or object recognition in the incoming data frames. However, analytics queries typically perform filtering and aggregation operations such as grouping, reduction and joins. The diverse set of operators, their data reduction capabilities and resource usage characteristics need to be considered when making placement decisions.
- Number of data sources: The number of video data sources sending data to remote servers in an edge/cloud system are typically smaller in number (in the order of 10s to 100s [51]). However, 1000s to hundreds of thousands of data sources can send text/numeric data to remote servers between edge and cloud (e.g., large scale datacenter monitoring pipelines). Thus, partitioning strategies need to operate at different scales, in terms of the number of data sources generating data streams and efficiently adapt their operator placement decisions.

We model our applications to capture the diverse types of application operators and resource nodes in our target application scenarios, and design efficient placement algorithms for the modeled applications in this thesis.

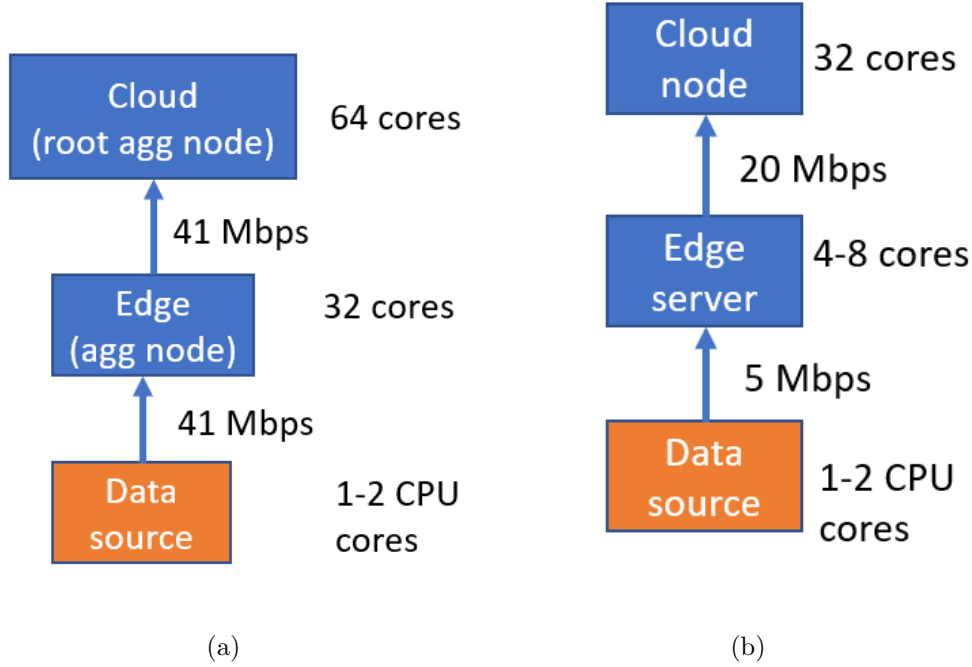


Figure 2.3: Example resource topologies depending on where the data source is located, along with possible compute and network resources available at each level in the resource hierarchy. Available network bandwidth is aggregate, across all applications running in this setup. (a) Data source within datacenter, and (b) Data source outside datacenter.

2.2 RESOURCE TOPOLOGIES

We now look at types of resource topologies that support near-data stream processing applications. We consider resource nodes both within and outside datacenters. As described in Section 1.1, an edge server is closer to the data source (in terms of network hops), compared to the cloud nodes. Each edge server has less compute resources compared to its cloud counterpart, however they can have similar types of resources to run applications, i.e., they may include dedicated CPU or GPU cores and other accelerators such as TPUs [52] for running machine learning operators within the application.

Resource topologies as described in Figure 2.1 can be of two types in the real world. The type of topology depends on whether the data sources exist within or outside datacenters. In case they are within datacenters, the network transfer characteristics are quite different from those in which they exist outside. Figure 2.3 shows an example hierarchical resource topology along with available compute and network resources. Within datacenters, Figure 2.3(a) shows data sources can be the platform servers themselves (generating monitoring data) or other hardware sensors which may have compute available for on-device processing. They connect

to dedicated server nodes used for analytics purposes. The network links typically provide 10 Gbps or more of bandwidth, which may be shared by up to 250 data sources or more, sending data to a single edge node. This gives us 41 Mbps network bandwidth from data source to edge node. Furthermore, a large datacenter can consist of up to 250,000 server nodes generating monitoring data [6], so we will have 1000 nodes in the edge layer (given the 1:250 ratio for edge:data source nodes). Assuming a 40 Gbps link between edge and cloud nodes, we obtain 41 Mbps available for each edge to cloud link. Note that the network link and compute resources may also be shared across a large number of queries running concurrently on each data source. E.g., up to hundreds or even thousands of monitoring queries can be executing concurrently [12]. The compute on data sources is also be shared by foreground services hosted on the server nodes, thereby leaving very limited resources for the monitoring query (1-2 cores). However, edge and cloud nodes are fully dedicated to the monitoring pipeline and have more compute cores for stream processing.

An example hierarchical resource topology for data sources outside the datacenter is shown in Figure 2.3(b). Data sources can consist of mobile devices and cameras sending video data to edge nodes over wireless networks (Wi-Fi, LTE, etc.), wired LAN or other radio-access networks [25]. The communication between edge and cloud nodes occurs over wide-area networks (WANs) and wired network infrastructure. The typical network bandwidths supported by WANs can be between 15-20 Mbps uplink bandwidth per-edge [53, 54]. The links between data sources and edge servers support anywhere between 2 and 5 Mbps for cellular networks [55]. In terms of network latencies, the additional overhead incurred for data sources to reach the cloud compared to edge can be between 10-100 ms [51]. This is significant as many applications target achieving end-to-end processing latencies as low as tens of milliseconds (e.g., user-interactive mobile applications). Of course, any data transmission latencies for processing within the data source node is negligible as the data originates on the node.

With the introduction of 5G technology, we note that there may be up to an order of improved speeds compared to existing technologies [56] between data source and edge nodes. However, this does not alleviate the potential network bottlenecks between edge and cloud nodes, and thus edge/cloud partitioning strategies can benefit applications. Furthermore, given the increasing need for low latency streaming application requirements in the order of milliseconds, there is a need for being able to parallelize application processing by leveraging the aggregate compute resources available across all the data sources. As an example, consider the cloud hosting a video processing pipeline, processing data from 10,000 mobile devices (i.e., user-base for the video application) sending their video feed to the cloud. Even one core assigned on each device for partial processing can result in 10,000 cores worth of compute that

can be harnessed to meet the application processing needs and reduce the cost of provisioning cloud resources for the developers of the mobile application.

2.3 SUMMARY

This section summarizes our observations of real-world application characteristics and underlying resource topologies, which we use to guide the design of our systems:

- Analytics queries running in resource topologies within datacenters can benefit from near-data processing strategies (e.g., in server monitoring pipelines). As shown in Figure 2.2(a), if a single query utilizes 26.2 Mbps to upload input data within one-second epochs from the data source, multiple queries sharing the network bandwidth can quickly saturate the available bandwidth. This limits the number of data sources that can be handled by each edge node, thereby limiting the ability of such systems to scale to large number of data sources. The entire query, which takes up to 0.93 seconds to fully execute a data item, may not be executed on the data source either, since a single query can exhaust most of the available compute time on data source, leaving very little compute for the remaining monitoring queries executing concurrently.
- It becomes even more critical for analytics queries to leverage near-data processing capabilities in resource topologies with data sources outside datacenters (e.g., smartphones and IoT device monitoring). This is because of the very limited bandwidth between data source and edge node, which can quickly get saturated due to data sources such as mobile devices being shared by multiple applications. Furthermore, the large number of data sources in such settings can saturate the network bandwidth at the edge node. The entire query cannot be executed on the data source either, since compute can be saturated very quickly if a single core is available and the incoming data coming in at 1 epoch per second, requires up to 0.93 seconds to sequentially process the query.
- For the application in Figure 2.2(b), let's consider an example with an epoch of 10 frames being generated every 5 seconds. Given the frame size of 800 KB, we observe data rates of 1.56 MBps or 12.5 MBps at the source. In resource topologies with data sources outside datacenters (e.g., cameras, mobile devices), we may exhaust the available network bandwidth from the data source (5 Mbps). So, unless the video is generated at a lower resolution or frame rate to reduce the data rates, it is infeasible to support the application in our resource topology by just uploading the raw data to remote nodes for processing. Furthermore, with compute resources available on the

data source (e.g., mobile device), the expensive video processing pipeline which requires a lot of compute resources (it takes a total of 12 seconds to execute all operators on a video frame when executed in sequence) may not be able to process the incoming data if 2 frames are generated every second. Given that we cannot upload all the raw data to the remote nodes for processing in this case, nor can we run the entire application on the data source, it becomes critical to explore near-data processing strategies which partition the application between data source and remote nodes.

Scaling the infrastructure to meet the resource needs of large number of data sources is critical for existing stream processing engines. In scenario 1 described at the beginning of Section 2.1, enabling datacenter monitoring pipelines to scale better to hundreds of thousands of data source nodes, will significantly reduce the monetary costs for datacenter operators. Furthermore, it also enables the streaming infrastructure to meet the query performance SLAs. In scenario 2, there can be more paramedics with body cameras sending data to the same edge node or multiple different ambulances with their own edge servers using the hospital server as the cloud node (in case of large-scale disasters such as earthquake) to meet the application processing needs. Improving the utilization of edge and cloud resources helps effectively parallelize the processing of incoming data streams and improve application performance. This not only enables paramedics to quickly detect and react to the critical needs of patients but also reduces the chances of the pipeline getting saturated with more data sources utilizing the edge/cloud resources.

Our observations indicate that the combination of large number of data sources and multiple applications executing concurrently on each data source, makes it challenging to meet the resource needs of modern streaming applications. Near-data processing is a critical mechanism to enable existing resource topologies to scale to large number of data sources and meet the near-real time performance needs of modern applications. In subsequent chapters, we define the system models and assumptions for streaming applications and the underlying resource topologies. We establish the hardness of the partitioning problems based on the system models and assumptions. We then design algorithms which enable us to quickly and effectively identify partitioning plans for the considered streaming applications executing in dynamic resource conditions. To validate the effectiveness of our proposed algorithms, we present three different case studies which consider diverse real-world motivating scenarios for scalable placement mechanisms across the two types of target applications, i.e., analytics queries and video processing. The case studies highlight system-level contributions made in their respective target domains.

CHAPTER 3: RELATED WORK

Application performance (e.g., end to end latency, throughput) and energy consumption are critical considerations when assigning resources to operators in streaming systems ranging from datacenter monitoring to camera and mobile video applications. Several works in literature have addressed challenges in monitoring and placement of application operators, with the goal of optimizing end-to-end latency and energy consumption in battery powered devices. In this section, we begin by describing lower-level frameworks that host streaming applications in the data source-edge-cloud setting and enable designing resource management policies on top of them. We then provide an overview of works on the following topics and compare them with our contributions in this thesis: (1) placement for optimizing application performance which explores the compute-communication tradeoff involved with different placement decisions and its impact on processing latency and throughput, (2) energy estimation for mobile application operators, to enable the design of energy-aware placement strategies. For each section, we discuss different approaches and assumptions for systems that utilize resources in the data source-edge-cloud hierarchy.

3.1 LOW-LEVEL ORCHESTRATION FRAMEWORKS

Traditional architectures for stream processing applications have focused on cloud-only approach to processing data where the stream processor is entirely running in a dedicated and centralized cluster and hosted within the datacenter. It requires provisioning compute resources and managing them within cluster to process incoming streams. In the last decade, several stream processing systems have been developed to support large-scale stream analytics on the cloud platform. These include Apache Spark Streaming [46], Apache Storm [57] and Apache Flink [45], Apache Nifi [58]. However, given the tremendous amount of data transfer costs associated with transporting high bandwidth data such as video streams or metrics/logs collected from a large number of data source nodes, the network latency and bandwidth requirements at the stream processor become significantly high. Compute resources provisioned within the centralized cluster also need to scale with the incoming data size, as data needs to be processed in a timely manner to meet application SLA and it can only be processed after arrival at the cloud. Thus, large-scale data processing can result in network and compute bottlenecks at the stream processor.

To reduce network transfer and compute cost of cloud-centric analytics, edge [59] and fog [60] resources have been considered, which are closer to the data source (in terms of

network hops). They serve as additional compute resources to complement the cloud-centric model and reduce data transfer over the network, thereby also reducing compute burden on the cloud. Several stream processing frameworks have been developed for the edge, such as Edgent [61], IBM’s Node-RED [62], Apache MiNiFi [63] and VMWare Liota [64]. However, these frameworks are primarily focused on the programming model and associated orchestration runtime. They do not address the challenges with performing dynamic operator placement with the goal to optimize application performance and energy costs, while scaling the placement solutions to large scale application and resource topologies. In our work, we design and validate the placement techniques to quickly adapt to dynamic resource conditions in a scalable manner, and implement our techniques over low-level frameworks such as Apache Flink, Edgent and MiNiFi.

Our placement framework for edge/cloud placement is built on top of ECHO platform [65] which uses Apache NiFi engine running on the cloud and edge nodes. For making placement decisions at the data source layer, we use the light-weight Apache MiNiFi engine. Our energy estimation techniques are targeted towards Android platform [66] and can be integrated with low-level streaming engines [20, 67].

3.2 PERFORMANCE OPTIMIZATION

We consider existing works that make placement decisions for application operators between data source, edge and cloud resources, with the goal to optimize end-to-end performance metrics such as throughput and latency. We discuss the ability of existing solutions to scale with increasing number of application operators and resource nodes.

3.2.1 Placement Between Edge and Cloud

Within datacenters. The first generation of stream processors such as Aurora [68] describe scheduling techniques which batch data items and introduce the notion of stream priorities to improve the efficiency of in-memory stream processing. They also investigate load shedding to improve query performance at runtime by sacrificing accuracy. However, Aurora does not address the challenges in making operator placement decisions across a distributed set of resource nodes. Medusa [69] utilizes an agoric model to create the right incentives for participating resource nodes to handle each other’s generated data load. Contracts are used to determine the cost of migrating load to a different node. Participants move load only if a contract exists with that node and cost of processing a task locally is larger than the payment

the participant would have to make to its partner for remote processing. When applied to our systems, we note that it addresses a different problem of avoiding over-subscription of compute resources, by using a local and greedy approach. Our goal is different. We focus on optimizing end-to-end latency at the global, application-level, and incorporate operator pipelining into our execution models to obtain more accurate solutions to the operator placement problem, and use dynamic programming techniques to scale to large number of application operators. Borealis [70] is built on top of Aurora and Medusa. It uses QoS based control mechanisms while employing hierarchical (i.e. local and global) optimizers to make placement decisions. However, it focuses on balancing the load across resource nodes, by identifying compute or network bottlenecks and then finding the best neighborhood resources to offload the processing. TelegraphCQ [71], which is built on top of Flux [72], is focused on improving load balancing using buffering and operator reordering across homogeneous compute nodes. Our optimization goal is to minimize the end-to-end latency for the application, however. Greedy load balancing strategies may alleviate one resource bottleneck but create another one in the process of resolving it. Furthermore, current works do not address the challenges with developing a placement solution based on an accurate execution model, which explicitly considers pipelining of operators when they are distributed across a heterogeneous compute and network resources, and process multiple data items within an epoch in parallel. The accurate model provides opportunities to find better quality placement solutions which reduce application processing time.

Recently, the growing interest in stream data analytics has led to extensive research into building streaming engines for single machine [73, 74, 75] and for distributed servers [46, 57, 76, 77, 78, 79], with their focus mainly on improving parallelism, responsiveness, scalability, programmability, and fault tolerance. Several adaptive scheduling algorithms to handle dynamic resource conditions have also been proposed for such engines. Storm [80] proposes a scheduling mechanism to minimize inter-node network traffic for stream processing applications. Dhalion [81] proposes rule-based scheduling policies for allocating resources to operators. Spark Streaming [82, 83] adopts threshold-based policies for resource management, while DS2 [11] allocates resources to operators based on predicting the true cost of executing each operator given the input stream. Different operator placement heuristics have been proposed to minimize the application end-to-end latency [84], and the inter-node traffic [80]. As they are designed to run on high-capacity servers with high-bandwidth network connectivity, they are best used in building centralized stream processing engines running within a cluster in the cloud. Other systems built on top of Storm [84] do not consider pipelined parallelism when processing streaming epochs, and solve an expensive NP-hard problem which would not scale for our setup with large number of application operators requiring

fast adaptation in the order of tens of seconds or less 6.2. These works make assumptions such as homogeneity in compute and network resources and resource nodes assumed to be physically co-located in the same datacenter site while connected over local area network. But such assumptions do not apply to our setup where resources are (1) heterogeneous (e.g. Raspberry Pis, smartphones and high-end servers) and, (2) geographically distributed and not fully connected resource nodes. Heterogeneity increases the search space to consider for our placement algorithm because each mapping of an operator to a resource node is unique and thus, different permutations of operator placements need to be jointly considered across all operators. Furthermore, our approach jointly addresses two problems to find better quality operator placement solutions compared to existing works: identifying the operators to be executed in parallel to process multiple data items within the same epoch, i.e., pipelined parallelism (see 5.1.2) and finding a mapping between operators and underlying resource nodes on which they execute.

Edge servers outside datacenters. Not much work customizes stream processing engines for resource-constrained edge nodes. EdgeWise [85] incorporates a scheduler for congestion control on each edge node. This congestion-aware scheduler results in improvements in throughput and latency by prioritizing bottleneck operators whose queues are experiencing backpressure. However, unlike our approach, EdgeWise uses a reactive approach which greedily schedules a congested operator at the next compute thread that becomes available. It doesn't consider the potential communication/network transfer costs for the operator to send data to its downstream operators, if the compute threads are distributed across different resource nodes. Thus, it doesn't address the compute-communication tradeoff challenges that our work does efficiently for large scale application graphs. Furthermore, the pipeline parallelism resulting from operators executing on data items in an epoch in parallel, is not modeled in such systems. To enable flexible placement, Edgent [61] provides primitives to dynamically configure the analytic flows to run on the edge, but is limited because it still relies on programmers to manually specify the flows to execute. Branch and bound technique [86] has been proposed to reduce the search complexity in solving the distributed operator placement problem. However, it doesn't explicitly model queuing delays and operator pipelining. Moreover, techniques to scale placement to large number of operators by fixing an upper time bound for running the optimization solver or sampling a subset of the resource nodes for placement, is compared against our approach in Section 6.4. Costless [87] address the operator placement problem between edge and cloud resources for video analytics applications in the serverless systems domain. Its placement mechanism between edge and cloud nodes is built on top of our proposed algorithm for edge/cloud

systems.

Certain video analytics frameworks such as Spatula [88] exploit spatio-temporal correlations for cross-camera video analytics. Given a query image of a person of interest flagged on one camera in a surveillance camera network, cross-camera identity tracking is done on edge servers, to identify subsequent frames across all cameras in which the person is likely to appear. Doing so enables us to transport video feed from a subset of the cameras for remote processing, thereby reducing compute load on the cloud. However, filtering at the edge may not necessarily lead to reduced completion time for processing an epoch of video frames, which is the optimization goal of our work on edge/cloud placement. This is because the compute cost of executing the filtering operations on edge servers can be very high. Moreover, to incorporate the results of edge-side filtering in our placement algorithm, the resulting data load on downstream operators after filtering is carried out, can be reflected in the operator profiling information (i.e., the output data size at each operator).

Mobile computation Offloading. The concept of cloudlets [89] has been proposed as an additional layer that sits between mobile devices such as smartphones and the cloud to help reduce latencies while offering intermediate-class compute resources: weaker than cloud-class servers but stronger than data source resource nodes. Previous work in mobile computation offloading addresses partitioning one application between smartphone and cloud/cloudlet from the perspective of a single user and assuming infinite compute resources at the cloud [90]. This assumption does not hold in our setup with fixed resources in the remote edge and cloud layers. The work in [91] addresses the tradeoff between computation and communication delays for dependent operations and proposes an optimal algorithm for single-user computation offloading for sequence-based operator graphs. We, however, look at more complex graphs such as DAGs. We also compare against the single-user algorithm in Section 6.4. The work in [92] addresses the NP-complete operator placement problem for DAG-based operator graphs and it proposes to solve the problem by searching the combinatorial space of solutions using genetic algorithms. We also empirically compare our approach with this approach in Section 6.4. [16] proposes an actor-based [14] code-offloading model for optimizing application performance. However, it is not evaluated at scale for large application graphs. Moreover, a fundamental limitation of above approaches is also that they do not address challenges such as: (1) Modeling the queuing delay introduced by having a stream of multiple data items within an epoch, and (2) Modeling the pipelined execution that results from multiple data items being processed concurrently by different operators.

Grid Computing. Software frameworks for managing a grid of resources such as Globus [93] provide distributed resource management/scheduling capabilities that could be

seen as being similar to our proposed approach. However, the fundamental difference is that such frameworks are designed to handle long running batch processing jobs. Such jobs rarely need to transfer data across geo-distributed grid resources. However, in our scenario we explore dataflow jobs in a streaming setting that are latency-sensitive and the tradeoff between communication and computation cost becomes critical.

3.2.2 Placement on Data Source Nodes

Several works have explored utilizing data source node resources to minimize network data transfer both in the context of mobile data sources or server nodes within a datacenter for monitoring and analytics pipelines.

Datacenter nodes: State-of-the-art approaches on alleviating network and stream processor resource capacities target analyzing network packets and are mostly based on operator partitioning. EverFlow [17] and dShark [94] apply light-weight operations (e.g., filter) on commodity network switches and computing nodes near data source, respectively. Gigascope [95] classifies operators based on domain-specific knowledge and pushes low-cost operators close to data source. Sonata [12] comes up with a more sophisticated method that capitalizes on an Integer Linear Program (ILP) solver to iteratively optimize operator partitioning so that network switches under resource constraints run a part of the query to minimize the amount of data sent out. Operator-level partitioning is coarse-grained, and cannot fully utilize the very limited compute resources on the data source node. We propose a fine-grained data-level partitioning mechanism (see Section 5.3.1). Furthermore, fast changing compute resource conditions on data source nodes requires partitioning strategies to quickly adapt their placement solutions. Centralized and optimal placement techniques such as Sonata, based on solving expensive ILP solvers which consider all the operators and available resource nodes, are computationally hard. Thus, the mechanism is inefficient and cannot keep up with the fast changing resource conditions by quickly updating placement solutions.

Some collection systems apply aggregation functions such as averaging [96] or sampling [97] on the data over a coarser-grained time scale (e.g., a min) to mitigate network bottlenecks. Typically, this is a pre-processing step before stream data is pushed into monitoring queries. The use of an aggregation in this way leads to loss of information (e.g, missing low-density outliers during sampling), so is complementary to the lossless approaches that we explore in our work. We also empirically show the limitations of using lossy/approximate techniques for server monitoring pipelines, in Section 7.6.

Wide-area streaming analytics. Recently, new systems have been designed to operate

efficiently on low-bandwidth networks such as wide-area networks (WANs). Sol [98] introduced an early-binding of analytics tasks to workers and dedicated communication tasks to improve resource utilization over WAN. AWStream [35] adapts between accuracy and bandwidth tradeoff by controlling the sampling of data, for the purpose of maximizing the achievable accuracy given scarce WAN bandwidth. MQO [99] provides a technique to share input/output data and operators across stream queries, saving WAN bandwidth consumption. While we aim at different design goals, future IoT applications may need to consider the wisdom shared by these prior works as they are complementary to our work.

Video analytics. Reducto [100] performs on-device filtering of video frames independently per-data source. To meet query accuracy requirements, resource-efficient filtering is performed using frame differencing technique which uses low-level features and an adaptive mechanism to set the differencing threshold on-device. CrossRoI [101] exploits cross-camera data correlations in a fleet of closely located cameras (e.g., cameras at a traffic intersection), to filter out redundant/overlapping frames at each data source. Work in drone-based video analytics [102] proposes 4 different strategies to filter frames and reduce network transfer costs. EarlyDiscard strategy uses DNNs, which are trained to filter out uninteresting frames from aerial views. Given that drones are resource-constrained, weak detectors such as MobileNet [103] are run on the device after applying optimizations to the DNN models executed on the video feed. Just-in-time-learning (JITL) strategy reduces false positives for frames transmitted from the drone, by enhancing the DNNs filtering capability with a cheap cascade filter. Reachback strategy enables storing the video frames in the drone storage space so that cloudlets can fetch the stored frames to perform analysis in case of critical frames being missed by the filtering strategies.

We note that such techniques drop data at the source, thereby impacting query accuracy. Our focus is on lossless approaches to reduce network transfer costs to the cloud, which can be augmented to frame filtering systems. Frame filtering doesn't consider dynamic compute resource conditions on the data source node which can become significant for handling query-specific filtering needs for thousands of video analytics queries [10] on the data source node. Moreover, even though such techniques meet the query accuracy needs while performing frame filtering on the data source, they only filter out frames before query execution. With the advent of state-of-the-art smart cameras having more powerful compute resources [100, 104], there is potential for also executing analytics operators on the smart camera itself, subject to available compute. Our placement solution can be easily integrated with frame filtering techniques to achieve the goal of partial query processing on the smart camera nodes. Doing so improves the data reduction benefits from filtering as well as

fine-grained data-level partitioning which partitions incoming data at each query operator. In Section 7.6, we show the network transfer reduction gains of our proposed partitioning policy compared to a coarse-grained data partitioning strategy at the query-level (i.e., at the source), which also determines the amount of data to be processed by the query on the data source node. The gains have been evaluated for queries involving text or numeric data, and can be extended to video analytics queries for future work.

Video streaming. Adaptive BitRate (ABR) algorithms have been proposed for video streaming applications, to provide high quality-of-experience (QoE) to users while handling the limited and dynamic network constraints at the client. The goal of ABR algorithms is for the client to download video feed from a server at the highest bitrates possible without rebuffering, subject to available network bandwidth. A segment corresponds to a few seconds of video playback data. Rebuffering occurs when the buffers storing segments are empty and thus there are no more segments to play out, the video playback needs to freeze in such cases until the next segment is received. Spiteri, et.al. [105, 106] propose two ABR algorithms BOLA-E and DYNAMIC, to maximize average bitrate of the downloaded video stream to the client, while minimizing rebuffering and bitrate oscillations (which result from bitrates being switched frequently that impact user QoE). They also propose a FAST SWITCHING algorithm to replace low-bit rate segments in the client’s buffer with high-bit rate ones, when possible. However, ABR algorithms address a different problem from the focus of our optimization efforts for improving operator placement on data source nodes. ABR techniques do not consider the compute resource constraints on the data source nodes and their impact on making placement decisions, for executing operations that constitute the analytics query at the source. They make network-bandwidth aware bitrate optimizations when requesting data from servers, whereas our efforts are on minimizing the network transfer costs from the source, by performing fine-grained data-level partitioning for query operators on the data source nodes, subject to compute availability. However, the concepts from these techniques could be used to extend our placement work for network-constrained environments, e.g., WAN-based analytics (see Chapter 9), where predicting the network conditions can help determine how much computation can be performed on the data source node, while meeting the budget constraints of limited and highly varying network bandwidth.

In-situ analytics. In-situ data analytics reduce data movement by executing query operations locally on the data source nodes. iMR [107] uses data summaries to trade off accuracy for performance. We compared Jarvis against data summaries in Section 7.6.4. Rule-based heuristics [108, 109, 110] statically determine which operators to push to data source, e.g., operators beyond first stateful operator cannot be pushed. Such static partitioning

quickly becomes sub-optimal in our system (Section 7.6) as compute resource demands may vary widely during query execution (Section 7.2.2). MapReduce jobs adapted for streaming applications [111, 112], perform compute load balancing across map operator instances by splitting input load across them. We compare Jarvis with load balancing-based query partitioning in Section 7.6.2. DIRAQ [113] and FlexAnalytics [114] execute indexing and compression operators at the data source, in which placement at the operator-level is considered. Dynamic resource allocation is enabled by frameworks like Spark using Mesos and YARN [115] at the task level (tasks are akin to operators in our setup). Operator-level placement is evaluated extensively in Section 7.6.2 under our setup.

Stream operator placement. Operator placement determines how to place operators in the application DAG across computing nodes for efficient query execution. These approaches are based on operator-level partitioning [23, 116, 117, 118, 119, 120, 121], which is less effective than data-level partitioning when applied in our setup (see Section 7.6). Prior work finds an optimal placement on heterogeneous resources [122, 123]. This requires solving an NP-hard optimization problem which makes it challenging to apply to our setup requiring fast adaptation across large number of data sources (see Section 5.3). Flouris et al. suggests several heuristics to assign operators near data source to minimize network transmission costs under geo-distributed sites [124]. All the proposed algorithms consider coarse-grained operator-level partitioning which is compared with our approach in Section 7.6.2. Furthermore, apart from the greedy heuristics, they exhibit exponential time complexity in the number of data source nodes for our setup, when considering the placement of each query instance’s operator across data source and remote servers in the edge or cloud. Nardelli et al. propose both model-based and model-free heuristics for operator placement [125], concluding that there is no one-size-fits-all solution. Query cost model-based approaches have also been implemented in centralized [116] and decentralized [118] systems to make operator-level placement decisions. They rely on accurate query cost estimates, which are computationally expensive to profile on the data source. Cost model-agnostic approach based on rule-based heuristics has been studied in [117]. We show that combining model-based and model-agnostics approaches in our approach outperforms using them individually in Section 7.6.3

Parallel query processing. Much prior work improves query performance by dynamically alleviating compute bottlenecks through operator scaling on input data. DS2 [11] can make a scaling decision over all query operators at a time given the correct query performance tracing result. Stela [126] can prioritize scaling of operators that have the greatest impact on overall query throughput. Similarly, other works in this domain all make scaling decisions at operator level [81, 127, 128, 129, 130, 131, 132, 133, 134, 135]. Applying these approaches to

our setup requires that the operators be replicated between each data source and its parent cloud node. However, compute resource allocation on the data source in our system happens at the application level, and an application consists of multiple operators with different data reduction capabilities. Our optimization goal is thus different: incoming records need to be carefully apportioned across multiple co-located operators on the data source, in order to minimize data transfer costs within a compute budget.

Conventional key-based load splitting strategies [136, 137, 138, 139, 140, 141, 142, 143, 144] are complementary to our work. These strategies assign input key ranges to different stateful operator instances such that each instance keeps the state of a distinct, non-overlapping key range. Integrating key-based load splitting into Jarvis may further reduce network transfer costs by minimizing the number of output keys sent from the data source. Recent studies investigate assigning subsequences of records or windows to operator instances based on the currently assigned compute load in the host node [145, 146, 147]. These approaches do not assume that compute resources in each remote server (i.e., edge or cloud node) are shared by multiple child nodes or data sources (see Figure 7.4). Therefore, compute load on the remote server node depends on jointly considering splitting decisions over data source nodes. This is exponential in the number of data sources.

SkewTune [148] and Google Dataflow [149] employ work-stealing techniques for straggler mitigation while others [150, 151] use local estimates of compute load to detect a congested host node and re-route input records to a different node in a greedy manner. In our setup, each data source has a fixed compute budget allocated for a query consisting of multiple operators with different data reduction capabilities. We could consider re-routing input records to a remote server node, when an operator does not result in a significant data reduction. This boils down to “w/o LP-init” which is evaluated for convergence speed in Section 7.6.3. To facilitate throughput comparison of our approach with compute load balancing techniques at the query level, we evaluate “LB-DP” in Section 7.6.2.

EdgeWise [85] runs a streaming engine on the edge, improving throughput and latency by prioritizing query operators experiencing backpressure. Our work on a placement mechanism between edge and cloud resources (Droplet [152]) and the related work Costless [87], address operator placement problem between edge and cloud resources for video analytics applications. They neither leverage the benefits of data-level partitioning nor respond quickly to dynamic resource conditions. Our solution for data source-level placement, does both.

3.3 ENERGY-AWARE PLACEMENT

There have also been several works considering the dual goals of performance and energy optimization when deciding placement with energy as a critical resource to optimize for its usage. Such systems additionally involve a power prediction module that is used to infer power consumption of application operators, so that it can be used to predict power costs and make placement decisions. Here, we discuss the works pertaining to power prediction techniques for mobile data sources.

3.3.1 Energy Estimation in Code Offloading Frameworks

ThinkAir [22] and CloneCloud [153] are VM-based code offloading solutions. Spinning up large VM components and moving them around is expensive even within a local area network [154]. Moreover, ThinkAir uses a power profiler inspired by hardware resource-usage based power estimation tool called PowerTutor which we discuss below. Serverless architectures are being explored as alternatives, to offload bursty workloads but it is limited to handling stateless functions [155]. Recent works [156] are exploring containers since they have lesser overhead but they are still coarse-grained, targeting full application-level placement and offloading. They do not require fine-grained energy estimation techniques at the operator-level. MAUI [20] was an early implementation of fine-grained, power-aware code offloading which required program annotations and resource modeling to obtain fine-grained power consumption estimates. Instrumentation puts burden on developers, increases code complexity and maintenance costs. Coara [157] and Coca [158] are programming frameworks which use aspect oriented programming to reduce developer burden for building offloading applications. And static analysis has been used to fully automate code instrumentation in [159]. But doing so requires dynamic analysis to quantify computation of loop operations for example. And dynamic monitoring via auto-instrumentation of application operators can generate large amounts of data and become expensive. Other works [160, 161] have used resource-aware session types for worst-case resource usage analysis of message passing processes. But it has not been applied in a practical system. IMCM [162] proposed an actor based approach for offloading and accounts for migration costs in its offloading models to make placement decisions. But its power profiling mechanism is similar to that used by the PowerTutor tool discussed below.

These works are more focused on the offloading-related challenges and improving infrastructure that enables offloading (stateful) mobile components to the cloud. For making a placement decision, they either rely on ILP solvers or brute force exploration of the place-

ment search space. Furthermore, works like ThinkAir make placement decisions using a policy-based approach i.e. it relies on rules such as, if the network quality is good then we offload the application operators. So its optimization goal is not to reduce network traffic and central cloud compute costs, which is the focus of our placement work in this thesis. Other works rely on a central placement solution that utilizes an ILP solver for making placement decisions. CloneCloud uses an offline service to compute possible placement decisions and cache them for look up at runtime, while MAUI runs the ILP solver at runtime. These techniques do not scale with the number of data source nodes, which is critical for handling the requirements of fast placement adaptation in the systems we consider 7.2.2.

3.3.2 Power Prediction Tools

Placement frameworks which model the energy use of applications rely on accurate power consumption estimates of operators to explore the costs of different placement strategies. Thus, the ability to efficiently predict power costs is a critical and fundamental requirement for building energy-aware placement frameworks. Here, we talk about existing power prediction techniques and their limitations that are addressed by our work. To measure mobile device power, previous research [163, 164, 165, 166] relied on external meters [167, 168]. However, they are impractical as they are limited to laboratory settings. Some automated measurement solutions [169, 170] rely on estimates for the remaining battery capacity available on device (state-of-charge or SoC), but they estimate power at the application level only. PowerTutor [171] automatically generates a power model by exercising hardware components in different power states and mapping them to SoC estimates during training. DevScope [172] and AppScope [173] address limitations of PowerTutor in isolating shared hardware resource usage between applications. However, these works model the power cost of individual hardware components based on their resource usage, and use the trained models to provide energy estimates. Such a mechanism requires building analytical power models for each new hardware component and constantly keep up with evolving hardware such as multi-core devices [174], GPUs [175], TPUs [176] and complex memory sub-systems [177]. With increasing complexity and diversity of hardware components, it is becoming difficult to build accurate hardware power models for each component as they frequently exhibit high prediction error [174, 178, 179]. Our approach investigates a mechanism that does not rely on analytical models to provide energy estimates. Furthermore, as in [171], programs are manually instrumented to obtain resource usage estimates of individual functions or operations in the application. This usage information is used to map it to power consumption costs. Source code instrumentation burdens application developers, increases code complexity

and maintenance costs. Analytical model-based techniques limit the portability and adoption of energy estimation solutions across devices and applications.

RECON [166] focuses on debugging energy hotspots in web pages and optimizing them for power usage, by developing a hybrid model consisting of resource usage of the operations at the hardware level as well as executing application component information at the software-level. To avoid relying on hardware power models, we show that using the information about executing software components at the fine-grained operator-level and utilizing sophisticated machine learning models to estimate energy costs, can suffice to obtain accurate energy estimates for our target applications. Sesame [180] requires devices to have OCV-based fuel gauge chips so it has limited applicability. Carat [181] uses a collaborative black-box approach to enable developers to identify whether their application is showing anomalous energy usage behavior across devices, or causing significant energy drain when running on certain devices. Thus, developers can debug energy bugs in their application. However, it only provides energy estimates at the coarse-grained process level. eProf [165] tracks system calls to overcome limitations of resource-usage techniques in modeling tail energy such as wireless radio. However, it still requires understanding the power usage characteristics of hardware and the associated system calls that need to be mapped to power costs.

Fine-grained power prediction at the operator-level, relies on building hardware resource models and program instrumentation whose limitations are detailed in Section 4.8.3. We propose an approach that does not require either of those for power estimation and building prediction models, thereby enabling energy-aware placement of application operators across mobile devices and remote servers in the future.

3.4 SUMMARY

Following are the key features of our work that differentiate it from existing literature:

- We motivate the need for *adaptive placement* in several real-world stream processing systems, due to changing input and compute resource conditions on the resource nodes. We then design adaptive placement strategies which find new placement plans when resource conditions change at runtime. This is unlike existing techniques which make static partitioning decisions [17, 95].
- Finding an optimal placement plan for a given application and resource topology is computationally expensive. Thus, we design efficient partitioning heuristics which exploit improved execution modeling and fine-grained partitioning techniques to identify

high quality partitioning plans. This is unlike current works which solve for an optimal placement strategy [12, 84, 182].

- We consider lossless approaches which do not tradeoff application output accuracy for performance, unlike works such as AWStream [35, 100].
- Several partitioning strategies solve for a placement strategy with different optimization goals. E.g., DS2 [11] identifies an optimal scaling factor for each operator, so that the application can keep up with the incoming data rates, SkewTune [148] mitigates the effect of stragglers by balancing the compute load across resource nodes. In our work, we look for optimizing the compute-communication tradeoff when making operator placement decisions.
- In the edge-cloud setting, following are the key distinguishing features of our work:
 - Several existing works which focus on workloads within datacenters [11, 126] cannot be applied to our setting, as they do not consider heterogeneous compute and network resource environments.
 - We jointly solve for 3 optimization knobs: (1) placement plan to map operators to resource nodes, (2) data parallelism factor for each application operator, (3) pipelined parallelism for operators distributed across resource nodes to process an epoch of data items. This is unlike current works which do not jointly consider all three optimization knobs when making placement decisions [23, 32, 70, 87, 147, 148].
- For placement at the data source-level, following are the key distinguishing features:
 - Current systems [12, 123, 124] use operator-level partitioning: if a data source does not have sufficient resources to execute an operator on all records, the operator is executed only on the stream processor. Instead, we perform data-level partitioning, i.e., we allow an operator to be executed both on a stream processor and data sources.
 - We consider a fast adaptation technique which effectively handles the dynamic resource conditions on data source nodes, by combining a resource model-based and cost-model agnostic approach to solve for a placement plan. This is unlike existing works which consider one or the other approach, but do not combine the techniques [125].

- For predicting the power cost of application operators on mobile devices and enable energy-aware placement decision-making, we rely on an empirical technique to estimate power consumption based on device-level energy draw information in conjunction with the application’s execution traces. This is unlike current works, which use analytical models designed manually by developers, based on either hardware resource utilization or software configuration parameters [166, 171].

CHAPTER 4: MODELS AND PROBLEM DEFINITIONS

In this chapter, we explain the models defined for applications and underlying resource topologies, along with basic assumptions underlying the systems we built for scalable placement. We then formally define the problems we address in this thesis.

4.1 DATA MODEL

We model the data as an unbounded stream S of *data items*, $s = \langle d_1, d_2, \dots \rangle$. A *data item* can be seen as one unit of data or a measurement defined by the application (e.g. tuple, log record, video frame, sensor reading). For certain applications such as video processing pipelines, a stream/sequence of such data items is typically defined with fixed frequency f (e.g., 30 frames/s). We aggregate a sequence of data items into *epochs*, where each epoch has duration T . The k^{th} epoch $epoch_k$ can be defined as $epoch_k = \langle d_1^k, d_2^k, \dots, d_I^k \rangle$, where $I = f \cdot T$ is the epoch size. In the rest of the paper, we deal with epochs of data. The epoch size/duration is an application defined parameter.

4.2 RESOURCE MODEL

The resource topologies we consider consist of 3 different types of resource nodes, arranged in a hierarchical topology: data source, edge and cloud nodes. A data source is the node which generates the data stream. Some data sources may not have compute resources (e.g., camera without an embedded processor) or have very limited compute to perform simple operations such as sampling or aggregation, but is not sufficient to execute the entire application. Examples include smart phones, smart cameras, datacenter server nodes in monitoring pipelines, etc. The compute resources are shared with other external applications such as other apps on a smartphone or customer hosted foreground services on a datacenter server node.

The cloud consists of remote resource nodes which are the farthest from the data source, in terms of network hops. They have dedicated compute resources for performing analytics operations and have powerful cloud-class servers (e.g., server nodes provisioned within cloud platforms such as Amazon AWS or Microsoft Azure) with large amounts of compute resources. Edge nodes are remote resource nodes from the data source, which are closer to the data source compared to the cloud nodes (in terms of network hops). They have dedicated compute resources for analytics applications and consist of intermediate-class compute capabilities:

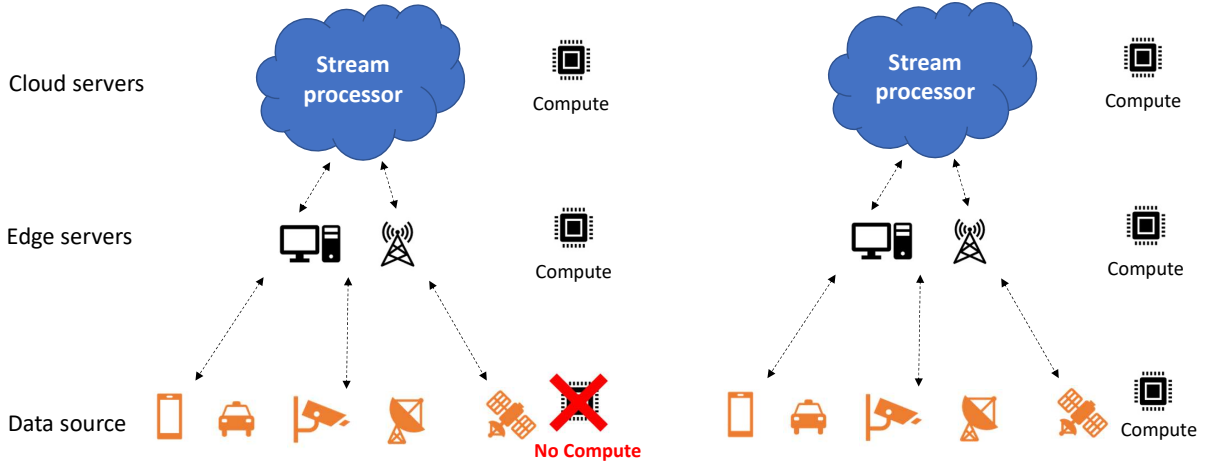


Figure 4.1: Type I resource topology on the left, data sources do not have compute capabilities. On the right, we have type II resource topology in which data sources have on-board compute resource to process data streams.

weaker than cloud-class servers, but stronger than typical data source nodes. Example edge servers include micro-datacenters near cell towers and intermediate aggregation server nodes in a monitoring pipeline.

As shown in Figure 4.1, we categorize real-world systems into one of two types of resource topologies depending on whether the data sources can perform computation or not. In the type I topology, we have data sources which do not have on-board compute resources and therefore transmit all the generated data streams to remote servers for processing. They represent data sources such as cameras or sensors with no compute capabilities, uploading raw data to a remote edge or cloud node. However, in topology of type II, data sources have on-board compute resources to perform limited processing of the generated data streams. Examples of such data sources include smart cameras or mobile devices which can process the data and reduce network transfer costs from data sources. They then send the intermediate data to edges and cloud for further processing. We make the distinction between these two types of systems because of the following reasons:

- Depending on whether the data sources have on-board compute resources and can be used to execute application operators, the total number of compute resource nodes to be considered for making placement decisions can vary significantly. We may have tens of edge nodes in real-world systems with type I topology whereas the number of data sources to consider for placement in type II topology can be in the order of hundreds or thousands of nodes in large scale services. As we will see in subsequent chapters, the



Figure 4.2: Illustrative analytics application graph representing operators which perform individual operations (e.g., filtering, aggregation, classification, etc.) along with edges representing dataflow dependencies.

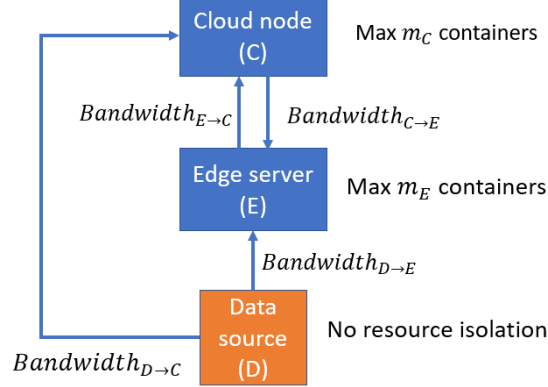


Figure 4.3: Example resource model for each node in data source, edge and cloud layers.

difference in scale has very different implications in terms of the types of techniques that can be considered for making efficient placement decisions.

- Edge and cloud nodes can be dedicated resources for analytics workloads and have significantly more compute resources compared to data source nodes. This enables the allocation of compute resources using containers or other virtualization technologies, with a container allocated for each application operator. However, data source nodes are typically shared by multiple applications. Thus, dedicating resources to an application or its operators can lead to significant under-utilization of the already very limited compute on these nodes. Applications on data sources operate under highly dynamic resource conditions in which compute available to an application can change quickly over smaller timescales. Placement solutions at the data source need to adapt to these fast changing resource conditions.

We model the physical resources as a weighted directed graph $G_R = (V_R, E_R)$. An example of the nodes in each of the data source, edge and cloud layers and their compute/network resources are shown in Figure 4.3. Vertices $V_R = \{D, E, C\}$ represent compute resources, i.e., data source D , edge server E and one cloud server C . The links $E_R = \{(B_{D,E}, B_{E,C}, B_{C,E}, B_{D,C})\}$ represent bandwidths from data source to edge, edge to cloud, from cloud to edge and data source to cloud respectively. Compute resources on edge

and cloud nodes are fixed and dedicated for the stream processor. It is possible to enable resource sharing for better utilization of resources while obtaining predictable compute costs via resource isolation, by allocating virtual resources at the operator-level and mapping each operator to execute in its own virtual resource. Each physical resource can have multiple virtual resources that can use a portion of the physical resources' CPU and memory. Virtual resources can be thought of as virtual machines (VMs), or containers. For simplicity, we refer to virtual resources as containers in this thesis. Each of E and C has maximum number of containers denoted as m_E, m_C respectively. We assume that all the containers across physical resources in the same layer, i.e., edge or cloud in the resource hierarchy, are homogeneous in terms of compute and memory resources reserved for them. Regarding data source nodes, even if they have on-board compute resources, they do not have containers or any other resource isolation mechanism for the stream analytics application or its operators executing on the node. In certain scenarios, e.g., cloud server monitoring pipelines where the server node is a data source, we may have a container with 1-2 cores and 1 GB RAM on each node for stream processing. Rest of the compute on the node is allocated for customer-deployed foreground applications such as web services. But within the container, multiple analytics queries and their operators may be concurrently executing, each with changing resource needs over smaller timescales.

