Towards Framework-Independent, Non-Intrusive Performance Characterization for Dataflow Computation

Huangshi Tian
HKUST
htianaa@cse.ust.hk

Qizhen Weng
HKUST
qwengaa@cse.ust.hk

Wei Wang
HKUST
weiwa@cse.ust.hk

ABSTRACT

Troubleshooting performance bugs for dataflow computation often leads to a "painful" process, even for experienced developers. Existing approaches to configuration tuning or performance analysis are either specific to a particular framework or in need of code instrumentation. In this paper, we propose a *framework-independent* and *non-intrusive* approach to performance characterization. For each job, we first assemble the information provided by off-the-shelf profilers into a DAG-based execution profile. We then locate, for each DAG node (operation), the source code of its executed functions. Our key insight is that code contains *learnable* lexical and syntactic patterns that reveal resource information. We hence perform code analysis and infer the operations' resource usage with machine learning classifiers. Based on them, we establish a *performance-resource model* that correlates the job performance with the resources used. The evaluation with two Spark use cases demonstrates the effectiveness of our approach in detecting program bottlenecks and predicting job completion time under various resource configurations.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Theory of computation →** *Distributed computing models.*

KEYWORDS

Performance Characterization, Dataflow Systems, Multi-Class Classification

ACM Reference Format:
Huangshi Tian, Qizhen Weng, and Wei Wang. 2019. Towards Framework-Independent, Non-Intrusive Performance Characterization for Dataflow Computation. In *10th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '19), August 19–20, 2019, Hangzhou, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3343737.3343743

1 INTRODUCTION

Dataflow is the *de facto* computing paradigm for a wide spectrum of parallel processing frameworks, ranging from batch and streaming analytics [10, 29, 31, 49] to distributed machine learning [1, 12]. These systems model computation as a directed acyclic graph (DAG) of operations with data flowing among them, and automatically manage the DAG execution in cluster environments. For example, Spark [49] allows programmers to compose a set of (dependent) transformations (e.g., `flatMap` followed by `reduceByKey`) to a distributed dataset; the framework then converts the transformations into a DAG and schedules its execution as parallel tasks on multiple machines. However, owing to the complexity of DAG execution, the performance of dataflow programs is usually hard to predict or reason about, which has led to a *painful* experience [50] for domain experts. In fact, even skilled programmers cannot avoid painstakingly troubleshooting configurations [18] and performance issues [52].

A leading factor that renders performance debugging of dataflow computation a painful process is the lack of handy toolchains (e.g., profiler, debugger, monitor) that can offer *informative* results with *actionable* advices. Existing tools provided by popular dataflow systems often produce an *overwhelming* amount of low-level execution traces. For example, when the user trains ResNet50 [22] with TensorFlow [39], the profiling traces of one iteration would include 9505 tensors, 2950 operators, and 5602 memory operations[1], in which the relevant performance information is easily drowned out. In addition, many profilers generate *framework-specific* information which requires users to have a solid understanding about the framework internals in the first place. As a result, an inexperienced developer may end up with more troubles making sense of the profiling traces (see "performance diagnosing" in §3). Moreover, almost all built-in profilers produce the execution traces only, without offering high-level actionable advices for performance debugging.

There are many approaches proposed recently to improve the status quo, but they either are *framework-dependent* or require *intrusive instrumentation.* Notably, blocked time analysis [35] reasons about the job performance only in Spark; [24] designs a self-tuning system that finds the optimal configuration for MapReduce-like frameworks [15]. These solutions are bound to a specific framework, limiting their applications

[1] The numbers are obtained from an official benchmark provided by TensorFlow (https://github.com/tensorflow/benchmarks).

to the increasingly diversified dataflow systems. Other approaches, such as Monotasks [34] and SnailTrail [26], heavily instrument the existing frameworks for performance clarity (Monotasks) or critical path analysis (SnailTrail). While accurate in profiling, such intrusive approaches fall short in two aspects. First, given the fast evolving codebase of popular dataflow frameworks, the instrumentation code must be manually updated accordingly. Second, the instrumentation itself can be framework-specific (e.g., Monotasks built upon the Spark task implementation), making it difficult to port to other systems.

In this paper, we propose a *non-intrusive* and *framework-independent* approach to performance reasoning and debugging for dataflow computation. Central to our approach is a *performance-resource model* that, for a dataflow job, *infers* from the execution traces how much time each operation has spent on different resources (e.g., CPU, network, disk). The model can then be used to predict the job completion time under varying resource configurations or reveal abnormal resource usage. Our model makes use of the information that is *readily accessible* in stock frameworks (e.g., job DAGs, runtime logs, execution traces given by the built-in profiler, source code), requiring no instrumentation.

We build the performance-resource model for a dataflow job in two stages. In the first stage, we construct a *DAG execution profile* out of the traces generated by the built-in profilers, where each node of the DAG corresponds to an *operation*. Here, "operation" is an umbrella term which may refer to a mathematical operator in a TensorFlow computation graph, a task in a Spark job, or an operation in a Heron [31] topology. For each operation, we extract its execution details from the traces, including the operation name, the start and finish time, and the call trace of its executed functions. We also locate the source code of those functions in the framework's codebase for further analysis.

In the second stage, we infer the resource usage of each operation and characterize it with a *resource vector*. Its components measure the time portion the operation spends on different resources (e.g., 30% on CPU and 70% on network). However, in many frameworks, the resource time cannot be directly obtained from the execution traces. To tackle this challenge, we resort to our insight that how an operation uses resources can be largely characterized from its source code. For example, an operation containing many routines for data encoding (or decoding) is likely bound on CPU; an operation frequently making RPC calls can be network-bound. In fact, many operations are implemented using functions provided by open-source toolkits and libraries (e.g., Netty [16], Akka [3], Parquet [17]), whose resource usage can be clearly identified from both the source code and documentation. Following this insight, we train *classification models* that learn the code patterns (lexical and syntactic) of different resource usage. The training datasets are prepared by extracting code from open-source projects with clear indication of resource usage, e.g., network and I/O primitives.

Combining the DAG execution profile with the inferred resource vector, we can easily generate performance diagnosis by inspecting abnormal resource usage patterns, or predict job runtime in what-if scenarios by simulating its DAG execution. We evaluate the effectiveness of our approach in two Spark use cases: performance debugging and runtime prediction. Preliminary results show that our approach accurately identifies performance bottlenecks, and predicts the job completion time with a 10% deviation on average.

2 OUR APPROACH

Performance-Resource Model Consider a programmer of a dataflow framework following the development workflow of coding, executing, and debugging. If the program runs much slower than she expects, she may ask: "Is there any resource malfunctioning [21, 28]?" After correcting all bugs, she may want to accelerate its execution, wondering "What would happen if I put in use more resources [23, 30]?" If the performance still falls short of her expectation, she has to optimize the program, typically starting with the question: "Which resource is the bottleneck during the execution [26]?" Bottleneck detection and performance debugging can be done by knowing how much time the job has spent on each resource; what-if questions involve predicting how long a job would take under a different resource configuration. Therefore, the answers to the questions above boil down to a *performance-resource