We make some comments about the scale at which the resource topologies are deployed in real-world settings and their impact on placement solutions. In type I topology, we consider the partitioning when scaling the number of data sources sending their data to remote edge and cloud servers. The analytics application needs to process data from each of the data sources, and parallelize its execution across the available resources between edge and cloud nodes. We note that the number of edge nodes per cloud node is limited, i.e., in the order of 5-10 nodes for each cluster of edge and cloud nodes considered for partitioning. Thus, the scale of resource nodes for which partitioning needs to be done is relatively small and limited. In systems with type II topology however, we need to harness the additional compute resources provided by data source nodes. Scaling placement solutions for such topologies is challenging. The compute resources on remote edge and cloud nodes are shared by multiple data sources. If the compute resources on a data source node are not sufficient to meet the processing needs of all the data generated on the node, then the application load is offloaded to a remote server. However, if all the data sources offload requests to remote nodes, then the compute and network resources on edge and cloud may get saturated. Thus, sharing of compute and network resources by multiple data sources means that placement decisions need to be made jointly across all data sources. It becomes prohibitively expensive as we increase data sources. In our case study in Chapter 7, we show how we address this problem

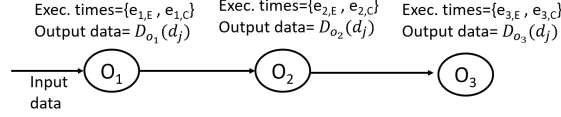


Figure 4.4: Example operator DAG.

in a real world setting.

4.3 APPLICATION MODEL

We model the dataflow in streaming applications as a directed acyclic graph (DAG) $G_o = (V_o, E_o)$ of operators (Fig. 4.4). An operator is a processing element that can execute user-defined code (e.g. grouping and reduction in datacenter server monitoring queries, convolution or face detection on video frames in face recognition applications). The vertices in the DAG represent operators $V_o = \{o_i \mid i = 1 \dots n\}$, and the links between them $E_o = \{o_i \rightarrow o_j \mid i \neq j \wedge 1 \leq i, j \leq n\}$ represent the dataflow dependencies, where $o_i \rightarrow o_j$ means o_i is applied to a data item in the stream before o_j is applied to the same data item. We can have forks and joins in the operator graph wherein a single operator may send its output to multiple successor operators in the graph (i.e., a fork operator) or an operator receives its input from multiple predecessor operators in the graph (i.e., a join operator).

We highlight the common characteristics of video stream processing and query analytics applications that make them suitable for near-data processing (see Section 2.1) as follows:

- **Support for incremental computation:** the application logic should be able to execute incrementally on the data. Consider an analytics application which calculates exact quantiles for a dataset being collected from multiple data sources. In this scenario, there is no partial computation possible on any of the nodes closer to the data source. This is because quantiles require processing the entire dataset in one go, without any partial processing that can be done and reused to obtain the final result. Thus, calculating exact quantiles is not a candidate for performing near-data processing. However, we note that approximate quantile computation [183] can be incrementally performed and can leverage compute resources closer to the data source.
- **Interface support for fine-grained application partitioning:** Coarse-grained partitioning techniques can make placement decisions at the application-level, i.e., deciding whether to execute an entire application on the cloud or near-data compute nodes. Fine-grained partitioning of an application can support partial execution of the

application workload on a resource node and send intermediate output to remote nodes for further processing. Such partitioning opens up opportunities to improve our resource management strategies. However, the underlying runtime needs the application to expose a well-defined interface so that it can call into such an interface and implement its fine-grained placement decision for the application. E.g., if the underlying runtime makes a placement decision at the operator-level, then each application operator should have an interface which the runtime can call into for executing the operator on the desired resource node. Similarly, partially executing the incoming data items at an operator on a resource node, requires the ability to filter out the remaining data and send it to a remote node for processing. Our definition of operator graphs to represent applications makes them amenable to define such interfaces and support the execution of fine-grained placement decisions by the distributed runtime.

- **Intermediate data size does not increase with near-data processing:** The granularity for partitioning should be defined in such a way that the output data size at each step does not increase compared to the input size. Otherwise, if the data size increases as a result of executing an operation closer to the data source, then it not only incurs a compute cost on the data source but also does not result in any network transfer savings. An example of an application in which such partitioning may not be directly beneficial is deep learning inference workloads such as convolutional neural networks (CNN) [184] for video processing pipelines. If we consider partitioning at the granularity of individual layers within the neural network (e.g., convolution, pooling and fully connected) so that each layer is an application operator, then the increase in number of channels at the intermediate layers prevent us from leveraging the benefits of partitioning by executing a subset of the layers closer to the data source. However, we note that the existence of an operator which increases data size does not mean the entire application cannot be partitioned in a fine-grained manner. Neural network partitioning has been explored for data batches instead of individual frames in a video stream [31, 32]. Furthermore, analytics queries with join operators can increase the output data size as they combine the records from multiple input sources (e.g., a table and a stream, 2 streams), and each record from one input source can satisfy the join predicate over multiple records at the destination input source. Even though executing joins on near-data compute resources may not be beneficial, a project operation following the join may filter out columns that are not of interest to downstream operators in the query. Thus, a partitioning decision can be made by combining the join and project operators to co-locate them in our placement strategy,

subject to compute constraints on the resource node.

There are additional constraints on the nature of applications that can be partitioned at the data source-level, which are described in detail in our case study in Chapter 7. Though it may appear that such constraints on applications as discussed above, may severely restrict the types of scenarios in which near-data processing can be beneficial, we note that several real world scenarios satisfy these constraints. It is reported by Drizzle [185] that 95% of aggregation queries in server monitoring pipelines on a popular cloud-based data analytics platform consist of aggregates supporting incremental updates, such as sum and count. Furthermore, several works have explored the benefits of near-data processing for video-processing pipelines [20, 41, 186].

Modeling application components as operators vs. actors. In our current work on placement, the operator abstraction we consider does not natively support dynamically migrating an operator instance from one node to another or the dynamic creation of new operator instances. When a new placement decision is made and an operator needs to be moved from one resource node to another, the distributed runtime needs to be programmed to stop the currently executing operator, re-deploy the operator instance on the new node and rewire the network topology based on operator dataflow dependencies, if needed. The operator can then continue processing the input stream on the new node. Similarly, when a new resource node is added to the system, the runtime needs to replicate the operator instances on the new node and update the network configuration so that predecessor operators can send their output data to the operator now running on the new node. The development of the distributed runtime can be simplified by using the actor [187] abstraction to represent operations within the streaming application.

Actors have been widely adopted by the cloud computing community and have been used in building several large-scale cloud-based applications [188, 189, 190, 191]. Within the actor model, distributed application components, each of which map to a specific functionality within the application, are seen as actors. They are autonomous objects operating concurrently and asynchronously (shown in Figure 4.5). While actors have many desirable properties that make them suitable for solving concurrency issues in large scale systems, their properties of state encapsulation, location transparency and mobility make them attractive for building code offloading solutions in mobile-cloud computing (MCC) applications. Offloading application components [20] involve dynamically transmitting code and/or intermediate state in memory, from one resource node to another. Actors provide a functional unit for stateful MCC application components, with several languages providing native support for migration [192] of actors. Fine-grained mobility enables the mobile device to adapt to workloads by exploring

different configurations of the application partitioned between mobile devices and remote servers [20, 193]. Actors can be seen as dealing with streams as well: their message passing interface means that they send and receive series of messages in order to transfer data from one place to another. Certain actor libraries such as C++ Actor Framework (CAF) [194] and Akka [195] are recently supporting stream processing semantics within actors.

Since the operator abstraction we consider for stream processing applications does not natively support state migration across nodes, we can leverage actors to transparently migrate operations and create new actors dynamically when the underlying resource topology changes. Changes in resource topologies can occur in the case of mobile users whose nearest edge server can change as they move their connection from one cellular tower to the next (assuming the edge server is hosted in a micro-datacenter near a cellular tower). Moreover, if a resource node fails in the middle of processing an epoch, then the placement algorithm is run on the new resource topology to update the placement decision. We assume a simplified failure recovery model in our current runtime, wherein subsequent data items arriving after the placement decision has been updated, are part of a new epoch. We do not consider the processing state that was lost for the epoch that was being processed when the node failed. However, to support more sophisticated failure recovery models in the future, it may be desirable to migrate the intermediate epoch processing state from before the node failure, to the new resource node and complete processing of the pending epoch. Actors can natively support such migration capabilities.

Note that while actors simplify the design and development of the underlying runtime, the runtime still needs to efficiently make placement decisions given an application and resource topology, which is the primary focus of our work in this thesis. We leave the porting of our applications to use the actor abstraction, and thereby simplify the design of our distributed runtime, for future work. A critical next step for us is to use the streaming support within actor libraries to migrate our algorithms and systems to use actors and enable fluid code migration. However, there are other system-level challenges as well. In addition to sending and receiving data, we also need to take care to not overflow any buffers or mailboxes in the process. Another pitfall is that actor messages can be lost and must be retransmitted in that case. Failure to do so would lead to missing data at the receiving side and accuracy loss.

4.4 ADAPTATION

We consider the following possible changes in resource conditions over time, which can result in the need to profile applications and recompute their partitioning plan:

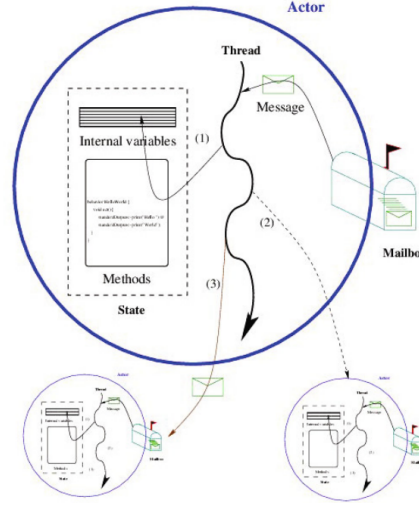


Figure 4.5: Each actor has a type, which defines its behavior. During execution, an instantiation of an actor type gets scheduled to process a message received in its mailbox and in response it can: change internal state and behavior to respond differently to the next message, create new actors and/or send messages to other actors [196]

- Change in the input load on edge and cloud nodes, due to changing number of data source nodes (within the type I resource topology).
- Change in available compute resources for the analytics application on data source nodes (type II resource topology). This can result from a change in the available compute for an analytics application on data source node because the resources are shared by multiple other applications (e.g., foreground web service running on datacenter server nodes) whose resource needs can vary.
- Change in input distribution for applications on data source nodes (type II resource topology) which can affect application resource needs.
- Change in the energy resource needs for application operators (battery-driven data sources in type II resource topology) due to changes in either the input stream content or data size. This change impacts the power prediction mechanism proposed in this thesis, which can be used for energy-aware placement.

We restrict our investigation to the above changes in resource conditions. Other types of resource changes can occur. As an example, the content of frames for video processing applications in type I resource topologies can change due to changes in video parameters like frame resolution. We can detect such changes by monitoring the end-to-end processing latency and comparing it with expected latency based on our execution model. A profiling

step then needs to be triggered at runtime, wherein the updated compute costs of operators are obtained on containers deployed at the edge and cloud layers. Once profiling is done, we can rerun our placement algorithm with the new profiling information. In this thesis, we focus on how to efficiently determine a placement plan once profiling information is obtained. A detailed investigation of improving the cost of profiling in such scenarios is left for future work.

When resource conditions change on data source nodes, since they do not support virtualized resources running, profiling needs to be done on each data source separately. Data source node resources are allocated at the application-level, and shared by multiple operators with no resource isolation available for each operator. This makes profiling of each operator challenging at runtime. Furthermore, profiling needs to also be done within the very limited compute resources on data sources, despite the potentially high compute cost of obtaining accurate profiling estimates. We investigate in our case study in Chapter 7, how profiling can be done efficiently despite the limited compute resources and lack of resource isolation on data sources.

Adaptation of placement decisions is done at the granularity of adaptation epochs, which may be aligned with the epochs defined by the processing time boundary for operators (within the data model). When resource conditions change at runtime, the state of the executing application is monitored to detect such a change at the end of the next adaptation epoch that gets processed in its entirety, after the resource change occurred. To initiate the computation of a new placement plan at runtime, data sources periodically inject *adaptation watermarks* into the stream (similar to [75]) and they delimit ordered consecutive *adaptation epochs* of records. Adaptation watermarks are different from the traditional watermarks used to indicate progress of time in stream processing systems.

4.5 OPERATOR PROFILE

When profiling is being done, following information is collected for each operator in Figure 4.4, which is then associated with its *profile* and includes:

1. The cost of executing operator o_i on a data item d_j , when placed on each resource $r_l \in V_R$. Operator o_i can be placed on data source, edge or cloud node, so we denote their corresponding execution costs as $e_{i,D}$, $e_{i,E}$ and $e_{i,C}$. Note that the cost is measured in terms of time units for which the operator executes on a data item. We assume that the data items within epoch $epoch_k$ have equal sizes therefore the execution cost is the same for all data items.

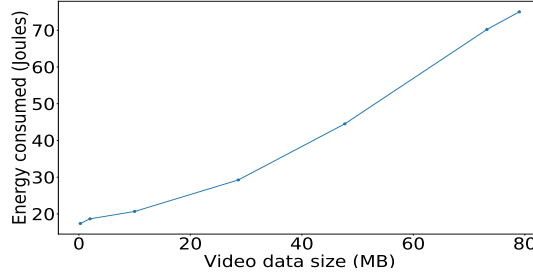


Figure 4.6: Communication power cost with increasing size of video data, sent from mobile device to a remote server for processing. Power is measured using an external power meter when the mobile operator is scheduled to execute.

2. The size of output data of each operator o_i . We denote it as D_{o_i} bytes.
3. The transmission time of the result of o_i from the edge to the cloud, $tr(E \xrightarrow{D_{o_i}} C) = \frac{D_{o_i}}{B_{E,C}}$, where $B_{E,C}$ is the bandwidth (bytes/sec) from the edge to the cloud. Since we assume that the data items within an epoch have equal sizes, the output data of o_i is the same for all data items i.e. $\forall d_j^k \in epoch_k \ D_{o_i} = D_{o_i}(d_j^k)$.

Operator power costs can be used to make energy-aware placement decisions. However, estimating the power cost of each operator within an application is itself challenging. Our work addresses this challenge and propose a tool which make power predictions for operators within mobile video applications. Details on the energy cost model used by the power prediction tool, is discussed below.

4.6 OPERATOR ENERGY COST MODEL

We model the energy usage characteristics of applications on battery-driven data sources which can be used to predict operator-level energy costs and can be used for making fine-grained placement decisions to optimize for energy consumption on the data source.

When an operator is scheduled to process a data item on the mobile device, it uses hardware resources (e.g. CPU, memory, etc.) to carry out the computation, each of which has an associated energy cost. Also, when an operator sends/receives data to/from a remote node, the request is transmitted over the network, along with data required by the destination operators to process that request. Network data transfers also incur energy costs and highly depend on the type of network interface used for transferring the data (e.g., WiFi, cellular). We first seek to understand whether the measured energy drain correlates with the executing operators in that time interval. To confirm that the mobile energy cost associated with this

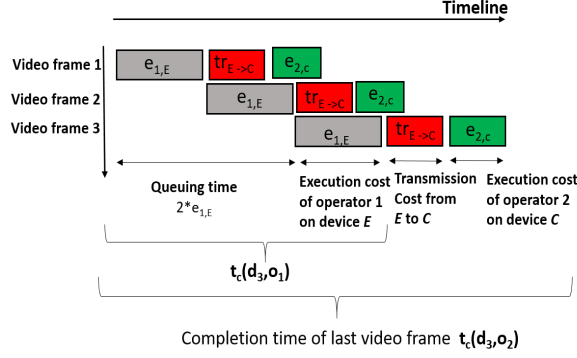


Figure 4.7: Pipelined Execution of operator o_1 on resource E and o_2 on resource C .

communication is actually incurred when the operator performing asynchronous send/receive operations is scheduled, we ran an experiment with an operator uploading different sizes of video data to a remote server. When the video data is uploaded, we concurrently measure the corresponding energy cost and log it along with the uploaded data size. Figure 4.6 shows that the measured energy cost correlates and increases with the uploaded data size, which is what we would expect based on network power models proposed in previous papers [171].

The mobile operating system exposes APIs for measuring the consumed energy at the device-level, over a given time interval [197]. However, to characterize the energy costs of application operators, we need to attribute our energy measurements to the corresponding operators. Moreover, operators can exhibit complex energy usage characteristics. An operator can change resource usage during its execution and potentially be in multiple power states. E.g. an operator could process video frames (that incurs CPU/memory cost) and based on the output of pre-processing, either upload the frame (which incurs network cost) or discard it, the latter resulting in no energy expended. Thus, operator power state is sensitive to the input stream. Our work in Chapter 8 discusses techniques to address energy attribution challenges and handle input dependencies of operator power states.

4.7 EXECUTION MODEL

One of our key contributions for edge-cloud placement, is the ability to model concurrent execution of multiple dependent operators in a pipelined fashion across one or more resource nodes. If there are multiple dependent operators executing on different containers, pipelining results from concurrently processing all the data items in an epoch. Such parallelism can exist when operators are distributed across available containers in edge and cloud layers. Consider the example video processing pipeline in Fig. 4.7. An epoch of 3 video frames is processed by two operators o_1 and o_2 . o_1 is placed on resource E and o_2 on resource C . First o_1 is applied

to the first data item and the output is transmitted to resource C where operator o_2 can be applied, while o_1 is concurrently applied to the second data item. Assuming $e_{1,E}$ is a large value, the time for processing the entire epoch $T(epoch_k)$ is equivalent to $t(d_j^k, o_i)$, which is the completion time of the last data item d_j^k in $epoch_k$ and o_i is the last operator applied to it. So for the above example, $T(epoch_k)$ is:

$$t(d_3, o_2) = \underbrace{t(d_3, o_1)}_{\text{prev. operator}} + \underbrace{0}_{\text{queuing}} + \underbrace{tr(E \xrightarrow{D_{o_1}} C)}_{\text{transmission}} + \underbrace{e_{2,C}}_{\text{execution}} \quad (4.1)$$

$$t_c(d_3, o_1) = \underbrace{0}_{\text{prev. operator}} + \underbrace{2 \cdot e_{1,E}}_{\text{queuing}} + \underbrace{0}_{\text{transmission}} + \underbrace{e_{1,E}}_{\text{execution}} \quad (4.2)$$

Given the limited resources on data source nodes with 1-2 cores, we consider the potential for parallelism on data sources to be limited. So, on data source nodes, we assume a simpler execution model of sequential execution of operators per data item, on each data source. Multiple analytics applications or queries can still be running concurrently on the data source, processing different data streams.

4.8 PROBLEM DEFINITIONS

We address three major challenges in this thesis. All of the problems address requirements within the distributed runtime to enable it to make scalable placement decisions. We first look at problem of computing a placement plan for application operators given a type I resource topology, with the goal to enable fast and adaptive placement for systems with varying number of data sources. Second, we look at efficiently computing placement plans for type II resource topologies, in which the focus is on making placement decisions for operators executing on data source nodes. Third, we look at the problem of estimating energy use of operators and training a power prediction model based on our estimates, as a first step towards enabling energy-aware placement decisions.

4.8.1 General Operator Placement Problem

Let's consider N_d data sources running an application with operators from V_o set. The set V_r consists of data sources along with edge and cloud resources. To determine a placement plan, we need to find a mapping Map from operators to resources defined as $Map : V_o \rightarrow V_r, Map(o_i) = r_j \mid r_j \in V_r$. And let $T(epoch_k, Map)$ be the completion time of processing an entire epoch $epoch_k$ given the mapping Map . The operator placement problem is to find a map Map for which $T(epoch_k, Map)$ is minimized subject to the following constraints:

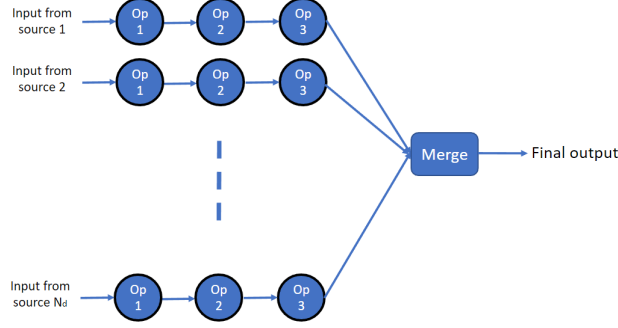


Figure 4.8: Physical representation of the application graph for multiple data sources, with one instance of the application for each data source.

- If o_i and $o_{i'}$ are dependent operators then:

$$(Map(o_i) = r_l) \wedge (Map(o_{i'}) = r_p), \implies B(r_l, r_p) \neq 0 \quad (4.3)$$

- Total resource cost of containers consumed by operators executing on resource r_l does not exceed $Capacity(r_l)$, which defines the available compute capacity of resource r_l

$$\forall l \mid r_l \in R, \sum_{i=1}^{|V_o|} \mathbb{1}_{Map(o_i)=r_l} \cdot \alpha_l \leq Capacity(r_l) \quad (4.4)$$

where α_l is the resource cost a container on resource node r_l .

Now consider the illustrative example graph in Figure 4.2. It is a logical representation of operators and their dataflow dependencies within the analytics application. The graph can be optimized to generate a physical operator graph with the goal to fully utilize available compute and optimize the input epoch completion time. We focus on the optimization which parallelizes operators given the available resources. A parallelization factor $f_{maxparallel}$ is used to denote the maximum number of physical instances of the application that can be running in parallel. If $f_{maxparallel} = 1$, then a single instance of the application and its operators sequentially processes the generated data items from all data sources. However, such a decision does not fully utilize the available compute and becomes compute bottleneck-ed as we increase data sources. On the other hand, fully parallelizing the execution by utilizing all available resource nodes may not be favorable if network transfer costs are very high between resource nodes which host operators with dataflow dependencies. Thus, one needs to carefully consider the compute-communication tradeoff involved in making placement decisions for each application operator instance. We set $f_{maxparallel} = N_d$ in our setup. A smaller value will not be able to potentially exploit parallelism in the incoming streams across

all data sources, as there are not enough operator instances to consider all the incoming streams separately. If there are more than N_d application instances, then it will enable further parallelizing the processing of stream from within a data source. However, we cannot parallelize the operators beyond the available nodes in resource topology, which is the upper bound for number of concurrently running operators. Since $N_d \gg |V_R|$ in our setup for type II resource topologies, we can safely limit the parallelism factor to N_d . We could have set $f_{maxparallel} = |V_R|$, however, this requires us to additionally determine the mapping from the generated stream of each data source to the physical operator instance to which it needs to be mapped for processing. This is challenging to do manually, so we leave the value of $f_{maxparallel} = N_d$ and leave it to our proposed algorithm to explore the possible placement configurations in which different subsets of the data source streams are processed sequentially vs. concurrently.

Now given the application graph G with $|V_o|$ operators, the total number of operators to process for making a placement decision is $N_d \cdot |V_o|$, as shown in Figure 4.8. It consists of an additional merge operation that is performed on the node where the final output is generated. Since it is pinned to a resource node in the cloud, it is not considered in the application operators for which placement decisions need to be made. We can redefine the general operator placement problem for the optimized physical application graph, by considering an updated set of operators $V'_o = N_d \cdot |V_o|$, and dataflow dependencies between operator instances similar to the original graph, with no additional dependencies across application instances. Figure 4.8 shows the dataflow dependencies for our example application. The optimization objective and constraints are similar to the ones defined earlier, but now for the operator set V'_o .

The resource nodes considered for our placement problem depend on the target resource topology. In type I resource topology, V_R is the set of all resource nodes in the edge and cloud layers. Each of the operators in the set of physical operator instances of size $|V'_o|$ can be placed on any of the available edge and cloud nodes. If a target resource topology exists in which not all edge nodes are accessible to a data source node for remote execution (e.g., if the edge nodes are geo-distributed), we can limit the possible placements considered by limiting the nodes considered in set $|V_R|$ to only consist of those running in the same geo-location. In the latter case, we make the simplifying assumption that there are dedicated cloud nodes for each geo-location. We will see in Chapter 5 that this assumption simplifies our algorithm and subsequent analysis, as it prevents us from having to consider potential interference in placement decisions on nodes in the cloud layer, in case the nodes were shared across geo-locations.

In resource topology of type II, the set $|V_R|$ can consist of all the resource nodes across

data source, edge and cloud layers. Each data source node is a compute resource on which only the operator instances corresponding to this node can be placed. Since data source nodes do not have network links connecting them directly, a placement plan which places an operator instance from one data source on a different data source node, is not feasible. We could just rely on pinning constraints to handle this case, but considering pinning constraints does not lend itself to easy analysis as we will see in Chapter 5, since they are user-provided configurations. Instead, we consider a logical representation of the data source node D which captures for each operator instance, the corresponding data source node with which it is associated during placement. Note that D represents the logical data source node (for the purpose of making placement decisions), and for a given operator instance, it is associated with the corresponding physical data source node $r_l \in V_R \forall l \in [1, N_d]$. This logical data source node D is unique in that unlike other resource nodes in edge and cloud layers where only one operator can be mapped to a container within the node, there are no containers in the corresponding physical node. So, all the operators can be concurrently mapped to the resource node D since the operator instances which are mapped to D , can be placed on their corresponding data source nodes.

We will see in Chapter 5 that solving the generalized placement problem is challenging. It is NP-hard, and thus requires us to consider heuristics to efficiently solve the problem. The chapter describes such a heuristic for resource topology of type I. However, when considering all the resource nodes in type II topology, the proposed heuristic does not scale when considering large number of data sources for placement. This is because, apart from the possible placements that need to be considered for type I resource topology, we also need to additionally consider possible operator combinations that can be concurrently scheduled at their corresponding data source nodes. More detailed analysis is provided in Chapter 5.

Given that solving for operator placement is computationally hard when considering the additional resource nodes at the data source layer, we look for techniques to decouple the placement strategies at the data source layer vs. edge and cloud layers. Real-world systems of type II resource topology are bottleneck-ed due to the massive volumes of data being transferred from the data source nodes. The goal of near-data processing strategies is to reduce network transfer costs to the remote edge/cloud nodes. Consequently, the compute load on edge and cloud resource nodes is also reduced. Thus, at the data source layer, can we greedily place the operators as close to the data source as possible, only considering the resource nodes close to the data source and not compute resources on all the remote nodes? With this intuition to reduce the number of resource nodes to consider between data source and edge/cloud layers, we define a different placement problem based on near-data processing.

4.8.2 Near-Data Operator Placement Problem

Given an application DAG and a type II resource topology where computation can be done on data source nodes, near-data operator placement plan identifies at each layer of resource nodes, the minimal number of operators that need to be executed remotely vs. those operators that can be executed locally in that layer. Specifically, at the data source level, we need to identify operators that can be placed on the data source node vs. those that require remote execution on edge or cloud nodes. However, this is a greedy approach and may result in increased end-to-end latency cost if the network transfer is not a bottleneck to begin with. So we need to only consider those placement plans that do not result in an increased end-to-end latency overhead due to local execution at the data source layer, compared to a placement plan that involves remote execution.

To formally define the problem, we refer to a boundary operator for the application running on each data source node, such that the data source executes only the operators prior to the boundary operator (including itself) in the topological order of the DAG. Next, we define the near-data operator placement problem.

Problem Definition. Each data source can execute an instance of the application with $M = |V_o|$ operators in the topologically sorted application DAG. Let $\mathbf{b}$ represent the vector of boundary operators for a given query, where b_i denotes the boundary operator of the i^{th} data source. Our goal is to find $\mathbf{b}$ which minimizes the number of operators sent to cloud for remote execution, without sacrificing query processing time as a result of the operators executing locally on data source, as follows:

$$\min_{\mathbf{b}} \sum_{i=1}^{N_d} \sum_{j=1}^M rc_j x_{ij} \quad (4.5)$$

subject to

$$T_{local}(i, b_i) \leq T_{remote}(i, b_i) \quad \forall b_i > 0, i \in [1, N_d] \quad (4.6)$$

Here, x_{ij} indicates if the boundary operator of i^{th} data source is j , rc_j is the processing cost on the remote server due to query partitioning at the j^{th} operator, $T_{local}(i, b_i)$ computes the local computation time cost for operators 1 to b_i on the i^{th} data source node, and $T_{remote}(i, b_i)$ denotes the network transmission time cost of intermediate data from b_i^{th} operator to its parent stream processor node along with computation time cost of executing operators $b_i + 1$ till M on the parent node. As we want to incentivize executing operators on data sources, the partitioning costs are ordered as $rc_1 > rc_2 > \dots > rc_M$.

In the next chapter, we discuss the computational hardness of the general and near-data

operator placement problems. We also discuss efficient heuristics designed for addressing the two problems and analyze them.

4.8.3 Predicting the Power Cost of Application Operators

For data sources which are battery operated in type II resource topologies, e.g., smartphones, energy becomes another important resource that needs to be optimized. Previous works have shown the benefits of making fine-grained operator-level placement with the goal to optimize energy usage [16, 20]. E.g., consider an application performing pre-processing steps on camera frames in a video analytics application. The input data may be large so making placement decisions at the application level enables us to either execute the entire application on the data source or remotely. Both are not ideal because large network transfer costs may incur significant battery drain [198] while running compute-intensive analytics tasks such as running deep learning models for classification in a video stream, may also incur large energy costs. Fine-grained placement enables the compute intensive classification operation to be placed in remote server nodes, while performing pre-processing steps such as filtering on the data source, to reduce network data transfer cost.

A critical requirement in designing fine-grained, energy-aware placement solutions is that we need to first estimate the energy costs of different placement configurations of application operators between data source and remote nodes. However, unlike compute cost estimates obtained by profiling an operator’s running time, estimating the power and thereby energy costs turn out to be more challenging to obtain efficiently. So, in this thesis, we focus primarily on addressing the problem of efficient estimation and prediction of operator power costs which can then be used by an energy-aware placement solution to optimize the energy costs of executing the application on the data source.

Consider a fixed amount of energy discharged from the battery due to the executing application on data source (e.g., 1% drop in battery state-of-charge, i.e., SoC, on a mobile device such as smartphone) referred to as *battery discharge interval*. A set of application operator instances will be scheduled on the mobile device, during the discharge interval. This set may consist of one or more physical operator instances, each representing an operator o_i in the application and running on the corresponding mobile data source node. An operator is said to be *active* at a given time instant if it has been scheduled to run by the underlying scheduler, and has not yet completed execution. Note that the scheduler can be instrumented to determine when an operator is scheduled, and track when it completes execution in the current scheduling interval. We define an *execution segment* as the time duration during which the composition of active operators does not change. Execution segments are non-overlapping.

Let the i^{th} discharge interval be divided into N_S^i segments such that each segment has a set of operators belonging to one or more operator types, which run concurrently and do not change for the duration of the entire segment.

We can define a sequence of segments in the i^{th} discharge interval as $SS_i = \{s_j^i \mid j \in \{1, \dots, N_S^i\}, s_j^i \subseteq V_o\}$ where s_j^i is the set of concurrently active operators in the j^{th} segment of i^{th} discharge interval. Also let $\mathcal{E}_j^i$ be the actual (i.e., ground-truth) energy that is consumed by segment s_j^i during time t_j^i that it was executing ($\mathcal{E}_j^i \in \mathbb{R}_{\geq 0}$, the set of real numbers). Now if $EMap$ is an estimated energy map from segments to their corresponding energy consumption values defined as $EMap : SS_i \times \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}_{\geq 0}$, $EMap(s_j^i, t_j^i) = ecost_j^i$, where $ecost_j^i$ is the energy consumption of segment s_j^i as it executes for duration t_j^i . Note that $i \in [1, 100]$ for a mobile device such as smartphone where each discharge interval corresponds to 1% battery SoC drop. Then the energy prediction problem is to find an energy map $EMap$ for which the following mean-squared error objective is minimized for each discharge interval i :

$$\text{Mean-squared error} = \frac{1}{N_S^i} \sum_{j=1}^{N_S^i} (\mathcal{E}_j^i - EMap(s_j^i, t_j^i))^2 \quad (4.7)$$

The goal of our power prediction technique is to measure coarse-grained, mobile device-level energy/power consumption values and attribute them to fine-grained operator-level application operators executing on the mobile device.

CHAPTER 5: ALGORITHMS AND ANALYSIS

In the previous chapter, we defined the general and near-data operator placement problems to compute efficient placement plans for type I and type II resource topologies. We also defined the energy estimation problem to enable the development of energy-aware placement strategies. We now look at algorithms we designed to efficiently compute a placement plan for the two types of resource topologies, as well as a measurement technique for power estimation on mobile devices.

5.1 GENERAL OPERATOR PLACEMENT FOR TYPE I RESOURCE TOPOLOGIES

For the general operator placement problem defined in Section 4.8, we start with making placement decisions for the single data source case, with $M = |V_o|$ application operators, for which placement needs to be computed. The design and analysis is then extended to the case with multiple data sources. We need to find a map Map from operators to resource nodes in edge and cloud nodes, such that the time to completely process an epoch $epoch_k$, $T(epoch_k, Map)$ is minimized.

Note that the chunk completion time $T(epoch_k, Map)$ can be defined in terms of the completion time of last data item d_I^k in $epoch_k$ when processed by the final operator o_M as:

$$\begin{aligned}
 T(epoch_k, Map) = t(d_I^k, o_M) = & \underbrace{t(d_I^k, o_{M-1})}_{\text{prev. operator completion}} + \underbrace{t_{queue}(d_I^k, o_M)}_{\text{queuing time}} + \\
 & + \underbrace{\mathbb{1}_{Map}(o_{M-1}, r_p) \cdot \mathbb{1}_{Map}(o_M, r_l) \cdot tr(p \xrightarrow{D_{o_{M-1}}} l)}_{\text{transmission cost}} + \underbrace{\mathbb{1}_{Map}(o_M, r_l) \cdot e_{M,l}}_{\text{execution cost}}
 \end{aligned} \tag{5.1}$$

where

$$\begin{aligned}
 \mathbb{1}_{Map}(o_i, r_l) & \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } o_i \text{ is placed on resource } r_l \\ 0 & \text{otherwise} \end{cases} \\
 t_{queue}(d_j^k, o_i) & \stackrel{\text{def}}{=} \text{queuing time of item } d_j^k \text{ on operator } o_i
 \end{aligned} \tag{5.2}$$

Here, $t(d_I^k, o_0) = 0$ is the base case for Eq.5.1 and it is the time just before first operator o_1 starts.

The major challenge to solve the operator placement problem is that $T(epoch_k, Map)$ (Eq. 5.1) depends on the mapping Map of all the operators that are running concurrently in a pipelined fashion. Hence, an exhaustive search of the solution space is required to determine

the optimal values of Map (in $O(2^M)$ time). There have been heuristics proposed, such as branch-and-bound [86], to decrease the search complexity. But in the worst case, it is still exponential in M .

5.1.1 NP-Completeness

Theorem 5.1. The general operator placement problem for type I resource topologies, to map application operators to resource nodes between edge and cloud layers, such that the streaming epoch completion time is minimized, is NP-hard.

Proof. Except for 3 restrictive cases which involve constraints on the resources/DAG structure, DAG scheduling in general is NP-complete [199]. Our problem does not meet any of the 3 conditions. We now transform a previous DAG scheduling problem that is proved to be NP-complete[200], into ours in polynomial time. Let $\mathcal{T} = \{T_1, T_2, \dots, T_p\}$ be a set of tasks that have partial order to form a DAG, and $\mathcal{R} = \{R_1, R_2, \dots, R_q\}$ be resources with each having bounded capacity $\mathcal{B} \forall j \in [1, q]$. For each task $T_i, i \in [1, p]$, it is associated with a latency time τ_i and capacity requirement $Cap(T_i, R_j) \leq B_j$ where $Cap(T_i, R_j)$ gives the capacity requirement of T_i when run on resource R_j . Previous work [200] has shown that checking the existence of a valid schedule from tasks $\mathcal{T}$ to resources $\mathcal{R}$ while meeting a deadline D , the resource capacity bounds and partial task ordering is NP-complete for more than two resources [201]. We can get the fastest schedule by testing different integer values of D to find the smallest with a valid schedule.

We transform this known NP-complete problem to our optimization problem, which is more complex, in polynomial time, as follows. We map each task $T_i \in \mathcal{T}$ to an operator $o_i \in V_o$, with running time τ_i replacing the computation, communication and queuing cost of o_i . The task dependencies are the dataflow dependencies between operators in E_o . We also transform the resource bounds B_j for resource $R_j \in V_R$ into the compute bounds in constraint 4.4 above. Given the placement mapping Map , we can test its validity as a solution by checking constraints 4.3 and 4.4 at each operator and finding critical path in the DAG in $\mathcal{O}(|E_o| + |V_o| + (|V_R| \times |V_o|) + (|V_o| + |E_o|))$, which is $\mathcal{O}((|V_R| \times |V_o|) + |E_o|)$. This is polynomial and hence shows that our optimization problem is NP-complete.

We note that similar hardness results have been described for scheduling problems in literature [201]. Next, we propose a heuristic algorithm to solve the problem in polynomial time with respect to the number of operators. QED.

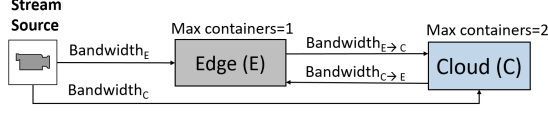


Figure 5.1: Example resource graph.

5.1.2 Proposed Approach for General Operator Placement in Type I Resource Topologies

The key idea in our approach lies in the fact that due to the limited number of resources in edge and cloud layers, the number of operators that can be concurrently processing data items in a pipeline fashion is limited. For example, if we have a DAG of 10 operators executing on two physical resources, each having two containers, then we can assume that up to 4 operators can be concurrently executing in parallel and the rest of the operators will wait until at least one of the first 4 operators finishes processing an epoch of data items and releases the container.

It may seem like fully utilizing all the available containers by running 4 operators concurrently can minimize the chunk completion time. However, doing so introduces transmission cost when two dependent operators are placed on containers that are running on different resources. Hence, depending on trade-off between execution and transmission costs, we might end up with a situation that is ideal to execute the 10 operators on only two containers that are collocated on the same physical resource. Based on the above intuition, our problem becomes choosing the operators that can run concurrently, so that the end-to-end computation and communication times are reduced, as will be illustrated in the following example.

Illustrative Example

As shown in Figures 4.4 & 5.1, let's say we have a sequence of three operators o_1, o_2, o_3 belonging to the dataflow graph G and two different resources: (1) one edge device E with one container, and (2) one cloud node C with two containers. We assume that intermediate output data can flow from E to C or vice versa. In this case, pipelining can happen for at most three operators which is equivalent to the total number of containers in the system. Each possible solution to the operator placement problem requires mapping the three operators to E or C , and up to two operators can be mapped to C (many-to-one) since it has two containers. For each placement decision however, we can have different subsets of operators running concurrently in pipeline fashion. We represent the combinatorial possibilities of placing the 3 operators across E and C as shown in Figure 5.2. Each row in the figure is a possible placement solution. The label $o_1@E$ means that o_1 is placed on device E (i.e., $Map(o_1) = E$), while the label $o_1@C$ means that o_1 is placed on device C (i.e., $Map(o_1) = C$).

1. $(o_1@E); (o_2@E); (o_3@E)$
2. $(o_1@C); (o_2@C); (o_3@C)$
3. $(o_1@E); (o_2@E \parallel o_3@C)$
4. $(o_1@C); (o_2@C \parallel o_3@C)$
5. $(o_1@C); (o_2@C \parallel o_3@E)$
6. $(o_1@E \parallel o_2@C); (o_3@E)$
7. $(o_1@E \parallel o_2@C); (o_3@C)$
8. $(o_1@C \parallel o_2@C); (o_3@C)$
9. $(o_1@C \parallel o_2@E); (o_3@E)$
10. $(o_1@C \parallel o_2@E); (o_3@C)$
11. $(o_1@E \parallel o_2@C \parallel o_3@C)$
12. $(o_1@C \parallel o_2@E \parallel o_3@C)$
13. $(o_1@C \parallel o_2@C \parallel o_3@E)$

Figure 5.2: Different possibilities for operator placement

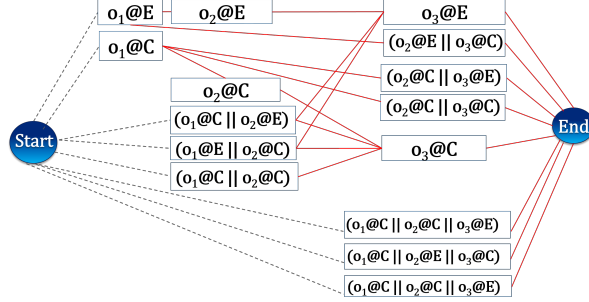


Figure 5.3: Placement graph corresponding to Fig. 5.2

The operators inside parentheses show the operators that are being executed concurrently in a pipelined manner as shown in Figure 4.7. More precisely, the parentheses denote how a given epoch of data items is executed and we describe the two general cases:

- **Pipelined Execution:** if we have a parentheses with two or more operators such as $(o_1@E \parallel o_2@C)$, then the operators are running in different containers and the execution goes as follows: (1) d_1^k gets processed by o_1 in resource E , (2) once d_1^k finishes execution, d_2^k starts being processed by o_1 in resource E while d_1^k is being processed by the next operator o_2 in resource C . Hence, o_1 and o_2 keep processing operators in pipeline fashion until o_2 processes d_I^k .
- **Sequential Execution:** If the parentheses have only one operator such as $(o_1@E)$, then o_1 processes all the data items in the epoch. The occurrence of an operator in the next parentheses separated by ';', such as $(o_1@E); (o_2@E)$, indicates that o_2 starts execution after o_1 processes the epoch. This behavior typically happens when the second operator o_2 is placed on the same resource E and the resource has one container. Hence, o_2 has to wait for o_1 to finish execution.

Since we have a total of 3 containers across the edge device and cloud, each parentheses can have at most 3 operators that can run concurrently.

The representation in Figure 5.2 for the placement of operators on resources, helps us draw an analogy between the operator placement and the matrix chain ordering problem (MCOP) [202]. This analogy and its significance in reducing complexity of operator placement is explained in the next section 5.1.2.

Solution

We first note that there exists a dynamic programming algorithm for solving MCOP in polynomial time. MCOP is defined as follows: given a sequence of matrices, find the most efficient way to multiply the matrices together. Because matrix multiplication is associative, no matter how we parenthesize the product, the result will be the same. However, the way in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product. Similarly, in our operator placement problem, the way in which we parenthesize operators affects the computation and communications costs. It turns out that the solution of MCOP is also the solution to our distributed operator placement problem.

Instead of enumerating the combinatorial solutions in Figure 5.2, we construct a *placement graph* as shown in Figure 5.3. Each node represents one unique parenthesis in Figure 5.2 and is labeled by one operator or multiple operators running in parallel to complete processing an epoch. An edge from node i to node j in the placement graph corresponds to the ending of one pair of parenthesis and the beginning of the next one in Figure 5.2. It represents the computation cost of operators in node i , the communication cost to transmit output data from node i to node j , and queuing delay at node j . We discuss more details about constructing the placement graph in Section 5.1.2.

The shortest path between the start and the end nodes is the solution of the distributed operator placement problem. For example, if the shortest path is $[(o_1@E \parallel o_2@C); (o_3@C)]$ then o_1 is placed on device E while o_2 and o_3 are placed on device C . The intuition behind reduction in the placement problem complexity is that the cost for each parentheses is calculated once even though the cost might be shared between multiple paths. If you look at Figure 5.2, the parentheses $(o_3@C)$ appears in lines 7-10 because it is shared between 4 different paths. However, in the placement graph (Figure 5.3, $(o_3@C)$ is represented as one node and its cost is calculated only once for the 4 paths.

Algorithm Steps

The algorithm can be described in 4 main steps:

Step 1: Topological Sort For a given dataflow graph G , we sort operators in a topological

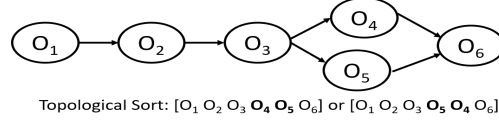


Figure 5.4: Example Operator graphs and their topological order.

order as shown in Figure 5.4. Topological order has the property that parent nodes appear before their successors, so the order of operator execution is maintained. If a node in the graph has multiple successors, the successors can be ordered arbitrarily. We denote this ordered list of operators as $OpList = \langle o_1, o_2 \dots, o_M \rangle$

We note that by doing this, we still maintain the original dependency relationship between operations from graph representation in Figure 5.4.

Step 2: Create Placement Graph The list $OpList$ obtained in Step 1 has a sequence of operators that follow certain execution order, which has an analogy with a chain of matrix multiplications. We use parenthesis around adjacent operators in $OpList$ to create unique sublists of size up to the number of containers in all resources. The unique sublists S for $OpList = \langle o_1, o_2, o_3 \rangle$ are:

$$S = \{ \underbrace{(o_1), (o_2), (o_3)}_{\text{parentheses of size 1}}, \underbrace{(o_1 \ o_2), (o_2 \ o_3)}_{\text{parenthesis of size 2}}, \underbrace{(o_1 \ o_2 \ o_3)}_{\text{parenthesis of size 3}} \} \quad (5.3)$$

We define a function $P(s)$ that denotes the set of possible placements of sublist $s \in S$. For example:

$$\begin{aligned} P(s = (o_1)) &= \{o_1@E, o_1@C\} \\ P(s = (o_1 \ o_2)) &= \{(o_1@E \parallel o_2@C), (o_1@C \parallel o_2@E), \\ &\quad (o_1@C \parallel o_2@C)\} \end{aligned} \quad (5.4)$$

We use the placement sets to generate a placement graph $G_p = (V_p, E_p)$ (See Figure 5.3) where the vertices are the union of all the placement sets: $V_p = \cup_s P(s)$.

Step 3: Add Placement Graph Edges and Edge Weights: Once we have the nodes as shown in Figure 5.3, we start by adding edges from *START* node to all nodes that start with first operator in $OpList$, e.g. o_1 . Similarly we add edges for all nodes that end with last operator in $OpList$ e.g. o_3 in the figure, to an *END* node. To add edges between intermediate nodes, we look for parent nodes that end with operator i , and we look for the candidate child nodes that starts with operator j , where j is the successor of i in $OpList$. Next we check the resource that operator j is placed on. If it is placed in a resource that appears in the parent node, we add an edge between the parent node and the candidate child node. The

intuition behind adding an edge is that if there is an overlap between compute resources used for the child node and the parent node then the child node has to wait for the resources to be free before it starts execution. Therefore, the weight on the edge denotes the computation, communication. We discuss the details of calculating edge weights in Section 5.1.2

Step 4: Shortest Path: Find the shortest path from *START* to *END* node in the placement graph constructed in Steps 2 and 3. The shortest path gives the full solution of placement and the consumed containers in each resource.

Cost Calculation

In this section, we describe how to calculate the overall computation and communication cost of all operators inside the parentheses such as calculating the cost of $(o_1@E \parallel o_2@C)$ in the path $[(o_1@E \parallel o_2@C); (o_3@C)]$ from Figure 5.3. We note that due to sequential execution (denoted by $;$), the cost of operators inside one pair of parentheses is added to the cost of operators inside the next pair. Hence, in this section we focus on calculating the cost for one pair of parentheses.

Dependent Operators: In this example we show the cost calculation $(o_1@E \parallel o_2@C)$. Figure 4.7 illustrates the execution of $(o_1@E \parallel o_2@C)$ and the cost is given by:

$$\underbrace{(I-1)e_{1,E}}_{\text{queuing delay}} + \underbrace{e_{1,E} + e_{2,C}}_{\text{execution cost}} + \underbrace{tr(E \xrightarrow{D_{o_1}} C) + tr(C \xrightarrow{D_{o_2}} E)}_{\text{transmission cost}} \quad (5.5)$$

The cost is composed of queuing, execution and transmission costs as illustrated in Figure 4.7, There is an additional term in the transmission cost $tr(C \xrightarrow{D_{o_2}} E)$, which specifies the transmission cost for the output of the current parentheses. The queuing delay in the previous equation is derived based on the assumption that $e_{1,E} > tr(E \xrightarrow{D_{o_1}} C), tr(C \xrightarrow{D_{o_2}} E), e_{2,C}$. The general formula for queuing delay is:

$$(I-1) \cdot \max(e_{1,E}, tr(E \xrightarrow{D_{o_1}} C), e_{2,C}, tr(C \xrightarrow{D_{o_2}} E)) \quad (5.6)$$

The queuing delay is dominated by the operator with the maximum computation cost or the maximum communication cost between two operators.

Fork Node: From Figure 5.4, we now consider the cost of $(o_3@E \parallel o_4@C \parallel o_5@C)$, where the parentheses contain a fork node (i.e., a node with multiple successors) such as o_3 . Since the fork node splits the DAG into two or more branches, the cost is given by considering the maximum cost across the different branches so in this example the cost will be the

maximum of: (1) Computation and communication costs of o_3 and o_4 , and (2) Computation and communication cost of o_3 and o_5 .

Summary of Algorithm Analysis

We first provide a summary of the proposed algorithm's computational complexity. We then provide a detailed explanation of the derivation in Section 5.1.3.

Among the 4 steps described in section 5.1.2, there are two steps that dominate the complexity of our placement algorithm: (1) topological sort of input operator graph (Step 1), (2) the shortest path calculation (Step 4) on placement graph. The complexity of topological sort [203] is $O(|V_o| + |E_o|)$, where $|V_o|$ and $|E_o|$ are the number of operators and number of links in the operator graph, respectively. The complexity of Dijkstra's algorithm [204] for shortest path calculation when executed on placement graph is: $O(|V_p| \log(|V_p|) + |E_p|)$, where V_p and E_p represent the set of vertices and links between vertices of the placement graph, respectively. For simplicity of analysis, we assume that we have $L = |V_R|$ resources each having one container, hence each vertex in the placement graph can have up to L operators in the same parentheses. $|V_p|$ is calculated as the number of sublists of size k multiplied by the number of possible placements of operators in each sublist (See Step 2 in section 5.1.2). In case of L resources and a bracket of size k operators, the number of possible placements is equivalent to the number of ways to obtain an ordered subset of k elements, from a set of L elements i.e. the number of permutations ${}^L P_k$. Hence $|V_p|$ is defined as:

$$|V_p| = \sum_{k=1}^L (|V_o| - k + 1) \cdot {}^L P_k \quad (5.7)$$

We show in Section 5.1.3 below that $\sum_{k=1}^L |V_o| \cdot {}^L P_k$ can be approximated as $|V_o| \cdot L!$. We further show that $|E_p| = |V_p| \cdot L!$. Hence, the overall complexity of our placement algorithm can be given by: $O(|V_o| \cdot L! \cdot (\log(|V_o| \cdot L!) + L! \cdot L) + |E_o|)$

5.1.3 Detailed Complexity Analysis

Given a dataflow graph $G_o = (V_o, E_o)$ and resource graph $G_R = (V_R, E_R)$ as input, in order to determine the complexity of our algorithm, we need to find the cost of (1) topological sort, (2) generating the vertices and edges in our placement graph, (3) computing edge weights and (4) running shortest path algorithm for the constructed placement graph.

For the graph G_o , topological sort can be done in $\mathcal{O}(|V_o| + |E_o|)$. So we can get a list

of operators *OpList* from this first step in our algorithm. Let $f_s(OpList, k)$ be a function which computes the count of distinct sublists of size k , from list *OpList*. It follows that $f_s(OpList, k)$ can be written as:

$$f_s(OpList, k) = |OpList| - k + 1 \quad (5.8)$$

where $|OpList|$ denotes the length of list *OpList*. Also note that

$$|OpList| = |V_o| \quad (5.9)$$

Now, given L is the number of resources available for placement, the number of vertices $|V_p|$ that can be generated in our placement graph from the list *OpList* is (from equations 5.8 and 5.9):

$$|V_p| = \sum_{k=1}^L (f_s(OpList, k) \cdot {}^L P_k) = \sum_{k=1}^L |V_o| \cdot {}^L P_k - \sum_{k=1}^L k \cdot {}^L P_k + \sum_{k=1}^L {}^L P_k \quad (5.10)$$

As mentioned in previous section, ${}^L P_k$ represents permutations for the number of ways to obtain an ordered subset of k elements, from a set of L elements.

From [205], we know the summation on permutations in first and last terms of equation 5.10 evaluates to:

$$\sum_{k=1}^L {}^L P_k \approx L! \cdot e \quad (5.11)$$

We can also show that the second term in equation 5.10 evaluates to:

$$\begin{aligned} \sum_{k=1}^L k \cdot {}^L P_k &= L! \cdot \left[\frac{L}{0!} + \frac{L-1}{1!} + \cdots + \frac{1}{(L-1)!} \right] \\ &= L! \cdot \left[L + \frac{L}{1!} + \cdots + \frac{L}{(L-1)!} - \left[\frac{1}{1!} + \frac{2}{2!} + \cdots + \frac{L-1}{(L-1)!} \right] \right] \\ &= L \cdot (L! \cdot e) - L! \cdot \left[\frac{1}{1!} + \frac{2}{2!} + \cdots + \frac{L-1}{(L-1)!} \right] \quad \text{--- from Equation 5.11} \\ &= L \cdot (L! \cdot e) - L! \cdot \left[1 + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{(L-2)!} \right] \\ &= L \cdot (L! \cdot e) - L! - L! \cdot \left[\frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{(L-2)!} \right] \end{aligned} \quad (5.12)$$

$$\begin{aligned}
\sum_{k=1}^L k \cdot {}^L P_k &= L \cdot (L! \cdot e) - L! - [L! \cdot e - L! - {}^L P_1] && \text{--- from Equation 5.11} \\
&= L \cdot (L! \cdot e) - L! \cdot e + {}^L P_1 \\
&= (L! \cdot e) \cdot (L - 1) + \frac{L!}{(L - 1)!} \\
&\approx (L - 1)[L! \cdot e] + L
\end{aligned} \tag{5.13}$$

From equations 5.10, 5.11 and 5.13, we are able to show that the number of vertices $|V_p|$ can be written as:

$$|V_p| \approx \mathcal{O}(|V_o| \cdot L!) \tag{5.14}$$

We now need to determine the number of edges in our placement graph. Based on our proposed algorithm, the following constraint on the first operator of a child node has to be satisfied, for it to share an edge with its candidate parent: the resource on which first operator of a child node is placed, should appear in the candidate parent node. Based on this constraint, the number of such child nodes $n_{childnodes}^k$ for a vertex with k operators is determined to be:

$$n_{childnodes}^k = \sum_{p=1}^{|V_o|-k} k \cdot {}^{L-1} P_{p-1} \tag{5.15}$$

Equation 5.15 gives the number of edges for each vertex with k operators in our placement graph. Combining equations 5.10 and 5.15 and assuming the number of operators is greater than the number of resources i.e. $|V_o| \gg L$, the total number of edges is:

$$\begin{aligned}
|E_p| &= \sum_{k=1}^L (f_s(OpList, k) \cdot {}^L P_k \cdot (\sum_{p=1}^{|V_o|-k} k \cdot {}^{L-1} P_{p-1})) \\
&\approx \sum_{k=1}^L (f_s(OpList, k) \cdot {}^L P_k \cdot k \cdot (L - 1)! \cdot e) && \text{--- from Eq. 5.11} \\
&= ((L - 1)! \cdot e) \cdot [|V_o| ((L - 1)(L! \cdot e) + L) - \sum_{k=1}^L k^2 \cdot {}^L P_k + (L - 1)(L! \cdot e) + L]
\end{aligned} \tag{5.16}$$

The last step comes from substituting the summation term from Equation 5.13 and Equation 5.8. Using equation 5.11 and expanding square terms below, we now observe the following result to expand equation 5.16:

$$\begin{aligned}
\sum_{k=1}^L k^2 \cdot {}^L P_k &= L! \cdot \left[\frac{1^2}{(L-1)!} + \frac{2^2}{(L-2)!} + \cdots + \frac{L^2}{0!} \right] \\
&= L! \cdot \left[\frac{(L-(L-1))^2}{(L-1)!} + \frac{(L-(L-2))^2}{(L-2)!} + \cdots + \frac{(L-(0))^2}{(0)!} \right] \\
&= L! \left[L^2 \left[1 + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{(L-1)!} \right] - 2 \cdot L \cdot \left[\frac{1}{1!} + \frac{2}{2!} + \cdots + \frac{L-1}{(L-1)!} \right] + \right. \\
&\quad \left. \left[\frac{1^2}{1!} + \frac{2^2}{2!} + \cdots + \frac{(L-1)^2}{(L-1)!} \right] \right] \\
&\approx L^2 \cdot (L! \cdot e) - 2 \cdot L \cdot L! \left[e - \frac{1}{(L-1)!} \right] + \\
&\quad \left[1 + \frac{1}{1!} + \frac{2}{2!} + \cdots + \frac{L-2}{(L-2)!} + \frac{1}{1!} + \frac{2}{2!} + \cdots + \frac{L-2}{(L-2)!} \right] L! \quad (\text{from Eq. 5.11}) \\
&= L^2 \cdot (L! \cdot e) - 2 \cdot L \cdot L! \left[e - \frac{1}{(L-1)!} \right] + \\
&\quad \left[L! + \left[\sum_{k=1}^L (L-k) \cdot {}^L P_k - (L-1) \cdot L \right] + \sum_{k=1}^L {}^L P_k - L! - L \right] \\
&= L^2 \cdot (L! \cdot e) - 2 \cdot L \cdot L! \left[e - \frac{1}{(L-1)!} \right] + \left[L \cdot \sum_{k=1}^L - \sum_{k=1}^L k \cdot {}^L P_k - L^2 + L + {}^L P_k - L \right] \\
&= L^2 \cdot (L! \cdot e) - 2 \cdot L \cdot L! \left[e - \frac{1}{(L-1)!} \right] + [{}^L P_k \cdot [L+1] - \sum_{k=1}^L k \cdot {}^L P_k - L^2] \\
&= L^2 \cdot (L! \cdot e) - 2 \cdot L \cdot L! \left[e - \frac{1}{(L-1)!} \right] + [L! \cdot e[L+1] - (L-1) \cdot (L! \cdot e) + L - L^2] \\
&= L^2 \cdot (L! \cdot e) - 2 \cdot L \cdot L! \cdot \left[e - \frac{1}{(L-1)!} \right] + 2 \cdot L! \cdot e - L^2 + L \\
&= L^2 \cdot (L! \cdot e) - 2 \cdot L \cdot L! \cdot e + 2 \cdot L \cdot \frac{L}{(L-1)!} + 2 \cdot L! \cdot e - L^2 + L \\
&= L^2 \cdot (L! \cdot e) - 2 \cdot L \cdot L! \cdot e + 2 \cdot L^2 + 2 \cdot L! \cdot e - L^2 + L \\
&= (L^2 - 2 \cdot L) \cdot (L! \cdot e) + L^2 + L + 2 \cdot L! \cdot e \\
&= (L^2 - 2 \cdot L + 1) \cdot (L! \cdot e) + L^2 + L + L! \cdot e \\
&= ((L-1)^2 + 1) \cdot L! \cdot e + L^2 + L
\end{aligned} \tag{5.17}$$

The fourth step in equation 5.17 is obtained by simply expanding the factorial terms in the previous step. Using equation 5.17, we can expand the terms in equation 5.16 to be:

$$\begin{aligned}
|E_p| &= (L-1)! \cdot e[(|V_o| + 1)((L-1) \cdot (L! \cdot e) + L) - (((L-1)^2 + 1)(L! \cdot e) + L^2 + L)] \\
&= (L-1)! \cdot e[|V_o| \cdot L(L! \cdot e) - |V_o| \cdot (L! \cdot e) + L \cdot |V_o| + L - L^2 \cdot (L! \cdot e) + 3L \cdot (L! \cdot e) \\
&\quad - 3(L! \cdot e) - L^2 - L]
\end{aligned} \tag{5.18}$$

Looking at the dominant terms in the above equation for $|E_p|$, it can be computed in $\approx \mathcal{O}(|V_o| \cdot (L!)^2)$.

For each edge, we need to compute its edge weight. If an edge has source vertex with k operators, the execution time of each operator combined with transmission and queuing delays between operators results in k operations that need to be performed for calculating its edge weight. So the cost of computing all edge weights from equation 5.16 would be:

$$c_{edgeweight} = \sum_{k=1}^L (f_s(OpList, k) \cdot {}^L P_k \cdot k^2 \cdot (L-1)! \cdot e) \tag{5.19}$$

We can expand terms in equation 5.19 similar to how we did it for equation 5.16 to show that it can be done in $\mathcal{O}(|V_o| \cdot (L!)^2 \cdot L)$

Following are costs for individual steps in our algorithm: (1) topological sort ($\mathcal{O}(|V_o| + |E_o|)$ where $|V_o|$ and $|E_o|$ are the number of operators and links in operator graph respectively), (2) generating vertices ($\mathcal{O}(|V_o| \cdot L!)$), (3) generating edges ($\mathcal{O}(|V_o| \cdot (L!)^2)$), (4) computing edge weights ($\mathcal{O}(|V_o| \cdot (L!)^2 \cdot L)$), (5) running shortest path algorithm [204] on constructed placement graph ($\mathcal{O}(|V_o| \cdot L! \cdot \log(|V_o| \cdot L!) + |V_o| \cdot (L!)^2)$). The overall complexity of our algorithm can thus be given as:

$$\mathcal{O}(|V_o| \cdot (L!) \cdot (\log(|V_o| \cdot L!) + L! \cdot L) + |E_o|) \tag{5.20}$$

Limitation of our approach: Going back to Figure 5.3, our approach has an implicit assumption that when o_1 and o_2 are executed concurrently in $(o_1@E \parallel o_2@C)$, they will fully utilize available resources and that's why o_3 in $(o_3@C)$ has to wait for both operators to finish execution before starting execution on device C . An improved model would account for the fact that o_3 can start right after the fastest of o_1 and o_2 and provide better resource utilization. We plan to improve this model as part of future work.

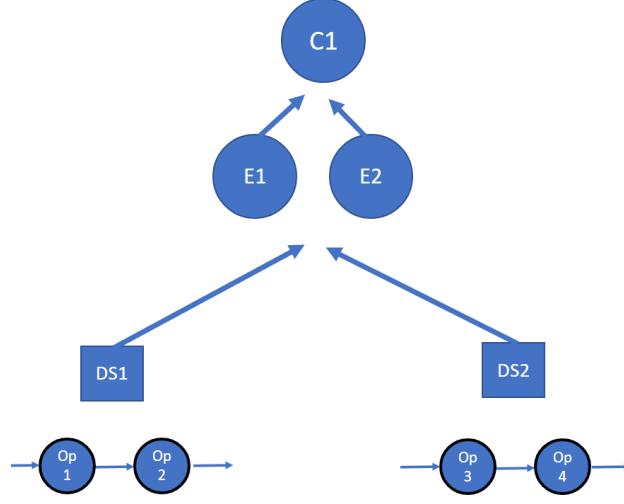


Figure 5.5: Example application and resource graph for our proposed approach to handle multiple data source nodes. DS: data source, E1 and E2: two edge nodes, C1: single cloud node

5.1.4 Extending to Multiple Data Sources

In the previous section, we considered one data source node and therefore, one instance of the application graph with $|V_o|$ operators for which placement decision needs to be made. However, for a large scale service handling data from multiple data sources, each data source may be generating data items at different rates. E.g., a camera at a busy traffic intersection may be set at a higher frame rate compared to another camera with relatively slower traffic patterns at a certain time of the day. Furthermore, the network bandwidth and consequently the network transfer cost for the raw data from each node, may vary across data sources. These factors have an impact on determining which operator instances from across data source nodes need to be parallelized, and the resource nodes on which the operator instances need to be placed, to minimize the overall application processing time.

A single instance of the application may not fully utilize all the available resources and meet the data processing needs. Consider a single instance of a video processing application with operators distributed across edge and cloud nodes. If a single data source node generates 10 frames in an epoch, the application will process the current epoch completely before processing frames from the next epoch. If we now consider 2 data source nodes, each generating an epoch of 10 frames in parallel, the single application instance will still process the epochs one after another in our execution model. If there were resource nodes available, we miss the opportunity to process video frames from different source nodes in parallel. Thus, as we increase the data sources, more data results in degradation of processing time experienced

by larger fractions of data sources. Parallelizing the operations can help reduce the overall application performance. To elaborate on the effects on parallelism with the addition of more data source nodes, consider two data sources as shown in Figure 5.5, with an instance of the application graph replicated on incoming data from each source node. Each data source generates data at different rates, so the number of data items in the epoch is different; $n_1 = 10$ for the number of data items in an epoch for the first data source, while $n_2 = 50$ for the number of data items in an epoch for the second data source. Let's consider the following values for the execution cost variables involved, $t_{1E} = 10s, t_{2E} = 4.5s, t_{1C} = 1s, t_{2C} = 4s$. To simplify the explanation, we will assume the network costs are negligible between data source, edge and cloud in our example. Initially, there is just one data source with operators o_1 and o_2 . Considering all the possible placements between edge and cloud, we can compute their costs using the equations in Section 5.1.2. The possible placements are $o_1@C_1; o_2@C_1, o_1@E_1; o_2@E_1, o_1@E_1||o_2@E_2, (o_1@E_1||o_2@C_1)$ and $(o_1@C_1||o_2@E_1)$. The placement with the lowest latency cost is $o_1@C_1||o_2@E_1$ with a cost of 46s. So our placement algorithm executes this placement. Now a new data source gets added, the operator execution costs are similar to those for the first data source, except that the number of data items in an epoch n_2 is 50. However, if we choose to execute the same placement decision without rerunning our placement algorithm, then the placement would look as follows: $(o_1@C_1||o_2@E_1); (o_3@C_1||o_4@E_2)$. The sequential processing of data from one data source after another is because there is a single cloud resource available. Now the cost of this placement is 272s because the cost of executing operators from the second data source is 2260s ($2260+460=2720$). However, consider an alternative placement as follows: $o_1@C_1; (o_2@E_1||o_3@C_1||o_4@E_2)$. The cost of this placement turns out to be 236s, which is lower and more desirable than the placement which cost 272s. The reason for the lower cost is because our approach was able to find a placement where it parallelized the execution of operators from second data source with the second operator from first data source, to fully utilize the 3 nodes available in resource topology. This placement solution is represented as $o_1@C_1; (o_2@E_1||o_3@C_1||o_4@E_2)$. Furthermore, it avoided the pitfall of executing o_1 on E_1 or E_2 , which is expensive and incurs a cost of 10s. We would have never obtained this new placement strategy when a new data source was added, unless we reran the placement algorithm to explore this strategy. Based on this discussion, we focus on extending our approach to determine the operator instances from each data source to be parallelized, along with their placement on the resource nodes. In summary, every time a new data source is added, it is critical to rerun the placement algorithm to obtain a placement strategy with the goal to minimize epoch processing time. Running our placement algorithm at runtime means that it needs to be efficient, and scale to large number of data sources. Otherwise, we will experience degraded performance and may not be fully utilizing available

resources until our algorithm finds a better placement strategy. Such a condition may be catastrophic if the input rates are much higher than the rate at which our application is able to process the incoming data stream.

To extend our placement algorithm to handle multiple data sources, a valid topological sort of the operators for step 1 of our algorithm can be considered in which each of the application instances are placed one after the other in a sequence, and then operators of each instance are topologically sorted. Based on the physical operator graph constructed in Figure 4.8, we can see that topologically sorting it would take $\mathcal{O}((|V_o| + |E_o|) \cdot N_d)$ steps. Step 2 of the algorithm remains the same, in which we construct a placement graph based on the topologically sorted list from step 1. The number of nodes in the placement graph is $\mathcal{O}(N_d \cdot |V_o| \cdot L!)$. Step 3 adds edges between nodes in the placement graph when there is a resource dependency between the two nodes. So, if the first operator in a candidate child node is placed on a resource node which appears in any of the resource nodes on which the operators in the parent node are placed, then there is a resource dependency. This is because, the operators in the parent node need to be fully processed, before the operator in the candidate child node can use the shared resource node. Note that this step does not change even if there is no dataflow dependency between operators in the original operator graph, e.g., if the 2 adjacent nodes in the topologically sorted operator list belong to different data sources. They still have a resource dependency due to the shared resource node between the parent and candidate child node in the placement graph. The number of edges is $\mathcal{O}(N_d \cdot |V_o| \cdot (L!)^2)$. Similarly, edge weights can be calculated in $\mathcal{O}(N_d \cdot |V_o| \cdot (L!)^2 \cdot L)$. The overall complexity is $\mathcal{O}(N_d \cdot |V_o| \cdot L! \cdot (\log(N_d \cdot |V_o| \cdot L!) + L \cdot L) + |E_o|)$. Thus, we are still log-linear in the number of data source nodes.

Given $f_{\maxparallel}$, our algorithm jointly determines the number of instances of each operator to execute in parallel as well as the placement of each of the instances on the resource nodes between edge and cloud nodes. Each data source has an instance of the application operators running on the incoming data so we have adopted a simple data splitting strategy in which we assign the data items in the input stream to be processed by operator instances corresponding to the data source node on which the stream was generated. More interesting splitting strategies can be explored as part of future work. E.g., grouping frames together in video analytics application and co-locating their processing based on traffic camera location can help identify spatial patterns/trends that are localized to a particular site. Colocating the processing of such frames using the same physical operators can help reduce network transfer costs.

5.2 GENERAL OPERATOR PLACEMENT ALGORITHM FOR TYPE II RESOURCE TOPOLOGIES: CHALLENGES

The cost of solving the placement problem for type I resource topologies depends on the number of nodes in the placement graph (see Section 5.1.3). If we consider extending our proposed type I placement solution to type II resource topologies, it results in a large number of nodes in the placement graph which makes it prohibitively expensive to compute a placement plan at runtime. Below, we provide more details on this issue:

Theorem 5.2. The number of nodes $|V_p|$ in the placement graph, when the edge-cloud placement heuristic for type I resource topologies, is applied to type II resources which includes compute nodes at the data source layer, has the following lower bound:

$$|V_p| > \frac{((N_d \cdot |V_o|) + L)!}{L! \cdot (N_d \cdot |V_o|)!} \quad (5.21)$$

Proof. Consider the placement graph nodes when placement needs to also be considered at the data source layer. Let's consider a case of k operators running in parallel on resource nodes, then we have to determine the number of operators which can execute at the corresponding data source layer, and the rest need to be placed between remote edge and cloud layers. So for each placement solution, we pick a given number of operators to push to remote nodes, and the rest are executed locally. This gives us the following relation:

$$|V_p| > \sum_{i=0}^L {}^k C_i {}^L P_i \quad (5.22)$$

The above equation comes from picking i operators out of k , to send to remote nodes (i.e., edge and cloud nodes) for execution, and then permuting them between the L containers available between edge and cloud nodes. Now, the placement graph node with the largest number of operators running in parallel is when $k = |V_o| \cdot N_d$ where all the operators are in the same placement graph node running in parallel. In this case, we have

$$|V_p| > \sum_{i=0}^L {}^{|V_o| \cdot N_d} C_i {}^L P_i \quad (5.23)$$

$$> \sum_{i=0}^L {}^{|V_o| \cdot N_d} C_i {}^L C_i i! \quad (5.24)$$

We note that dropping $i!$ in the above equation, which is a positive term, should not change

the inequality as we are looking for a conservative lower bound. We obtain the following:

$$|V_p| > \sum_{i=0}^L |V_o|^{N_d} C_i^L C_i \quad (5.25)$$

$$> (N_d \cdot |V_o|)^L C_L \quad (5.26)$$

$$> \frac{((N_d \cdot |V_o|) + L)!}{L! \cdot (N_d \cdot |V_o|)!} \quad (5.27)$$

The reduction in the above step follows from the Vandermonde's identity [206] in combinatorics.

QED.

The factorial term in the number of data source nodes, makes the solving of the placement graph very expensive. Considering a real-world case study described in Chapter 7 as an illustrative example, we note that server monitoring pipelines in large scale datacenters need to handle data generated from up to 250,000 data source nodes sending monitoring data to be processed by a cluster of up to 1000 nodes between the edge and cloud layers. At full scale, even if we have a small monitoring query running per data source node with just 2 operators for each query instance, we have $N_d \cdot |V_o| = 250000 * 2 = 500000$ operators and $L = 1000$ resource nodes. If solve for a subset of the available resource nodes, with 250 data source nodes sharing 3 containers running on an edge node, we will have more than 21M nodes in the placement graph. Current results in Section 6.4 show that even with 1000 operators, it can take up to 10 seconds for a placement decision to be made. The corresponding placement graph will have number of nodes in the same order. When considering data source nodes as well for placement, handling nodes up to 4 orders of magnitude more of placement graph nodes can become prohibitively expensive. Such performance is not acceptable for the fast placement adaptation requirements in the order of minutes or even 10s of seconds on server nodes in datacenters, as we describe in Section 7.2.2.

Other challenges with applying the proposed type I topology placement heuristic, for type II resource topologies include:

- The need for fine-grained data-level partitioning for each of the application operators becomes critical, given the very limited compute resources on data source nodes. The type II heuristic currently uses operator-level partitioning which is insufficient to effectively use the available compute resources on data source nodes. More details are described in Section 5.3.1.
- The placement solution for type II topologies is a *cost model-based heuristic*, which builds an execution model based on resource usage costs of operators and solves an

optimization problem to obtain a placement solution. Such techniques rely on accurate profiling estimates being available to the placement algorithm. However, as we will see in Section 5.3.1 and Chapter 7, profiling costs can be very high on data source nodes, making it challenging to obtain accurate execution costs of operators at runtime. This inaccuracy limits the ability of placement algorithms to generate good placement plans.

- In WAN systems, the network transfer costs can cause the coordination overheads for exchanging control messages (e.g., operator profiling costs) between data source nodes and remote edge, cloud resource nodes [98] to be prohibitively expensive for the fast adaptation requirements of real-world systems with type II resource topologies.

In summary, the proposed heuristic for type I resource topologies does not scale when considering large number of data sources for placement in a type II topology. This is because, apart from the possible placements on edge and cloud resource nodes that need to be considered in a type I topology, additionally there are possible operator combinations that can be concurrently scheduled at their corresponding data source nodes. The main challenge is the additional resource nodes to consider and the possible combinations of operators that can execute concurrently on the nodes. So, we now seek to understand if considering all the resource nodes, with a global view of the resource topology when making placement decisions, is necessary, or can we compute effective placement plans by limiting the number of considered resource nodes? Our goal is to reduce network transfer costs to the remote edge and cloud nodes from data sources to alleviate network and compute bottlenecks. So, we consider greedily placing the operators as close to the data source as possible, at each layer of the type II resource topology.

5.3 NEAR-DATA OPERATOR PLACEMENT ALGORITHM FOR TYPE II RESOURCE TOPOLOGIES

To deal with the problem of efficient placement in type II resource topologies, we earlier defined a different placement problem for near-data operator placement at the data source layer . To recap– the problem dealt with minimizing for each data source, the number of operators that are sent to a remote node for execution, without sacrificing the overall application processing time as a result of the operators executing locally on the data source. Unfortunately, solving this placement problem is also NP-hard.

Theorem 5.3. The problem of data source-level operator placement across data source, edge and cloud nodes, with the objective of maximizing the resource utilization of near-data resources without degrading query processing time, is NP-hard.

Proof. The key idea of the proof is that the generalized assignment problem (GAP) [207] can be reduced to a special case of our query partitioning problem in polynomial time. Therefore, if we have an algorithm that can minimize the number of operators sent to the cloud without sacrificing query processing time, then we can obtain an optimal solution to GAP. Since GAP is NP-hard, our problem is also NP-hard.

Consider the hierarchical architecture of resource nodes in type II topologies in which each node can communicate with its parent node. An application with $M = |V_o|$ operators are to be executed on streams being generated on data sources. We make a simplifying assumption in this proof that an application instance on each data source consists of a computational pipeline of operators executing in a sequence. On the remote edge and cloud layers, a fixed number of compute nodes are provisioned for processing the data from all data sources. Each data source needs to identify operators which need to be executed locally vs. those that require remote execution on parent node. Due to dataflow dependencies, we can partition the query graph at $M + 1$ possible locations in the graph by defining boundary operators (defined in Section 4.8.2) i.e. a boundary operator splits the application operators into two sets. Operators before the boundary operator (including itself) are executed on the data source and those after the partition are executed remotely. As an example, for a data source i with $M = 4$ operators, $b_i \in [0, M]$ represents the operator where partition occurs. So $b_i = 3$ means first 3 operators execute on data source and the last operator executes remotely.

Compute resources on remote nodes (i.e., edge and cloud layers) are used to process operators sent for remote execution after partitioning. Data transport also incurs a network overhead cost. Our goal is to parallelize processing across data sources and remote nodes, while minimizing the utilization of finite compute and network resources available remotely. As we increase the number of data sources, processing load increases on the system. And the processing load from different data sources can result in network/compute bottleneck on edge and cloud nodes, hurting overall processing time. Thus, we cannot independently partition operators for each data source, instead joint partitioning decision needs to be made across data sources. When the remote resources are saturated, it is more beneficial for the data source to execute operators locally to avoid long processing time. Similarly, if all data source nodes decide to execute their operators locally, then there may exist one or more data source nodes which may have gained from remote processing, because the remote compute and network resources bottlenecks do not exist.

We now introduce the generalized assignment problem (GAP) [207] which finds a minimum cost assignment of n items to m bins such that each item is assigned to precisely one bin

subject to capacity restrictions on the bin. Following is the definition:

$$\min \sum_{i=1}^m \sum_{j=1}^n a_{ij} x_{ij} \quad (5.28)$$

subject to,

$$\begin{aligned} \sum_{i=1}^m x_{ij} &= 1, j \in \{1, \dots, n\}, \\ \sum_{j=1}^n w_{ij} x_{ij} &\leq \text{bin}_i, i \in \{1, \dots, m\}, \\ x_{ij} &\in \{0, 1\}, i \in \{1, \dots, m\}, j \in \{1, \dots, n\} \end{aligned} \quad (5.29)$$

where a_{ij} is the assignment cost associated with assigning item j to bin i , w_{ij} the claim on the capacity of bin i by item j if it is assigned to bin i , bin_i the bin capacity of bin i and x_{ij} a 0-1 variable indicating whether item j is assigned to bin i ($x_{ij} = 1$) or not ($x_{ij} = 0$). This problem is known to be NP-hard [207].

Consider a stream processing system in which each remote node is dedicated to processing data corresponding to a boundary operator, i.e., it only processes data from the data source node assigned to that boundary operator. If the boundary operator is M for the i^{th} data source, then only the corresponding source node executes all the operators. Whereas, if the boundary operator $b_i = M - 1$, then the remote node(s) processing data for $b_i = (M - 1) \forall i \in [1, N_d]$, will receive the data to execute the final operator. This restricts the number of data source nodes that are assigned a particular boundary operator, because there is a finite capacity of resource nodes that can process data corresponding to that boundary operator. Note that we are not losing generality with this assumption because in a different implementation which does not have such dedicated resource nodes, not all the data sources can be assigned to the boundary operator value of 0. This is because, there may now exist one or more data sources for which remote processing may have been beneficial, and violating the constraint in Equation 4.6.

A data source i ($i \in [1, N_d]$) with boundary operator b_i can execute operators remotely while improving query processing time compared to local execution, if and only if total resource usage on remote nodes, due to remaining data sources is below a threshold data rate T_i i.e. $\sum_{p \in [1, N_d] \setminus \{i\} : b_p = b_i} d_p(b_p) \leq T_i$. Here, $d_p(b_p)$ is a function that provides rate of data leaving data source p given the boundary operator b_p , and T_i is the threshold data rate for all the remaining data source nodes with boundary operator b_i , so that data source i with

b_i has enough resources to execute operators remotely. We use the data rate as a measure for the network and compute overhead on the edge and cloud nodes. Based on this, we transform generalized assignment problem to a special case of our problem of configuring minimum number of data sources to execute operators remotely. We can regard the items and bins in GAP as the data source nodes and possible boundary operators i.e. values in the boundary operator vector $\mathbf{b}$, respectively. Then the weight of an item n' assigned to bin m' is $w_{m'n'} = d_{n'}(m')$ where $m' = b_{n'}$, capacity constraint of each bin $bin_{m'} = T_{n'} + d_{n'}(b_{n'})$ and the cost of assignment $a_{m'n'}$ is such that $a_{0n'} > a_{1n'} > \dots > a_{Mn'}$. The cost of assignment ordering incentivizes local execution on data source. By this, we can ensure that as long as data source i on its assigned boundary operator b_i is able to improve processing time compared to executing the query locally, total size of items assigned to bin corresponding to b_i will not violate capacity constraint bin_{b_i} . This is because
$$\sum_{p \in [1, N_d] \setminus \{i\} : b_p = b_i} d_p(b_p) \leq T_i$$
 which

$$\text{implies } \sum_{j=1}^n w_{m'j} x_{m'j} = \sum_{p \in [1, N_d] \setminus \{i\} : b_p = b_i} d_p(b_p) + d_i(p_i) \leq bin_{b_i}.$$

Therefore, if we have an algorithm that can minimize the number of operators sent to the cloud and edge nodes without sacrificing processing time, then we can also obtain optimal solution to GAP. Since GAP is NP-hard, our problem is also NP-hard. QED.

We note that previous works have described the hardness of optimal query partitioning in other related domains [208, 209]. The complexity comes from having to consider a global view of the resource topology to make placement decisions. In this thesis, we investigate a greedy heuristic which locally executes on each node and is embarrassingly parallel. We enable the potential latency overhead resulting from our greedy solution to be configured so that it can be bound by an upper limit. Our approach is implemented in a fully decentralized manner to ensure high scalability of our placement system.

Having established the NP-hardness of near-data operator placement, we note that in our systems, each data source runs an instance of the application on its incoming input stream, and we need to determine the boundary operator for each application instance. Thus, we need to evaluate M^{N_d} possible partitioning solutions, where N_d is the number of data sources and M is the number of operators per query instance. Given that resource conditions can change in the order of minutes in real-world systems (Section 7.2.2), we want to quickly compute a new partitioning plan. We can consider applying other approaches using data summaries to trade off accuracy for reducing network transfer costs. However, these approaches have limitations as we will highlight in Chapter 7.

5.3.1 Insights for Partitioning Heuristic at the Data Source

Following are our insights used to design our partitioning heuristic at the data source for addressing the near-data operator placement problem:

Data-level application partitioning. Operator-level partitioning, which makes placement decisions at the operator-level and executes each operator either on the data source or the remote server, could be undesirable when attempting to run a computationally expensive operator within limited compute resources provided by the data source. Instead, we adopt data-level partitioning to allow an operator to process a fraction of its input records on data source and drain the rest for remote processing. This data-level partitioning is fine-grained and can improve the overall utilization of limited compute resources on each data source node. By leveraging data-level partitioning techniques, we are exploiting the benefits of operator-level parallelism where operators are executing concurrently on different nodes, as well as data-level parallelism where multiple instances of an operator (i.e., one on data source node and the others on remote edge and cloud nodes) are processing subsets of the input data in parallel. Several existing works exploit either operator-level parallelism or application-level data parallelism, in isolation, but not the fine-grained data-level partitioning at each operator, which is the focus of our investigation in this thesis.

Combining cost model-based and model-agnostic techniques for query refinement.

To refine query plans with high efficiency on the data source, we exploit a greedy and embarrassingly parallel heuristic, which reduces the network traffic leaving each data source node. We utilize a combination of cost model-based and model-agnostic techniques to allow each data source node to make fast and effective query refinement decisions *independently*. The cost model-based technique quickly finds a new partitioning plan based on online and fine-grained profiling of each operator’s compute cost. However, such profiling might be inaccurate if the compute resources available for executing operators are insufficient for accurate profiling. Thus, we apply a model-agnostic process to iteratively fine-tune the output plan produced by the cost model-based approach. The proposed heuristic is simple, computationally tractable and a reasonable first attempt to address the partitioning problem at data source in large scale stream processing systems.

We can consider applying other approaches using data summaries to trade off accuracy for reducing network transfer costs. However, these approaches have limitations for several scenarios in real-world stream processing systems as we will highlight in Chapter 7. We thus focus on techniques which do not result in accuracy loss, while still enabling network transfer cost reduction by performing near-data processing. Furthermore, approximate techniques based on data summaries could be implemented on top of our near-data processing

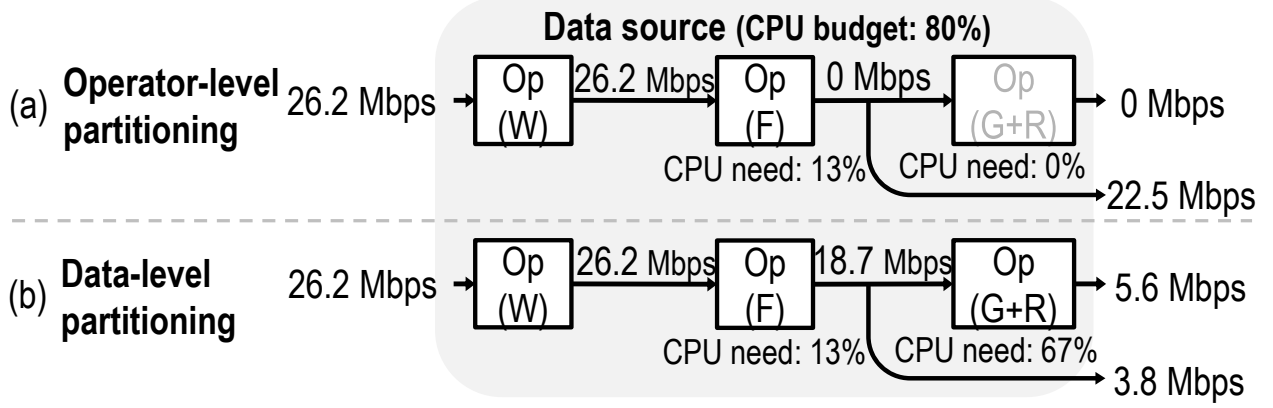


Figure 5.6: Coarse-grained operator-level vs. fine-grained data-level partitioning on data source with 80% CPU budget. G+R requires 80% CPU to process all output data from F operator.

techniques and can be viewed as complementary to our approach. Investigation of modeling data synopsis operations as separate operators in the application graph and leveraging our proposed placement solutions for such graphs, is left for future work.

5.3.2 Near-Data Placement Algorithm

We now describe an algorithm based on the above insights, to achieve fast and fine-grained placement of application operators and the data they process on resource nodes. This is done by partitioning the application workload across data sources and remote nodes, using a data-level partitioning algorithm which relies on hybrid cost model-based and model-agnostic techniques for making partitioning decisions.

To enable data-level partitioning, we introduce the following notions: (1) load factors for each application operator, which controls the amount of data in an epoch that is processed by the operator, (2) the ability to offload/drain excess input data items from an operator queue on data source to its remote node, that could not be processed before the arrival of the next epoch. Load factor is denoted by p value, where $0 \leq p \leq 1$. It selects the fraction p of incoming data items to be inserted to the downstream queue and consumed by the downstream operator for local processing on the node. Remaining data is drained over the network to a downstream instance of the same operator type, in the remote edge node. Load factors are configurable at runtime for each operator, and it is done by solving an optimization problem that computes initial load factors and then iteratively fine-tunes the load factors.

Data-level partitioning problem definition. We summarize variables used in our data-level partitioning problem in Table 5.1. Let us consider a computation pipeline which

Data-Level Partitioning Problem	
D_I	Size of input data (in bytes) injected into the query in an epoch.
o_j	j^{th} operator in the query.
λ_j	Relay ratio of o_j , i.e., ratio of its output to input data size.
c_j	Compute cost per byte of input data for o_j .
dr_j	Size of drained data in an epoch at the j^{th} control proxy.
C	Compute budget available to the query.
p_j	j^{th} control proxy's load factor, i.e., fraction of incoming records to be processed by downstream operator.
pe_j	Effective load factor for j^{th} control proxy, i.e., product of load factors of upstream query operators until o_j .

Table 5.1: Variables in the data-level partitioning problem.

contains M operators $\{o_i : 1 \leq i \leq M\}$. Here, we want to decide load factors that minimize the size of network data transfer from data source node, i.e., total number of drained records (i.e. $\sum_{i=1}^M dr_i$) from data source given the compute budget C available to the query. The problem can be defined as follows:

$$\min_{p_1, p_2, \dots, p_M} \sum_{i=1}^M [\prod_{j=0}^{i-1} p_j \lambda_j] (1 - p_i) \quad (5.30)$$

subject to

$$\begin{aligned} \sum_{i=1}^M [\prod_{j=0}^{i-1} p_j \lambda_j] p_i c_i &\leq C/D_I, \\ 0 \leq p_i \leq 1, \quad 0 \leq \lambda_i \leq 1, \quad c_i \geq 0 \quad \forall i \in [1, M], \quad p_0 = 1, \lambda_0 = 1 \end{aligned} \quad (5.31)$$

Note that D_I , M , and C are fixed for a problem instance.

Unfortunately, there are two challenges in solving the optimization. First, it can be proved that the Hessian matrix is not positive semi-definite even for the two-variable cases, resulting in a computationally hard non-convex formulation [210]. While it is possible to enumerate all possible combinations of load factor values, doing so is expensive for online optimization. Second, the formulation assumes certain conditions, which may not always be satisfied in practice. For instance, to estimate c_i accurately, each operator needs to be evaluated on a sufficient number of input records. Furthermore, relay ratio λ_i can vary non-linearly with input size for certain operations. E.g., in an analytics query with grouping, λ_i is affected by grouping key distribution in the input.

StepWise-Adapt algorithm. We now describe how we address the data-level partitioning

problem. We employ *StepWise-Adapt*, a novel hybrid algorithm that combines two techniques: (1) a model-based technique which searches for near-optimal load factors based on the modeling assumptions of the data-level partitioning problem defined in Equation 5.30. and (2) a model-agnostic technique that monitors query execution using load factors obtained from step (1) and fine-tunes them if the query behavior deviates considerably from stable state—i.e., the available resources are either over/under-subscribed by the application. For fast fine-tuning, step (2) uses a heuristic inspired by the first fit decreasing (FFD) heuristic for bin packing problem [211], to prioritize load factor updates of operators that contribute to more network data reduction.

To efficiently solve the non-convex optimization problem in Equation 5.30, we consider if a transformation (i.e., change in optimization variables) exists to make the objective and constraint functions convex. We analyzed the performance of optimization solvers on problem formulations resulting from different transformations. The cost of solving data-level partitioning is very sensitive to the nature of transformation applied to the problem. For example, let's consider the transforming the problem to a geometric program (GP) [212] for a small scale instance of the problem with just 2 optimization variables. When varying different parameters in the optimization, GP solver takes up to 7 seconds for just solving the problem using CVXOPT solver [213] on a machine with four 2-GHz i7 CPUs. And then the fine-tuning step takes additional time to converge and find load factors which stabilize the query. This is expensive for the fast convergence requirement of data-level partitioning, especially when resource conditions change in the order of 10s of seconds or minutes for target scenarios such as server monitoring pipelines. To enable fast optimization, we thus identified another simple transformation of the original problem that converts it into a linear program (LP) and can be solved in sub-second duration. The transformation is done by introducing a new optimization variable pe_i for the i^{th} control proxy where $pe_i = \prod_{j=0}^i pe_j$. Then the optimization problem in Equation 5.30 can be rewritten as:

$$\min_{pe_1, pe_2, \dots, pe_M} \sum_{i=1}^M \left[\left(\prod_{j=0}^{i-1} \lambda_j \right) \cdot (pe_{i-1} - pe_i) \right] \quad (5.32)$$

subject to

$$\begin{aligned} \sum_{i=1}^M \left[\left(\prod_{j=0}^{i-1} \lambda_j \right) \cdot pe_i \cdot c_i \right] &\leq C/D_I, \\ 0 \leq pe_i &\leq pe_{i-1} \quad \forall i \in [1, M], \quad pe_0 = 1 \end{aligned} \quad (5.33)$$

Rest of the conditions on $D_I, M, C, c_i, \lambda_i$ remain the same as in the original formulation.

When a resource change occurs on data source, load factors need to change to stabilize the query.

A feasible solution provided by LP solver assumes that operator relay ratios/costs of operators (i.e., λ_i and c_i) are fixed and independent of load factors. However, if these parameters are unsteady, the solver provides load factors which would either over-subscribe or under-subscribe the compute budget, making the query execution unstable. To address the issue, StepWise-Adapt takes a post-LP solver step to further tune the load factors. In this step, StepWise-Adapt observes the resource conditions after the application executes an epoch with current load factors of operators and updates them based on the priorities of their downstream operators. An operator that has the *lowest* data relay ratio is assigned the *highest* priority, and StepWise-Adapt aims to increase its load factor first when the application under-subscribes the compute resources (until its $p = 1$). When compute resources are over-subscribed, load factor for the operator in the lowest priority will be decreased first (until its $p = 1$). In this way, we can give more resources to operators that potentially bring about larger data reduction. All the information necessary for StepWise-Adapt is unknown beforehand, but can be readily obtained with profiling.

Note that a high priority operator may have a high compute cost, but since our objective is to minimize the network transfer cost within the compute budget, we choose to find the smallest number of load factor updates to fully utilize available compute. Furthermore, when updating a load factor for an operator, we execute a binary search over discretized load factor values to further improve convergence time—i.e., from the current load factor to the maximum value when compute resources are under-subscribed or to the minimum value when compute is over-subscribed.

Having described the algorithms for solving the general operator placement problem for type I resource topologies, and data-level partitioning for near-data processing in type II resource topologies, in subsequent chapters we look at different case studies which use the designed algorithms and implement them using system abstractions we introduce to efficiently address the placement problem for data source-edge-cloud systems.

5.3.3 Placement for Network-Constrained Environments

We briefly look at how to adapt our compute-constrained query partitioning strategy in Section 5.3.2, for near-data processing in *network-constrained* geo-distributed stream analytics. WAN bandwidth typically sits between 3-20 Mbps and can drop down to below 25% of the maximum bandwidth [35], while compute resources on edge servers can be dedicated for analytics processing. That is, the constraint factor is mainly attributed to dynamic network

bandwidth in WAN streaming analytics. Note that such systems have been considered for near-data processing in recent works [214]. Prior work reveals that many WAN-based systems suffer from performance degradation due to high WAN latencies in $O(100\text{ms})$, which result in significant coordination overhead in the system runtime [98]. The decentralized implementation of our approach helps avoid this expensive coordination between edge and cloud servers, when making query partitioning decisions.

We change our algorithm in Section 5.3.2 such that the compute cost of query processing is minimized while subscribing to the available network bandwidth from each edge server. WAN-based partitioning benefits from minimizing the compute cost of query processing on edge servers, because it improves resource utilization of the servers—service providers can then support a larger number of data sources with the provisioned compute resources. While minimizing compute costs, draining data without utilizing compute resources from the servers is not favorable as it can lead to network bandwidth over-subscription. Bandwidth under-subscription is not preferred either, due to higher compute costs incurred on the edge servers than necessary. Thus, for a bandwidth B allocated to a query and epoch duration T , we can reformulate the optimization problem in Equation 5.30 for geo-distributed environments as follows:

$$\min_{p_1, p_2, \dots, p_M} \sum_{i=1}^M [\prod_{j=0}^{i-1} p_j \lambda_j] p_i c_i \quad (5.34)$$

subject to

$$\sum_{i=1}^{M+1} [\prod_{j=0}^{i-1} p_j \lambda_j] (1 - p_i) \leq B.T, \quad (5.35)$$

$$0 \leq p_i \leq 1, \quad 0 \leq \lambda_i \leq 1, \quad c_i \geq 0 \quad \forall i \in [1, M], \quad p_0 = 1, \quad p_{M+1} = 0, \quad \lambda_0 = 1$$

Similar to Equation 5.30, the new optimization problem can also be transformed and solved efficiently using a LP solver.

5.4 ENERGY ESTIMATION ON THE DATA SOURCE FOR TYPE II RESOURCE TOPOLOGIES

We now address the third problem described in Section 4.8.3. Being able to efficiently estimate energy costs of application operators, enables us to build prediction models which can be leveraged to make fine-grained and energy-aware placement decisions at the granularity

of operators.

To estimate the energy costs of individual operators in an application running on a data source, current works [20, 171, 215] profile the hardware resources via an external power meter [216] to build a hardware resource utilization-based power model (similar to [171]). The operators are then instrumented to obtain their resource usage, and the power model is used to determine their energy costs. However, such power estimation techniques have several disadvantages:

- External power meters are less practical outside of a laboratory setting and many modern data sources such as mobile devices do not have a removable battery [169].
- Source code instrumentation burdens application developers, increases code complexity and maintenance costs [217].
- Resource power models frequently exhibit high prediction error [179], have to constantly keep up with evolving hardware [174] and need to be characterized for each new hardware resource [175, 176].
- Power costs of operators depend on the input in applications such as face detectors and video conferencing applications which provide background filters. In such applications, changing parameters such as frame rate, resolution or the content in the input frames (i.e., face vs. no face) can significantly impact operator power costs. Power prediction models should be able to handle complex energy characteristics of mobile applications.
- Other techniques [166, 218] have attempted to model the energy costs using application configured software parameters. E.g., frame rate and frame resolution can be input features to a prediction model which outputs the operator power costs in a video analytics application. However, they use external power meters for measurements [166] or are tightly integrated with specific platforms such as NVIDIA GPU toolchain [218]. Moreover, simple linear models proposed in literature [218] cannot capture complex energy behaviors of modern applications described earlier.

To overcome the above disadvantages, we propose the following design goals for our solution: (i) *Empirical and resource model-agnostic* solution which does not require manual work to build analytical power models for each of the hardware or application components., (ii) *Easy to adopt/Zero-effort*, not requiring any changes to application code, (iii) *Dynamic*, in order to support complex runtime resource usage behaviors of operators, e.g., operator costs of video analytics applications depend on input video content, (iv) *Lightweight*, in order to have minimal resource overhead on the data source node.

A major challenge to solve the energy estimation problem with the above-mentioned design goals is that data sources such as mobile devices do not provide any mechanism for direct power measurement of each operator. Battery status information (State-of-charge or SoC) is easily accessible for querying on the device but is extremely noisy and coarse-grained. And since we do not have access to application code, we do not have visibility into operator behavior and its energy characteristics to attribute these coarse-grained energy measurements to executing operators.

Power measurement on the data source. In our proposed approach, we leverage coarse-grained power measurements available on data sources via battery status API, which provides SoC estimate or remaining amount of battery charge as a percentage of the total. We also concurrently monitor the time duration for executing operators on the mobile device, by instrumenting the scheduler in the underlying system runtime. The key observation is that depending on the input to the streaming application, the composition of operators running during each battery SoC level is going to vary. This variation is reflected in terms of certain downstream operators not executing at all, e.g., in the case of filtering operations. Or the time duration of operators executing in an epoch can change. This insight is used to train energy models which take as input the executing operators and output their predicted power cost for processing an epoch of data items.

For mobile devices, the duration that a battery resides in a SoC level is inversely proportional to the average device power consumed in that duration. So changes in battery SoC duration across different levels can be attributed to the difference in composition of operators between those levels. The following equation captures the conversion of SoC duration to an estimate of power consumed by the executing operators in that interval [171]:

$$P * (t_1 - t_2) = E * (SoC(V_1) - SoC(V_2)) \quad (5.36)$$

where P is the average power consumption in time interval $[t_1, t_2]$, E is the rated battery energy capacity, and $SoC(V_i)$ is the battery state-of-charge at voltage V_i (i is 1 or 2).

Application execution profile. While periodically sampling battery SoC on data source, we concurrently log the active operators. Monitoring active operators enables attributing the measured energy consumption in the discharge interval, to the corresponding active operators in that interval. To avoid making changes to the application itself, we instrument the underlying streaming runtime scheduler to obtain the type of operators and the number of operator instances running.

Execution segments can be obtained by continuously tracking the scheduling and completion of each operator in the application. This can be done in the runtime scheduler by storing

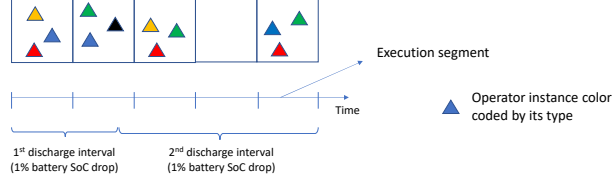


Figure 5.7: A snapshot showing execution segments and discharge intervals for operators executing on the data source.

the state of active operators in a map which contains the operator type (e.g., filter, feature extractor, etc.) to the number of instances of the operator type that are currently scheduled. An API exposed by the runtime is queried by the energy estimator to train power prediction models.

Power attribution. We now build a power predictor that directly maps operators to their power costs (no analytical model required). We describe how to interpret the parameters of the trained power model which are used to attribute energy measurements to the corresponding operators executing during the measured time interval. In Equation 4.7, we defined our minimization objective for a time duration with a fixed amount of energy consumption. The time duration was broken down into execution segments, each of which contain non-changing set of operators. Note that each measurable battery discharge interval such as 1% drop in SoC on a mobile device, gives us one training data sample. Training data needs to be collected over multiple such discharge intervals. Figure 5.7 shows how the execution segments can align with SoC discharge intervals.

The sequence of execution segments in the i^{th} discharge interval SS_i , can consist of either active operators executing (referred to as *active segments*) or no active operators during which time the device is idle (referred to as *idle segments*). If there is at least one operator executing in a segment, then the device is not idle during that segment. The arrangement of active and idle segments in a discharge interval depends on the arrival characteristics of the input stream as well as the scheduling behavior of the underlying runtime. As shown in Figure 5.7, a segment may consist of one or more active operators executing concurrently. Let's represent the j^{th} segment $s_j^i \in SS_i$ as $s_j^i = \{o_k \mid k \in [1, M], o_k \text{ is an active operator in segment } s_j^i\}$. If $s_j^i = \{\}$ then it is an idle segment. Based on Equation 5.36, if we fix the SoC at 1%, then the length of a discharge interval is inversely proportional to the expected power consumed during that time interval. We use $E_{1\%}$ to denote the fixed energy consumed in a discharge interval. The energy consumption within the i^{th} discharge interval can be represented as

follows:

$$\begin{aligned}
E_{1\%} &= \sum_{j=1}^{N_S^i} EMap(s_j^i, t_j^i) = \sum_{j \in [1, N_S^i]: s_j^i \neq \{\}} EMap(s_j^i, t_j^i) + \sum_{j \in [1, N_S^i]: s_j^i = \{\}} EMap(s_j^i, t_j^i) \\
&= \sum_{j \in [1, N_S^i]: s_j^i \neq \{\}} p(s_j^i) \cdot t_j^i + \sum_{j \in [1, N_S^i]: s_j^i = \{\}} p(s_j^i) \cdot t_j^i \\
&= \sum_{j \in [1, N_S^i]: s_j^i \neq \{\}} p(s_j^i) \cdot t_j^i + \sum_{j \in [1, N_S^i]: s_j^i = \{\}} p_{idle} \cdot t_j^i \\
&= \sum_{j \in [1, N_S^i]: s_j^i \neq \{\}} p(s_j^i) \cdot t_j^i + p_{idle} \cdot (T_{1\%}^i - \sum_{j \in [1, N_S^i]: s_j^i \neq \{\}} t_j^i)
\end{aligned} \tag{5.37}$$

where $p(s_j^i)$ is the power consumption of executing the j^{th} segment of i^{th} discharge interval, p_{idle} is the constant idle power on the mobile device, $T_{1\%}^i$ is the duration of i^{th} discharge interval. Re-arranging terms in Equation 5.37, gives us the following equation:

$$\begin{aligned}
T_{1\%}^i &= \frac{E_{1\%} - \sum_{j \in [1, N_S^i]: s_j^i \neq \{\}} p(s_j^i) \cdot t_j^i}{p_{idle}} + \sum_{j \in [1, N_S^i]: s_j^i \neq \{\}} t_j^i \\
&= \frac{E_{1\%} - \sum_{j \in [1, N_S^i]: s_j^i \neq \{\}} p(s_j^i) \cdot t_j^i + p_{idle} \cdot \sum_{j \in [1, N_S^i]: s_j^i \neq \{\}} t_j^i}{p_{idle}} \\
&= \frac{E_{1\%}}{p_{idle}} - \frac{\sum_{j \in [1, N_S^i]: s_j^i \neq \{\}} (p(s_j^i) - p_{idle}) \cdot t_j^i}{p_{idle}} \\
&= \frac{E_{1\%}}{p_{idle}} + \frac{\sum_{j \in [1, N_S^i]: s_j^i \neq \{\}} (p_{idle} - p(s_j^i)) \cdot t_j^i}{p_{idle}}
\end{aligned} \tag{5.38}$$

As we can see in the equation above, if $p(s_j^i) > p_{idle}$ and if more the segments in discharge interval that we observe operators within segment s_j^i , the smaller the total number of segments in that interval. Given the traces for battery SoC and application traces, we can obtain the value of $T_{1\%}^i$ for the i^{th} discharge interval, active segments along with their durations in segment sequence SS_i and the operators in each segment. This enables us to estimate parameters of Equation 5.38. Note that for placement purposes, it suffices to estimate these parameters, which compute for a given segment s_j^i , the coefficient $\frac{p_{idle} - p(s_j^i)}{p_{idle}}$. We refer to the absolute value of this coefficient as the *relative power* for the corresponding segment.

CHAPTER 6: OPERATOR PLACEMENT BETWEEN EDGE AND CLOUD RESOURCES

In this chapter, we describe a case study which investigates the effectiveness of the placement algorithms we designed for type I resource topologies, with the target applications being video processing pipelines in Internet-of-things (IoT) systems. We built a proof-of-concept system to validate our approach for several real world applications and show the effectiveness of our approach compared to related works.

6.1 INTRODUCTION

The continued growth of IoT applications requires the ability to extract insights from massive amount of data streams observed in real-time. Such data is collected from multiple input sources including: (1) surveillance cameras for traffic monitoring and intrusion detection, (2) wearable cameras and medical devices for healthcare, (3) sensor networks deployed on bridges and buildings in smart cities.

Due to the limited computational capability of devices that collect such data, the conventional approach of performing analytics on data streams is to send the streams to a centralized data center (cloud) and leverage its powerful and seemingly abundant resources to execute the analytics remotely. However, given the tremendous amount of data transfer required by video data, the latency and bandwidth requirements become significantly high. Moreover, if the cloud infrastructure belongs to a public cloud provider that charges for consumed compute, network, and storage resources, this approach of cloud-centric analytics becomes prohibitively expensive.

Figure 6.1 shows an existing deployment of IoT applications which includes *edge devices* that act as gateways to aggregate and forward IoT-captured data or to protect IoT devices from being exposed to security vulnerabilities. Such edge devices are typically few hops away from the data source, with non-trivial computational and network resources. Therefore, the edge servers can be used as a computational resource to complement the cloud-centric approach and reduce the latency of data analysis.

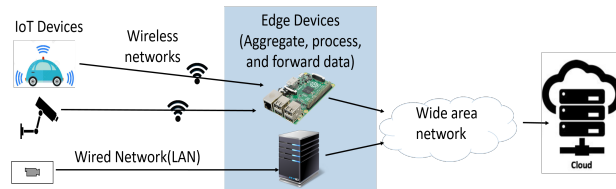


Figure 6.1: IoT edge and cloud infrastructure.

IoT applications consists of a set of dependent operations modeled as dataflow graphs. Each operation describes some computation on the incoming data such as filtering, convolution or encryption. A key challenge is to automatically decide how to partition such operations among edge and cloud compute resources, in order to minimize the overall completion time of the entire graph of operations. This is the general operator placement problem we defined in Chapter 4 with the target system being type I resource topologies.

The unique nature of IoT applications poses several challenges for operator placement problem because IoT applications have the following properties: (a) Data comes from various sources as continuous streams of observations. Hence, there exists queuing delay for observations within the same data stream and across different data streams that share the same compute and network resources; (b) Compute resources are heterogeneous and geo-distributed (i.e., edge devices are more resource-constrained but closer to the data source). Such resource asymmetry requires careful modeling of the trade-off between computation and communication delays; (c) Dataflow operations are interdependent and the placement algorithm has to optimize their execution without violating the dependency constraints; (d) As the number of data sources increase, we need to process incoming data streams in parallel to meet the real-time processing needs of IoT applications, which requires jointly determining the scaling/parallelism factor for the application and placement of operators within each parallel application instance.

Following are the contributions made in this thesis, with regard to addressing the placement problem for edge and cloud nodes for IoT systems:

- We proposed a general model for estimating the execution time of IoT dataflow graphs that operate on data streams of observations. The model considers computation, communication and queuing delays, and captures pipeline parallelism (due to multiple inputs being processed concurrently by different operators in the graph).¹
- We drew an analogy between the general operator placement problem and the Matrix Chain Ordering Problem (MCOP) which allowed us to design a dynamic programming algorithm called to solve the operator placement in log-linear time with respect to the number of operators. The algorithm minimizes the completion time of executing dataflow graphs on edge and cloud nodes.
- We validated our proposed algorithm by building a real system Droplet, which is based on the edge-cloud orchestration framework Echo [65]. Our empirical results show that

¹The work on addressing general operator placement problem for IoT applications was done jointly with Tarek Elgamal, who is the primary contributor for accurate execution modeling of video streaming pipelines. This thesis focuses on contributions to scalability aspects of our proposed approach.

our algorithm provides near optimal results in three real world applications while scaling to large number of operators and outperforming the closest counterpart in computation partitioning literature [91] [92], for large number of data sources.

The rest of this chapter is structured as follows. Section 6.2 discusses real-world scenarios where we can apply the proof-of-concept system we built based on the proposed placement heuristic in Section 5.1.2. In Section 6.3, we discuss the architecture of systems that implement our proposed heuristic. In Section 6.4, we evaluate our method and we compare it with the closest approaches in literature.

6.2 MOTIVATING SCENARIOS

In Section 5.1, we defined the general operator placement problem for type I resource topologies and designed heuristics to efficiently address the problem for data streams generated by one or more data sources. Type I resource topologies are very common in several real world deployments including edge-cloud systems for traffic surveillance [25], drone based target tracking systems which offload video processing to nearby edge servers [219] and cloud gaming which uses edge servers provide low-latency, interactive experience to gamers [220]. Furthermore, there are several applications in video analytics and other domains which can leverage near-data processing capabilities such as telehealth services/analytics that enable paramedics and remote doctors to coordinate during emergency medical environments [41], car counting application in road traffic monitoring systems [10] and edge-based monitoring systems for IoT devices [50].

In this thesis, we restrict the scope of our case study to focus on video processing pipelines running on edge and cloud systems. As an example, consider a real-world scenario with paramedics wearing body cameras to analyze video feed of patients to detect strokes. Video processing is done using an edge device in the ambulance and a cloud server in the hospital. The application consists of three operators which perform face detection, facial landmark detection and finally, mouth droop estimator as an indication of stroke. The camera communicates with edge device using WiFi P2P network. The edge device will have internet connectivity to the cloud server. Depending on the emergency conditions, the video feeds may be generated from body cameras of one or more paramedics, and the data from all these data sources may need to be processed concurrently [41]. An accident scene may not require more than one or two paramedics, but a large scale disaster such as building collapse, earthquakes or a war-zone may require several paramedics attending to multiple patients. Furthermore, there may be multiple ambulances and paramedics at different locations communicating with

the cloud servers in a hospital at the same time. The dynamic conditions of data sources (i.e., head-mounted cameras of paramedics) joining and leaving the network, makes it critical for placement solutions to quickly adapt, in the order of minutes or less.

Several other scenarios have similar resource and application conditions which require a fast and scalable placement solution. For resource efficiency, a surveillance camera network may dynamically select subsets of cameras (i.e., number of data source nodes change) to collect and analyze video feeds in motion detection and target tracking tasks. Such a mechanism enables focusing on regions of interest in public spaces or indoor locations such as malls and airports, and perform real-time face recognition. Identifying a new camera configuration within 20 seconds or less, can be sufficient to keep up with the dynamics observed in real-world camera networks [221]. Another scenario involves a bus or other mass transit vehicle that has an edge server [222, 223] to keep up with the real-time demands of video processing workloads, which enhance traveler experience. The travelers may be playing augmented reality games or using other social media-based video applications. Executing expensive video operators (e.g., deep neural network inference) on the mobile device with limited resources can create compute bottlenecks. The high frequency of travelers moving in and out of the vehicle requires the edge systems to quickly adapt to varying number of data sources. Other similar scenarios include augmented reality games like Pokemon Go servers processing video feeds from varying number of players [224] in a certain geo-location. Application operations that cannot be executed on the local mobile device need to be placed between edge and cloud servers, e.g., Raid Battle [225] that requires coordination across players, or Trainer Battle [226] which provides teamwork-oriented experience in a multi-player Pokemon go game. Another interesting scenario is one in which multiple traffic camera feeds along the path of an autonomous vehicle may be streamed to edge servers in the vehicle and cloud platform to make real-time driving decisions.

6.3 SYSTEM ARCHITECTURE

Figure 6.2 shows an overview of the management framework for distributed operator placement framework. We assume that the cloud and edge resources are under the same administrative control, with management components of the framework being deployed in the cloud and communicating with edge devices over wide area network in order to deploy operators on them. The computing infrastructure consists of edge and cloud resources. Each edge device has a local dataflow engine that handles execution of operators that are deployed on it. The cloud has a clustered dataflow engine that handles the execution of operators in a cluster with multiple machines. The system consists of four main components **App**

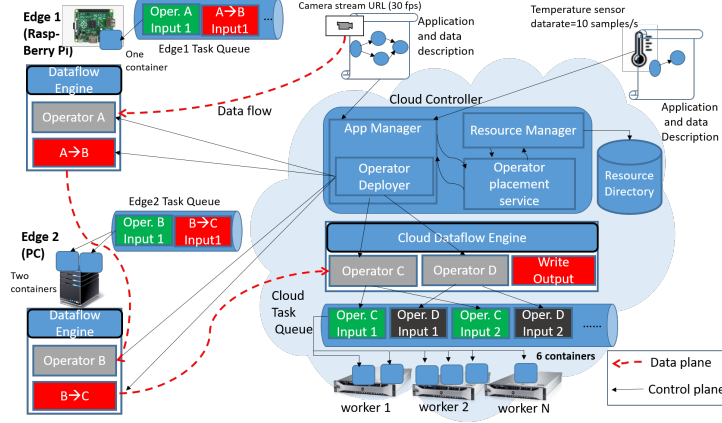


Figure 6.2: Distributed operator placement framework in shared edge and cloud infrastructure.

Manager, Dataflow Engine, Resource Manager, and Operator placement service.

In the rest of the section, we describe the details of each component:

6.3.1 App Manager

The core part of the cloud controller is the AppManager which receives the application description and location of the data input stream for multiple IoT applications. The application description contains information about input data source, output data directory, operators that compose the application and their dependency relationship. The AppManager parses the application description into a directed acyclic graph (DAG) of operators and it communicates with the operator placement service to decide the mapping between each operator and an edge or cloud resource. Once the operator placement service decides the mapping, an operator deployer sends a request to the local dataflow engine of each edge and cloud resource to deploy the subset of the operator graph assigned to it.

The operator deployer allows data to flow from one dataflow engine to another by deploying a *transmission operator* in each dataflow engine. The transmission operator pushes the output of the final operator in one dataflow engine to the first operator in the next dataflow, hence allowing dependent operators to be concurrently processing inputs in a pipeline fashion without violating their dependency relationship. For example, the first input processed by operator i can be processed by operator j while the second input is being processed in parallel by operator i . After deploying all the execution and transmission operators, the operator deployer starts dataflow execution. The data starts to flow from one operator to another within one dataflow engine and across dataflow engines. The last dataflow engine has a *Write Output Operator* which by default routes the result to the initiator of the workflow

and in addition it permanently stores the results of dataflow execution in a cloud database. Throughout the execution of the dataflow a *Resource Manager* component monitors the workload on all edge and cloud resources. We present more details about the Resource Manager in Section 6.3.3.

6.3.2 Dataflow Engine

We use Apache NiFi, a light-weight engine designed for composing user-defined operators and executing dataflows in a single or multiple machines. NiFi’s native data model is a FlowFile which is an in-memory reference to a collection of bytes that have attributes such as filename, file size, and timestamp at which the FlowFile was ingested in the system. In our system, a FlowFile can be seen as one unit of data defined by the application (e.g., packet, tuple, video frame). The IoT application is typically defined for a stream/sequence of such data units (e.g., 30 frames/s).

An operator in Nifi is user-defined Java logic that receives a FlowFile from the upstream operator, apply some operation (e.g., face detection), and generates a new FlowFile (e.g., face patch FlowFile) that will be passed to the downstream operator.

When an operator is deployed on a dataflow engine, the operator generates a task that has both operator code and the FlowFile data, and pushes the task to the queue. Multiple containers can then pull tasks from the task queue and executes them. Containers are units of processing that are capable of executing one task for one input at a given time. The container can be thought of as a VM or a Docker container [227]. A Docker container is a lightweight virtual machine that carries dependencies needed to run an application. In our system we have one container image that is capable of executing any task, if a new task is implemented in the system then a new container image is created in a central repository that all edge devices can access. Based on how powerful the CPU of the edge device is, it can spawn multiple containers to run more than one task in parallel. In Figure 6.2, we show two registered edge devices (raspberry pi and PC) in addition to one cloud. In this example, the raspberry pi can have maximum of one container, however the server can have two containers and the cloud can have up to six containers. We aim to avoid consuming the maximum number of containers by designing an operator placement algorithm that minimize both completion time and resource consumption. The overall cost comprises a dataflow completion cost and a penalty for each consumed container on a resource, to avoid over consumption of expensive resources. The details of the operator placement model is described in Section 4.8.

6.3.3 Resource Manager

It carries information about which compute resources are available in the geo-distributed cluster and receives requests to dynamically register new resources and remove old ones. The resource manager has a separate API than the App Manager and it keeps the information in a directory that can be queried by the Operator placement service and the Operator deployer. When an application is deployed in the system, the resource manager keeps track of the operators currently deployed in the edge and cloud resources. The operator placement service uses this information to decide the placement of a new application given the placement of another application.

6.3.4 Operator Placement Service

It receives application information from the App manager and resource information from the resource manager, then it finds a mapping between each application operator and an available resource where such operator is allowed to be placed. The placement service decides both the mapping of operators' to resources and the number of containers to spawn in each resource in order to minimize the makespan (i.e., maximum completion time of applying all the operators to all inputs) and the overall resource consumption. The underlying placement algorithm addresses the tradeoff between deploying operator code near the data stream source (i.e., edge computing) compared to moving the data to a remote cloud where computation resources are more powerful (i.e., cloud computing).

We assume that the operator placement service is invoked for an epoch of the data stream. For example, if the body camera produces 30 fps then the placement service can be invoked for 10 seconds of data (i.e., 300 frames). This epoch size is provided by the application and ideally the placement service should be called for this application again after the 300 frames finish execution in order to decide if any replacement is required. We present the details of the placement service underlying algorithm in Section 5.1.2

Profiling

In order to decide the most efficient placement, the operator placement service has to keep profiling information about the execution of each operator on the resources where it is allowed to sit.

We profile each operator by running it on a single input and measuring the average execution time on multiple runs on the same edge device. The profile then becomes independent of

length of the data stream. For example, if the input data is a video stream with 1000 frames but all the frames have same resolution (e.g., 1080p) then it is enough to profile the information for one video frame. The profile is however sensitive to the frame resolution. To account for this, we represent the execution time of some operators as a function of the resolution. For example, it is known that convolution operators have $n \times m$ Flops where $n \times m$ is the number of pixels in an image. Dividing this number by the speed of the container’s virtual CPU, gives an estimate of the execution. In the future work, we plan to explore other dynamic profiling techniques such as using feedback control loops [228].

6.4 EVALUATION

In this section, we implement the proposed Droplet algorithm and evaluate its performance against the closest algorithms in the literature using various applications from different domains.

Compared approaches. We compare four methods for operator placement:

1. **Droplet (Our Approach):** We use the proposed algorithm described in 5.1.2 to decide the operator placement.
2. **Brute Force (BF):** We execute all different placements and we empirically measure the completion time of all the operators and we report the placement with the smallest execution time.
3. **SCPP (Single-User computation partitioning) [91]:** We implement the SCPP approach which finds an optimal solution for the placement problem with some assumptions such as (1) Operator graph is a sequence of operations (i.e., not a DAG or a tree), (2) Pipeline parallelism is not considered (i.e., all the video frames in one chunk have to be processed by operator 1 before they move together to operator 2). We use this approach to evaluate the significance of pipeline parallelism.
4. **Genetic algorithm (GA):** We implement genetic algorithm as a representative of search strategies that have been proposed in the literature [92] [86] to solve the placement problem. Droplet is compared with GA when the number of operators is significantly high (i.e., more than 100 operators) to evaluate it’s scalability and performance characteristics at large scale. We use the fitness function that we modeled in Equation 5.1 and we use similar parameters to the ones reported in [92]: population size=50, crossover probability=0.5, mutation probability=0.15. However, for compari-

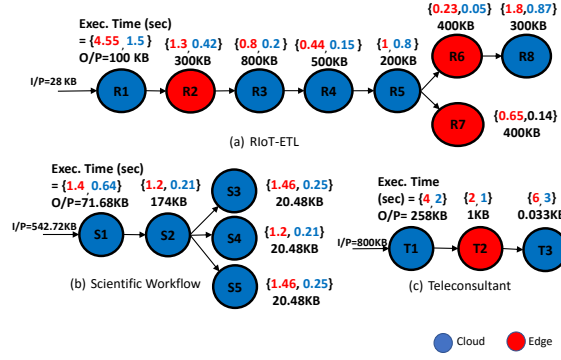


Figure 6.3: Operator graphs for real world applications. Color coding of operators shows Droplet’s placement output

son purposes we terminate GA after it runs for 10 times the duration taken by Droplet to obtain shortest path placement.

Applications. We evaluate Droplet through three real-world applications from various domains:

1. Teleconsultant [41]: a Telehealth application in which paramedics wear body cameras to analyze video feed of patients to detect strokes. Analysis is done using an edge device in the ambulance and a cloud server in the hospital. The application consists of three operators, and we use a stream of images obtained from the image gallery for facial paralysis [229] with chunks defined to consist of 10 images.
2. RIOT-ETL [65]: an IoT benchmark that performs data preprocessing and cleaning on the incoming data stream. The benchmark is modeled as a dataflow graph of 8 operators as shown in Figure 6.3. In our experiments, we generate data that mimics the original dataset and use a chunk size of 10 observations.
3. Scientific Dataflow [230]: We use a sub-dataflow of the *LIGO* scientific workflow that processes astronomical data. [230] provides the profiling measurements for computation and communication cost for the 5 operators that comprise the sub-dataflow (as shown in Figure 6.3). We generate data that follows same profiles and we use a chunk size of 10 observations.

System setup: The computing infrastructure consists of edge and cloud resources. We use 1 laptop as edge device and one server as the cloud. The edge device has Intel Core i7-5600 CPU with 4 GB of memory and the cloud server has Intel Core i7-3770 CPU with 8 GB of memory. We use one thread on the edge device and one thread in the cloud server and

control the bandwidth from edge to cloud server to be 12 Mbps and from cloud server to edge device to be 13 Mbps, which simulates an average wide area network connection. Each of the edge and cloud servers has a local dataflow engine, Apache NiFi, that handles execution of operators that are deployed on it. Nifi is an engine designed for composing user-defined operators and executing dataflows in a single machine or across multiple machines. Each of the edge and cloud servers has a local deployment of Nifi and we use Echo [65] orchestration framework to handle the communication between the two Nifi instances. Echo does the following: (1) receives the operator graph from the user, (2) reads the resource graph from a directory of registered resources, (3) communicates with Droplet to find out the mapping from operators to resources, (4) deploys the operators that belong to the edge and cloud resources in their local Nifi engines, and (5) manages communication of intermediate results between Nifi engines.

Evaluation Metrics: We conduct experiments to evaluate Droplet performance in terms of:

- *Model Accuracy:* For each placement, we compare our estimate of completion time of operators with observed completion time in the real system deployment and evaluate accuracy of our estimates.
- *Chunk Completion time:* Given an operator graph and a resource graph, we deploy the operators according to the placement obtained from Droplet that provides the shortest completion time of a chunk of data frames and measure it. This is compared against chunk completion time from other approaches.
- *Time to find Placement:* We measure the time that Droplet takes to obtain the placement decision and focus on its behavior for large scale operator graphs.

6.4.1 Model Accuracy

In order to show that our modeling of completion time for operator graph execution closely reflects the observed running time from real-world deployments, we compare the two measurements for our applications. Figure 6.4 shows the results for the *Teleconsultant* application. As we can see, the estimated completion times for each of the possible operator placements in the application are very close to empirically measured values. The average error in estimated completion time across placements is only about 0.42% and Pearson correlation coefficient is 0.998.

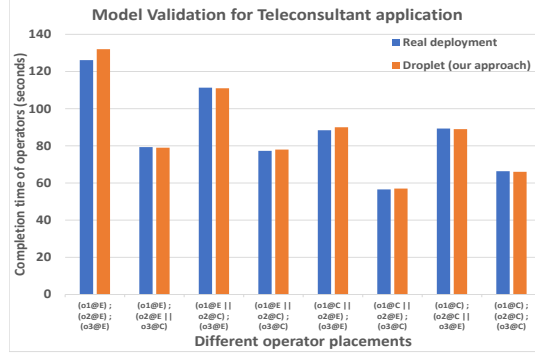


Figure 6.4: Comparing model estimated completion time for *Teleconsultant* application with observed times during deployment of different operator placements

6.4.2 Chunk Completion Time

Real-World Application

To determine how well Droplet does on real-world applications, we solve the operator placement problem with different approaches and plotted the results in Figure 6.5. BF gives us the reference minimum time for each application because it is obtained from an exhaustive search over all placements. So it was feasible to use BF to obtain optimal completion time in order to quantify the error introduced by our modeling approximations. SCPP provides the execution time when the operators in the input graph are executed in a sequence with no pipeline parallelism across inputs. The results show that Droplet is within an average of 4.09% error compared to Brute Force, in finding the operator placement with minimum execution time. Moreover, Droplet and GA are able to leverage pipeline parallelism to find placements that outperforms SCPP in two of the above applications. We note that Droplet and SCPP give the same execution time for the Scientific dataflow. The reason is that the ideal placement for this application is to place all the operators in the cloud. This reduces the significance of the pipeline parallelism that happens when some operators are running in the edge and other operators are concurrently running in the cloud.

Large Scale of Operators

In case of large scale of operators, it was infeasible to find the optimal placement solution via BF. So we compared the placements, obtained by Droplet, with GA, by increasing the number of operators. Figure 6.6 shows a comparison between the chunk completion for both algorithms. The results show that as the number of operators increases, the search space

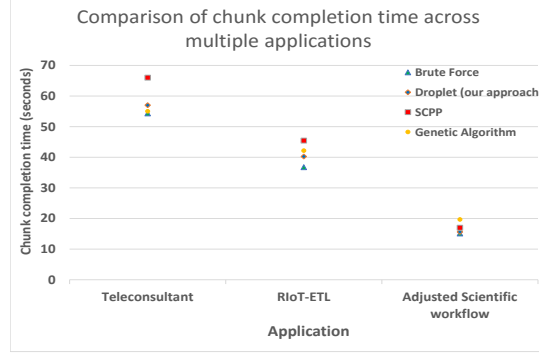


Figure 6.5: Droplet is evaluated against other approaches for performance in finding the operator placement with shortest chunk completion time, across multiple applications

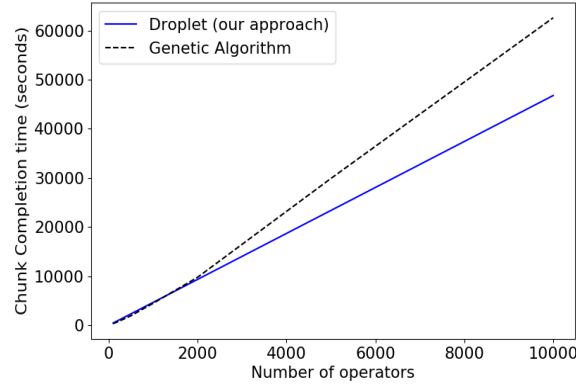


Figure 6.6: Comparison with genetic algorithm with increasing number of operators that GA has to navigate, becomes huge (in the order of 2^n) and GA finds less optimal results. This result confirms that as the number of operators becomes in the order of thousands which is the case in several Deep Neural Networks models [231], Droplet is able to find placements that are more than 20% better than GA in terms of completion time while being 10 times faster.

6.4.3 Time to Find Placement

We now analyze the scalability characteristics of Droplet with increasing numbers of operators. Figure 6.7 compares Droplet with BF, in the time to find shortest path placement of operators. This time includes the construction time of placement graph and executing shortest path algorithm. With BF, we are not able to complete the search in a reasonable amount of time when number of operators increased beyond 15. Droplet however is able to complete the search within 10 seconds for 1000 operators. It is shown to scale polynomially

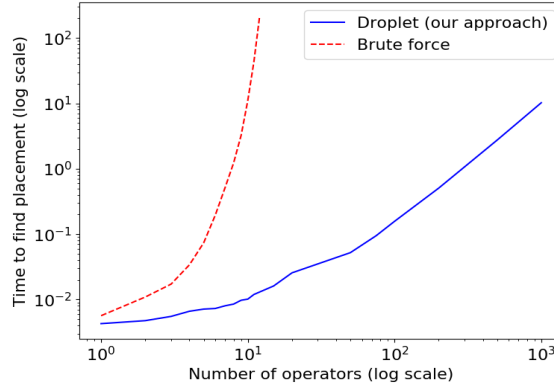


Figure 6.7: Scalability in time to search for the best placement with increasing number of operators

with increasing the number of operators.

6.5 CONCLUSION

We studied the problem of distributed operator placement for IoT applications. We studied the scenario where IoT applications are sharing resources across the edge and cloud. We addressed several challenges that are unique to IoT applications such as: (1) Modeling the trade-off between computation and communication delays when placing operators across heterogeneous and distributed compute resources, (2) scaling our placement solution to handle the incoming data volumes from large number of data sources and, (3) processing continuous stream of inputs in pipelining fashion, and modeling the queuing delays involved in this scenario. We formulated our problem as a shortest path problem that navigates various possibilities of operator placement and chooses the one with the minimum total cost with respect to operator graph completion time. We showed through analysis and experimentation that our proposed solution scales log-linearly in the number of data source nodes.

CHAPTER 7: OPERATOR PLACEMENT BETWEEN DATA SOURCE, EDGE AND CLOUD RESOURCES

We now describe our second case study which focuses on type II resource topologies. Such topologies are already deployed at scale in server monitoring systems within datacenters (in which server nodes themselves are the monitoring data sources and can perform computation) and autonomous vehicles with on-board SoCs, while the advent of smart sensors and smart cameras are expected to make such topologies increasingly prevalent. In this chapter, we investigate datacenter server monitoring systems and apply the heuristics we developed for placement in Chapter 5. We build a proof-of-concept system to validate the effectiveness of our proposed placement techniques.

7.1 INTRODUCTION

Today’s datacenters use thousands of servers to host large-scale services, such as web search, database systems, and machine learning (ML) pipelines, for millions of users. Operating these services with high availability requires that datacenter operators quickly detect performance and reliability issues, such as high network latency, disk failures, and service outages from software bugs [43, 232, 233, 234, 235, 236, 237], and resolve these issues to restore normal service operation [12, 24, 238].

To facilitate highly available services, datacenter operators deploy a dedicated monitoring system that analyzes real-time events collected across datacenter servers using a stream processing engine running in a hierarchical architecture of resources consisting of edge and cloud nodes, as shown in Figure 7.1. Note that to keep the use of terminology consistent in the thesis, we use the terms edge and cloud in this chapter to refer to the intermediate aggregation servers close to the data sources and the dedicated cluster with root aggregation nodes running the stream processing engine, respectively. Note that the hierarchical topology of resource nodes may be completely located within a single datacenter. Once the collected data is analyzed, the cloud enables visualizing the behavior of the monitored system via dashboards, allowing the operators to generate alerts upon observing issues impacting service availability. Data streams for server monitoring are diverse, including both service-level application logs and host-level metrics (e.g., network latency) that represent the health of various system resources. Thus, large-scale monitoring pipelines can generate up to 10s of PBs per day from hundreds of thousands of servers [6, 238], making transferring the data to a remote cluster a significant bottleneck. Moreover, the edge and cloud nodes require thousands of CPUs to keep up with such massive data streams while maintaining a low mean

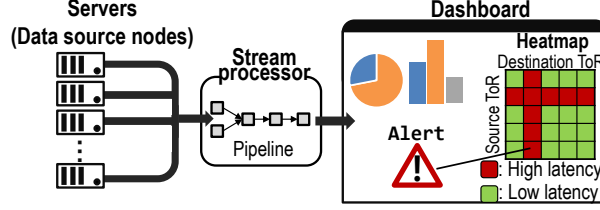


Figure 7.1: Overview of a datacenter monitoring system.

time to detection (MTTD) [24].

Challenges. To reduce the amount of data delivered to the edge and cloud nodes, several monitoring pipelines propose to leverage available compute resources on data sources (i.e., server nodes) and aggregate data locally. Despite the great potential, because available resources on the data source typically result from resource over-provisioning to handle peak resource demands [24, 239, 240, 241, 242], a monitoring query may be restricted to run a subset of operations (e.g., filtering) within a given compute budget to minimize interference with the hosted services. Such compute budget varies widely across data sources and different time slots in practice (Section 7.2.2). However, prior approaches *statically* decide which operations to run on the data source [17, 95, 243]. This makes static partitioning either too conservative or, alternatively, run out of the given compute budget during peak resource demands by the hosted services, affecting overall query performance.

In the network telemetry domain, Sonata [12] proposed a *dynamic* approach based on a query cost optimization model to distribute query operators across programmable switches and remote edge/cloud nodes. Query placement in Sonata occurs at the operator level [17, 95, 243], and is modified at runtime by a central query planner running in the cloud. Thus, the query planner deploys an operator to the switch only if available compute resources on the switch are sufficient to process all the incoming data at the operator. Unfortunately, this coarse-grained operator-level query partitioning is not effective in scenarios where available compute resources are highly constrained, as is the case with data sources in monitoring systems. Furthermore, solving an expensive optimization using a centralized query planner is unsuitable for making frequent decisions in systems with data sources that exhibit fast changing resource conditions.

Our proposal. We propose Jarvis, a new monitoring engine that targets large-scale systems generating *high-volume data streams*. For the query workload on each data source, Jarvis identifies fine-grained *data-level partitioning* strategies by controlling the amount of data processed by each query operator, i.e., *load factor*. This partitioning strategy uses a query cost model-based technique to estimate the initial load factors of query operators and then iteratively adapts the estimates by monitoring the query’s execution. Jarvis can significantly reduce network data transfers while utilizing the limited and dynamic compute resources

over data sources. As Jarvis is implemented in a fully decentralized manner, it can scale to a large number of data sources.

Jarvis introduces the following novel extensions to the conventional query execution pipeline: *Control proxy* and *Jarvis runtime*. The control proxy is a light-weight routing logic—associated with a query operator—that decides “how many” incoming records should be forwarded to the associated query operator. It is used to implement the load factors at the operator-level. At each data source, the local Jarvis runtime interacts with all control proxies within a query in order to identify their state (i.e., idle, congested, or stable). After observing state changes for the control proxies of the query, the Jarvis runtime refines its plan for data-level partitioning in order to keep the query execution in each data source stable.

Results. We have implemented Jarvis and evaluated it with monitoring queries on host-level network latency metrics and application logs. The results show that Jarvis enables a cloud node to handle up to 75% more data sources while improving query throughput by up to 4.4× over the state-of-the-art partitioning strategies. Moreover, Jarvis converges to a stable query configuration within seven seconds of a resource change occurring on data source.

7.2 BACKGROUND & MOTIVATION

7.2.1 Monitoring Systems Overview

While large-scale server monitoring [24, 43, 232, 234] is used in many scenarios, [24, 43] motivate our work. We briefly describe the two scenarios that guide the design of Jarvis:

- *Scenario 1:* Network engineers deploy Pingmesh [43] agents on datacenter nodes to collect latency data between node pairs. A web search team uses Pingmesh to monitor network health of their latency-sensitive service and generate an alert if more than a predefined proportion of hosting nodes (e.g., 1%) have probe latencies exceeding a threshold such as 5 ms [43].
- *Scenario 2:* A log processing system, Helios [24], enables live debugging of storage analytics services such as Cosmos [244]. A bug in a cluster resource manager may lead to service resources being under-provisioned. To temporarily mitigate performance degradation, several TBs of log streams [24] are processed quickly to identify impacted tenants. Their latency and CPU/memory utilization data are queried to predict resource needs and make scaling decisions.

These scenarios rely on a diverse set of computations. Scenario 1 requires processing of periodically generated metrics composed of structured numerical data; queries consist of

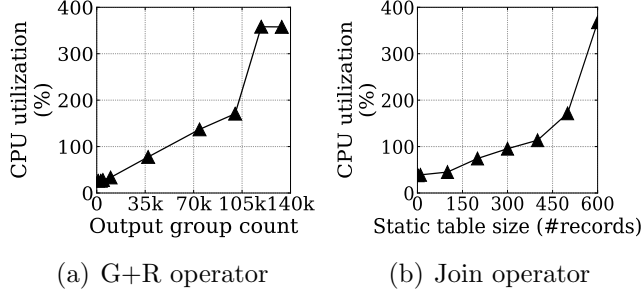


Figure 7.2: Profiling of query operators.

computations specific to numerical data, such as filtering and aggregation. On the contrary, Scenario 2 requires processing aperiodically generated logs composed of unstructured strings; queries accompany computations for string processing such as parsing, splitting, and search.

Moreover, these computations rely on various streaming primitives, which are computationally different and whose costs are highly dynamic depending on the input data. For example, filter (F) drops uninteresting records by applying the predicates on each record and typically requires little compute resource. Grouping (G) organizes records by key fields—this requires key lookups in a hash table. Join (J) joins an input stream with a static table—this requires key lookups on the table while joining two inputs into an output stream [245]. Unlike filter, both grouping and join can be expensive due to the irregular access patterns of hash table lookups. Their cost depends on the hash table size, which corresponds to the group count and the static table size, respectively. Map (M) performs user-defined transformations on the input (e.g., parsing and splitting text logs) and its cost depends on the transformation logic.

To demonstrate the variability in resource usage of operators, based on the nature of input data, we profiled commonly used operators in monitoring queries. In particular, we profile the operators F, G+R,¹ and J operators on a real-world Pingmesh trace using a cloud node with 4 CPU cores. We configure the data input rate to 26.2 Mbps (Section 7.6 details why we consider this rate) and adjust several parameters that affect operator resource use: (i) filtering predicate for F to obtain different filter-out rates (between 20-80%), (ii) output group counts for G+R reflecting a range of grouping granularity and (ii) the size of static table for J which is joined with input stream. We observe that the F operator requires low compute resources (i.e., 17-25% of the CPU) with low variance even for such wide range of filter-out rates. On the contrary, as shown in Figure 7.2, G+R and J operators can be quite expensive with high variance (i.e., 50%-350% of the CPU) as they are quite sensitive to the number of groups and table size, respectively.

¹Reduction commonly follows grouping to apply aggregation functions incrementally per group, so both are often combined.

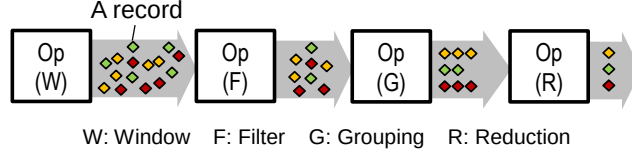


Figure 7.3: DAG of stream operations for the query in Listing 7.1.

Programming model. We adopt a declarative programming model [45, 46] to define a monitoring query in Jarvis. By this programming model, a monitoring query can also be generally expressed as a directed acyclic graph (DAG), similar to the applications modeled in Section 4.3. The vertices in the DAG denote monitoring operations which transform their input records and emit output records to downstream operations, and the edges denote how data flows between the operations, i.e., dataflow dependencies. Listing 7.1 shows how programmers can declaratively describe a real query on the Pingmesh data. In Figure 7.3, we present a simple DAG of how data flows between operations for the Pingmesh query.

```

/* 1. create a pipeline of operators */
query = Stream
.Window(10_SECS)
.Filter(e => e.errCode == 0)
.GroupApply((e.srcIp, e.dstIp))
.Aggregate(c => c.avg(rtt), c.max(rtt), c.min(rtt))
/* 2. Execute the pipeline */
Runner r( /* config info */ );
r.run(query);

```

Listing 7.1: A temporal query for server-to-server latency probing in every 10-second fixed-size window.

Hierarchical query engines. Physical resources involved in monitoring query execution can be viewed as a *tree structure* (Figure 7.4(b)), where leaf nodes represent the data source (i.e., server nodes) and the other nodes represent edge servers which are the intermediate aggregation nodes and root cloud node to generate final query output. Each data source node allocates compute and memory resources (e.g., 1 or 2 cores and 1 GB RAM) to facilitate monitoring queries. Such allocation of the monitoring resources is done in isolation without affecting the resources utilized by foreground services. Each data source sends data to its parent node and can leverage compute resources available in the parent node for query processing. The root cloud node receives data relayed by all intermediate edge nodes and completes the query to output the final result.

7.2.2 Adaptive Query Partitioning in Query Engines

Consider the volume of data transferred over the network observed in Microsoft’s Pingmesh trace. Each record in the trace corresponds to a single probe message for a server pair. A

record is 86B in size, including timestamp (8B), source IP address (4B), source cluster ID (4B), destination IP address (4B), destination cluster ID (4B), round trip time in microseconds (us) (4B), and error code (4B). Now, assuming that a datacenter consists of 200K servers and each server probes 20K other servers with a probing interval of 5 seconds [43], the data generation rate is estimated to be ~ 512.6 Gbps. Transferring such a high volume of data would strain network capacity and delay the query execution. Our observations are corroborated by many existing monitoring scenarios which exhibit *high traffic volume* [6, 237, 246] due to a large number of data sources and diverse data streams.

Monitoring queries typically permit data processing that incrementally reduces the data volume—e.g., Figure 7.3 shows that records are first filtered and then aggregated into a few statistical numbers. Thus, an effective approach to alleviating the network bottleneck is to process data *near the data source* and reduce its size before transmitting over the network. We use the terms placement and partitioning interchangeably, to refer to the placement of query operators and the data items that each operator processes, between data sources and remote edge, cloud nodes.

Changing resource availability. Resource demands of foreground services change dynamically over time; thus unused compute resources on data source nodes available for monitoring queries exhibit temporal variability. This is a common characteristic of data source nodes as they usually host services whose workloads change over time. As a consequence, several large-scale web services (e.g., Alibaba and Wikipedia) and ML inference services report experiencing bursty request loads in the order of minutes [247, 248, 249, 250]. These services require variable amount of compute resource to meet their SLAs despite changing request loads. Subsequently, query partitioning decisions need to be made promptly *at runtime* on *each data source*, to be compatible with dynamic resource conditions available to exploit on each data source.

Changing resource needs. Service failures in the datacenter can generate traffic bursts in error log volumes on each data source, until the failure is mitigated. Similarly, several real-world log analytics systems report high temporal variability in the record count of log streams, even across 1 minute time windows [6, 24, 251].

Anomalous behavior on server nodes can also cause monitoring data distributions to change. In our running example, network issues can cause spikes in server probe latencies. Analyzing our Pingmesh traces, we observed that such spikes can last in duration from 40 seconds up to a minute. Also, such spikes complicate query resource usage characteristics. Each operator’s compute demand is affected by the incoming data rate after executing all preceding operations in the query. In Figure 7.3, F operator performs filtering and leads to the first

data volume reduction from the input stream. Since erroneous (or high latency) records are usually not dominant, the operator happens to drop only a small fraction of records. A majority of records are then to be processed by G+R operator, increasing compute cost for the query. However, if we were to have network issues for inbound or outbound traffic for some servers, there will be a high degree of data reduction in F operator, thus lowering resource usage of the G+R operator.

Furthermore, data generation rates can vary across data source nodes. We observed highly variable data rates in the Pingmesh trace across servers, with 58% of the servers generating 50% or lower of the highest rate (details in [252]). The reason for this is that some servers are configured to probe a larger set of peers to cover a larger network range on behalf of other servers in the same ToR switch.

Summary. Each edge node in our setup can receive data from up to 250 data sources, and since edge node compute is shared by all these data sources, placement decisions need to be made jointly across all data sources. The exponential cost to make such placement decisions (in terms of number of data sources) would not be practical for online adaptation. The ability to quickly adjust partitioning plans (i.e., *query refinement*) would reduce the duration for which queries either over-subscribe or under-subscribe available compute resources. The over-subscription can lead to interference with foreground services in each data source node, whereas the under-subscription loses an opportunity to further reduce network traffic leaving the node. Prior work uses an ILP solver for online query refinement, but this is too costly: using it takes up to 20 minutes to compute a new query partitioning plan in Sonata [12]. Our goal is to develop an approach which not only performs fast query partitioning but also effectively reduces network traffic given the compute budget on each data source.

7.3 SERVER MONITORING INSIGHTS FOR PARTITIONING ON DATA SOURCE NODES

We discussed the potential benefits of data-level partitioning in fully utilizing the limited resources on data source nodes in Section 5.3.1. To highlight the effectiveness of data-level partitioning in large scale monitoring systems, we perform a simple empirical study using an analytics query in Figure 7.3 with a real-world Pingmesh system trace. More details on the trace are provided in Section 7.2.1. We run the query on 80% of a single 2.4GHz CPU set as the compute budget on a data source node. Figure 5.6(a) shows that the operator-level partitioning cannot execute the expensive G+R operator entirely under the compute budget, because F operator drops only a small portion of input records. This leads to network traffic as high as 22.5 Mbps, which is close to the input rate, without fully utilizing compute budget.

On the contrary, Figure 5.6(b) shows that the operator G+R can process 83% of its input under the data-level partitioning, resulting in total network traffic of 9.4 Mbps, i.e., 2.4x lower over the operator-level partitioning.

Can we use data synopses instead? Data synopsis techniques, such as sampling, histograms, and sketches [253], are commonly used to enable scalable stream analysis by trading off accuracy for performance [254, 255]. Despite their effectiveness and practicality, losing output accuracy may be undesirable for monitoring tasks such as anomaly detection, in which anomalies are hard to detect before their samples are fully processed [256]. Data that corresponds to anomalies is typically infrequent. Thus, the lossy output could lead to undetected anomalies that affect server reliability and additional costs for anomaly detection [257]. In Section 7.6.4, we compare the effectiveness (i.e., accuracy and network transfer costs) of Jarvis and a state-of-the-art window-based sampling protocol [254] using the query in Listing 7.1. In general, when network transfer costs between the two approaches are almost at par, we observe sampling misses a number of anomalous data points, resulting in high errors in the final query result.

7.4 JARVIS SYSTEM

Jarvis is a proof-of-concept system which implements our partitioning heuristic in a fully decentralized manner using novel system abstractions called Control proxy and Jarvis runtime. It aims at being fully transparent to users. To achieve this goal, Jarvis extends the query DAG to enable data-level partitioning, once it is constructed by the compiler for a user query. In this section, we discuss Jarvis architecture, its overall design, and the proposed workflow used to achieve fast and accurate query partitioning.

7.4.1 High-Level architecture

Figure 7.4(a) shows key system components involved in Jarvis *query manager* when a user submits a query. The query manager includes the *resource manager* that maintains the current list of data source, edge and cloud nodes along with their network topology in the *resource directory*. The *query optimizer* uses the topology information to generate an optimized physical plan from the logical plan for the query (as done in most streaming engines [45]). The *query deployer* compiles the query plan into an executable code and deploys it on Jarvis running on each node. Figure 7.4(b) shows how the query manager interacts with data source, edge and cloud nodes. Our work focuses on changing the query

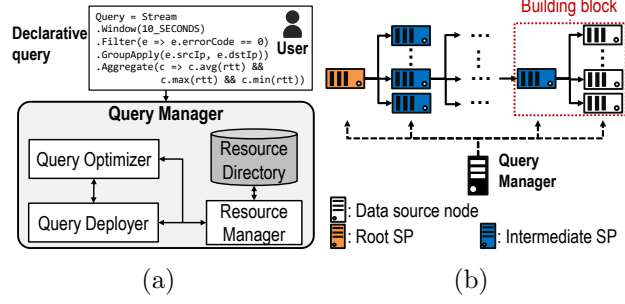


Figure 7.4: (a) Jarvis query manager, and (b) Monitoring pipeline architecture. SP refers to stream processor. L_0 to L_H refer to the hierarchy levels in a tree topology of height H .

optimizer to generate a Jarvis-friendly query plan and runtime optimizations after the query is deployed on the nodes.

To realize data-level partitioning at runtime, we introduce two new primitives to the query execution pipeline: (i) *Control proxy*, a unified abstraction for stream operator and (ii) *Jarvis runtime*, a system runtime that coordinates executions of all control proxies. Control proxy is a light-weight operator bridging two adjacent stream operators and decides “how many” records shall be forwarded to the next downstream operator v.s. drained to the remote edge and cloud nodes. Conceptually control proxy implements data-level partitioning at each operator.

Jarvis modifies the query optimizer to add control proxies between adjacent stream operators. Figure 7.5 illustrates the query plan deployed on data source and each of the remote edge nodes for the example query in Figure 7.3. Jarvis runtime deployed at each data source configures control proxies to execute a data-level partitioning plan for the query. Jarvis runtime continually probes the states of control proxies to observe the query state. If the query is in either idle or congestion state due to changes on resource conditions, Jarvis runtime compute a new data-level partitioning plan by reconfiguring the control proxies.

Jarvis runtime is fully decentralized, i.e., each instance of the query on a data source has a dedicated runtime instance which functions independent of other query instances and data sources. Its interactions with control proxies are also local to the node, requiring no coordination with an external planner or the edge/cloud nodes. This design essentially avoids having a central service performing a joint optimization across queries and data sources [12], which might be costly with a large number of data sources.

Control proxy can enable data-level partitioning to relieve congestion on the data source, but its design is distinct from conventional backpressure mechanisms. Rate throttling and data dropping [258, 259] can be used instead of control proxy-based data draining, to mitigate over-subscription of compute resources at data sources. However, they incur

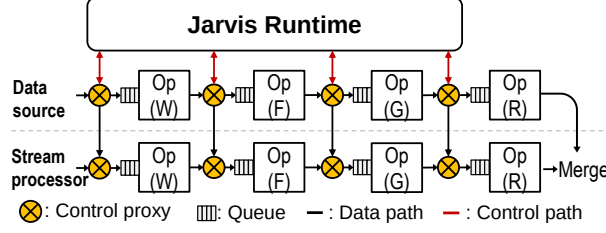


Figure 7.5: An overview of control proxies and Jarvis runtime. Data path: routing path for incoming data. Control path: interaction between control proxies and Jarvis runtime.

lossy processing such as data synopses (due to data sources currently not being persistent) and thus, have similar limitations. Lossless backpressure mechanisms such as resizing of operator queues [259, 260] and operator scaling [115] have limited effectiveness due to resource constraints on data source node. A detailed comparison with operator scaling is discussed in Chapter 3.

7.4.2 Query Plan Generation

The query plan generation in Jarvis is built on the conventional workflow of existing streaming engines [261]. The input query is first parsed to do syntactic checks. Then, logical optimizations (e.g., constant folding, predicate pushdown) are introduced to produce a logical plan. The optimized logical plan is used to generate a physical plan for deployment and execution. Jarvis inserts a control proxy between each of the adjacent stream operators in the logical plan, for deployment to data source and intermediate edge nodes—Jarvis runtime is also deployed on data sources.

Jarvis does *not* currently support some operations to execute on data sources. The supported operations are derived based on the constraints on application operators described in Section 4.3. The following describes the set of rules to identify such operations for server monitoring pipelines: **R-1**) Aggregation operations that are not incrementally updatable, such as exact quantiles. However, their approximate versions, such as approximate quantiles [183, 262], can benefit from Jarvis; **R-2**) Downstream operators succeeding stateful operations that require aggregation across multiple data sources; **R-3**) Stateful joins across streams. Prior work [12] has also excluded such operations because they are expensive and may not reduce data that needs to be sent out of data source; **R-4**) Multiple physical operators for the same logical operator, which parallelize operator execution (e.g. [11]). Data sources have constrained compute budget and hence benefits of exploiting operator parallelism are limited.

All rules except R-4 apply also to intermediate edge nodes. As the latter are dedicated to run monitoring queries, hardware-level parallelism can be exploited to accelerate query

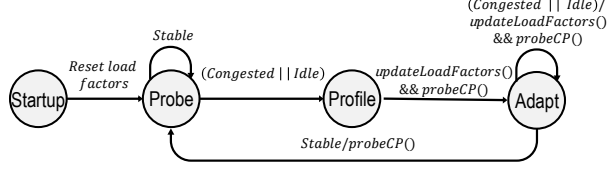


Figure 7.6: Jarvis runtime state machine.

operators. It may appear that R-1 limits queries that can leverage Jarvis. However, a significant number of real-world queries use operators that support incremental updates. For example, Drizzle [185] has reported 95% of aggregation queries on a popular cloud-based data analytics platform consist of aggregates supporting incremental updates, such as sum and count.

After applying the above rules, queries deployed on data sources typically consist of sequences of operators. Hereafter, while discussing query refinement on data sources, we restrict our discussion to sequential operator graphs. Nevertheless, our approach is extendable to handle graphs with split patterns that may execute on data sources—i.e., one operator output is an input to multiple downstream operators, and we leave this for future work. Also, all rules are described in a configuration file, so they can be extended.

7.4.3 Dynamic Query Refinement

Jarvis takes an approach of “replicating” query operators across data source, edge and cloud nodes to implement data-level partitioning, as shown in Figure 7.5. Once a query is deployed, Jarvis runtime steers each control proxy to select a portion p (i.e., $0 \leq p \leq 1$) of incoming data to be inserted to the downstream queue and consumed by the downstream operator. Remaining data is drained over the network to be processed by the control proxy associated with the same downstream operator in the edge and cloud nodes.

Jarvis can continuously refine load factors of control proxies in the data source node to adjust a query partitioning plan. This query refinement occurs at an adaptation epoch boundary defined by a time interval. During epoch processing, each control proxy monitors its downstream operator to identify the state of the operator to be one of the following: **Congested**) Operator contains more than a predefined number of pending records due to compute resource over-subscription on the data source, experiencing backpressure; **Idle**) Operator stays empty for longer than a predefined time duration due to under-subscription of compute resources on the data source; **Stable**) Operator is neither congested nor idle. Jarvis runtime collects state information from all control proxies at the end of an epoch and classifies the current data-level partitioning plan as *non-stable* if all operators are idle or

at least one operator is congested. Upon identified as non-stable, Jarvis runtime triggers adaptation to bring the query back to the stable state.

Now, we show in Figure 7.6 an overall workflow of operational phases in Jarvis to keep stable query executions:

- **Startup: initialization.** All load factor values are initialized to zero, so all records are processed by edge and cloud nodes.
- **Probe: normal operation.** At the end of every epoch, Jarvis runtime executes `ProbeCP()` function to query all control proxies and determine their congestion states. It continues to do so until it identifies the computation pipeline as congested or idle. At this point, the runtime enters `Profile` phase.
- **Profile: query plan diagnosis.** Jarvis obtains new estimates for the following during an epoch: (1) compute cost of each operator by executing an operator at a time, (2) reduction in the size of input stream by executing each operator, and (3) available compute budget for the query. These estimates are used to adapt load factors in the next `Adapt` phase.
- **Adapt: load factor adaptation.** Jarvis runtime computes a new data-level partitioning plan. Initial load factors are first calculated using the profiling estimates and set for each control proxy. Jarvis runtime executes `ProbeCP()` to probe the query state and perform iterative fine-tuning if necessary, until the computation pipeline is back to stable state. At this point, it returns to `Probe` phase.

Small workload variation in congested state or idle state can trigger a series of `Profile-Adapt` phases that put the control system in an oscillating behavior with small implications on the optimal load factors. To avoid this undesired behavior, each control proxy is configured with a threshold fraction of pending records in an epoch (`DrainedThres`) that can be drained by control proxies and tolerated by `ProbeCP()` without signaling congested state. Similarly, each control proxy is configured with a threshold fraction of epoch duration (`IdleThres`) that allows control proxies to stay idle and can be tolerated by `ProbeCP()` without signaling idle state.

7.4.4 Computing Data-level partitioning Plan

Jarvis runtime in its `Adapt` phase needs to compute a new data-level partitioning plan to stabilize the query execution. The details of how a new data-level partitioning plan is computed is already described in Section 5.3.2. It is done by solving an optimization problem that computes initial load factors and then iteratively fine-tuning load factors to stabilize the query.

7.4.5 Scaling to More Data Sources

Jarvis optimizes a core building block (shown in Figure 7.4), composed of a single cloud node and a number of data sources as high as 250. As the number of data sources increases, Jarvis replicates the core building block, resulting in a set of intermediate cloud nodes; each node aggregates results obtained from its data source nodes. A root node computes the final query result by aggregating results obtained from intermediate cloud nodes. As there is no communication between building blocks, the system can scale better with improved scaling of the core building block and availability of upstream network bandwidth between intermediate nodes and the root node.

7.4.6 Limitations

To maintain the effectiveness of Jarvis, (1) the epoch duration should be set properly to avoid invoking query partitioning decisions too frequently, and (2) query workload characteristics should not change dramatically during **Profile** and **Adapt** phases. We set an epoch as one second for our evaluation, and observe that Jarvis can take up to 7 seconds to stabilize a query during adaptation, making it applicable to scenarios where resource conditions change in the order of minutes (Section 7.2.2).

Currently, multiple queries in a data source node require that a single Jarvis runtime instance is dedicated to a single individual query. We adopt a fair resource allocation policy [263] to allocate the compute budget among the competing queries. Running multiple queries is evaluated in Section 7.6.6. Opportunities exist for a better resource allocation strategy as well as for operator sharing across multiple queries [23, 99]. We leave this for future work.

7.5 IMPLEMENTATION

Jarvis is implemented using lightweight query execution runtime Apache MiNiFi [63] for data source side and Apache NiFi [58] for cloud side runtime. RxJava is used to implement query computation pipelines within NiFi/MiNiFi custom processors. Kryo serialization framework [264] is used for transferring data over the network between data source and cloud nodes. Below, we highlight major implementation issues that Jarvis addresses, with a comprehensive discussion provided in [252].

Integration with existing streaming engines. Jarvis can be integrated with existing query engines on cloud side. On the data source with limited compute resources, unlike

stream processors which require dedicated server resources (e.g., Flink [45]), it is preferable to run a lightweight dataflow runtime such as MiNiFi. MiNiFi allows us to design and deploy custom dataflows so it is easy to incorporate Jarvis runtime and control proxy during query compilation (Section 7.4.2). MiNiFi agents send data to NiFi on cloud and we leverage NiFi’s integration already available with existing query engines. E.g. Flink provides NiFiSource and NiFiSink connectors to exchange data with NiFi [265]. An alternative architecture is to leverage Kafka and use NiFi as a producer [266]. Query instances on cloud only require adding control proxy to the dataflow (Figure 7.5), which is implemented using output tags [267] for splitting the incoming stream to multiple downstream operators.

Accurate query processing. Streams from multiple data sources are processed on the cloud side. Accurate stream processing requires: (1) consuming multiple streams by correctly merging watermarks from each stream to indicate time progress, (2) routing records to the right operator on cloud for further processing. For (1), we leverage the methodology used by Flink [268]. Each operator advances its time based on the minimum of all its incoming input streams’ event times. Since each control proxy on the data source generates an additional stream for drained records, incoming watermarks need to be replicated by control proxy for the drained path to reflect time progress. For (2), control proxy attaches an identifier for the operator on cloud which should receive records for further processing. Stateless operators can send their output to the next operator in the query. However, stateful operator outputs need to be sent to the same operator instance on cloud node, for reprocessing and merging with data from other streams.

Externalizing operator results on time: Congestion control method in Jarvis is designed for continuous stream computation model [269]. Therefore, to estimate congestion state for each operator precisely, all query operators must externalize computed results continuously to the next operator. With the implementation of G+R operator in RxJava, this was not the case as it collects all epoch records during grouping into a hash table and externalizes it with grouped records when an epoch ends. This does not reflect processing and idle time of next operator R as records never arrive during the current epoch, reporting an inaccurate congestion state and breaking our query refinement algorithm. We address this issue by dividing an epoch into sub-epochs and externalizing grouping results on a sub-epoch basis. This does not require changes on stream operators; instead, it requires each control proxy to initiate query refinement only when the control proxy encounters the end of the epoch, not sub-epoch. Jarvis currently divides an epoch into 100 sub-epochs.

Probing downstream queues. Some streaming engines may hide the operator’s internal queue state by abstracting data transfer between two operators with well-defined APIs [78].

Our RxJava based implementation also shared this concern originally. We address it by forcing synchronous queueing on the incoming data stream. This allows us to manage our own custom queues external to query operators, and the queueing time of records in the queue can correctly reflect the processing time for the operator.

7.6 EVALUATION

7.6.1 Methodology

Testbed setup. We deploy our data source on Amazon EC2 t2.micro nodes, each with one 2.4 GHz Intel Xeon E5-2676 core and 1 GB RAM. We use a larger Amazon EC2 t2.medium node with two 2.4 GHz Intel Xeon E5-2686 cores and 4 GB RAM as data source for the experiment of multiple queries. A cloud instance is deployed on an Amazon EC2 m5a.16xlarge node with 64 2.5 GHz AMD EPYC 7000 cores and 256 GB RAM. All nodes run Ubuntu 16.04.

We conduct our experiments on two types of setups: (1) a single data source node connected to a single cloud node to evaluate performance of query partitioning and refinement strategies in Jarvis, and (2) multiple data source nodes (up to 250) connected to a single cloud node to evaluate Jarvis as we increase the monitoring scale.

Performance metrics. We measure *query processing throughput* in Mbps (megabit per second) with a latency bound of 5 seconds, *epoch processing latency* in seconds, and *convergence duration* in number of epochs after resource conditions change. Performance results are obtained after running three minutes for the system warm-up.

Workloads. We use two datasets: **Pingmesh** dataset as described in Section 7.2.2 and **LogAnalytics**, a text-based log which includes tenant name, job running time in milliseconds along with CPU and memory utilization for handling tenant-wise performance issues for jobs running in an analytics cluster. We run the following queries:

- **S2SProbe** (Listing 7.1) on Pingmesh dataset. The filter predicate delivers 14% filter-out rate.
- **T2TProbe** (Listing 7.2) on Pingmesh dataset. It measures network latency aggregates for ToR-to-ToR pairs by joining the input stream with a table that maps server IP address to its ToR switch ID.
- **LogAnalytics** (Listing 7.3) on LogAnalytics dataset. It parses unstructured logs and bucketizes per-tenant latency and resource utilization to create histograms.

```
Stream
.Window(10_SECS).Filter(e => e.errCode == 0)
.Join(m, e => e.srcIp, m => m.ipAddr,
      (e,m) => (e, srcTor=m.torId))
.Join(m, e => e.dstIp, m => m.ipAddr,
      (e,m) => (e, dstTor=m.torId))
.GroupApply((e.srcToR, e.dstToR))
.Aggregate(c => c.avg(rtt), c.max(rtt), c.min(rtt))
```

Listing 7.2: A temporal query for ToR-to-ToR latency probing. *m* is a table to map server IP address to its ToR switch ID.

```
patterns={"*tenant name*", "*job running time*", "*cpu util*", "*memory util*"}
Stream
.Window(10_SECS)
.Map(l -> l.trim().toLowerCase())
.Filter(l -> patterns.stream().anyMatch(s->l.contains(s)))
.Map(j -> new JobStats(j.split('=')))
.Map(j -> j.stat = width_bucket(j.stat,0,100,10))
.GroupApply(j.tenant_name, j.stat_name, j.stat)
.Aggregate(c -> c.count())
```

Listing 7.3: A text query for computing histogram data for per-tenant analytics job latency and resource utilization. JobStats is an object to store job related information.

For Pingmesh, guided by [43], we set each server to probe 20K other servers at a time interval of 5 seconds. As a probe record is 86B, each server generates data approximately at 2.62 Mbps. For LogAnalytics, guided by [6] which reports text log data generated at 10s PB per day across 200K data source nodes in a production system, we set each server to generate 0.62 MBps (or 4.96 Mbps) of log data. For experimentation purpose, we scale up the data generation rate by 10x, i.e., 26.2 Mbps for Pingmesh and 49.6 Mbps for LogAnalytics per data source node.

Network configuration. We assume that a cloud node would have a network link of 10 Gbps [17]. For ease of experiment, we assume this bandwidth is fairly utilized across 250 nodes (guided by conversations with a large-scale datacenter operator) and 20 queries (guided by [12]) per node, allowing 2.048 Mbps effective bandwidth per query per data source node. We again scale up the obtained bandwidth by 10x to match with data rate scaling.

Baseline systems. we compare Jarvis with the following systems: (1) **All-SP** that runs a query entirely on cloud (i.e., Gigascope [95]), (2) **All-Src** that runs a query entirely on data source, (3) **Filter-Src** that applies static operator-level partitioning and runs only filter operations on data source (i.e., Everflow [17]), (4) **Best-OP** that applies a solver to dynamically allow the best operator-level partitioning (i.e., Sonata [12]), and (5) **LB-DP** (or **LoadBalance-DP**) that applies coarse-grained data partitioning at query level to split the input stream between data source and cloud proportional to available compute on the nodes (i.e., M3 [111]).

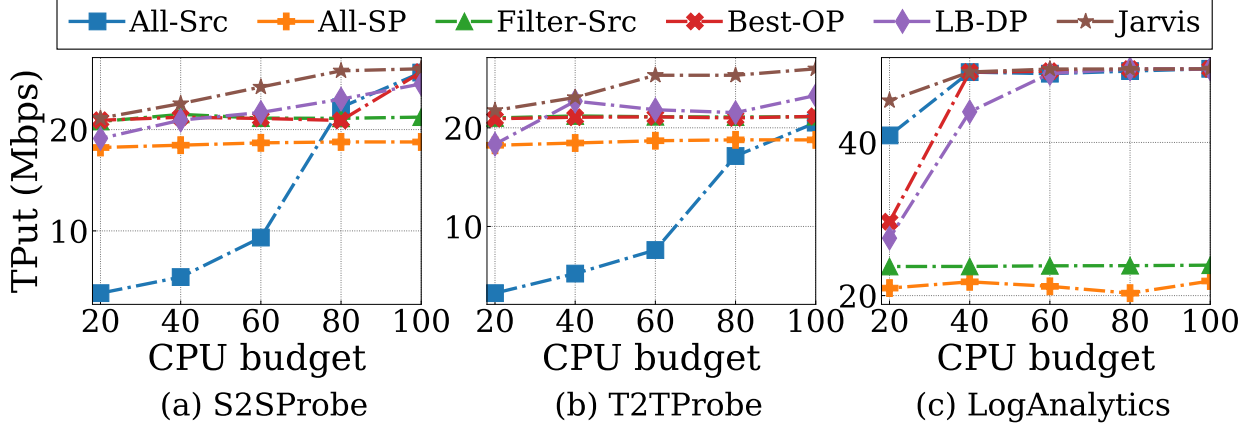


Figure 7.7: Query throughput over varying CPU budgets in % of a single core. Stream joined with a table of size 500 in (b).

7.6.2 Query Throughput Analysis

We use the single data source setup and evaluate query throughput for different partitioning strategies while varying available compute resource on data source. We note that Jarvis incurs little overhead, consuming less than 1% of a single core during **Profile** and **Adapt** phases.

S2SProbe query. Figure 7.7(a) shows the results of query throughput on S2SProbe. Jarvis outperforms other techniques in the 40-80% CPU budget range, with throughput gains of 2.6x and 1.16x over All-Src and LB-DP respectively at 60% CPU, and 1.25x over Best-OP at 80% CPU. The query requires nearly 85% CPU to execute entirely on data source. Thus, the throughput in All-Src declines drastically with CPU budgets lower than 80%. Network bandwidth becomes a bottleneck for Filter-Src as F operator is not effective in filtering out data. Best-OP executes F and G+R on data source only at 100%. For lower CPU budgets, it hits compute bottleneck and runs only F at the source as its compute cost is just 13%.

Jarvis partially processes the input of the G+R operator within available compute resource to reduce network traffic. In comparison, All-SP is restricted by available network bandwidth, and thus its throughput does not change with available CPU. LB-DP generates higher amounts of network traffic compared to Jarvis since its goal is to balance the compute load between data source and cloud nodes.

T2TProbe query. Figure 7.7(b) shows the results of query throughput on T2TProbe. This query requires more than a core to execute due to an expensive join (J) operator. Thus, All-Src cannot handle the input rate even at 100% CPU, and its throughput declines drastically as CPU budget lowers further. Jarvis does data-level partitioning to process the

input partially on the J operator, thereby reducing network traffic, outperforming other techniques in the 40-100% CPU range. Jarvis performs 4.4x better than All-Src at 40% CPU and 1.2x better than Best-OP and LB-DP between 60-100% CPU range. Note that the J operator is followed by a project on the fields srcToR, destToR and rtt, so the output size of the project is less than the input size of the J operator, leading to data reduction. Both Filter-Src and Best-OP execute only F at the source while Best-OP cannot accommodate J operator even at 100% CPU. LB-DP relieves compute load on data source but the reduction is not sufficient for significant throughput gains.

LogAnalytics query. LogAnalytics is relatively cheaper and uses 31% CPU to process the input at 49.6 Mbps. Nonetheless, Jarvis does data-level partitioning to reduce compute cost at 20% CPU budget, as shown in Figure 7.7(c), outperforming Best-OP and LB-DP by 1.5x. All-Src shows lower throughput than Jarvis at 20% CPU as it is resource constrained. Filter-Src executes filtering on data source, but is bound by network cost due to low filter-out rate. On the contrary, Best-OP can perform the filter and map operators at the source, thus outperforming Filter-Src. All-SP is always bound by the network, and hence Jarvis outperforms it by 2.3x in the 40-100% CPU range. For 20-40% CPU range, LB-DP transfers up to 45% of the input over the network, resulting in significant network bottlenecks.

7.6.3 Convergence Analysis

We evaluate how fast Jarvis adapts to changes in resource conditions on data source by measuring the convergence time in number of epochs. We compare Jarvis with **LP only** which runs StepWise-Adapt without fine-tuning (i.e., model-based approach in [125]) and **w/o LP-init** which runs StepWise-Adapt by initializing load factors to zero (i.e., model-agnostic approach in [125]). We do not compare with other methods that take minutes to compute a new query plan (e.g., Sonata [12]).

S2SProbe query. Figure 7.8(a) shows the results as we vary the available compute resource on the data source. Note that three epochs are required to detect that compute budget has changed to avoid triggering adaptation due to scheduling noise in the system. When the compute budget changes at the 3rd epoch (10%→90% CPU), Jarvis reduces the convergence time from six epochs in w/o LP-init, down to one epoch when employing initialization using LP solver. LP only also stabilizes the query using the output of LP solver. When the compute budget drops at the 18th (90%→60% CPU), the query reaches a stable state within two and four epochs for Jarvis and w/o LP-init, respectively. The additional epoch for Jarvis is required because profiling within a one-second epoch is not sufficient for G+R to process all

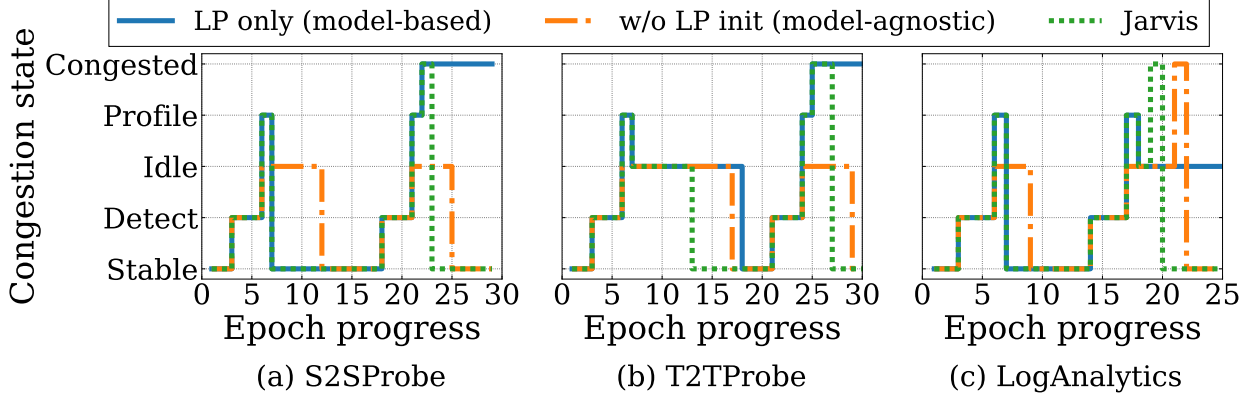


Figure 7.8: Convergence analysis of Jarvis compared with cost model-based and model-agnostic approaches.

records, resulting in less accurate estimates for the cost of G+R. The inaccurate profiling also prevents LP only from stabilizing the query.

T2TProbe query. Performance of a join-bound query is affected by the size of the static table. As shown in Figure 7.8(b), we vary the available compute and the size of the static table by starting with 10% CPU and a static table of size 50, then switching to 100% CPU in the 3rd epoch, and finally increasing the static table size by 10x to cause congestion.

Convergence duration is reduced from 11 epochs (in w/o LP-init) to 7 epochs in Jarvis, when the budget is increased to 100%. The additional epochs for Jarvis after profiling is attributed to the fact that the expensive J operator cannot be executed on all records to get accurate profiling estimates. As a result, the downstream G+R operator is not profiled accurately. Thereafter, fine-tuning plays a critical role in stabilizing the query in Jarvis. When the table size increases, the compute cost of J operator increases leaving no resources for G+R to execute. It takes 5 and 3 epochs to converge in w/o LP-init and Jarvis, respectively. The inaccurate profiling causes LP only to not converge for both changes in resource conditions. Note that we manually reset load factors to stabilize query for the next run, at 18th epoch.

Figure 7.8(c) shows the results of LogAnalytics. We see similar trends as observed in S2SProbe and T2TProbe queries.

Impact of number of operators. As we increase number of query operators, we expect w/o LP-init which runs StepWise-Adapt without LP init, to take longer to converge. We analyze algorithm convergence via a simulator that performs an exhaustive search of different execution configurations with possible query resource needs and compute budgets, while measuring associated convergence cost for each configuration. We find that convergence time can increase to as high as 21 epochs in the worst case, with four operators. Detailed

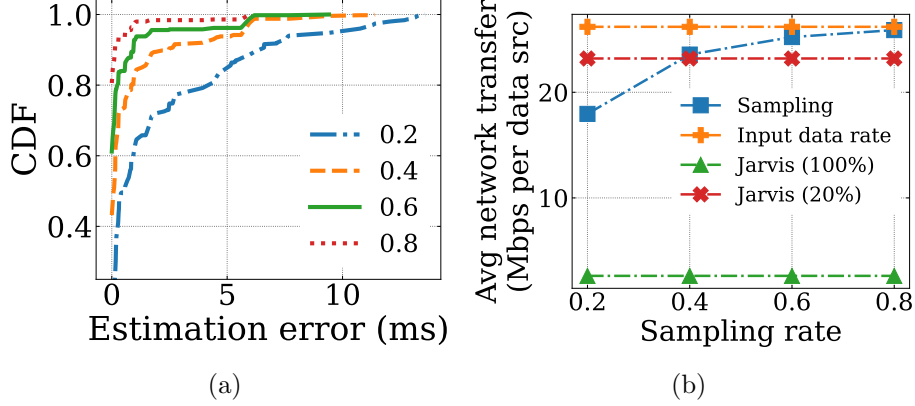


Figure 7.9: (a) CDF of estimation error for server probe latency using sampling. (b) Network transfer costs in sampling vs. Jarvis with 100% and 20% CPU budget.

analysis can be found in [252]. Our LP solver can be a valuable part of design in improving convergence cost in such cases.

7.6.4 Comparing Jarvis with Data Synopses

Data synopses can be used in our target scenarios to reduce network transfer costs. To see how promising they are, we choose a popular window-based sampling protocol (WSP) in [254], which constructs continuous samples from distributed streams in multiple data sources, to be applied to the query in Listing 7.1 on Scenario 1 in Section 7.2.1. When the search service latency SLA is violated, it degrades user satisfaction and reduces revenues [13]. Developers need to quickly correlate SLA violations with the alerts in Scenario 1 to determine if the violation is due to a network issue [43]. This necessitates accurate alerts to be generated. Thus, with different sampling rates in WSP, we plot the error in estimating range of probe latencies for each server pair. Figure 7.9 shows the results.

Given the alert threshold of 5 ms used in Scenario 1, we observe that with 0.6 and 0.8 sampling rates, we have 85-90% of the estimation errors within 1 ms. However, network transfer savings are not significant for these high rates (Figure 7.9(b)). For lower sampling rates (i.e., 0.2 and 0.4), we get significant reductions in network transfer costs (10-32% of input rate). However, the reduction that can be obtained using query partitioning in Jarvis is comparable or better (i.e., 11.4-90% of input rate) depending on available compute on data source. Critically, we note that 20-40% of the estimation errors exceed 1 ms for sampling rates of 0.2 and 0.4, and 20% even exceeds 5 ms for 0.2 sampling rate. The reason for the high errors is because high probe latencies which correspond to network issues are sparse in the dataset. When such probes are missed during sampling, the query can significantly

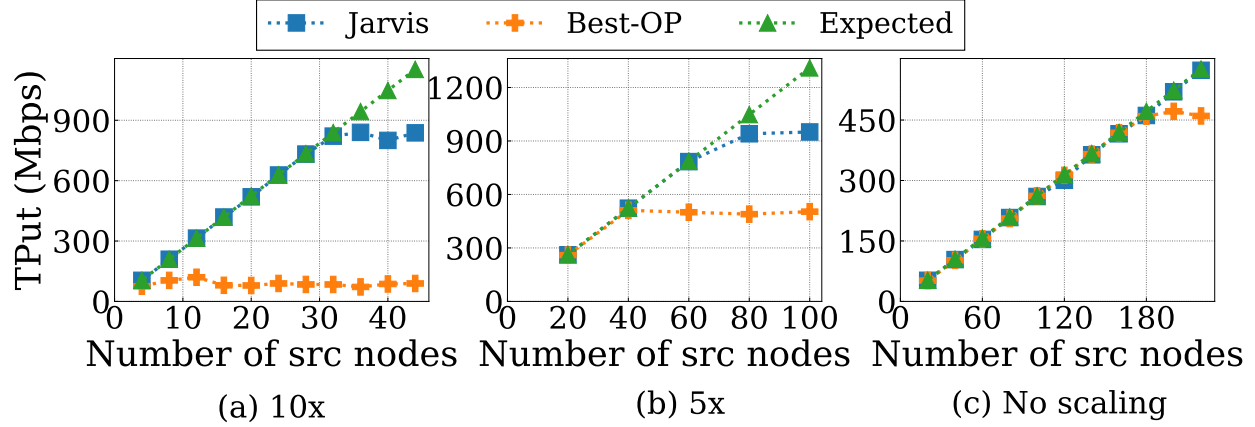


Figure 7.10: Query throughput over varying number of source nodes and different input rates.

underestimate the probe latencies observed by a node. We further observe that for 0.2 and 0.4 sampling rates that result in high network transfer savings, between 10%-38% of the alerts that should be generated in Scenario 1 are missed.

In summary, given the challenges due to accuracy loss in sampling, similar to the works in [12, 24, 246, 256], our work focuses on optimizing server monitoring pipelines which demand lossless and fully accurate query outputs.

7.6.5 Multiple Data Source Nodes

We discuss the efficacy of Jarvis when multiple data sources are sending data to a single cloud node. We compare Jarvis and Best-OP, i.e., the state-of-the-art in dynamic operator-level partitioning, on S2SProbe query while varying the following dimensions: number of data source nodes and input data rate per data source node.

We start with the input rate of 26.2 Mbps for Pingmesh, which is scaled by 10x over the calculated rate for this dataset. On each data source, we set CPU to 55% to ensure that Best-OP executes only the F operator while not fully utilizing the given CPU budget. Figure 7.10(a) shows the results as we increase the number of data sources. In Best-OP, F operator does not reduce data significantly, so the policy suffers from network bottleneck as soon as we add more data sources. In contrast, Jarvis scales up to 32 nodes without throughput degradation. Further scaling stops due to network bottleneck. Beyond 32 nodes, we observe backpressure which causes epoch processing latency to keep growing.

We now decrease the input rate to 13.1 Mbps (5x scaling) on each data source and set available CPU to 30%, to reflect the change in query compute demand from decreasing the

input rate. Figure 7.10(b) shows that Best-OP scales to 40 nodes after which it becomes network bottlenecked. Jarvis scales up to nearly 70 nodes, 75% improvement in number of data sources supported over Best-OP. When we further reduce the input rate to 2.62 Mbps in Figure 7.10(c) and allocate 5% CPU to the query, Best-OP starts to degrade in throughput at 180 nodes while Jarvis is seen to scale even for 250 data sources.

Query latency. Jarvis also improves epoch processing latency as a result of network traffic reduction. Based on the observed epoch latency distributions, we note for example, that Jarvis improves the median latency of Best-OP by around 3.4x in the configuration with 5x scaling and 40 data sources, when both policies are able to handle the input data rate. Moreover, Best-OP exhibits 1.8 and 5.2 seconds for the median and max latency, respectively, which are much larger than 500 ms and 2 seconds for the two measures in Jarvis. For configurations where Best-OP could not keep up with input data after it hits network bottleneck, such as 5x scaling with 60 nodes, we see the max latency of Jarvis to be within 5 seconds while Best-OP latency grows beyond 60 seconds.

7.6.6 Multiple Queries on Data Source Node

This experiment aims to investigate potential interference when we execute multiple queries with Jarvis on a single data source node. For this purpose, we run multiple instances of S2SProbe query while each instance is configured to use a fixed amount of CPU resource. Specifically, we measure an *aggregate* query throughput while fixing load factors for each query instance such that it utilizes 55% CPU for the input rate of 26.2 Mbps (i.e., 10x scaling). Figure 7.11 shows the results while varying the following dimensions: input data rate per query and number of cores on data source.

We observe that there is no significant interference among query instances until the system is bottlenecked by compute budget. Under system stress at 10x input scaling, single-core throughput saturates at 2 queries given per-query CPU demand. Two-core throughput does not increase beyond 3 queries, indicating saturation at 70 Mbps. At 5x scaling, per-query CPU demand drops to 30% and Jarvis supports up to four and six queries on a single- and two-core setup, respectively. At no input scaling, Jarvis supports 15 queries and 25 queries with one and two cores, respectively.

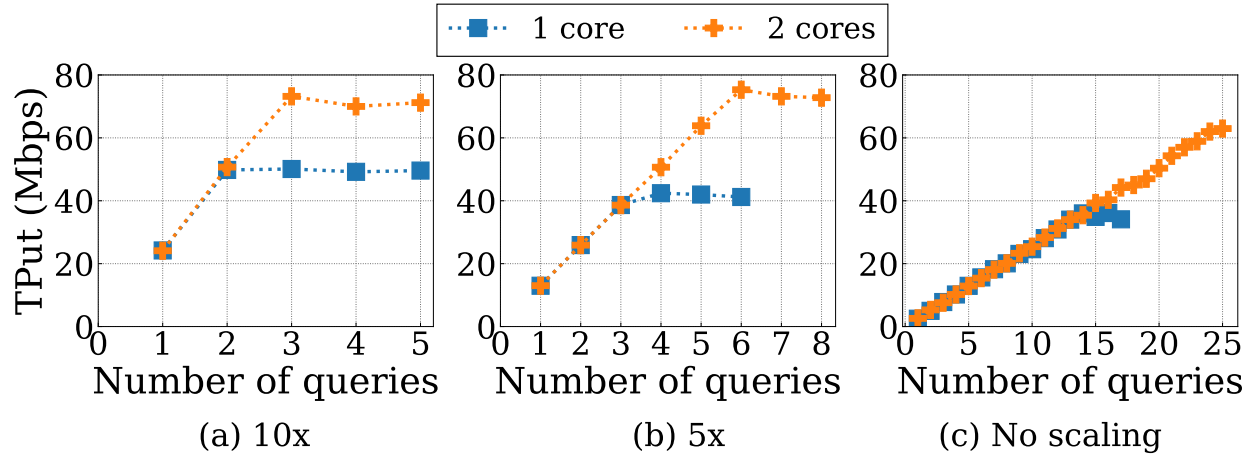


Figure 7.11: Query throughput for executing multiple queries on data source at different input rates.

7.7 CONCLUSION

We presented Jarvis, a new monitoring engine that targets large-scale systems generating high volume data streams. Jarvis performs fine-grained data-level partitioning of monitoring queries, to effectively reduce resource bottlenecks at the stream processor. Our technique can be implemented in a fully decentralized manner to scale to a large number of data sources. Our analysis using real-world monitoring query workloads suggests that Jarvis substantially improves system throughput while quickly adapting to fast-changing resource conditions on data source nodes.

CHAPTER 8: RESOURCE MODEL-AGNOSTIC POWER PREDICTION FOR MOBILE VIDEO ANALYTICS APPLICATIONS

In this case study, we address the problem of building power predictors for application operators on battery-driven data sources such as modern smartphones with increasingly complex hardware resources such as multi-core CPUs, GPUs and TPUs. Energy is an important resource consideration for making placement decisions. We highlight the challenges in predicting energy consumption of compute-intensive video processing pipelines running on mobile devices at the fine-grained operator-level, and address the challenges using the design and implementation of a proof-of-concept system called *eScope*. We then show how *eScope* could be used to perform placement decisions between a mobile device and remote server node to improve the application energy consumption cost on the mobile device.

8.1 INTRODUCTION

Mobile users want to run video stream analytics applications such as augmented reality (AR), face recognition, and machine learning (ML) programs on their devices. Such applications require significant compute resources on the mobile device and are energy hungry [270, 271, 272]. However, mobile devices have limited resources making it challenging [273] to support such applications. In particular, battery capacity is a severe limitation on mobile devices and a major source of dissatisfaction for mobile users [274, 275]. Understanding the energy consumption of applications running on the mobile device has been of interest to the research community. Several efforts have focused on model-based approaches, i.e, they develop power models for each hardware component on the mobile device, by profiling their power cost based on resource usage information. Resource use of operators within an application are then determined to estimate their power consumption. However, we observe that existing techniques are unable to meet the requirements of modern video analytics applications due to increasing complexity of power models to characterize energy costs of the underlying target hardware [169, 175, 176, 177] and the complexity of modern application operators which run in different power states depending on input. To address this problem, we develop *eScope*, a resource model-agnostic power prediction tool for operators within a mobile video analytics application, without requiring manual work to model the applications or their usage of the underlying hardware resources.

Existing tools provide application level power estimates based on the utilization of different hardware components (e.g. CPU, GPU, NIC, memory). Each operator within a video analytics pipeline performs an operation such as face detection, segmentation, classifier, etc.

Fine-grained energy accounting, i.e. on a per-operator basis, presents opportunities for making more accurate power predictions and optimization decisions. Power hungry operators can be placed on remote cloud servers and processed results received back to reduce energy use on the mobile device [20, 276, 277]. Configuration parameters (e.g., sampling rate) can be tuned at the operator-level for improved energy efficiency and performance [271, 278, 279]. Previous approaches [166, 171] generate power models on a target device, by exercising hardware components in different power states using either external power meters or mapping them to battery state-of-charge (SoC) estimates during training. Operator-level energy accounting is done by tracing hardware resource use of each operator, i.e., via code instrumentation. Such analysis requires profiling energy use of each hardware resource and annotating (instrumenting) the operator source code to estimate its resource use. We believe such analysis limits the potential for wider adoption of power prediction and associated optimization tools. Instrumentation adds to program complexity. And it is difficult to accurately model complex power characteristics of modern hardware [169, 174, 177, 178]. Recent works have proposed mapping software parameters (e.g., frame rate, resolution) to operator power costs using linear models [166, 178, 218]. However, an operator’s power cost not only depends on such configuration parameters, but also on the content within video frames (e.g., whether the incoming frame to a face detector contains a face or not). Moreover, operator power costs can have a complex and non-linear mapping to input features within the prediction model. With increasing software complexity, it becomes challenging to identify all the software configuration parameters of interest and their dependencies across operators. Constant maintenance is also required to keep track of the power profile for evolving software components.

eScope does not require manual work to build analytical power models for each of the hardware components or using application configuration parameters (model-agnostic/automatic), and is easy to adopt (transparent to the application code). eScope accounts for changing input data (dynamic) when making power predictions and does so at the granularity of each operator (fine-grained). Achieving these objectives is challenging because there are no tools to directly measure the power cost of each operator and we do not have visibility into operator code, to understand its resource usage. eScope collects easily accessible device-level energy draw information on the mobile device (i.e., SoC) and concurrently monitors changing composition of executing application operators, to attribute measured power draw in a time interval to the observed operators within that interval. A prediction model, which directly maps operator execution times to their corresponding power use, is then trained using this data and deployed on the mobile device to provide runtime predictions for making power optimization decisions. When used for energy management, we provide a discussion how eScope can be integrated existing placement frameworks to make energy-aware operator

placement decisions. Our evaluation shows that eScope is able to provide high confidence power predictions with 98% accuracy for operators across a wide range of execution profiles, and compute overhead of less than 3% on the mobile device. For complex execution profiles, it can provide prediction accuracy gains of up to 40% compared to traditional approaches.

This chapter describes the following contributions:

- A power prediction tool for mobile video analytics applications which is *fine-grained* (i.e., at the operator-level), *transparent* to the application code, infers *dynamic* information (thus handling changing input streams) and is *resource model-agnostic* (no manual work to support different hardware components or applications).
- A quantitative analysis of several state-of-the-art machine learning models which identifies ones that are best suited to make on-device power predictions for applications with complex power characteristics such as operators executing in different power states depending on their input, while incurring negligible overhead.
- A fully-functional proof-of-concept (PoC) implementation of eScope supporting mobile applications across two different languages: (1) SALSA [192] which supports actor-based programming and provides native support for code migration (to be leveraged for future extensions of eScope), (2) the popular Kotlin language [280] due its wide adoption for mobile application development. Our PoC is evaluated across several benchmarks and real-world applications, to show that it accurately predicts operator power costs and incurs low overhead on the mobile device.

8.2 MOTIVATING SCENARIOS

We consider mobile video analytics applications for a variety of scenarios such as face filters on social media applications (e.g., Snapchat [281]), video conferencing applications and augmented reality games (e.g., Pokemon Go [224]). These applications typically consist of compute-intensive operators such as face detection, object recognition, segmentation and so forth. These operations need to be executed on the incoming camera streams in real-time, with the goal to process data at user-interactive speeds.

Consider a mobile application developer who writes a face detector application consisting of a computational pipeline of operators (i.e., feature extraction, bounding box detection, face classifier) as shown in Figure 8.1. A mobile user downloads the application and decides to execute it on their camera feed. However, the user wants to configure it to not exceed a given energy budget, based on how long they want their battery to last. We could consider

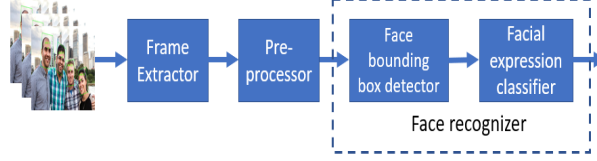


Figure 8.1: Pipeline of operators in a face recognition application.

placing resource-intensive operators on remote server nodes, so that the mobile device doesn't expend energy to execute those operators locally and instead directly gets the computation results from the remote node. However, a critical first step to enable such an optimization is to understand the energy cost of executing each operator on the mobile device. This chapter addresses the problem of efficiently estimating the operator energy costs on mobile devices.

Traditional approaches adopt a hardware utilization based modeling approach to predict the power cost of application operators. They assume that an analytical power model, to predict the power cost of each hardware component based on its utilization, is known beforehand. Building such a model requires identifying for each hardware component (e.g., CPU, GPU, memory), different variables (e.g., CPU utilization, frequency for the processor) affecting the component's energy use, as well as the nature of the function (e.g., linear, polynomial) that maps these identified variables to the corresponding power cost. Tools such as PowerTutor [171] can be trained on each target device to estimate parameters of each of its hardware component power models. An application operator's resource usage data is then profiled for each hardware component, and the outputs of pre-trained power models for the hardware components are summed up to obtain the total operator power cost. Other approaches rely on software parameters such as frame rate, and frame resolution, and train an analytical model to map them directly to operator power cost.

8.2.1 Challenges

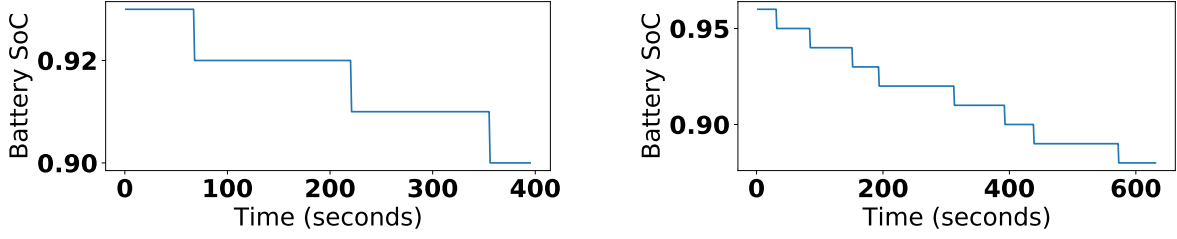
Existing power prediction techniques rely on analytical models which are becoming increasingly challenging to characterize for modern applications and complex hardware that they run on. With hardware advancements, each new compute resource (i.e., CPU, GPU, TPU) introduced in the market needs to be studied to understand variables that affect power/energy use. An analytical model then needs to be developed which accurately captures the relationship of each variable to energy cost, along with potential dependencies between the variables. When hardware evolves, we may need to re-evaluate the analytical models to account for changes that affect energy behavior. Each hardware resource poses different

challenges. E.g., models that estimate the power consumption of a CPU are typically based on CPU frequency and utilization, while modelling power consumption of GPU is more complex, and dependent on multiple variables (such as memory bandwidth and frequency of shader cores in the GPU and their interconnection bus topology) [175, 282]. Recent works [174] have also found that CPU idle power states play a critical role in power consumption of modern multicore CPUs. Previous works corroborate with our observation that building accurate analytics models for hardware power costs is a challenge to understand the energy usage behavior of mobile applications [169, 178, 283, 284].

Furthermore, every mobile device needs to be characterized separately for each of its hardware components. To observe the degree of variations in energy use of ML Kit [285]-based face detector application and a fixed input workload, across devices with different processors, we run the application with its input on two different mobile devices. For our setup, we use a **Pixel5**, which has Snapdragon 765G chip with Adreno 620 GPU (referred to as *Phone1*) and **MotoG5** which has Snapdragon 430 chip with Adreno 530 GPU (referred to as *Phone2*). While running the same workloads over the two phones we observe different behaviours in terms of battery drain characteristics. Figure 8.2 shows that we observe a high degree of variation (49%) in the energy consumption of the face detector between the two devices.

Application-level parameters have also been used directly to model the power cost of different operators [166, 284]. Such models can map software parameters such as frame sampling rate, image resolution and precision of weights for a deep learning model running in an operator, to the corresponding operator’s power cost. Similar to analytical power models built for hardware components, this approach also requires developers to manually identify the software components and their parameters affecting power, which becomes challenging for complex and fast evolving software architectures. Several recent works have identified the challenges with tuning application knobs in a large and exponential configuration space to improve the energy use of video analytics pipelines [27, 278]. Energy costs maybe different at each point in the configuration space, thus making it difficult to efficiently characterize their energy use. More critically, the content of input video frames affects power cost and is not accounted for by existing techniques. We discuss below in more detail, the impact of input video content on the power consumption of analytics operators. Finding input content representations and extending existing analytical models to estimate operator power costs is not trivial, and is a limitation of current software-based power models.

Input-dependent power consumption. Energy costs of video analytics applications not only depend on the input frame parameters such as frame rate, resolution, etc., as observed



(a) Total energy consumed on Phone1 to run the fixed workload is 465mWh. (b) Total energy consumed on Phone2 to run the same fixed workload as Phone 1, is 912mWh.

Figure 8.2: Comparing battery SoC over time for a fixed load, between *Phone1* and *Phone2*.

by previous works [27], but also on the content of each input frame. To quantify the degree of dependence of power cost on input frame content, we change the input to face recognition application (Figure 8.1) running on **Pixel5** mobile device, which has Snapdragon 765G chip with Adreno 620 GPU. The application consists of a face detector and classifier which classifies the facial expressions of people in the frames (e.g., smiling, open eyes). In the first case, the input stream consists of video frames with no faces. Each frame is processed by the face detection operator and upon detecting that there are no faces in the frame, it is dropped and not processed by downstream operators. In the second case, the input video frames contain one or more faces, and thus they are processed by all the application operators, including the facial expression classifier. We collected the data across three different resolutions for the input frames. Figure 8.3 shows the results. We observed a significant power difference with the second case consuming 13% to 92% more power than the first case, across different input frame sampling rates. This is due to conditional branching logic within the face recognizer, which discards frames early on in the computation pipeline, if they do not contain features of interest i.e., faces in our case. While for frames that contain faces, resource-intensive operators which extract the bounding box for the faces and classify facial expressions of interest are executed. The rate of frames processed by the application was increased by decreasing the incoming frame resolution, enabling more frames to be processed in a unit of time. We note that power costs increase with increasing frame rates, indicating that it is a dominant factor which affects power consumption.

We could argue that the power dependency on input content can be captured by defining fine-grained operators, at the granularity of face detection and classifier operations. Doing so can enable us to simply capture the execution of an operator as an indicator variable, within software-based analytical models proposed in previous works. However, the application operators are defined by developers and while we can trace the application execution at

the operator-level, we have no control over the granularity at which these operators are defined during application development time. Coarse-grained definitions of operators (e.g., at the application level) can result in sequential execution of operations with no parallelism benefits across video frames, while too fine-grained definitions of operators (e.g., at each branching statement), can result in significant control/scheduling overheads. Typically, analytics operations are defined in third party libraries (e.g., ML Kit face detector) and are called within end-user applications such as generating avatar’s from user images in a video chat or games that respond to the player’s expressions. Branching logic defined within application functions/operations is common and we find several branching statements within ML Kit code base as well [285].

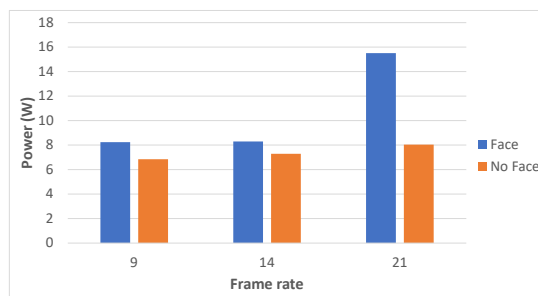


Figure 8.3: Input sensitivity of face recognition application when running with different frame resolutions, while changing the input content having faces vs. no faces at each resolution.

The dependency of resource usage and performance of video analytics applications on input data content (due to conditional branching logic within operator code), has also been corroborated by recent works [278]. Operators may have complex power characteristics due to their dependency on input frame content. Thus, we investigate an empirical approach for building powerful predictors to directly attribute power costs to application operator execution times. Our approach does not rely on analytical models built for specific hardware components nor does it require developing software-parameter based models. Instead, we investigate a resource model-agnostic approach which can be applied to different hardware and software components.

8.3 PROPOSED SOLUTION

Given an application to profile for power costs, the developer provides a training dataset which is representative of the different types of input data expected by the application. The application is deployed on a target mobile device, and a training phase is initiated. During this phase, we require disabling any user activity so that noise is reduced in collected training measurements. Such an initial training phase mechanism has also been adopted by previous works [171].

Section 5.4 describes the core modeling aspects of our approach. During training, we leverage coarse-grained power measurements available on the mobile device via battery status API, which provides state-of-charge (SoC) estimate or remaining amount of battery charge as a percentage of the total capacity. We also concurrently monitor executing operators on the mobile device, by instrumenting the scheduler of the underlying language runtime. In this section, we discuss the development of a tool called eScope, which implements our prediction technique and go over system level aspects of our solution, as it applies to mobile devices.

8.3.1 Power Measurement on Mobile Device

Mobile operating systems expose a battery status API, capable of showing its state-of-charge (SoC), which is the remaining amount of battery capacity as a percentage of the total. SoC cannot be directly measured. As described in [216], it is typically estimated using a hardware component called *battery fuel gauge*, which has an estimation error within 2% [286]. Our approach uses these SoC estimates to measure power consumed on the mobile device.

eScope registers with the battery status API, which notifies its listeners whenever there is a change in the SoC estimate. This estimate can be converted to energy consumed by the equation described in Section 5.4. Note that the minimum granularity at which a change in SoC can be sampled in most mobile devices is 1%. We refer to this time duration for 1% drop in battery SoC as the *discharge interval*. We chose to use the battery SoC estimates directly since it provides an easily accessible (does not require a power meter), cross-platform (relies on phone’s built-in battery sensor) solution for measuring power on the mobile device [169]. However, these estimates can be extremely noisy. In Section 8.3.3, we discuss pre-processing steps used to extract the relative difference in operator power estimates for offloading purpose. Another issue is that battery characteristics can change due to temperature and age [171, 173, 287]. In such cases, we retrain eScope on the device, when the predicted power costs do not match the observed costs at runtime.

8.3.2 Operator Execution Profile and Power Attribution

An execution profile captures within each discharge interval, time series data for when each operator instance in the video analytics application is scheduled, along with the timestamps when they complete execution. We refer readers to Section 5.4 for a detailed discussion on how the operator execution profiles are used, as well as how the power attribution is done for SoC changes with the executing application operators on the mobile device.

We collect the operator execution profile by instrumenting the underlying language runtime scheduler and collecting segment-level execution information. However, querying the scheduler API at the end of each segment (whose duration is in the order of 10s of milliseconds in our evaluated workloads) will increase the data collection overhead on the mobile device. Instead, we configure a *polling interval* for the frequency with which we query the scheduler API for execution profile data. At the end of each polling interval, the scheduler returns the segment information in the previous polling interval, including the operators running in each segment and the execution duration for the segment. Note that a large polling interval may also increase the amount of state that needs to be stored by the scheduler. We find that using a polling interval of 1 second worked well in our experiments.

8.3.3 Power Prediction Model

In Section 8.2.1, we saw that an operator’s power cost is sensitive to the input stream content. The operator can thus be in multiple power states during execution. To account for input dependency, we propose using the operator execution time on the epoch of video frames, as an input feature for our power prediction model. Power consumption of compute-intensive operators is affected by processor utilization (apart from other variables such as processor frequency). We note that operator execution time, as calculated from the underlying scheduler traces can capture the effects of processor utilization on power cost. When other application-level variables change (e.g., frame rate, high vs. low accuracy DNN models within an operator, frame resolution, etc.), we observe that it also affects the operator execution time. Thus, we use the operator execution time as our input feature for prediction, and rely on high quality input datasets (i.e., covering a wide range of input variables that affect power cost) for training our power prediction models. Moreover, we perform training directly on the target device and develop custom power prediction models per device, to handle variations across target hardware and their configurations. We show in our evaluation that with such custom, device-specific models, eScope can provide high accuracy power predictions for diverse video analytics applications.

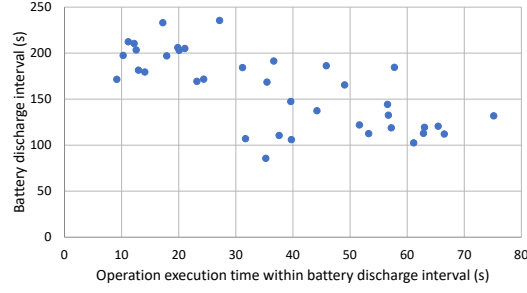


Figure 8.4: Example training data generated for eScope by the developer of a mobile video conferencing application, with the operator execution time and battery discharge interval being the independent and dependent variables respectively, from Equation 5.38.

Linear models which map power costs to hardware or application-level parameters (e.g., video frame rate, resolution) have been popular in modeling application power use [166, 171, 218]. In Section 5.4, we also saw that if the power cost of an operator is fixed, then a linear model can map the operator execution time to the battery discharge interval. However, operator power cost may not always be fixed and the mapping between power costs and operator execution time can be a complex, non-linear function depending on the workload. To demonstrate this, we ran experiments using a mobile video conferencing application (see Figure 8.8) that supports background subtraction and background blurring features during a video call. We varied multiple parameters for the input stream including (1) frame sampling rate, (2) frame resolution, (3) whether segmentation occurs within background removal operator or not (depending on the incoming frame containing a face vs. not). Note that the training data is currently expensive and time consuming to collect, as each data point requires a 1% battery SoC drop to occur. So, it is impractical to do a full sweep of the possible range of values that each parameter can take, for every combination of stream parameters. A video conferencing application may only operate in 2 modes: (1) when battery SoC is high, process frames at higher sampling rate, (2) when battery SoC is low, process at lower sampling rates, with the goal to maximize user quality of experience, subject to energy constraints. Thus, the developer can provide a representative set of training inputs, which depend on the target use cases for their application. An example training dataset for the video conferencing app generated by the developer is shown in Figure 8.4, with the X and Y-values representing the predictor and predicted variables for the linear regression model in Equation 5.38. We observe that the battery discharge interval varies widely, between 100-200 seconds depending

on the operator execution duration, with a correlation coefficient of -0.7. In Section 8.7, we will see that more complex prediction models (e.g., support vector regressor [288]) can perform significantly better than linear regressors on such data. Depending on the nature of input training data, an operator can exist in multiple power states. A frame consisting of a face requires executing an additional segmentation operation inside the background removal operator. So if the frame consists of faces, then the background removal operator is observed to be in a higher power state, compared to when the frames do not contain any faces and the operator executes in lower power state as the expensive segmentation operation does not need to run. eScope needs to be able to handle potentially more complex training datasets than the one visualized in Figure 8.4, depending on the target application behavior.

Based on above discussion, we need to explore more powerful prediction models (beyond linear regressors) to map the active duration for an operator execution to its corresponding discharge interval length in Equation 5.38 (Section 5.4). Before doing so, we consider pre-processing steps to clean up the data and reduce noise/measurement errors.

Pre-Processing

There are several sources of noise in the data that we collect from mobile device. We identify the following which can significantly affect our prediction models:

- Error in estimating discharge interval length due to variance in discharge duration across different intervals, for the same fixed load. Higher the load however, we have observed lower variance which makes it possible to better isolate resource-intensive operators of interest.
- Due to scheduling noise, we have observed active operator sets that may not be consistent with the expected execution pattern from application logic. E.g. two instances of the same operator type may appear in some segments when you expect only a single operator to be scheduled at all times. This could be because an operator that was scheduled in a previous segment, was able to execute only in the next segment.
- A background task collects the active operators during each execution segment, at the end of a polling interval. However, the background task may not get scheduled on the device if it is under very high load, or the memory buffers to store the execution segment information may be filled. These issues cause segment information to be potentially missed in a polling interval.

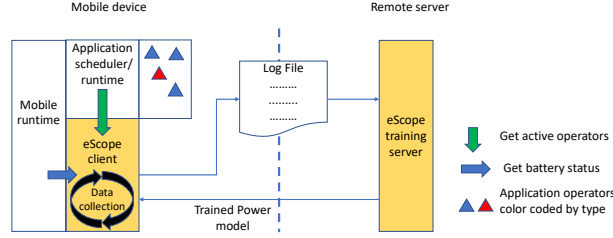


Figure 8.5: eScope system architecture.

Cleaning recipe: To eliminate the effects of measurement errors identified above, following are the pre-processing steps we take before feeding the data to our training models:

- Discard discharge intervals where (*expected number of times the segment data is polled* - *actual number of times the segment data is polled* > 10%) in a discharge interval. Expected number of polls in a discharge interval can be obtained from the duration of the discharge interval, polling interval timestamps and the size of each polling interval.
- We want to focus our analysis on operators which occur frequently in the execution profile and thereby provide us with sufficient training data to understand their power usage. So, for each segment type (defined by the set of operators constituting the segment) and discharge interval, we calculate the total segment execution duration within that interval. We then bucketize the observed segment execution durations across discharge intervals, by creating multiple segment duration bins. Discharge intervals are assigned to bins based on how long the corresponding segment executed within the interval. To make statistically significant estimations from the collected data, for each bin, we discard discharge interval durations which are outside two standard deviations for that bin.
- Before reporting the accuracy of a trained model, examine residual plots and the Cook's distance [289] to discard observations which are outliers for the trained model.

Training and Prediction

After pre-processing, we now have segment-level information for each discharge interval. This data is uploaded from the mobile device to the remote eScope server, where it is fed to a bag of prediction models for training. Note that eScope client on the mobile device continues to collect execution segment and discharge interval data in the background, improving the power model and redeploying it when model parameters change due to dynamic input conditions or application updates.

Even though Equation 5.38 suggests a linear regressor is sufficient to model the power consumption of active operators, complex components in several real-world applications including web browsers [166] and real-time face detectors (Section 8.7 show more complex behaviors. The profiling results of a face detector based on ML Kit library [285]) show that there are multiple power states depending on the duration for which the components execute. So for such complex components exhibiting multiple power states, training models which capture non-linear dependencies between segment power costs and their execution durations, would provide better accuracy. We analyze a comprehensive set of ML models for their accuracy and resource overhead for making power predictions on the mobile device:

- Linear regression: Based on Equation 5.38, linear regressor can model energy behaviors of active operators executing in fixed power states.
- Generalized additive models (GAMs): GAMs [290] are more flexible than linear models, since they estimate functions of the predictor variables which are connected to the dependent variable via a link function. But they require manually selecting features and choosing optimal smoothing parameters for those features.
- Support Vector: Support Vector Regression (SVR) [288] is useful when the dimensionality of feature space representation is much larger than number of training observations. It is suited for handling limitations of linear functions in high dimensional feature spaces.
- Tree-based regressors: Even though considered black box techniques that are hard to interpret, random forest (RFR) [291] and gradient boosting regressors (GBRs) [292] are ensemble learning methods which are easy to train, and capture complexities in the data very well. And unlike RFR, GBR is a boosting technique which can outperform RFR if tuned well but it also has tendency to overfit if the data is noisy.
- Neural Network (NN): A popular learning technique, neural networks are extremely flexible and can learn important features from complex input data. Even though they are difficult to train and require larger amounts of data, they can be computationally less expensive than techniques such as RFR.

We use nested cross-validation (Nested CV) [293] to evaluate each of our prediction models. The best prediction model after training is deployed back to the mobile device for making on-device and runtime power predictions. So eScope identifies the best model in terms of prediction accuracy which also minimizes overhead on the mobile device.

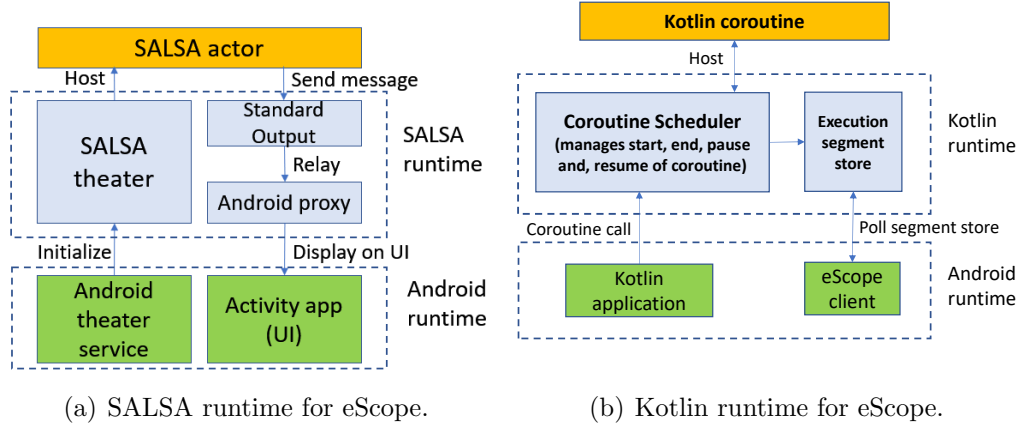


Figure 8.6: Modifications made to SALSA and Kotlin runtime to implement eScope in the two languages.

8.4 SYSTEM OVERVIEW

Figure 8.5 shows a high level overview of eScope. It consists of (i) a lightweight client which runs in the background on mobile device and, (ii) eScope server on a remote cloud machine. eScope client running on mobile device periodically queries GetActiveOperators API (green arrow), which we implement in the application runtime scheduler. The API returns the active operators within segments from the last polling interval. The client also queries GetBatteryStatus API (blue arrow) for battery SoC information, which is provided by the mobile device platform. The collected logs during training phase are transported over the network to the remote eScope server. Power prediction models are then trained on the collected logs on the remote server. The best model is identified for the collected training data, and downloaded by the client to provide application operator-level power predictions.

8.5 IMPLEMENTATION

To demonstrate the portability of eScope across different programming frameworks, we implemented eScope within two different language runtimes: SALSA [192] and Kotlin [280]¹. SALSA is an actor-based programming language, which enables leveraging the actor abstraction for implementing video analytics operators and providing power predictions at the actor level. The scheduler in the underlying runtime is instrumented to obtain the actor execution

¹This thesis highlights our contributions towards eScope design, quantitative analysis of ML models for on-device inference, eScope implementation and evaluation in SALSA. Dipayan Mukherjee is the primary contributor for characterizing energy use for real-world applications in Kotlin and investigating eScope applicability for Kotlin applications.

profile. Actors have native support for code migration, which can be leveraged for future work to extend eScope and support energy-aware code offloading on mobile devices. Kotlin is a popular and mainstream general-purpose programming language with strong support for Android development. We use Kotlin coroutines to model video analytics operators and instrument its underlying runtime scheduler to obtain coroutine execution profiles for eScope.

We used the Android [66] platform for our mobile device. SALSA compiles down to Java byte code, making it easier to port the runtime to Android through an extension. We leveraged previous work to run SALSA applications on Android [294], for our implementation. The ported architecture is shown in Figure 8.6(a), along with how SALSA actors interact with Android via the SALSA runtime. SALSA has the notion of *theaters* which provide a layer beneath actors for message passing, remote communication and migration [295]. `AndroidTheaterService` is implemented as an extension theater service in Android for hosting SALSA actors. The figure also shows the Java-based `AndroidProxyService`, which enables SALSA actors to communicate with Android driver functions. Standard Output actor is shown interacting with mobile display. The scheduler in the ported SALSA runtime is instrumented to implement *GetActiveActors* API (shown in Figure 8.5), which maintains a map of active operators to be read when the API is called.

Figure 8.6(b) shows the modifications to Kotlin runtime to implement eScope. In Kotlin, video analytics operators are implemented as coroutines [280]. A coroutine is an instance of suspendable computation in Kotlin. It is conceptually similar to a thread, in that it takes a block of code to run that works concurrently with the rest of the code. However, a coroutine is not bound to any particular thread. It may suspend its execution in one thread and resume in another one. Coroutines can be thought of as light-weight threads. We modify the underlying language runtime to extract execution information of co-routines and the calling operator. We trace the Kotlin run-time code path for co-routine scheduling. For an operator being executed within the application, we track each of the coroutine instances which are started/resumed or stopped/paused. We add the coroutine information in a dictionary in which each entry is keyed with its unique co-routine ID. The unique ID allows us to distinguish multiple calls from the same coroutine and leads to accurate power attribution. The Kotlin runtime API exposed to eScope client for querying segment information, uses the dictionary to store execution profiles of application operators. The API is queried by our client to obtain the operator execution information to send to the remote server during training phase. During inference phase (i.e., after training is completed on server and the best prediction model is deployed on mobile device), the Kotlin runtime API is queried to obtain the executing operators for which power costs are to be predicted, and monitor the accuracy of our prediction model.

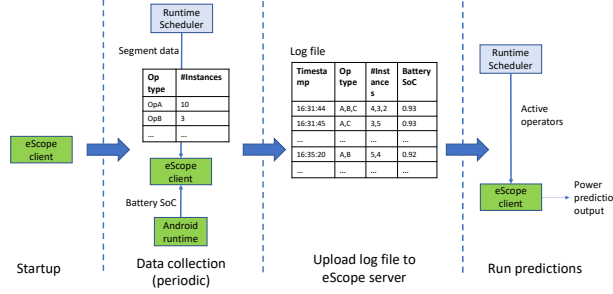


Figure 8.7: Phases of execution: (1) Language runtime scheduler is launched to enable operators to run, (2) eScope client periodically polls for segment information along with battery status (which is provided by Android’s BatteryManager service [197]), and appends timestamp before writing this data to log file, (3) Client uploads generated log file after the training data collection phase, to eScope server, (4) Client downloads trained model from server to provide on-device power predictions.

eScope client is an Android application (Figure 8.7 shows phases of its execution). Prediction models on eScope server are implemented using Scikit-learn [296], which is a Python module integrating a wide range of state-of-the-art ML algorithms. Neural networks were implemented using Keras [297], which is a high level API capable of running Tensorflow [298] in the backend. Keras is primarily designed for fast experimentation. For comparing the performance of ML models, we implemented nested CV [293] using grid search hyperparameter tuning [299] on top of the Scikit-learn APIs. Once we have obtained the best prediction model for the log data, we export the model to a PMML [300] file on the training server. It is then sent back to the mobile device where the client can import and load back the PMML file for runtime predictions.

8.6 DISCUSSION

8.6.1 Scaling to Large Number of Mobile Devices

The server can concurrently process logs from multiple mobile devices and eScope clients. It can be easily scaled by provisioning more cloud servers to carry out the training jobs. This is because each of the collected training logs is specific to a mobile device, and thus needs to be trained independently of the other devices sending data to the server. There is no data communication between training jobs across clients, so each job can be mapped to a server node. The training jobs are not currently distributed as we execute each job entirely on a server node. In the future, we can consider distributing the training jobs for faster training. In case of retraining the prediction model for a client, we currently adopt a simple policy of

considering each training job to be stateless, i.e., it does not accumulate the training data from previous runs for the client. This simplifies the need to train on the latest data which captures the hardware and application conditions on the device. We can extend the policy to incorporate more sophisticated strategies in the future.

8.6.2 Using eScope During Inference

eScope needs to provide operator-level power predictions to a placement framework, which enables making energy-aware placement decisions. For this purpose, we leverage the notion of adaptation epochs (time interval at which adaptation decisions are made) introduced in Chapter 4. Once the trained eScope model is deployed on the mobile device for inference, it is used to make operator-level power predictions for the next epoch, based on the observed operator execution time in the previous epoch. eScope is queried by a placement framework to make placement decision per-operator for the upcoming epoch, to decide whether to execute each operator locally on the device or remote server.

If the input stream content changes very quickly, then the adaptation epoch size may need to be configured to a small enough duration so that the system can keep up with the input changes and not be stuck in a sub-optimal placement plan. However, doing so requires frequently running eScope inference on the device, thereby increasing inference overhead. Thus, the effectiveness of eScope is maintained as long as the epoch duration is long enough to avoid making power predictions and invoking placement decisions too frequently. We note that input changes occur slowly in our target scenarios, in the order of 10s of seconds [301]. This allows our adaptation epoch sizes to be long enough (our evaluation was done for 1 second epoch size in Section 8.7 to stress the system but it can be configured for a longer period) so that the placement framework does not query for the operator power costs too frequently, while still keeping up with the rate at which input stream changes occur. E.g., for face recognition using within a social media application such as Snapchat, we use a video stream where the mobile user is using the front camera to classify their facial expressions and the stream is steady as long as the user is in front of the camera. Similarly video conferencing applications where the background blurring occurs, has a steady input stream for the duration of the call which can be in the order of several minutes.

Note that there is a key challenge with using the trained eScope prediction model during inference. The input features for the trained model are the operator active durations within a battery discharge interval. However, this means during inference, we need to wait for a discharge interval to occur (which can be ~ 140 seconds for real-world applications we evaluated), so the predictions may not keep up with the change in input data, which can be

in the order of 10s of seconds. To resolve this issue, we propose introducing an additional trained linear regressor which simply maps the operator active duration within a 1 second segment, to the corresponding total operator active duration within a battery discharge interval. E.g., if during training an operator is determined to execute for 400 ms in each segment of 1 second duration, then the trained model outputs that the operator was active for a total of 40% of the discharge interval duration, if there exists a linear mapping from the operator active duration within a segment and the corresponding duration within a discharge interval during training. In fact, analysis of our data collected for real-world applications suggests the correlation coefficient between operator active duration within a segment and total operator active duration within a discharge interval is between 0.82-0.93. Thus, as a first step towards enabling eScope inference at fine-grained time scales, we propose using the simplified linear regression model for resolving the mismatch in input features for training vs. inference. For future work, we will analyze the data for more real-world applications and the use of more powerful prediction models beyond the linear regressor, to find more accurate mappings between segment-level and discharge interval-level operator active durations.

We can safely assume that input streams change slowly enough for our target scenarios, so that power predictions and associated placement adaptation decisions can keep up with the input. In Chapter 9, we discuss ways to integrate the design of eScope with Jarvis to enable energy-aware placement decision making on mobile data sources.

8.7 EVALUATION

In this section, we evaluate eScope for its effectiveness in predicting power consumption across a wide range of benchmarks and real-world applications.

8.7.1 Experimental Setup

We ran real world experiments on the following mobile devices: (1) Samsung Galaxy Nexus phone (GTI9250)- Android 4.1.2, 2 ARMv7 CPU, 710.8 MB RAM, Li-Ion battery voltage 3.9V and 1750mAh. It has a removable battery which helped obtain ground truth for power drawn by the workloads, when directly connected to a power supply, (2) Samsung Galaxy S6 (SM-G920T)- Android 7.0, Octa-core (Cortex-A57 & Cortex-A53) CPU, 3 GB RAM, Li-Ion battery capacity 2550 mAh. (3) Google Pixel 5 - Android 12.0, Qualcomm Snapdragon 765G system-on-chip, with Octa-core (1×2.4 GHz Kryo 475 Prime & 1×2.2 GHz Kryo 475 Gold & 6×1.8 GHz Kryo 475 Silver), 8 GB RAM, Li-Ion battery capacity 4080 mAh and, Adreno 620 GPU. Real-world applications were run on Pixel 5.

For the power supply, we use a Topward 6306D Dual Tracking DC Power Supply [302]. eScope server is a 16 core machine with 2.7GHz Intel Xeon CPU ES-2680 processors, 6 GB RAM and running CentOS Linux release 7.4.1708.

Workloads: Based on benchmarks used in [215], we implemented several workloads in SALSA which exercise different resources on the mobile device: compute, network, memory and I/O. We used the workloads to generate different execution profiles. For complex profiles with operators in different power states, we implemented a simulator to allow us to experiment with large training data and a variety of execution configurations.

Real-world applications were written in Koltin. We evaluated two applications: (1) ML Kit-based face recognition application in Figure 8.1 and, (2) video conferencing application shown in Figure 8.8. The face recognizer identifies human faces and detects facial expressions such as smiling, frowning etc. We used ML Kit’s face detector library for running the experiments. ML Kit face detector extracts the generated video frames from the camera API, and then processes them to a resolution configured within the application. This pre-processing is done before sending the frames to the face detector. The face detector identifies the bounding box around the face and then a facial expression classifier identifies the type of facial expression in the processed frame.

The video conferencing application detects a face, identifies the segmentation mask of the background and then removes the background. Such a feature has become popular for video conferencing applications, which allows users to remove/change the background during conference calls. We leverage the face detection library from the first application, and then after detecting the face, a segmentation module is executed on the frame. This module provides a mask for the background, which can be applied to the selected frame, allowing us to remove the background from the frame and overlaying with a different background image. Note that the segment data polling interval is configured to 1 second in all our experiments.

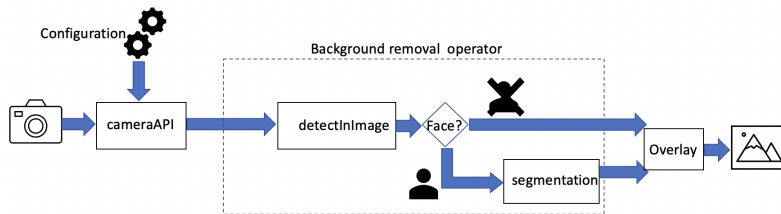


Figure 8.8: Video conferencing application which detects faces in the input video frames, and conditionally executes segmentation operation to identify the background and overlay it with a different background image.

8.7.2 Feasibility Analysis of eScope for Mobile Devices

In eScope, training and prediction happen alongside other workloads running on the mobile device so a critical design goal is to minimize monitoring resource overhead on the device. So we first evaluate the feasibility of deploying eScope on a mobile device, by investigating its power and CPU utilization overhead.

eScope client has an overhead on the mobile device associated with each of data collection, training data upload and prediction phases shown in Figure 8.7:

- **Collect training data:** Periodic sampling of training data incurs an additional power cost of 225mW on average. As a reference, idle power on the mobile device is 717mW on average. We could further reduce the collection cost by increasing the sampling interval duration which is currently 1 second.
- **Upload training data:** The size of a log file is 802.1KB on average, and sending it to the training server using our test device had negligible impact on energy (~ 17.94 J). This is about the energy cost of a single load for certain Web pages [166].
- **Prediction:** Once we have trained the best model, we deploy it to the mobile device and run it on periodically sampled operator execution profiles. Rest of this section describes the power and CPU utilization overhead associated with these models.

CPU utilization overhead: The prediction models are compute-intensive, so we compare different models based on their CPU utilization overhead. We use a fixed NQueens workload as reference and show the additional overhead introduced by each of the prediction models trained with the NQueens workload, in Figure 8.9. We can see that linear models and GAMs are the least expensive since they involve simple arithmetic operations once the coefficients of the model are determined during training. RFRs and GBRs use regression trees as their base learners. Though larger number of trees make them more stable predictors, it comes at a significantly high computation cost as can be seen in the figure. And finally, SVR and NN have similar additional overheads.

Power overhead: Since RFRs and GBRs have prohibitively high compute cost, we measure the power overhead of remaining prediction models. Linear models and GAMs present negligible power overhead while NNs are more expensive than SVR. But even NNs have additional power cost of only 72.5 mW, which is significantly cheaper ($\sim 30\%$) than the power overhead of data collection. Note that we continuously ran the predictions for each execution segment in this experiment so optimizations like caching power predictions on previously observed set of active operators can further reduce this overhead.

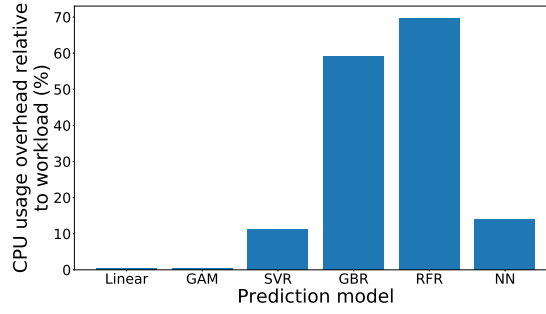
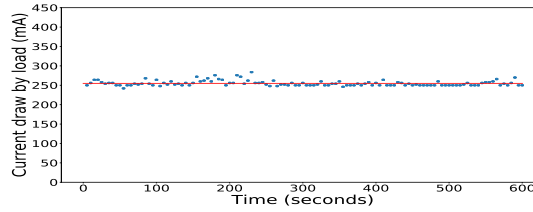
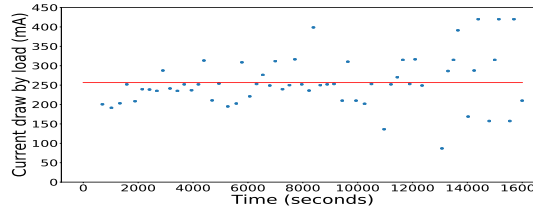


Figure 8.9: CPU utilization overhead relative to a fixed NQueens application workload



(a) Current draw measured from power supply



(b) Current draw measured from battery

Figure 8.10: Comparing current draw over time for a fixed load, between ground truth (power supply) and battery. Red line indicates mean current draw in both graphs.

We observed that running eScope presents low overhead even when compared to idle power on the mobile device. We also determine that it is not feasible to deploy certain prediction models such as RFRs and GBRs given their high overhead so we discard them from subsequent analysis.

8.7.3 Accuracy of Power Measurements on Mobile Device

As discussed in Section 8.3.1, we rely on battery SoC estimates for measuring power draw on the mobile device. It directly affects power predictions so we now evaluate the accuracy of SoC estimates.

We implemented a *Displayapp*, which fixes the screen brightness at the value input by

the user, providing a fixed workload for battery drain. Depending on the screen brightness setting, it can be a significant consumer of energy on the mobile device [181, 303].

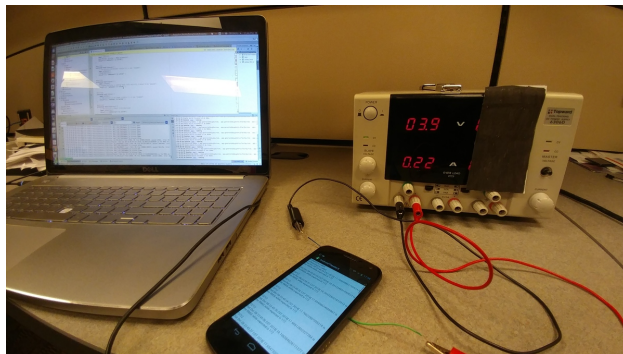


Figure 8.11: Experimental setup with mobile device connected to power supply

Figure 8.10 shows our results for accuracy of power measurements via change in battery SoC. The ground truth measurements are obtained by removing the battery from the phone and connecting the positive/negative terminals to the power supply (as shown in Figure 8.11). We fix the voltage at 3.9V (to match the rated voltage on the battery) and start sampling the current drawn by the *DisplayApp* (shown in Figure 8.10(a)). We then run *DisplayApp* with the same brightness setting but this time with the battery being the power source. We log the time intervals when the battery level changes. Given the total battery capacity is 1750mAh or 24.57kJ, a discharge interval can be estimated to account for 0.2457kJ ($0.01 \times 24.57\text{kJ}$) or 17.5mAh of charge. We can use the discharge interval length to obtain average current drawn from the battery during that interval (shown in Figure 8.10(b)).

The mean current drawn from the battery is within 0.59% error of the mean current drawn from power supply. Though the standard deviation in current drawn is higher for the battery (66.58mA), this is expected because the battery is not an ideal constant voltage source. The p-value obtained for the null hypothesis (that the two current series readings are coming from the same source) was large ($p=0.13$), indicating that we do not have evidence to reject the null hypothesis.

8.7.4 Prediction Accuracy of eScope for Different Types of Workloads

We now evaluate eScope to understand if it can provide high confidence predictions for operator power consumption across several different execution profiles.

Fixed Workloads

Running periodic tasks is a common requirement in mobile applications e.g. running manual animations, uploading data over the network, etc. To evaluate eScope for such tasks, we implemented fixed workload profile which has a fixed time period and duration.

Load	Description	Dominant Re-source	Linear RMSE	SVR RMSE	NN RMSE	Relative Power
NQ	Places N Queens on $N*N$ chess board	Compute	15.56	13.76	102.8	4.43 ± 1.06
Fib	Calculates Fibonacci series	Memory	11.9	11.63	88.52	2.56 ± 0.52
Trap	Area under the curve using trapezoidal rule	Compute	14.85	16.82	55.56	1.98 ± 0.2
Nums	Concurrent arithmetic operations	Compute	17.65	16.82	57.93	2.34 ± 0.17
Sort	External sort of contents of a file	Compute+I/O	4.32	5.63	71.99	6.16 ± 0.7
Scp	Copy file from mobile to server	Net+I/O	12.32	12.3	122.86	0.72 ± 0.1
Image	Read image and extract Haar-like features [304]	Compute+I/O	9.92	9.94	70.03	1.24 ± 0.6
NQ + Fib	Concurrently run NQ & Fib	Compute+Memory	13.87	16.58	90.38	NQ(4.87 ± 2.18), Fib(3.48 ± 2.13)
NQ + Sort	Concurrently run NQ & Sort	Compute+I/O	12.35	9.5	50.69	NQ(3.74 ± 1.25), Sort(5.56 ± 1.32)
NQ + Scp	Concurrently run NQ & Scp	Compute+Net+I/O	11.2	10.72	85.63	NQ(3.91 ± 0.84), Scp(1.22 ± 0.58)

Table 8.1: Benchmarks for fixed workload, with accuracy of different predictions models and estimated relative power of benchmark operators. Accuracy is in terms of average RMSE (in seconds) for discharge interval length, across folds using nested CV

Table 8.1 shows the fixed workloads along with accuracy of linear, SVR and NN prediction models (in terms of average Root Mean Squared Error (RMSE) in seconds) using 8-fold nested CV. We can see that the RMSE values are quite close for linear and SVR (SVR RMSE is within 10.7% of linear RMSE and its R^2 is within 8.2%). This means that linear models work well for fixed workloads, because operators execute in a fixed power state when they are scheduled. So it is preferable given its low resource overhead.

Table 8.1 also shows relative power of operators at the 95% confidence interval (last column). We used a bucket size of 10 seconds to pre-process the data, which includes discharge intervals with no active operators observed i.e. idle discharge intervals. Note that even though workloads contained multiple operator types, eScope provides relative power at 95% confidence level, for those operators which significantly affect discharge interval size. E.g. Trap had ExpStarter operator which was only responsible for creating multiple worker operators to calculate area under the curve in parallel, so its relative power was insignificant.

We can see in the table that NNs perform very poorly on our workloads. Training NNs is difficult because of the hyperparameter configurations we need to explore and also its accuracy suffers due to the lack of large amounts of training data. We validated that this was the case by merging training data from five runs of a complete battery discharge. We observed that the accuracy of NNs improved with increase in training data. GAMs weren't included in our analysis because they required manual parameter tuning for each application. Limitations of manual tuning considerably increased variance in accuracy we could obtain across applications. Since such a training approach is not scalable, we excluded it from our analysis.

Based on accuracy and overhead considerations of different prediction models, we identify linear and SVR as being ideal candidates for training on eScope server. Linear models have minimal resource overhead while SVR can better handle complex execution profiles (as we will see in Section 8.7.8) with acceptable increase in overhead.

Variable Workloads

Mobile applications can contain tasks that may not have a fixed duration/period, such as interactive tasks whose running time depends on user input. To model such applications and evaluate whether the battery SoC measurements are reactive to changes in workload across discharge intervals, we run variable compute and network intensive workloads for different durations across discharge intervals.

Compute: A compute-intensive operator was implemented to create 6 operator instances to run on as many cores, and create random number generators to perform arithmetic operations over the generated numbers. This application was scheduled to alternate between 120 seconds and 10 seconds across discharge intervals. eScope was able to obtain relative power of this operator to be 1.18 ± 0.14 at 95% confidence.

Network: A network-intensive *Upload* operator was written to upload 11.29 MB of data every second using WiFi network interface with upload bandwidth of 115 Mbps. The data to upload is maintained in memory so unlike *Scp*, there is no I/O cost but we upload 3x more

data. This application was also scheduled to alternate between 120 seconds and 10 seconds across discharge intervals. eScope was able to obtain relative power of this operator to be 1.27 ± 0.22 at 95% confidence.

We see that eScope is able to extract high confidence relative power estimates using linear models since operators execute in the same power state each time they are scheduled, even though the workloads do not have a fixed duration/period.

High Resource Utilization Workloads

[305] discusses how concurrently running applications such as game engines, web browsers, streaming music, etc. can cause full utilization/saturation of CPU cores. Increasing mobile traffic via applications such as video streaming and VoIP also cause network bandwidth saturation which can affect power consumption characteristics.

Unlike resource-usage based power modeling, relying only on operator information can potentially result in non-linear behaviors during resource saturation. So we evaluate eScope’s performance during saturation of compute and network resources:

Compute: We ran multiple compute-intensive operators concurrently in order to fully utilize up to a maximum of 8 cores available on our test devices. We varied the range of concurrently running operators in each segment between 4, 6, 8 and 10 operators. We observed an average RMSE of 0.9 ± 0.1 seconds and an average R^2 of 0.998 in estimating the discharge interval size for these workloads with a linear model.

Network: In order to exercise network bandwidth, we concurrently ran multiple *Upload* operators in each segment, each one uploading 3.7 MB of data to a server. Up to 7 operators were executing in each segment, on a network link with mean available bandwidth of 88 Mbps. We see that the RMSE in estimating drain interval length across these workloads is 9.35 ± 4.18 seconds and R^2 is 0.98 with a linear model (see Figure 8.12).

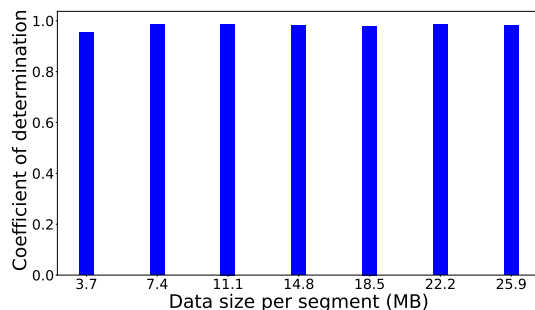


Figure 8.12: eScope accuracy under network bandwidth saturation

eScope’s linear model doesn’t suffer from performance degradation under resource saturation conditions because our representation for operator execution profiles explicitly captures concurrently running operators in each segment.

8.7.5 Comparison of Power Modeling with Related Work

Based on the works described in Section 3.3, we use the following two representative power modeling approaches to compare eScope with in this section: (1) MAUI [20] is a code offloading framework which relies on profiling energy use of each resource and then instrumenting source code to estimate resource usage of each application component, (2) RECON [166] is an energy debugging tool which monitors both hardware resource usage and executing application components in order to do fine-grained application power modeling.

Figure 8.13 shows that for workloads utilizing a range of resources such as compute, memory and network, eScope is able to capture the variance in power estimates as well as the other two techniques based only on the operator information which is much easier to obtain. Tracking of *fine-grained* operator execution traces closely corresponds to resource utilization changes during its execution that affect power consumption behavior.

For more complicated workloads where the operators may execute in significantly different power states when they are scheduled, eScope relies on non-linear models such as SVR to accurately model the operators power behavior.

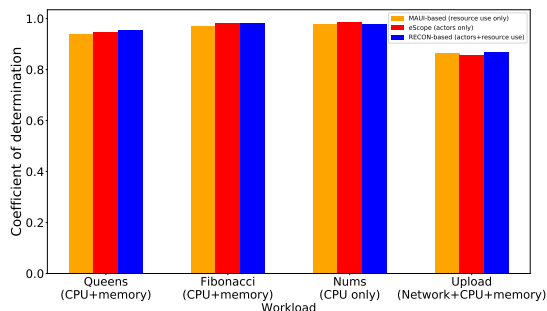


Figure 8.13: Comparing the accuracy of eScope with other power modeling techniques

8.7.6 Summary

We now provide a summary based on our evaluation results in section 8.7. We have established that battery SoC based energy measurements can provide accurate runtime estimates of energy consumption of operators in applications. We then evaluated our bag of

prediction models on the training server, for their performance on our application workloads. Below we summarize our observations regarding the performance/tradeoff of each prediction model:

- **Linear Models** have very low additional overhead on the mobile device and show very good accuracy values (± 17.65 seconds error in estimating drain interval length) on the benchmark and real world applications we evaluated. The reason it works well is because operators provide modularity in application design, making them good candidates as an unit for energy monitoring. Further, we are observing operator behavior at the granularity of an execution segment but our predictions are being done at the coarser granularity of a drain interval. So as long as complexities in energy consumption behavior are contained within a drain interval, they will not affect estimation model accuracies. This is the reason linear models did well when compared to resource utilization based models. For placement scenarios, such coarse-grained predictions suffice to make optimization decisions such as those described in [215].
- **GAMs** are impractical due to the difficulty in training these models and resulting variance in accuracy across applications.
- **RFRs and GBRs** have a very high overhead of running the models on the mobile device. Even though they have good average accuracy, they still have high variance in their accuracy estimate across applications. These observations are attributed to the fact that given sparse training data, these tree based learners are at the risk of selecting a subset of uninformative features at each split, negatively affecting predictive performance by creating multiple trees voting for the majority class [306].
- **SVRs** have acceptable overhead, while their accuracy is only slightly better than linear models. They are more powerful since they can capture non-linearities and work especially well for complex datasets with large number of features and small training observations.
- **NNs** have acceptable overhead but have unacceptably low accuracy on the benchmark applications.

So we have seen that linear models perform really well on the applications we evaluated and have very low overhead. SVRs are also very accurate and have acceptable overhead, though their CPU overhead is relatively higher than linear models at $\sim 10\%$. Lasso regression has its limitations (see [[307]]) however, so the higher overhead of SVRs would be acceptable if they can better capture complex execution profiles. E.g. we know lasso regressors would suffer

if operators execute in multiple power states or there are dependencies between features in training data. Each feature in the training data corresponds to an operator type and its instance count.

8.7.7 Real-World Applications

We evaluated both the real-world face recognizer and video conference applications with a variety of inputs and configurations. We configured the resolution of the video frames by sweeping through the possible resolution values between 144p to 1080p, which can provide insight into how the input image resolution impacts power costs. We also varied the sampling rates at which the frames are consumed by application. Both the frame rate and resolution have been identified as parameters which significantly affect the energy cost of video analytics applications in previous works [27]. Furthermore, we varied the input frame content as follows: (1) *Without faces*: this was achieved by pointing the camera towards an empty background (similar to the input feed if the caller or human user is away from the phone), (2) *With faces*: this was achieved by pointing the camera towards a screen containing a human face covering 50% of the screen area. Based on the data, we observed that overlaying the background image was consuming negligible power compared to segmentation in the video conference application, hence we didn't consider the final overlay operator to simplify our subsequent analysis. For the face recognizer, we observed that if eScope used a linear regressor for predicting the discharge interval duration for the workload, the RMSE with cross-validation was 17.6 ± 11.24 seconds. However, if eScope used SVR model, the RMSE was 14.3 ± 9 seconds. Thus, we have a significant drop in accuracy when using the linear regressor instead of SVR for our power predictions. For the video conference application, we observe RMSE of 26.37 ± 12.9 seconds using a linear regressor, while SVR gave RMSE of 24.7 ± 16.5 . The difference in performance between SVR and linear model is relatively higher for the face recognizer, which incentivizes deploying SVR model for inference on the mobile device, at the cost of higher overhead compared to linear model. A heuristic is used to make this decision, wherein if SVR performs better than linear model by $> 15\%$, then the SVR model is chosen by eScope despite the higher overhead cost. In the video conference application, the difference in performance is not very significant and eScope chooses the linear model for inference. We note that the RMSE in predicting the discharge interval duration can be reduced by collecting more execution logs on the mobile device during training phase. Thus, more powerful models such as SVR can be beneficial in improving the power prediction accuracy for complex real-world applications.

8.7.8 Detailed Analysis of Operators in Multiple Power States

We saw in Section 8.7.7 that eScope can benefit from complex prediction models such as SVR for real-world applications. Other mobile applications also exhibit complex resource usage characteristics similar to video analytics. Browsers load web pages which contain components utilizing different resources, such as compute and then network in an image/gif download component for *fico.com* [166]. Below, we use a simulator to simulate more complicated execution profiles to further investigate non-linear power characteristics between segment power cost and its execution duration, along with benefits of using SVR compared to linear models.

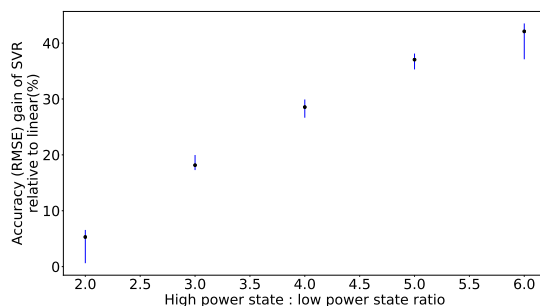


Figure 8.14: Accuracy gain (in terms of RMSE) of SVR relative to linear models with increasing power ratio between high and low power states of an operator. The bars represent the range of accuracy gains for each power ratio, when varying the training data size from 100 to 1000 discharge intervals

The simulator generates different execution configurations, while using real world datasets we had collected from mobile devices to model discharge intervals, active operators in execution segments and noise characteristics. Figure 8.14 shows that SVR can perform significantly better than linear models (up to 40%) for reasonably large ratios between operator power states. For a fixed power state ratio of an operator, Figure 8.15 shows the trend of increasing accuracy gain of SVR relative to linear, with increase in the training data size. We observed a similar trend for other power ratios we evaluated as well. So we see that linear models have low resource overhead but SVR can provide significant accuracy gains when monitoring non-linear workloads.

8.8 CONCLUSION

We presented a power prediction mechanism for estimating the energy costs of compute-intensive video-analytics applications running on mobile devices. Our proposed empirical

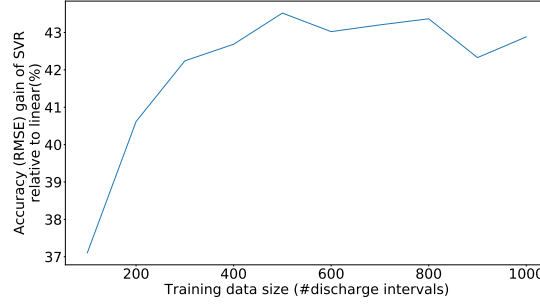


Figure 8.15: Accuracy gain (in terms of RMSE) of SVR relative to linear models with increasing training data size (i.e. number of discharge intervals). The power ratio between high and low power states for this operator was 6.

method is resource-model agnostic, so it doesn't assume an accurate power model based on resource usage of application operators, being available for each target hardware component. Our technique predicts operator-level power consumption using device-level energy draw information in conjunction with an application's execution traces. Our approach meets the stated design goals of being fine-grained (i.e., predicts power costs at the granularity of individual operators within the analytics pipeline), easy to adopt with no application-code level changes required, dynamic (i.e., can capture the impact on energy costs, of changing input at runtime) and easily portable to new hardware and applications. Such a technique can facilitate energy management solutions which can execute energy-intensive operators on a remote server, thereby reducing on-device computational power costs. We implemented our prediction technique on Android devices in a tool called eScope. Our evaluation showed that eScope is able to model the energy costs with high accuracy across a diverse set of workloads and does so with minimal overhead. We also quantitatively evaluated several ML models to provide a feasibility and performance analysis for making power predictions on a mobile device.

CHAPTER 9: CONCLUSIONS AND FUTURE WORK

We summarize and conclude the work we presented in this thesis. We then discuss interesting directions for future work.

9.1 SUMMARY AND CONCLUSIONS

We presented several different challenges with regard to placement of streaming application operators in near-data processing systems. We began by going over several motivational scenarios where near-data processing is beneficial for improving application performance in the video processing and query analytics domains. We then identified two resource topologies with different resource characteristics at the data source layer. These differences result in treating the two topologies separately from a placement point-of-view, and with different algorithms to scale placement solutions for the two topologies. For type I topology, we established the NP-hardness of the problem and proposed a dynamic programming-based heuristic to scale our placement solution for large number of data sources. We then proved that the proposed heuristic does not scale to handle type II resource topologies. A greedy version of the problem for near-data placement on data source nodes was formulated, to reduce the number of resource nodes to consider for making fast placement decisions. The new formulation was also proved to be NP-hard, and therefore another greedy heuristic algorithm was designed to make placement decisions on each data source node. The algorithm was implemented in a fully decentralized manner to make node-local placement decisions. Finally, we take the first steps towards enabling energy-aware placement decisions on battery-driven data source nodes by addressing challenges in making fine-grained (i.e., at the operator-level) power predictions for video analytics applications without making application code-level changes. Our power predictor does not rely on analytical resource models that are becoming increasingly complex and inaccurate with the advent of modern hardware and accelerators such as multi-core CPUs, GPUs and TPUs.

With the rise in streaming data volumes from various sources such as cameras, IoT devices, server nodes (generating logs and health metrics) and smart sensors, there is increasing demand from data owners to process the collected streams in a timely manner to obtain actionable insights from it. Current systems infrastructure cannot scale to handle the resource needs of modern stream processing applications. Near-data processing presents a promising opportunity to scale stream processing systems and support resource-efficient processing of massive data streams. However, efficiently computing placement solutions to map operators

to resource nodes so that we can leverage near-data resource nodes at the edge as well as compute resources on data source nodes, is not trivial. This is because of the large scale and heterogeneity of data source nodes that generate massive data volumes that need to be considered for placement as well as dynamic resource conditions on near-data resources due to compute being shared across multiple resource-intensive applications and the nature of application logic which causes compute resource needs to be sensitive to changes in the incoming data streams to be processed. We show that there does not exist a single placement solution which can scale to large number of operators in the application graph and handles large number of resource nodes for placement. Instead a hybrid approach is required, wherein (1) a centralized solution with a global view of resource nodes can be considered for making fast placement decisions between edge and cloud nodes, thus enabling the minimization of end-to-end application processing time subject to compute resource availability and, (2) a fully decentralized placement mechanism which makes local placement decisions can be considered at the data source-layer, due to the large number of resource nodes at this layer. Moreover, resource model-agnostic approaches to profile the execution costs of application operators on resource nodes, can significantly improve the quality of placement solutions. This thesis is a first step towards the development of adaptive and scalable placement solutions for distributed and heterogeneous resource environments, with the goal to optimize application performance and resource-efficiency.

9.2 DISCUSSION AND FUTURE WORK

Our work in this thesis lays the groundwork for several interesting directions to explore for future work on resource-efficient, adaptive and scalable placement strategies for streaming analytics applications over data source-edge-cloud environments. We discuss them below:

9.2.1 Integrating Placement Solutions Across Data Source-Edge-Cloud Resources

In this thesis, we have implemented the proposed placement algorithms for type I and type II resource topologies, with the architectural components designed for the target application domains. One could however, integrate these solutions to potentially improve the quality of placement solutions at the edge and cloud layers. Specifically, in Jarvis, we implement the greedy placement solution in fully decentralized manner at all the 3 layers (i.e., data source, edge and cloud) in our target resource topologies. However, given a placement decision made by Jarvis locally on the data source nodes, the aggregate data load sent for processing at the edge and cloud layers can be profiled. Furthermore, the pending operators to be executed on

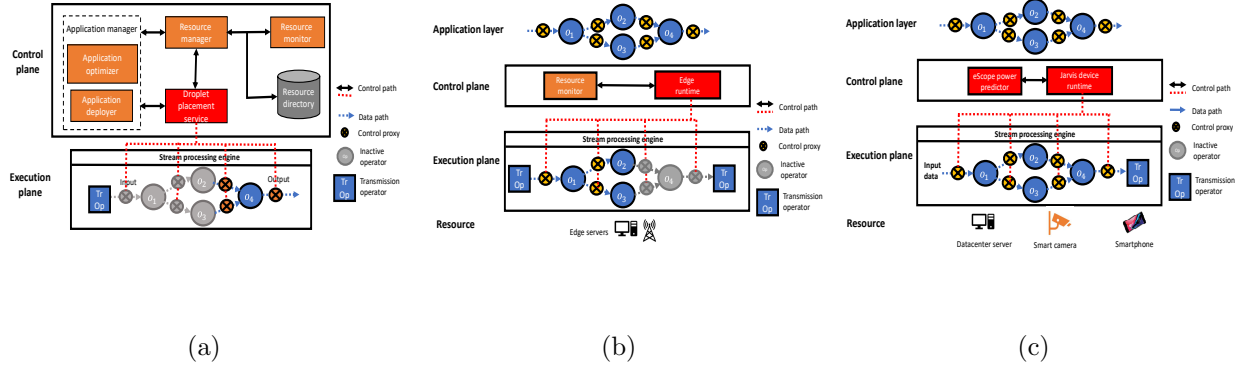


Figure 9.1: (a) System overview integrating Droplet and Jarvis architecture components on cloud nodes, (b) System overview integrating Droplet and Jarvis architecture components on edge nodes and, (c) System overview integrating Jarvis and eScope architecture components on data source nodes

the incoming data at the edge and cloud layers can also be determined based on the data-level partitioning plan implemented on the data source nodes. Droplet can use this information to make operator-level partitioning decisions between edge and cloud layers. With a global view of resources in edge and cloud layers that Droplet can leverage, we expect the quality of placement solutions to be better than the approach used by Jarvis which makes local placement decisions. Similarly, at the data source layer, data-level partitioning plans can be integrated with the power prediction capabilities provided by eScope, to enable energy-aware data-level partitioning plans on each data source node.

Figure 9.1 shows the architecture diagram integrating the placement and power prediction solutions proposed in this thesis. In Figure 9.1(a) each cloud node consists of the architectural components of Droplet in the control plane. In the execution plane, the application running on the stream processing engine is augmented with control proxies to implement a partitioning plan computed by the Droplet placement service. This is done by setting the load factors of operators already executed in the edge and data source layers, to be zero, while the other control proxies have their load factors set to 1. Figure 9.1(b) shows the system components for the runtime deployed on edge nodes. The resource monitor sends the available resource information to Droplet in the cloud. Edge runtime then receives placement decisions from Droplet placement service in the cloud. It configures the load factors of control proxies within the stream processing engine, to implement the computed partitioning plan on the edge nodes. In Figure 9.1(c), we note that there are no control paths between the system components on data source nodes, with the control plane elements in cloud or edge nodes. Thus, we can make fully decentralized placement decisions at the data source layer. Each

data source node consists of the Jarvis device runtime augmented with the ability to query a power predictor and make energy-aware placement decisions. The control proxies in the execution plane can then be configured to minimize the network transfer costs from data source node, subject the compute and energy constraints on the node.

9.2.2 Fault Tolerance

Our placement solution can be extended to support fault tolerance. When data source fails in the middle of processing a streaming window, no further data is generated from that node. Intermediate state accumulated by the data source for the current window can be periodically checkpointed by re-using the drain path from control proxies. We are investigating how Flink’s mechanism to draw consistent snapshots while guaranteeing exactly-once semantics [308] (and its use in supporting rescaling of stateful operators [309]), can be adapted to Jarvis. The edge node then identifies data source failures (using heartbeats), uses the latest checkpoint for the node, and processes the remaining data for the current window. When the edge node fails within a micro-datacenter, its processing state can be checkpointed using existing fault recovery techniques [310] so that a new node can restart processing from the last checkpoint. An external data store may need to be updated by edge so that data sources can read its last successful checkpoint. Data sources can then replay records as they contain the upstream query operators. Our setup consists of operators on edge nodes which receive multiple input streams from data sources. Taking snapshots using the Flink approach requires aligning the snapshot barriers across the data sources to guarantee exactly-once semantics. This introduces processing latencies. Exploring mechanisms to address this issue and scale fault tolerance mechanisms for Droplet and Jarvis, consisting of large number of data sources is an interesting direction of future work. Note that a similar coordination mechanism as above, for fault tolerance between edge and cloud nodes can be considered. Several existing works [311, 312, 313] can be considered to guide the design of our fault tolerant strategies.

9.2.3 Extending Data Source-Level Placement Solution for WAN-Based Analytics

The proof-of-concept system Jarvis which was designed and implemented for server monitoring pipelines, can be extended to IoT scenarios where billions of devices can sense, communicate, compute and potentially actuate. IoT scenarios exhibit similar constraints as the ones assumed in our work, however, they are characterized by very limited and highly varying network bandwidth (e.g., WAN) due to geo-distributed nature of underlying resource nodes.

The advent of edge datacenters have enabled data processing close to the data source in a geo-distributed setup, thus alleviating the network and compute bottlenecks at a central cloud cluster hosting the stream processor. Query execution can be distributed between edge and cloud datacenters, similar to how we apply our partitioning approach within a datacenter for server monitoring pipelines. However, the resource conditions under which partitioning decisions need to be made can vary widely depending on the considered target system. For instance, datacenter monitoring is constrained by compute resources at the data source, while partitioning for geo-distributed analytics needs to consider scarce and variable bandwidth in a wide-area network (WAN). Transporting data over WAN can introduce significant network overhead for coordination [98]. To support geo-distributed analytics [314], large cloud platforms today deploy edge servers to process data close to where it is generated, enabling low-latency, intelligent, and real-time analysis. Examples of geo-distributed analytics include (1) CDN log analysis to identify issues related to network bandwidth and latency, to improve CDN data placement [315], and (2) large-scale sensor monitoring in industrial IoT for predictive maintenance [316]. The edge servers, sitting between data sources (over cellular or WiFi) and cloud (over WAN), have compute resources which can be used to identify data points of interest and perform computations such as aggregations and joins as part of a user-defined query. Similar to datacenter server monitoring in Chapter 7, processing on the edge datacenters can greatly reduce the outbound network traffic to the cloud connected via WAN. However, WAN bandwidth is very limited and varies significantly over time [35], requiring stream analytics to operate efficiently under limited and dynamic network constraints.

Based on proposed optimization formulation to address partitioning in network-constrained environments (Section 5.3.3), we now briefly summarize the potential changes to Jarvis runtime and control proxy to facilitate such partitioning. In Section 7.4.3, the query is probed to detect its congestion state and when it is congested or idle, Jarvis runtime refines load factors to keep the query execution stable. For network-constrained partitioning, we can instead measure the aggregate network traffic leaving the edge server due to data drained by control proxies and the final query output of the last operator. When the aggregate traffic exceeds a configured bandwidth threshold for the query, Jarvis runtime initiates adaptation. During adaptation, the optimization in Equation 5.34 is solved in the first step of StepWise-Adapt. For the second step, query state is determined based on the aggregate network traffic leaving the edge server. The load factors are fine-tuned based on the observed state. Operators are assigned higher priority based on if they have lower compute costs during data processing. If the aggregate network traffic exceeds the configured bandwidth threshold, then the load factor of the operator with the lowest compute cost is increased. If the aggregate network traffic is below the given network bandwidth limit, then the load

factor of the operator with highest compute cost is reduced.

9.2.4 Power Prediction and Placement for Mobile Data Sources

We now provide discussion on the characteristics of our proposed power prediction method and how it can be extended for future work.

Speed of convergence for power measurements. In eScope, we use SoC readings to obtain energy measurements that can be used for training the eScope energy model. However, SoC readings are coarse-grained. A battery SoC change can only be observed at the granularity of 1%. Thus, it becomes expensive to collect training data for eScope. V-edge [317] is an alternative power measurement mechanism which uses battery voltage dynamics to estimate power draw from a running workload. When the load changes, the resulting current draw decreases the terminal voltage measured using the battery API. This decrease is proportional to the power drawn by the workload. Integrating V-edge power measurement capabilities into eScope can greatly improve the convergence speed for building energy models. We are investigating the limitations of V-edge when estimating power draw for long running workloads (since it requires an initial calibration and can subsequently accumulate errors in voltage measurements). And how the errors, if any, can be compensated by SoC measurements.

Integration with Jarvis. eScope is focused on addressing challenges for fine-grained power prediction for mobile video application operators. A Jarvis-integrated system for power management can benefit not only the mobile application service providers hosting their services on a remote cloud platform, but also end-users. Users can impose power budget constraints for their compute-intensive video analytics applications. A Jarvis-integrated system can adapt the placement policy of video analytics operators to maximize the reduction in network transfer costs, without violating the compute and power resource budget constraints. To achieve our goal, we can extend the optimization formulation for data-level partitioning in Equation 5.30 as follows:

$$\min_{p_1, p_2, \dots, p_M} \sum_{i=1}^M \left[\prod_{j=0}^{i-1} p_j \lambda_j \right] (1 - p_i) \quad (9.1)$$

subject to

$$\begin{aligned}
& \sum_{i=1}^M [\prod_{j=0}^{i-1} p_j \lambda_j] p_i c_i \leq C/D_I, \\
& \sum_{i=1}^M [\prod_{j=0}^{i-1} p_j \lambda_j] p_i e_i t_i \leq E/D_I, \\
& 0 \leq p_i \leq 1, \quad 0 \leq \lambda_i \leq 1, \quad c_i \geq 0 \quad \forall i \in [1, M], p_0 = 1, \lambda_0 = 1
\end{aligned} \tag{9.2}$$

where e_i is the compute resource power consumption of operator i to process each data item in the epoch, t_i is the active execution duration for operator i to process an item in the epoch and E is the compute energy consumption budget for an epoch, Note that D_I , M , E and C are fixed for a problem instance.

The problem can be transformed into a linear program, following the steps we used in Equation 5.32. Note that we have focused on the power cost of using compute resources on the data source. However, maximizing compute resource utilization on the mobile device can incur high power costs, while minimizing compute resource use on the device can also incur high power costs depending on network conditions on the data source. E.g., using cellular connection to transport data to edge servers could be more energy-intensive compared to using WiFi connection [20]. To address this issue, based on previous works which estimate energy costs based on the size of data transferred over the network [171], we rely on the critical assumption that reducing the data size for network transfer also reduces the network power costs. Combined with the property of analytics operators which result in intermediate data size reduction as we process more operators in the computational graph, we can leverage the cost model-based and model-agnostic heuristics within StepWise-Adapt algorithm, to identify a partitioning configuration which meets the compute and energy budget constraints. Leveraging the ideas described above, we can extend the Jarvis runtime to incorporate energy budget constraints when making data-level partitioning decisions on the data source. Figure 9.1(c) shows how the runtime can query the eScope power predictor at the end of every adaptation epoch, and implement a new partitioning plan depending on the observed power costs of operators in the previous epoch. Integrating the concepts from Jarvis and eScope and implementing them in a system to improve energy usage costs on mobile devices is an interesting direction for future work.

Code Offloading Model. We have discussed the problem of power prediction and its use in making placement decisions. Code offloading requires system support to dynamically migrate operators and internal state, from one resource node to another. To do so, we also need to account for the migration overhead that results from transferring the data and code from one node to another, as well as possibly halting the pipeline execution on the source

node, before continuing processing on the destination node [215]. Moreover, certain operators cannot be offloaded (e.g. GPS sensor, screen display, etc.) so the offloading strategy should take as input, a list of pinning constraints for such operator types to be pinned to the mobile device. IMCM [215] has addressed challenges associated with considering migration overhead and pinning constraints in actor-based offloading frameworks. Extending our work to support code offloading, by integrating our techniques with offloading frameworks to make scalable placement decisions, is also an interesting direction for future work.

REFERENCES

- [1] M. Steen and A. S. Tanenbaum, “A brief introduction to distributed systems,” *Computing*, vol. 98, no. 10, p. 967–1009, 2016. [Online]. Available: <https://doi.org/10.1007/s00607-016-0508-7>
- [2] D. C. Knowledge, “Google Data Center FAQ,” 2017, <https://www.datacenterknowledge.com/archives/2017/03/16/google-data-center-faq>.
- [3] R. Miller, “A Look Inside Wikipedia’s Infrastructure,” 2008, <https://www.datacenterknowledge.com/archives/2008/06/24/a-look-inside-wikipedias-infrastructure>.
- [4] S. Kliff, “America’s 4,150 traffic cameras, in one map,” 2015, <https://www.vox.com/a/red-light-speed-cameras>.
- [5] P. BISCHOFF, “Surveillance camera statistics: which cities have the most CCTV cameras?” 2021, <https://www.comparitech.com/vpn-privacy/the-worlds-most-surveilled-cities/>.
- [6] L. Mai, K. Zeng, R. Potharaju, L. Xu, S. Suh, S. Venkataraman, P. Costa, T. Kim, S. Muthukrishnan, V. Kuppala, S. Dhulipalla, and S. Rao, “Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems,” *Proc. VLDB Endow.*, vol. 11, no. 10, p. 1303–1316, June 2018. [Online]. Available: <https://doi.org/10.14778/3231751.3231765>
- [7] P. Chang and G. Agha, “Towards context-aware web applications,” in *Distributed Applications and Interoperable Systems, 7th IFIP WG 6.1 International Conference, DAIS 2007, Paphos, Cyprus, June 6-8, 2007, Proceedings*, ser. Lecture Notes in Computer Science, J. Indulska and K. Raymond, Eds., vol. 4531. Springer, 2007. [Online]. Available: https://doi.org/10.1007/978-3-540-72883-2_18 pp. 239–252.
- [8] S. Agarwal and B. R. Prasad, “High speed streaming data analysis of web generated log streams,” in *2015 IEEE 10th International Conference on Industrial and Information Systems (ICIIS)*, 2015, pp. 413–418.
- [9] A. Sandur, C. Park, S. Volos, G. Agha, and M. Jeon, “Jarvis: Large-scale server monitoring with adaptive near-data processing,” 2022. [Online]. Available: <https://arxiv.org/abs/2202.06021>
- [10] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman, “Live video analytics at scale with approximation and delay-tolerance,” in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017. [Online]. Available: <https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zhang> pp. 377–392.

- [11] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe, “Three Steps is All You Need: Fast, Accurate, Automatic Scaling Decisions for Distributed Streaming Dataflows,” in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’18. USA: USENIX Association, 2018, p. 783–798.
- [12] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, “Sonata: Query-Driven Streaming Network Telemetry,” in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM ’18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: <https://doi.org/10.1145/3230543.3230555> p. 357–371.
- [13] M. Jeon, S. Kim, S.-w. Hwang, Y. He, S. Elnikety, A. L. Cox, and S. Rixner, “Predictive parallelization: Taming tail latencies in web search,” in *Proceedings of the 37th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR ’14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: <https://doi.org/10.1145/2600428.2609572> p. 253–262.
- [14] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986.
- [15] P. Thati, P. Chang, and G. Agha, “Crawlets: Agents for high performance web search engines,” in *Mobile Agents, 5th International Conference, MA 2001 Atlanta, GA, USA, December 2-4, 2001, Proceedings*, ser. Lecture Notes in Computer Science, G. P. Picco, Ed., vol. 2240. Springer, 2001. [Online]. Available: https://doi.org/10.1007/3-540-45647-3_9 pp. 119–134.
- [16] G. Agha, M. Charalambides, K. Mechtov, K. Palmskog, A. Sandur, and R. Shiftehfar, “Theoretical considerations: Inferring and enforcing use patterns for mobile cloud assurance,” in *Assured Cloud Computing*. Wiley-IEEE Press, in press, 2018.
- [17] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng, “Packet-Level Telemetry in Large Datacenter Networks,” *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, p. 479–491, Aug. 2015. [Online]. Available: <https://doi.org/10.1145/2829988.2787483>
- [18] C. A. Varela and G. Agha, “Programming dynamically reconfigurable open systems with salsa,” *SIGPLAN Notices*, vol. 36, no. 12, pp. 20–34, 2001.
- [19] M. Gupta, *Akka Essentials*. Packt Publishing Ltd, 2012.
- [20] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, “Maui: making smartphones last longer with code offload,” in *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM, 2010, pp. 49–62.

- [21] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, “Clonecloud: Elastic execution between mobile device and cloud,” in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys ’11. New York, NY, USA: ACM, 2011. [Online]. Available: <http://doi.acm.org/10.1145/1966445.1966473> pp. 301–314.
- [22] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, “Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading,” in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 945–953.
- [23] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, “Network-aware operator placement for stream-processing systems,” in *22nd International Conference on Data Engineering (ICDE’06)*, 2006, pp. 49–49.
- [24] R. Potharaju, T. Kim, W. Wu, V. Acharya, S. Suh, A. Fogarty, A. Dave, S. Ramanujam, T. Talus, L. Novik, and R. Ramakrishnan, “Helios: Hyperscale indexing for the cloud & edge,” *Proc. VLDB Endow.*, vol. 13, no. 12, p. 3231–3244, Aug. 2020. [Online]. Available: <https://doi.org/10.14778/3415478.3415547>
- [25] G. Ananthanarayanan, V. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. R. Sivalingam, and S. Sinha, “Real-time video analytics – the killer app for edge computing,” *IEEE Computer*, 2017. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/real-time-video-analytics-killer-app-edge-computing/>
- [26] B. Nushi, E. Kamar, and E. Horvitz, “Towards accountable AI: hybrid human-machine analyses for characterizing system failure,” in *Proceedings of the Sixth AAAI Conference on Human Computation and Crowdsourcing, HCOMP 2018, Zürich, Switzerland, July 5-8, 2018*, Y. Chen and G. Kazai, Eds. AAAI Press, 2018. [Online]. Available: <https://aaai.org/ocs/index.php/HCOMP/HCOMP18/paper/view/17930> pp. 126–135.
- [27] J. Jiang, G. Ananthanarayanan, P. Bodík, S. Sen, and I. Stoica, “Chameleon: Scalable adaptation of video analytics,” in *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/chameleon-video-analytics-scale-via-adaptive-configurations-cross-camera-correlations/>
- [28] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, “Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints,” in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys ’16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: <https://doi.org/10.1145/2906388.2906396> p. 123–136.
- [29] S. Garg, K. Kaur, N. Kumar, G. Kaddoum, A. Y. Zomaya, and R. Ranjan, “A hybrid deep learning-based model for anomaly detection in cloud datacenter networks,” *IEEE Transactions on Network and Service Management*, vol. 16, no. 3, pp. 924–935, 2019.

- [30] J. Emmons, S. Fouladi, G. Ananthanarayanan, S. Venkataraman, S. Savarese, and K. Winstein, “Cracking open the dnn black-box: Video analytics with dnns across the camera-cloud boundary,” in *Proceedings of the 2019 Workshop on Hot Topics in Video Analytics and Intelligent Edges*, ser. HotEdgeVideo’19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: <https://doi.org/10.1145/3349614.3356023> p. 27–32.
- [31] H.-J. Jeong, H.-J. Lee, C. H. Shin, and S.-M. Moon, “Ionn: Incremental offloading of neural network computations from mobile devices to edge servers,” in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC ’18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: <https://doi.org/10.1145/3267809.3267828> p. 401–411.
- [32] T. Elgamal and K. Nahrstedt, “Serdab: An iot framework for partitioning neural networks computation across multiple enclaves,” in *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGRID 2020, Melbourne, Australia, May 11-14, 2020*. IEEE, 2020. [Online]. Available: <https://doi.org/10.1109/CCGrid49817.2020.00-41> pp. 519–528.
- [33] Y. Zhu and V. J. Reddi, “High-performance and energy-efficient mobile web browsing on big/little systems,” in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 13–24.
- [34] Y. Li, M. Chen, W. Dai, and M. Qiu, “Energy optimization with dynamic task scheduling mobile cloud computing,” *IEEE Systems Journal*, vol. 11, no. 1, pp. 96–105, March 2017.
- [35] B. Zhang, X. Jin, S. Ratnasamy, J. Wawrzyniek, and E. A. Lee, “AWStream: Adaptive Wide-Area Streaming Analytics,” in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM ’18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: <https://doi.org/10.1145/3230543.3230554> p. 236–252.
- [36] Amazon, “Cloud Computing Services - Amazon Web Services (AWS),” 2022, <https://aws.amazon.com/>.
- [37] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, “Edge computing: Vision and challenges,” *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.
- [38] D. Faggella, “The Self-Driving Car Timeline – Predictions from the Top 11 Global Automakers,” 2020, <https://emerj.com/ai-adoption-timelines/self-driving-car-timeline-themselves-top-11-automakers/>.
- [39] Statista, “Number of smartphone subscriptions worldwide from 2016 to 2027,” 2022, <https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/>.

- [40] Statista, “Mobile data traffic per smartphone worldwide from 2014 to 2027,” 2022, <https://www.statista.com/statistics/738977/worldwide-monthly-data-traffic-per-smartphone/>.
- [41] T. Elgamal, B. Chen, and K. Nahrstedt, “Teleconsultant: Communication and analysis of wearable videos in emergency medical environments,” in *Proceedings of the 25th ACM International Conference on Multimedia*, ser. MM '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: <https://doi.org/10.1145/3123266.3127920> p. 1241–1242.
- [42] V. Frascolla, F. Miatton, G. K. Tran, K. Takinami, A. De Domenico, E. C. Strinati, K. Koslowski, T. Haustein, K. Sakaguchi, S. Barbarossa, and S. Barberis, “5g-miedge: Design, standardization and deployment of 5g phase ii technologies: Mec and mmwaves joint development for tokyo 2020 olympic games,” in *2017 IEEE Conference on Standards for Communications and Networking (CSCN)*, 2017, pp. 54–59.
- [43] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien, “Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis,” in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: <https://doi.org/10.1145/2785956.2787496> p. 139–152.
- [44] T. Soyata, R. Muraleedharan, C. Funai, M. Kwon, and W. Heinzelman, “Cloud-vision: Real-time face recognition using a mobile-cloudlet-cloud acceleration architecture,” in *2012 IEEE Symposium on Computers and Communications (ISCC)*, 2012, pp. 000 059–000 066.
- [45] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, “Apache flink™: Stream and batch processing in a single engine,” *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 28–38, 2015. [Online]. Available: <http://sites.computer.org/debull/A15dec/p28.pdf>
- [46] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, “Discretized Streams: Fault-Tolerant Streaming Computation at Scale,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: <https://doi.org/10.1145/2517349.2522737> p. 423–438.
- [47] “Genetec,” 2021, <https://www.genetec.com/>.
- [48] “Avigilon,” 2021, <https://www.avigilon.com/products>.
- [49] IBM, “IBM Edge Computing for Servers,” 2021, <https://www.ibm.com/docs/en/cloud-private/3.2.0?topic=servers-preparing-install-edge-computing>.
- [50] Microsoft, “Azure stack edge,” <https://azure.microsoft.com/en-us/products/azure-stack/edge/>.

- [51] B. Charyyev, E. Arslan, and M. H. Gunes, "Latency comparison of cloud datacenters and edge servers," in *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*, 2020, pp. 1–6.
- [52] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-I. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmamghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snellham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: <https://doi.org/10.1145/3079856.3080246> p. 1–12.
- [53] Aruba, "Configuring wan uplinks," https://help.central.arubanetworks.com/2.5.3/documentation/online_help/content/gateways/cfg/basic/basic-conf-wan-det.htm.
- [54] IBM, "Ibm wan bandwidth requirements," <https://www.ibm.com/docs/en/psfa/1.6?topic=architecture-wan-bandwidth-requirements>.
- [55] Verizon, "Verizon 4g lte speeds," <https://www.verizon.com/articles/4g-lte-speeds-vs-your-home-network/>.
- [56] Celona, "Lte vs. 5g: Differences in speed and performance for business," <https://www.celona.io/cbrs/lte-vs-5g-differences-in-speed-performance-for-business>.
- [57] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni et al., "Storm@twitter," in *ACM SIGMOD'14*, 2014. [Online]. Available: <http://doi.acm.org/10.1145/2588555.2595641>
- [58] A. S. Foundation, "An easy to use, powerful, and reliable system to process and distribute data," 2018, <https://nifi.apache.org/>.
- [59] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.
- [60] S. Yi, Z. Hao, Z. Qin, and Q. Li, "Fog computing: Platform and applications," in *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, 2015, pp. 73–78.

- [61] IBM, “A Community for Accelerating Analytics at the Edge,” 2016, <https://edgent.incubator.apache.org/>.
- [62] IBM, “IBM Node-RED,” 2021, <https://nodered.org/>.
- [63] A. S. Foundation, “A subproject of Apache NiFi to collect data where it originates,” 2018, <https://nifi.apache.org/minifi/>.
- [64] VMWare, “VMware Liota: Little IoT Agent,” 2021, <https://github.com/vmware/liota>.
- [65] P. Ravindra, A. Khochare, S. P. Reddy, S. Sharma, P. Varshney, and Y. Simmhan, “ $\mathbb{ECHO}$: An adaptive orchestration platform for hybrid dataflows across cloud and edge,” in *Service-Oriented Computing*. Springer International Publishing, 2017, pp. 395–410. [Online]. Available: https://doi.org/10.1007%2F978-3-319-69035-3_28
- [66] “Android,” <https://www.android.com/>, 2008.
- [67] Q. Ning, C.-A. Chen, R. Stoleru, and C. Chen, “Mobile storm: Distributed real-time stream processing for mobile clouds,” in *2015 IEEE 4th International Conference on Cloud Networking (CloudNet)*, 2015, pp. 139–145.
- [68] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, “Aurora: A new model and architecture for data stream management,” *The VLDB Journal*, vol. 12, no. 2, p. 120–139, 2003. [Online]. Available: <https://doi.org/10.1007/s00778-003-0095-z>
- [69] M. Balazinska, H. Balakrishnan, and M. Stonebraker, “Load management and high availability in the medusa distributed stream processing system,” in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’04. New York, NY, USA: Association for Computing Machinery, 2004. [Online]. Available: <https://doi.org/10.1145/1007568.1007701> p. 929–930.
- [70] Y. Ahmad, B. Berg, U. Cetintemel, M. Humphrey, J.-H. Hwang, A. Jhingran, A. Maskey, O. Papaemmanouil, A. Rasin, N. Tatbul, W. Xing, Y. Xing, and S. Zdonik, “Distributed operation in the borealis stream processing engine,” in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’05. New York, NY, USA: Association for Computing Machinery, 2005. [Online]. Available: <https://doi.org/10.1145/1066157.1066274> p. 882–884.
- [71] S. Ch, O. Cooper, A. Deshp, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah, “Telegraphcq: Continuous dataflow processing for an uncertain world,” in *In First Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.
- [72] M. A. Shah, J. M. Hellerstein, Sirish Chandrasekaran, and M. J. Franklin, “Flux: an adaptive partitioning operator for continuous query systems,” in *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*, 2003, pp. 25–36.

- [73] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin, “StreamBox: Modern Stream Processing on a Multicore Machine,” in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '17. USA: USENIX Association, 2017, p. 617–629.
- [74] H. Miao, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin, “StreamBox-HBM: Stream Analytics on High Bandwidth Hybrid Memory,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: <https://doi.org/10.1145/3297858.3304031> p. 167–181.
- [75] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing, “Trill: A High-Performance Incremental Query Processor for Diverse Analytics,” *Proc. VLDB Endow.*, vol. 8, no. 4, p. 401–412, Dec. 2014. [Online]. Available: <https://doi.org/10.14778/2735496.2735503>
- [76] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, “Twitter Heron: Stream Processing at Scale,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: <https://doi.org/10.1145/2723372.2742788> p. 239–250.
- [77] B. Chandramouli, R. C. Fernandez, J. Goldstein, A. Eldawy, and A. Quamar, “Quill: Efficient, Transferable, and Rich Analytics at Scale,” *Proc. VLDB Endow.*, vol. 9, no. 14, p. 1623–1634, Oct. 2016. [Online]. Available: <https://doi.org/10.14778/3007328.3007329>
- [78] W. Lin, H. Fan, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou, “STREAMSCOPE: Continuous Reliable Distributed Processing of Big Data Streams,” in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, ser. NSDI'16. USA: USENIX Association, 2016, p. 439–453.
- [79] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, “Naiad: A Timely Dataflow System,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: <https://doi.org/10.1145/2517349.2522738> p. 439–455.
- [80] L. Aniello, R. Baldoni, and L. Querzoni, “Adaptive online scheduling in storm,” in *ACM DEBS*, 2013. [Online]. Available: <http://doi.acm.org/10.1145/2488222.2488267>
- [81] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy, “Dhalion: Self-regulating stream processing in heron,” *Proc. VLDB Endow.*, vol. 10, no. 12, p. 1825–1836, Aug. 2017. [Online]. Available: <https://doi.org/10.14778/3137765.3137786>
- [82] Apache, “Dynamic allocation in Spark,” 2021, <https://www.slideshare.net/databricks/dynamic-allocation-in-spark>.

- [83] Apache, “Dynamic resource allocation in Spark,” 2021, <https://spark.apache.org/docs/latest/job-scheduling.html>.
- [84] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, “Optimal operator placement for distributed stream processing applications,” in *ACM DEBS*, 2016. [Online]. Available: <http://doi.acm.org/10.1145/2933267.2933312> pp. 69–80.
- [85] X. Fu, T. Ghaffar, J. C. Davis, and D. Lee, “Edgewise: A Better Stream Processing Engine for the Edge,” in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC ’19. USA: USENIX Association, 2019, p. 929–945.
- [86] L. Tong, Y. Li, and W. Gao, “A hierarchical edge cloud architecture for mobile computing,” in *IEEE INFOCOM 2016*, 2016.
- [87] T. Elgamal, “Costless: Optimizing cost of serverless computing through function fusion and placement,” in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, 2018, pp. 300–312.
- [88] S. Jain, X. Zhang, Y. Zhou, G. Ananthanarayanan, J. Jiang, Y. Shu, V. Bahl, and J. Gonzalez, “Spatula: Efficient cross-camera video analytics on large camera networks,” in *ACM/IEEE Symposium on Edge Computing (SEC 2020)*, November 2020. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/spatula-efficient-cross-camera-video-analytics-on-large-camera-networks/>
- [89] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, “The case for vm-based cloudlets in mobile computing,” *IEEE PerCom*, vol. 8, no. 4, 2009. [Online]. Available: <http://dx.doi.org/10.1109/MPRV.2009.82>
- [90] Y. H. Kao, B. Krishnamachari, M. R. Ra, and F. Bai, “Hermes: Latency optimal task assignment for resource-constrained mobile computing,” *IEEE TMC*, vol. 16, no. 11, 2017.
- [91] L. Yang, J. Cao, H. Cheng, and Y. Ji, “Multi-user computation partitioning for latency sensitive mobile cloud applications,” *IEEE Transactions on Computers*, vol. 64, no. 8, pp. 2253–2266, 2015.
- [92] R. Ghosh and Y. Simmhan, “Distributed Scheduling of Event Analytics across Edge and Cloud,” *ArXiv e-prints*, Aug. 2016.
- [93] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy, “A distributed resource management architecture that supports advance reservations and co-allocation,” in *IWQoS ’99.*, 1999.
- [94] D. Yu, Y. Zhu, B. Arzani, R. Fonseca, T. Zhang, K. Deng, and L. Yuan, “dShark: A General, Easy to Program and Scalable Framework for Analyzing In-network Packet Traces,” in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019. [Online]. Available: <https://www.usenix.org/conference/nsdi19/presentation/yu> pp. 207–220.

- [95] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, “Gigascope: A Stream Database for Network Applications,” in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’03. New York, NY, USA: Association for Computing Machinery, 2003. [Online]. Available: <https://doi.org/10.1145/872757.872838> p. 647–651.
- [96] M. L. Massie, B. N. Chun, and D. E. Culler, “The Ganga Distributed Monitoring System: Design, Implementation And Experience,” *Parallel Computing*, vol. 30, p. 2004, 2003.
- [97] Microsoft, “Windows and Linux performance data sources in Azure Monitor,” 2018, <https://datacenterfrontier.com/data-center-health-10-kpi/>.
- [98] F. Lai, J. You, X. Zhu, H. V. Madhyastha, and M. Chowdhury, “Sol: Fast Distributed Computation Over Slow Networks,” in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020. [Online]. Available: <https://www.usenix.org/conference/nsdi20/presentation/lai> pp. 273–288.
- [99] A. Jonathan, A. Chandra, and J. Weissman, “Multi-Query Optimization in Wide-Area Streaming Analytics,” in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC ’18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: <https://doi.org/10.1145/3267809.3267842> p. 412–425.
- [100] Y. Li, A. Padmanabhan, P. Zhao, Y. Wang, G. H. Xu, and R. Netravali, “Reducto: On-camera filtering for resource-efficient real-time video analytics,” in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM ’20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: <https://doi.org/10.1145/3387514.3405874> p. 359–376.
- [101] H. Guo, S. Yao, Z. Yang, Q. Zhou, and K. Nahrstedt, “Crossroi: Cross-camera region of interest optimization for efficient real time video analytics at scale,” in *Proceedings of the 12th ACM Multimedia Systems Conference*, ser. MMSys ’21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: <https://doi.org/10.1145/3458305.3463381> p. 186–199.
- [102] J. Wang, Z. Feng, Z. Chen, S. George, M. Bala, P. Pillai, S.-W. Yang, and M. Satyanarayanan, “Bandwidth-efficient live video analytics for drones via edge computing,” in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, 2018, pp. 159–173.
- [103] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” 2017. [Online]. Available: <https://arxiv.org/abs/1704.04861>
- [104] Qualcomm, “Qualcomm QCS603,” <https://www.qualcomm.com/products/technology/processors/application-processors/qcs603>, 2022.

- [105] K. Spiteri, R. Sitaraman, and D. Sparacio, “From theory to practice: Improving bitrate adaptation in the dash reference player,” *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 15, no. 2s, jul 2019. [Online]. Available: <https://doi.org/10.1145/3336497>
- [106] K. Spiteri, R. Urgaonkar, and R. K. Sitaraman, “Bola: Near-optimal bitrate adaptation for online videos,” in *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, 2016, pp. 1–9.
- [107] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum, “In-situ mapreduce for log processing,” in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC’11. USA: USENIX Association, 2011, p. 9.
- [108] R. Gracia-Tinedo, M. Sanchez-Artigas, P. Garcia-Lopez, Y. Moatti, and F. Gluszak, “Lamda-flow: Automatic pushdown of dataflow operators close to the data,” in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2019, pp. 112–121.
- [109] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm, “lprof: A non-intrusive request flow profiler for distributed systems,” in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014. [Online]. Available: <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zhao> pp. 629–644.
- [110] Y. Moatti, E. Rom, R. Gracia-Tinedo, D. Naor, D. Chen, J. Sampe, M. Sanchez-Artigas, P. Garcia-Lopez, F. Gluszak, E. Deschdt, F. Pace, D. Venzano, and P. Michiardi, “Too big to eat: Boosting analytics data ingestion from object stores with scoop,” in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, 2017, pp. 309–320.
- [111] A. M. Aly, A. Sallam, B. M. Gnanasekaran, L.-V. Nguyen-Dinh, W. G. Aref, M. Ouzzani, and A. Ghafoor, “M3: Stream processing on main-memory mapreduce,” in *2012 IEEE 28th International Conference on Data Engineering*, 2012, pp. 1253–1256.
- [112] C. Lei, Z. Zhuang, E. A. Rundensteiner, and M. Y. Eltabakh, “Redoop infrastructure for recurring big data queries,” *Proc. VLDB Endow.*, vol. 7, no. 13, p. 1589–1592, Aug. 2014. [Online]. Available: <https://doi.org/10.14778/2733004.2733037>
- [113] S. Lakshminarasimhan, X. Zou, D. A. Boyuka, S. V. Pendse, J. Jenkins, V. Vishwanath, M. E. Papka, S. Klasky, and N. F. Samatova, “Diraq: Scalable in situ data- and resource-aware indexing for optimized query performance,” *Cluster Computing*, vol. 17, no. 4, p. 1101–1119, Dec. 2014. [Online]. Available: <https://doi.org/10.1007/s10586-014-0358-z>
- [114] H. Zou, Y. Yu, W. Tang, and H.-W. M. Chen, “Flexanalytics: A flexible data analytics framework for big data applications with i/o performance improvement,” *Big Data Research*, vol. 1, pp. 4–13, 2014, special Issue on Scalable Computing for Big Data. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S2214579614000021>

- [115] Spark, “Configuration: dynamic allocation in Spark,” 2021, <https://spark.apache.org/docs/latest/job-scheduling.html>.
- [116] Y. Ahmad and U. Çetintemel, “Network-aware query processing for stream-based applications,” in *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, ser. VLDB ’04. VLDB Endowment, 2004, p. 456–467.
- [117] F. Starks and T. P. Plagemann, “Operator placement for efficient distributed complex event processing in manets,” in *2015 IEEE 11th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, 2015, pp. 83–90.
- [118] B. J. Bonfils and P. Bonnet, “Adaptive and decentralized operator placement for in-network query processing,” in *Information Processing in Sensor Networks*, F. Zhao and L. Guibas, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 47–62.
- [119] S. Bokhari, “A shortest tree algorithm for optimal assignments across space and time in a distributed processor system,” *IEEE Transactions on Software Engineering*, vol. SE-7, no. 6, pp. 583–589, 1981.
- [120] V. Kumar, B. Cooper, and K. Schwan, “Distributed stream management using utility-driven self-adaptive middleware,” in *Second International Conference on Autonomic Computing (ICAC’05)*, 2005, pp. 3–14.
- [121] Y. Ahmad and U. Cetintemel, “Network-aware query processing for stream-based applications,” in *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, ser. VLDB ’04. VLDB Endowment, 2004, p. 456–467.
- [122] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, “Optimal operator replication and placement for distributed stream processing systems,” *SIGMETRICS Perform. Eval. Rev.*, vol. 44, no. 4, p. 11–22, May 2017. [Online]. Available: <https://doi.org/10.1145/3092819.3092823>
- [123] T. Hiessl, V. Karagiannis, C. Hochreiner, S. Schulte, and M. Nardelli, “Optimal placement of stream processing operators in the fog,” in *2019 IEEE 3rd International Conference on Fog and Edge Computing (ICFEC)*, 2019, pp. 1–10.
- [124] I. Flouris, N. Giatrakos, A. Deligiannakis, and M. Garofalakis, “Network-wide complex event processing over geographically distributed data sources,” *Information Systems*, vol. 88, p. 101442, 2020. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0306437919304946>
- [125] M. Nardelli, V. Cardellini, V. Grassi, and F. L. Presti, “Efficient operator placement for distributed data stream processing applications,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 8, pp. 1753–1767, 2019.
- [126] L. Xu, B. Peng, and I. Gupta, “Stela: Enabling stream processing systems to scale-in and scale-out on-demand,” in *2016 IEEE International Conference on Cloud Engineering (IC2E)*, 2016, pp. 22–31.

- [127] G. Mencagli, “A game-theoretic approach for elastic distributed data stream processing,” *ACM Trans. Auton. Adapt. Syst.*, vol. 11, no. 2, June 2016. [Online]. Available: <https://doi.org/10.1145/2903146>
- [128] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo, “Decentralized self-adaptation for elastic data stream processing,” *Future Generation Computer Systems*, vol. 87, pp. 171–185, 2018. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167739X17326821>
- [129] B. Lohrmann, D. Warneke, and O. Kao, “Nephele streaming: Stream processing under qos constraints at scale,” *Cluster Computing*, vol. 17, no. 1, p. 61–78, Mar. 2014. [Online]. Available: <https://doi.org/10.1007/s10586-013-0281-8>
- [130] G. Mencagli, M. Vanneschi, and E. Vespa, “A cooperative predictive control approach to improve the reconfiguration stability of adaptive distributed parallel applications,” *ACM Trans. Auton. Adapt. Syst.*, vol. 9, no. 1, Mar. 2014. [Online]. Available: <https://doi.org/10.1145/2567929>
- [131] R. K. Kombi, N. Lumineau, and P. Lamarre, “A preventive auto-parallelization approach for elastic stream processing,” in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 1532–1542.
- [132] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu, “Elastic scaling of data parallel operators in stream processing,” in *2009 IEEE International Symposium on Parallel Distributed Processing*, 2009, pp. 1–12.
- [133] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, “Elastic scaling for data stream processing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1447–1463, 2014.
- [134] C. Hochreiner, M. Vögler, S. Schulte, and S. Dustdar, “Elastic stream processing for the internet of things,” in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, 2016, pp. 100–107.
- [135] R. Mayer, B. Koldehofe, and K. Rothermel, “Predictable low-latency event detection with parallel complex event processing,” *IEEE Internet of Things Journal*, vol. 2, no. 4, pp. 274–286, 2015.
- [136] O. Saleh and K.-U. Sattler, “The pipeline approach: Write once, run in different stream-processing engines,” in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS ’15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: <https://doi.org/10.1145/2675743.2776774> p. 368–371.
- [137] R. Mayer, C. Mayer, M. A. Tariq, and K. Rothermel, “Graphcep: Real-time data analytics using parallel complex event and graph processing,” in *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS ’16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: <https://doi.org/10.1145/2933267.2933509> p. 309–316.

- [138] C. Balkesen, N. Tatbul, and M. T. Özsu, “Adaptive input admission and management for parallel stream processing,” in *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS ’13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: <https://doi.org/10.1145/2488222.2488258> p. 15–26.
- [139] N. Rivetti, L. Querzoni, E. Anceaume, Y. Busnel, and B. Sericola, “Efficient key grouping for near-optimal load balancing in stream processing systems,” in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS ’15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: <https://doi.org/10.1145/2675743.2771827> p. 80–91.
- [140] E. Zeitler and T. Risch, “Massive scale-out of expensive continuous queries,” *Proc. VLDB Endow.*, vol. 4, no. 11, p. 1181–1188, Aug. 2011. [Online]. Available: <https://doi.org/10.14778/3402707.3402752>
- [141] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, “Millwheel: Fault-tolerant stream processing at internet scale,” *Proc. VLDB Endow.*, vol. 6, no. 11, p. 1033–1044, Aug. 2013. [Online]. Available: <https://doi.org/10.14778/2536222.2536229>
- [142] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, “S4: Distributed stream computing platform,” in *2010 IEEE International Conference on Data Mining Workshops*, 2010, pp. 170–177.
- [143] S. Wu, V. Kumar, K.-L. Wu, and B. C. Ooi, “Parallelizing stateful operators in a distributed stream processing system: How, should you and how much?” in *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS ’12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: <https://doi.org/10.1145/2335484.2335515> p. 278–289.
- [144] N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis, “A holistic view of stream partitioning costs,” *Proc. VLDB Endow.*, vol. 10, no. 11, p. 1286–1297, Aug. 2017. [Online]. Available: <https://doi.org/10.14778/3137628.3137639>
- [145] G. Mencagli, M. Torquati, F. Lucattini, S. Cuomo, and M. Aldinucci, “Harnessing sliding-window execution semantics for parallel stream processing,” *Journal of Parallel and Distributed Computing*, 11 2017.
- [146] N. Rivetti, E. Anceaume, Y. Busnel, L. Querzoni, and B. Sericola, “Online scheduling for shuffle grouping in distributed stream processing systems,” in *Proceedings of the 17th International Middleware Conference*, ser. Middleware ’16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: <https://doi.org/10.1145/2988336.2988347>

- [147] G. Mencagli, M. Torquati, and M. Danelutto, “Elastic-ppq: A two-level autonomic system for spatial preference query processing over dynamic data streams,” *Future Generation Computer Systems*, vol. 79, pp. 862–877, 2018. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167739X1730938X>
- [148] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, “Skewtune: Mitigating skew in mapreduce applications,” in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: <https://doi.org/10.1145/2213836.2213840> p. 25–36.
- [149] E. Kirpichov and M. Denielou, “No shard left behind: dynamic work rebalancing in Google Cloud Dataflow,” 2016, <https://cloud.google.com/blog/bigdata/2016/05/no-shard-left-behinddynamic-work-rebalancing-in-googlecloud-dataflow>.
- [150] R. Mayer, M. A. Tariq, and K. Rothermel, “Minimizing communication overhead in window-based parallel complex event processing,” in *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS ’17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: <https://doi.org/10.1145/3093742.3093914> p. 54–65.
- [151] M. A. U. Nasir, G. De Francisci Morales, D. García-Soriano, N. Kourtellis, and M. Serafini, “The power of both choices: Practical load balancing for distributed stream processing engines,” in *2015 IEEE 31st International Conference on Data Engineering*, 2015, pp. 137–148.
- [152] T. Elgamal, A. Sandur, P. Nguyen, K. Nahrstedt, and G. Agha, “Droplet: Distributed operator placement for iot applications spanning edge and cloud resources,” in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 1–8.
- [153] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, “Clonecloud: elastic execution between mobile device and cloud,” in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 301–314.
- [154] S. Imai, T. Chestna, and C. A. Varela, “Elastic scalable cloud computing using application-level migration,” in *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*. IEEE Computer Society, 2012, pp. 91–98.
- [155] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, *Serverless Computing: Current Trends and Open Problems*. Singapore: Springer Singapore, 2017, pp. 1–20. [Online]. Available: https://doi.org/10.1007/978-981-10-5026-8_1
- [156] S. Wu, C. Niu, J. Rao, H. Jin, and X. Dai, “Container-based cloud platform for mobile computation offloading,” in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 123–132.

- [157] R. Friedman and N. Hauser, “COARA: code offloading on android with aspectj,” *CoRR*, vol. abs/1604.00641, 2016. [Online]. Available: <http://arxiv.org/abs/1604.00641>
- [158] H. Chen, Y. Lin, and C. Cheng, “Coca: Computation offload to clouds using aop,” in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, 2012, pp. 466–473.
- [159] Y. Duan, M. Zhang, H. Yin, and Y. Tang, “Privacy-preserving offloading of mobile app to the public cloud,” in *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*. Santa Clara, CA: USENIX Association, 2015. [Online]. Available: <https://www.usenix.org/conference/hotcloud15/workshop-program/presentation/duan>
- [160] F. Pfenning and D. Griffith, “Polarized substructural session types,” in *Foundations of Software Science and Computation Structures*, A. Pitts, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 3–22.
- [161] A. Das, J. Hoffmann, and F. Pfenning, “Work analysis with resource-aware session types,” in *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, ser. LICS ’18. New York, NY, USA: ACM, 2018. [Online]. Available: <http://doi.acm.org/10.1145/3209108.3209146> pp. 305–314.
- [162] S. Shifteh Far, “A flexible fine-grained adaptive framework for parallel mobile hybrid cloud applications,” Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2015.
- [163] J. Flinn and M. Satyanarayanan, “Powerscope: A tool for profiling the energy usage of mobile applications,” in *Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, ser. WMCSA ’99. Washington, DC, USA: IEEE Computer Society, 1999. [Online]. Available: <http://dl.acm.org/citation.cfm?id=520551.837522> pp. 2–.
- [164] F. Belloso, “The benefits of event-driven energy accounting in power-sensitive systems,” in *Proceedings of the 9th ACM SIGOPS European Workshop*, Kolding, Denmark, 2000. [Online]. Available: <http://i30www.ira.uka.de/research/publications/>
- [165] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang, “Fine-grained power modeling for smartphones using system call tracing,” in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys ’11. New York, NY, USA: ACM, 2011. [Online]. Available: <http://doi.acm.org/10.1145/1966445.1966460> pp. 153–168.
- [166] Y. Cao, J. Nejati, M. Wajahat, A. Balasubramanian, and A. Gandhi, “Deconstructing the energy consumption of the mobile page load,” *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 1, no. 1, pp. 6:1–6:25, June 2017. [Online]. Available: <http://doi.acm.org/10.1145/3084443>
- [167] A. S. (student, T. Schmid, P. Dutta, and N. Spring), “Demo: Phone power monitoring with batter.”

- [168] “Monsoon power monitor.” <https://www.msoon.com/LabEquipment/PowerMonitor/>, 2014.
- [169] K. Shankari, J. Fürst, Y. Wang, P. Bonnet, D. E. Culler, and R. H. Katz, “Zephyr: Simple, ready-to-use software-based power evaluation for background sensing smartphone applications,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2018-168, 2018. [Online]. Available: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-168.html>
- [170] C. Wang, F. Yan, Y. Guo, and X. Chen, “Power estimation for mobile applications with profile-driven battery traces,” in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2013, pp. 120–125.
- [171] L. Zhang, B. Tiwana, R. P. Dick, Z. Qian, Z. M. Mao, Z. Wang, and L. Yang, “Accurate online power estimation and automatic battery behavior based power model generation for smartphones,” in *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2010, pp. 105–114.
- [172] W. Jung, C. Kang, C. Yoon, D. Kim, and H. Cha, “Devscope: A nonintrusive and online power analysis tool for smartphone hardware components,” in *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS ’12. New York, NY, USA: ACM, 2012. [Online]. Available: <http://doi.acm.org/10.1145/2380445.2380502> pp. 353–362.
- [173] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha, “Appscope: Application energy metering framework for android smartphone using kernel activity monitoring,” in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX, 2012. [Online]. Available: <https://www.usenix.org/conference/atc12/technical-sessions/presentation/yoons> pp. 387–400.
- [174] Y. Zhang, Y. Liu, L. Zhuang, X. Liu, F. Zhao, and Q. Li, “Accurate cpu power modeling for multicore smartphones,” Microsoft Corporation, Tech. Rep. MSR-TR-2015-9, 2015. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/accurate-cpu-power-modeling-for-multicore-smartphones/>
- [175] Y. G. Kim, M. Kim, J. M. Kim, M. Sung, and S. W. Chung, “A novel gpu power model for accurate smartphone power breakdown,” *ETRI Journal*, vol. 37, no. 1, pp. 157–164, 2015. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.4218/etrij.14.0113.1411>
- [176] S. Reif, B. Herzog, J. Hemp, W. Schröder-Preikschat, and T. Hönig, “Ai waste prevention: Time and power estimation for edge tensor processing units: Poster,” in *Proceedings of the Twelfth ACM International Conference on Future Energy Systems*, ser. e-Energy ’21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: <https://doi.org/10.1145/3447555.3466579> p. 300–301.

- [177] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne, “Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks,” in *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, 2016, pp. 262–263.
- [178] D. Di Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. De Lucia, “Software-based energy profiling of android apps: Simple, efficient and reliable?” in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 103–114.
- [179] J. C. McCullough, Y. Agarwal, J. Chandrashekar, S. Kuppuswamy, A. C. Snoeren, and R. K. Gupta, “Evaluating the effectiveness of model-based power characterization,” in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC’11. Berkeley, CA, USA: USENIX Association, 2011. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2002181.2002193> pp. 12–12.
- [180] M. Dong and L. Zhong, “Self-constructive high-rate system energy modeling for battery-powered mobile systems,” in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys ’11. New York, NY, USA: ACM, 2011. [Online]. Available: <http://doi.acm.org/10.1145/1999995.2000027> pp. 335–348.
- [181] A. J. Oliner, A. P. Iyer, I. Stoica, E. Lagerspetz, and S. Tarkoma, “Carat: Collaborative energy diagnosis for mobile devices,” in *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys ’13. New York, NY, USA: ACM, 2013. [Online]. Available: <http://doi.acm.org/10.1145/2517351.2517354> pp. 10:1–10:14.
- [182] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden, “Wishbone: Profile-based partitioning for sensornet applications,” in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI’09. USA: USENIX Association, 2009, p. 395–408.
- [183] G. Lim, M. S. Hassan, Z. Jin, S. Volos, and M. Jeon, “Approximate quantiles for datacenter telemetry monitoring,” in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. USA: IEEE, 2020, pp. 1914–1917.
- [184] J. Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy, B. Shuai, T. Liu, X. Wang, G. Wang, J. Cai, and T. Chen, “Recent advances in convolutional neural networks,” *Pattern Recognition*, vol. 77, pp. 354–377, 2018. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0031320317304120>
- [185] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica, “Drizzle: Fast and adaptable stream processing at scale,” in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP ’17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: <https://doi.org/10.1145/3132747.3132750> p. 374–389.

- [186] R. Bhardwaj, Z. Xia, G. Ananthanarayanan, J. Jiang, Y. Shu, N. Karianakis, K. Hsieh, P. Bahl, and I. Stoica, “Ekya: Continuous learning of video analytics models on edge compute servers,” in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022. [Online]. Available: <https://www.usenix.org/conference/nsdi22/presentation/bhardwaj> pp. 119–135.
- [187] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, “A foundation for actor computation,” *Journal of Functional Programming*, vol. 7, no. 01, pp. 1–72, 1997.
- [188] P. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin, “Orleans: Distributed virtual actors for programmability and scalability,” Microsoft Corporation, Tech. Rep. MSR-TR-2014-41, 2014. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/orleans-distributed-virtual-actors-for-programmability-and-scalability/>
- [189] “Facebook chat system,” <https://www.facebook.com/notes/facebook-engineering/chat-stability-and-scalability/51412338919>, 2009.
- [190] “Twitter uses scala for its actor library,” https://www.redfin.com/blog/how_and_why_twitter_uses_scala/, 2010.
- [191] “Paypal scales to billions of transactions daily using akka,” <http://highscalability.com/blog/2016/8/15/how-paypal-scaled-to-billions-of-transactions-daily-using-ju.html>, 2016.
- [192] “Salsa on android,” <http://osl.cs.illinois.edu/software/salsa-android/>, 2016.
- [193] X. Zhang, J. Schiffman, S. Gibbs, A. Kunjithapatham, and S. Jeong, “Securing elastic applications on mobile devices for cloud computing,” in *Proceedings of the 2009 ACM workshop on Cloud computing security*. ACM, 2009, pp. 127–134.
- [194] D. Charousset, “Streaming-experimental,” 2020, <https://actor-framework.readthedocs.io/en/stable/Streaming.html>.
- [195] Lightbend, “Akka streams introduction,” 2021. [Online]. Available: <https://doc.akka.io/docs/akka/current/stream/stream-introduction.html>
- [196] T. Desell and C. A. Varela, *SALSA Lite: A Hash-Based Actor Runtime for Efficient Local Concurrency*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 144–166. [Online]. Available: https://doi.org/10.1007/978-3-662-44471-9_7
- [197] “Android battery manager service,” <https://developer.android.com/reference/android/os/BatteryManager>, 2008.
- [198] “Gcmnetworkmanager: Api to schedule network-oriented tasks, and let google play services batch network operations across the system,” <https://developers.google.com/android/reference/com/google/android/gms/gcm/GcmNetworkManager>, 2008.

- [199] Y.-K. Kwok and I. Ahmad, “Static scheduling algorithms for allocating directed task graphs to multiprocessors,” *ACM Comput. Surv.*, vol. 31, no. 4, 1999. [Online]. Available: <http://doi.acm.org/10.1145/344588.344618>
- [200] M. R. Garey and D. S. Johnson, “Complexity results for multiprocessor scheduling under resource constraints,” *SIAM J. Comput.*, vol. 4, no. 4, p. 397–411, dec 1975. [Online]. Available: <https://doi.org/10.1137/0204035>
- [201] R. Ghosh and Y. Simmhan, “Distributed scheduling of event analytics across edge and cloud,” *CoRR*, vol. abs/1608.01537, 2016. [Online]. Available: <http://arxiv.org/abs/1608.01537>
- [202] P. G. Bradford, G. J. E. Rawlins, and G. E. Shannon, “Efficient matrix chain ordering in polylog time,” *SIAM Journal on Computing*, 1998. [Online]. Available: <https://doi.org/10.1137/S0097539794270698>
- [203] A. B. Kahn, “Topological sorting of large networks,” *Commun. ACM*, vol. 5, no. 11, pp. 558–562, Nov. 1962. [Online]. Available: <http://doi.acm.org/10.1145/368996.369025>
- [204] M. L. Fredman and R. E. Tarjan, “Fibonacci heaps and their uses in improved network optimization algorithms,” *J. ACM*, vol. 34, no. 3, pp. 596–615, July 1987. [Online]. Available: <http://doi.acm.org/10.1145/28869.28874>
- [205] N. J. A. Sloane and S. Plouffe, “The encyclopedia of integer sequences,” *Academic Press*, vol. abs/1608.01537, 1995. [Online]. Available: <https://oeis.org/A000522>
- [206] R. Askey, *Orthogonal Polynomials and Special Functions*. Society for Industrial and Applied Mathematics, 1975. [Online]. Available: <https://epubs.siam.org/doi/abs/10.1137/1.9781611970470>
- [207] M. Savelsbergh, “A branch-and-price algorithm for the generalized assignment problem,” *Operations Research*, vol. 45, pp. 831–841, 12 1997.
- [208] X. Chen, L. Jiao, W. Li, and X. Fu, “Efficient multi-user computation offloading for mobile-edge cloud computing,” *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 2795–2808, 2016.
- [209] B. Liu, X. Xu, L. Qi, Q. Ni, and W. Dou, “Task scheduling with precedence and placement constraints for resource utilization improvement in multi-user mec environment,” *Journal of Systems Architecture*, vol. 114, p. 101970, 2021. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1383762120302174>
- [210] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.
- [211] G. Dósa, R. Li, X. Han, and Z. Tuza, “Tight absolute bound for first fit decreasing bin-packing: $\text{Ffd}(l)_i = 11/9 \text{ opt}(l) + 6/9$,” *Theoretical Computer Science*, vol. 510, pp. 13–61, 2013. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0304397513006774>

- [212] A. M. Peterson E. L., *Geometric Programming*. Boston, MA: Springer US, 1980, pp. 31–94. [Online]. Available: https://doi.org/10.1007/978-1-4615-8285-4_3
- [213] L. V. Martin Andersen, Joachim Dahl, “Cvxopt: Python software for convex optimization,” 2015, <http://cvxopt.org/>.
- [214] D. Kumar, S. Ahmad, A. Chandra, and R. K. Sitaraman, “Aggnet: Cost-aware aggregation networks for geo-distributed streaming analytics,” in *ACM/IEEE Symposium on Edge Computing (SEC ’21)*, 2021. [Online]. Available: <https://groups.cs.umass.edu/ramesh/wp-content/uploads/sites/3/2021/06/sec21-final80.pdf>
- [215] R. Shiftehfar, K. Mechitov, and G. Agha, “A fine-grained adaptive middleware framework for parallel mobile hybrid cloud applications,” in *6th Annual International Conference on ICT: Big Data, Cloud and Security*, 07 2015.
- [216] M. A. Hoque, M. Siekkinen, K. N. Khan, Y. Xiao, and S. Tarkoma, “Modeling, profiling, and debugging the energy consumption of mobile devices,” *ACM Comput. Surv.*, vol. 48, no. 3, pp. 39:1–39:40, Dec. 2015. [Online]. Available: <http://doi.acm.org/10.1145/2840723>
- [217] G. Agha, M. Charalambides, K. Mechitov, K. Palmskog, A. Sandur, and R. Shiftehfar, “Inferring and enforcing use patterns for mobile cloud assurance,” in *Assured Cloud Computing*, R. H. Campbell and K. A. K. C. A. Kamhoua, Eds. Wiley/IEEE Press, 2018, pp. 237–376.
- [218] D. Stamoulis, E. Cai, D.-C. Juan, and D. Marculescu, “Hyperpower: Power- and memory-constrained hyper-parameter optimization for neural networks,” in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2018, pp. 19–24.
- [219] Y. Liu, C. Zhu, X. Deng, P. Guan, Z. Wan, J. Luo, E. Liu, and H. Zhang, “Uav-aided urban target tracking system based on edge computing,” *CoRR*, vol. abs/1902.00837, 2019. [Online]. Available: <http://arxiv.org/abs/1902.00837>
- [220] G. Elliott, “The future of cloud gaming is on the edge,” <https://www.datacenterdynamics.com/en/opinions/future-cloud-gaming-edge/>, 2020.
- [221] B. Dieber, C. Micheloni, and B. Rinner, “Resource-aware coverage and task assignment in visual sensor networks,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 21, no. 10, pp. 1424–1437, 2011.
- [222] Adlink, “6 trends driving edge AI in railway transportation,” 2021, <https://blog.adlinktech.com/en/2021/06/08/6-trends-driving-edge-ai-in-railway-transportation/>.
- [223] D. Technologies, “Transportation takes a leading edge with smart technology,” 2020, <https://techhq.com/2020/07/transportation-takes-a-leading-edge-with-smart-technology/>.

- [224] J. Paavilainen, H. Korhonen, K. Alha, J. Stenros, E. Koskinen, and F. Mayra, “The pokémon go experience: A location-based augmented reality mobile game goes mainstream,” in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, ser. CHI ’17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: <https://doi.org/10.1145/3025453.3025871> p. 2493–2498.
- [225] Niantic, “Raid Battle in Pokemon Go,” 2019, https://bulbapedia.bulbagarden.net/wiki/Raid_Battle.
- [226] Niantic, “Trainer Battle in Pokemon Go,” 2019, https://pokemongo.fandom.com/wiki/Trainer_Battle.
- [227] D. Merkel, “Docker: Lightweight linux containers for consistent development and deployment,” *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2600239.2600241>
- [228] P. Nguyen and K. Nahrstedt, “Monad: Self-adaptive micro-service infrastructure for heterogeneous scientific workflows,” in *2017 IEEE International Conference on Autonomic Computing (ICAC)*, 2017, pp. 187–196.
- [229] T. F. P. Institute, *Facial Paralysis Institute Photo Gallery*. <https://www.facialparalysisinstitute.com/photo-gallery/>: The Facial Paralysis Institute, 2022.
- [230] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, “Characterizing and profiling scientific workflows,” *FGCS*, 2013.
- [231] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, “Device placement optimization with reinforcement learning,” *CoRR*, vol. abs/1706.04972, 2017. [Online]. Available: <http://arxiv.org/abs/1706.04972>
- [232] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, “Fingerprinting the Datacenter: Automated Classification of Performance Crises,” in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys ’10. New York, NY, USA: Association for Computing Machinery, 2010. [Online]. Available: <https://doi.org/10.1145/1755913.1755926> p. 111–124.
- [233] C. Tan, Z. Jin, C. Guo, T. Zhang, H. Wu, K. Deng, D. Bi, and D. Xiang, “NetBouncer: Active Device and Link Failure Localization in Data Center Networks,” in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019. [Online]. Available: <https://www.usenix.org/conference/nsdi19/presentation/tan> pp. 599–614.

- [234] Q. Zhang, G. Yu, C. Guo, Y. Dang, N. Swanson, X. Yang, R. Yao, M. Chintalapati, A. Krishnamurthy, and T. Anderson, “Deepview: Virtual Disk Failure Diagnosis and Pattern Detection for Azure,” in *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI’18. USA: USENIX Association, 2018, p. 519–532.
- [235] S. Karumuri, F. Solleza, S. Zdonik, and N. Tatbul, “Towards observability data management at scale,” *SIGMOD Rec.*, vol. 49, no. 4, p. 18–23, Mar. 2021. [Online]. Available: <https://doi.org/10.1145/3456859.3456863>
- [236] S. Das, F. Li, V. R. Narasayya, and A. C. König, “Automated Demand-Driven Resource Scaling in Relational Database-as-a-Service,” in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD ’16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: <https://doi.org/10.1145/2882903.2903733> p. 1923–1934.
- [237] Uber, “The Billion Data Point Challenge: Building a Query Engine for High Cardinality Time Series Data,” 2018, <https://eng.uber.com/billion-data-point-challenge/>.
- [238] Z. Chothia, J. Liagouris, D. Dimitrova, and T. Roscoe, “Online Reconstruction of Structural Information from Datacenter Logs,” in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys ’17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: <https://doi.org/10.1145/3064176.3064195> p. 344–358.
- [239] S. Srivatsan, “Observability at Scale: Building Uber’s Alerting Ecosystem,” 2018, <https://eng.uber.com/observability-at-scale/>.
- [240] Nutanix, “Nutanix Agent - collector agent,” 2021, <https://docs.epoch.nutanix.com/setup-guide/collectors/configuration/>.
- [241] W. Song, Z. Xiao, Q. Chen, and H. Luo, “Adaptive Resource Provisioning for the Cloud Using Online Bin Packing,” *IEEE Trans. Comput.*, vol. 63, no. 11, p. 2647–2660, Nov. 2014. [Online]. Available: <https://doi.org/10.1109/TC.2013.148>
- [242] X. Sun, N. Ansari, and R. Wang, “Optimizing Resource Utilization of a Data Center,” *Commun. Surveys Tuts.*, vol. 18, no. 4, p. 2822–2846, Oct. 2016. [Online]. Available: <https://doi.org/10.1109/COMST.2016.2558203>
- [243] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman, “Aggregation and degradation in jetstream: Streaming analytics in the wide area,” in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014. [Online]. Available: <https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/rabkin> pp. 275–288.

- [244] R. Chaiken, B. Jenkins, P.-r. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, “Scope: Easy and efficient parallel processing of massive data sets,” *Proc. VLDB Endow.*, vol. 1, no. 2, p. 1265–1276, Aug. 2008. [Online]. Available: <https://doi.org/10.14778/1454159.1454166>
- [245] G. Pekhimenko, C. Guo, M. Jeon, P. Huang, and L. Zhou, “TerseCades: Efficient Data Compression in Stream Processing,” in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018. [Online]. Available: <https://www.usenix.org/conference/atc18/presentation/pekhimenko> pp. 307–320.
- [246] Z. Li, Q. Cheng, K. Hsieh, Y. Dang, P. Huang, P. Singh, X. Yang, Q. Lin, Y. Wu, S. Levy, and M. Chintalapati, “Gandalf: An Intelligent, End-To-End Analytics Service for Safe Deployment in Large-Scale Cloud Infrastructure,” in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020. [Online]. Available: <https://www.usenix.org/conference/nsdi20/presentation/li> pp. 389–402.
- [247] S.-U.-R. Baig, W. Iqbal, J. L. Berral, A. Erradi, and D. Carrera, “Adaptive prediction models for data center resources utilization estimation,” *IEEE Transactions on Network and Service Management*, vol. 16, no. 4, pp. 1681–1693, 2019.
- [248] H. Fernandez, G. Pierre, and T. Kielmann, “Autoscaling web applications in heterogeneous cloud infrastructures,” in *2014 IEEE International Conference on Cloud Engineering*, 2014, pp. 195–204.
- [249] A. Bhattacharjee, A. D. Chhokra, Z. Kang, H. Sun, A. Gokhale, and G. Karsai, “Barista: Efficient and scalable serverless serving system for deep learning prediction services,” in *2019 IEEE International Conference on Cloud Engineering (IC2E)*, 2019, pp. 23–33.
- [250] C. Zhang, M. Yu, W. Wang, and F. Yan, “Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving,” in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019. [Online]. Available: <https://www.usenix.org/conference/atc19/presentation/zhang-chengliang> pp. 1049–1062.
- [251] M. Lin, A. Wierman, L. L. H. Andrew, and E. Thereska, “Dynamic right-sizing for power-proportional data centers,” in *2011 Proceedings IEEE INFOCOM*, 2011, pp. 1098–1106.
- [252] A. Sandur, C. Park, S. Volos, G. Agha, and M. Jeon, “Extended version of jarvis: Large-scale server monitoring with adaptive near-data processing,” *Zenodo*, 2021. [Online]. Available: <https://doi.org/10.5281/zenodo.5120252>
- [253] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine, “Synopsis for massive data: Samples, histograms, wavelets, sketches,” *Found. Trends Databases*, vol. 4, no. 1–3, p. 1–294, Jan. 2012. [Online]. Available: <https://doi.org/10.1561/19000000004>

- [254] G. Cormode, S. Muthukrishnan, K. Yi, and Q. Zhang, “Continuous sampling from distributed streams,” *J. ACM*, vol. 59, no. 2, May 2012. [Online]. Available: <https://doi.org/10.1145/2160158.2160163>
- [255] P. K. Agarwal, G. Cormode, Z. Huang, J. M. Phillips, Z. Wei, and K. Yi, “Mergeable summaries,” *ACM Trans. Database Syst.*, vol. 38, no. 4, Dec. 2013. [Online]. Available: <https://doi.org/10.1145/2500128>
- [256] Q. Huang, H. Sun, P. P. C. Lee, W. Bai, F. Zhu, and Y. Bao, “Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy,” in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM ’20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: <https://doi.org/10.1145/3387514.3405877> p. 404–421.
- [257] CSOOnline, “False positives still cause threat alert fatigue,” 2017, <https://www.csoonline.com/article/3191379/false-positives-still-cause-alert-fatigue.html>.
- [258] L. Jiang, “Enable Back Pressure To Make Your Spark Streaming Application Production Ready,” 2017, <https://www.linkedin.com/pulse/enable-back-pressure-make-your-spark-streaming-production-lan-jiang/>.
- [259] RxJava, “RxJava backpressure operators,” 2017, <http://reactivex.io/documentation/operators/backpressure.html>.
- [260] U. Celebi, “How Apache Flink™ handles backpressure,” 2015, <https://www.ververica.com/blog/how-flink-handles-backpressure>.
- [261] E. Begoli, J. Camacho-Rodriguez, J. Hyde, M. J. Mior, and D. Lemire, “Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources,” in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD ’18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: <https://doi.org/10.1145/3183713.3190662> p. 221–230.
- [262] Prometheus, “Prometheus-estimate percentiles from histograms.” 2021, <https://prometheus.io/docs/practices/histograms/>.
- [263] B. Radunovic and J.-Y. L. Boudec, “A Unified Framework for Max-Min and Min-Max Fairness with Applications,” *IEEE/ACM Trans. Netw.*, vol. 15, no. 5, p. 1073–1083, Oct. 2007. [Online]. Available: <https://doi.org/10.1109/TNET.2007.896231>
- [264] Esoteric, “Java binary serialization and cloning: fast, efficient, automatic,” 2020, <https://github.com/EsotericSoftware/kryo/>.
- [265] Apache, “Apache NiFi Connector,” 2021, <https://ci.apache.org/projects/flink/flink-docs-master/docs/connectors/datastream/nifi/>.

- [266] B. Bende, “Integrating Apache NiFi and Apache Kafka,” 2016, <https://bryanbende.com/development/2016/09/15/apache-nifi-and-apache-kafka>.
- [267] Apache, “Side Outputs,” 2021, https://ci.apache.org/projects/flink/flink-docs-master/docs/dev/datastream/side_output/.
- [268] A. Flink, “Apache Flink - Timely Stream Processing,” 2020, <https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/concepts/time/>.
- [269] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, “TimeStream: Reliable Stream Computation in the Cloud,” in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys ’13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: <https://doi.org/10.1145/2465351.2465353> p. 1–14.
- [270] X. Qiao, P. Ren, S. Dustdar, L. Liu, H. Ma, and J. Chen, “Web ar: A promising future for mobile augmented reality—state of the art, challenges, and insights,” *Proceedings of the IEEE*, vol. 107, no. 4, pp. 651–666, 2019.
- [271] P. Guo and W. Hu, “Potluck: Cross-application approximate deduplication for computation-intensive mobile applications,” *ACM SIGPLAN Notices*, vol. 53, pp. 271–284, 03 2018.
- [272] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, “Survey and benchmarking of machine learning accelerators,” in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, 2019, pp. 1–9.
- [273] Y. Li and W. Gao, “Interconnecting heterogeneous devices in the personal mobile cloud,” in *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, 2017, pp. 1–9.
- [274] A. Rahmati, A. Qian, and L. Zhong, “Understanding human-battery interaction on mobile phones,” in *Proceedings of the 9th International Conference on Human Computer Interaction with Mobile Devices and Services*, ser. MobileHCI ’07. New York, NY, USA: ACM, 2007. [Online]. Available: <http://doi.acm.org/10.1145/1377999.1378017> pp. 265–272.
- [275] C. Min, S. Kang, C. Yoo, J. Cha, S. Choi, Y. Oh, and J. Song, “Exploring current practices for battery use and management of smartwatches,” in *Proceedings of the 2015 ACM International Symposium on Wearable Computers*, ser. ISWC ’15. New York, NY, USA: ACM, 2015. [Online]. Available: <http://doi.acm.org/10.1145/2802083.2802085> pp. 11–18.
- [276] P. Guo, B. Hu, R. Li, and W. Hu, “Foggycache: Cross-device approximate computation reuse,” in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom ’18. New York, NY, USA: ACM, 2018. [Online]. Available: <http://doi.acm.org/10.1145/3241539.3241557> pp. 19–34.

- [277] S. Yao, J. Li, D. Liu, T. Wang, S. Liu, H. Shao, and T. Abdelzaher, “Deep compressive offloading: speeding up neural network inference by trading edge computation for network latency,” *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, 2020.
- [278] F. Romero, M. Zhao, N. J. Yadwadkar, and C. Kozyrakis, “Llama: A heterogeneous and serverless framework for auto-tuning video analytics pipelines,” in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC ’21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: <https://doi.org/10.1145/3472883.3486972> p. 1–17.
- [279] H. Sharif, Y. Zhao, M. Kotsifakou, A. Kothari, B. Schreiber, E. Wang, Y. Sarita, N. Zhao, K. Joshi, V. Adve, S. Misailovic, and S. Adve, “Approxtuner: a compiler and runtime system for adaptive approximations,” *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021.
- [280] “Actors in kotlin,” <https://github.com/Kotlin/kotlinx.coroutines/blob/master/docs/shared-mutable-state-and-concurrency.md>, 2016.
- [281] J. B. Bayer, N. B. Ellison, S. Y. Schoenebeck, and E. B. Falk, “Sharing the small moments: ephemeral social interaction on snapchat,” *Information, Communication & Society*, vol. 19, no. 7, pp. 956–977, 2016. [Online]. Available: <https://doi.org/10.1080/1369118X.2015.1084349>
- [282] R. A. Bridges, N. Imam, and T. M. Mintz, “Understanding gpu power: A survey of profiling, modeling, and simulation methods,” *ACM Comput. Surv.*, vol. 49, no. 3, sep 2016. [Online]. Available: <https://doi.org/10.1145/2962131>
- [283] S. V. Rajaraman, M. Siekkinen, and M. A. Hoque, “Energy consumption anatomy of live video streaming from a smartphone,” in *2014 IEEE 25th Annual International Symposium on Personal, Indoor, and Mobile Radio Communication (PIMRC)*, 2014, pp. 2013–2017.
- [284] E. Cai, D. Juan, D. Stamoulis, and D. Marculescu, “\emphneuralpower: Predict and deploy energy-efficient convolutional neural networks,” in *Proceedings of The 9th Asian Conference on Machine Learning, ACML 2017, Seoul, Korea, November 15-17, 2017*, ser. Proceedings of Machine Learning Research, M. Zhang and Y. Noh, Eds., vol. 77. PMLR, 2017. [Online]. Available: <http://proceedings.mlr.press/v77/cai17a.html> pp. 622–637.
- [285] B. Elbouchikhi, “Introducing ml kit,” <https://developers.googleblog.com/2018/05/introducing-ml-kit.html>, 2018.
- [286] M. 2014., “MAX17047/MAX17050, ModelGauge m3 Fuel Gauge.” <https://datasheets.maximintegrated.com/en/ds/MAX17047-MAX17050.pdf>, Maxim Integrated, Tech. Rep., 2014.

- [287] J. Lee, H. Joe, and H. Kim, “Automated power model generation method for smart-phones,” *IEEE Transactions on Consumer Electronics*, vol. 60, no. 2, pp. 190–197, 2014.
- [288] H. Drucker, C. J. C. Burges, L. Kaufman, A. J. Smola, and V. Vapnik, “Support vector regression machines,” in *Advances in Neural Information Processing Systems 9*, M. C. Mozer, M. I. Jordan, and T. Petsche, Eds. MIT Press, 1997, pp. 155–161. [Online]. Available: <http://papers.nips.cc/paper/1238-support-vector-regression-machines.pdf>
- [289] J. Li, T. Nagayama, K. A. Mechitov, and B. F. S. Jr., “Efficient campaign-type structural health monitoring using wireless smart sensors,” in *Sensors and Smart Structures Technologies for Civil, Mechanical, and Aerospace Systems 2012*, M. Tomizuka, C.-B. Yun, and J. P. Lynch, Eds., vol. 8345, International Society for Optics and Photonics. SPIE, 2012. [Online]. Available: <https://doi.org/10.1117/12.914860> pp. 270 – 280.
- [290] J. Maindonald, “Generalized additive models: An introduction with r,” *Journal of Statistical Software, Book Reviews*, vol. 16, no. 3, pp. 1–2, 2006. [Online]. Available: <https://www.jstatsoft.org/v016/b03>
- [291] L. Breiman, “Random forests,” *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001. [Online]. Available: <https://doi.org/10.1023/A:1010933404324>
- [292] J. H. Friedman, “Greedy function approximation: A gradient boosting machine,” *The Annals of Statistics*, vol. 29, no. 5, pp. 1189–1232, 2001. [Online]. Available: <http://www.jstor.org/stable/2699986>
- [293] G. C. Cawley and N. L. Talbot, “On over-fitting in model selection and subsequent selection bias in performance evaluation,” *J. Mach. Learn. Res.*, vol. 11, pp. 2079–2107, Aug. 2010. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1756006.1859921>
- [294] S. Imai, “Task offloading between smartphones and distributed computational resources,” M.S. thesis, Rensselaer Polytechnic Institute, May 2012.
- [295] C. Varela and G. Agha, “Programming dynamically reconfigurable open systems with salsa,” *ACM SIGPLAN Notices*, vol. 36, no. 12, pp. 20–34, 2001.
- [296] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in python,” *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Nov. 2011. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1953048.2078195>
- [297] F. Chollet and Others, “Keras,” <https://github.com/fchollet/keras>, 2015.

- [298] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “Tensorflow: Large-scale machine learning on heterogeneous distributed systems,” 2015. [Online]. Available: <http://download.tensorflow.org/paper/whitepaper2015.pdf>
- [299] M. Kuhn, “caret: Classification and Regression Training,” Astrophysics Source Code Library, May 2015.
- [300] A. Guazzelli, M. Zeller, W.-C. Lin, and G. Williams, “Pmml: An open standard for sharing models,” *The R Journal*, vol. 1, pp. 60–65, 2009.
- [301] T. Y.-H. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan, “Glimpse: Continuous, real-time object recognition on mobile devices,” in *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys ’15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: <https://doi.org/10.1145/2809695.2809711> p. 155–168.
- [302] “Topward 6306d dual tracking dc power supply 0-30v/0-6a x 2; 5v/5a,” <https://www.topward.com.tw/data/6000.pdf>, 1995.
- [303] A. Carroll and G. Heiser, “An analysis of power consumption in a smartphone,” in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC’10. Berkeley, CA, USA: USENIX Association, 2010. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1855840.1855861> pp. 21–21.
- [304] R. Lienhart and J. Maydt, “An extended set of haar-like features for rapid object detection,” in *Proceedings. International Conference on Image Processing*, vol. 1, 2002, pp. I–I.
- [305] “The benefits of multiple cpu cores in mobile devices,” https://www.nvidia.com/content/PDF/tegra_white_papers/Benefits-of-Multi-core-CPUs-in-Mobile-Devices_Ver1.2.pdf, 2010.
- [306] L. Breiman, “Random forests,” *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001. [Online]. Available: <https://doi.org/10.1023/A:1010933404324>
- [307] H. Zou and T. Hastie, “Regularization and variable selection via the elastic net,” *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 67, no. 2, pp. 301–320, 2005.
- [308] Apache, “Data Streaming Fault Tolerance,” 2017, https://nightlies.apache.org/flink/flink-docs-release-1.3/internals/stream_checkpointing.html.

- [309] S. Richter, “A Deep Dive into Rescalable State in Apache Flink,” 2017, <https://flink.apache.org/features/2017/07/04/flink-rescalable-state.html>.
- [310] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, “Integrating scale out and fault tolerance in stream processing using operator state management,” in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: <https://doi.org/10.1145/2463676.2465282> p. 725–736.
- [311] A. Javed, K. Heljanko, A. Buda, and K. Främling, “Cefiot: A fault-tolerant iot architecture for edge and cloud,” in *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*, 2018, pp. 813–818.
- [312] J. Grover and R. M. Garimella, “Reliable and fault-tolerant iot-edge architecture,” in *2018 IEEE SENSORS*, 2018, pp. 1–4.
- [313] H. Sun, H. Yu, G. Fan, and L. Chen, “Qos-aware task placement with fault-tolerance in the edge-cloud,” *IEEE Access*, vol. 8, pp. 77 987–78 003, 2020.
- [314] M. Calder, X. Fan, Z. Hu, E. Katz-Bassett, J. Heidemann, and R. Govindan, “Mapping the expansion of google’s serving infrastructure,” in *IMC*, 2013.
- [315] D. Berman, “CloudFront Log Analysis Using the Logz.io ELK Stack,” 2016, <https://logz.io/blog/cloudfront-log-analysis/>.
- [316] AWS, “Industrial Internet of Things,” 2022, <https://aws.amazon.com/iot/solutions/industrial-iot/>.
- [317] F. Xu, Y. Liu, Q. Li, and Y. Zhang, “V-edge: Fast self-constructive power modeling of smartphones based on battery voltage dynamics,” in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi’13. USA: USENIX Association, 2013, p. 43–56.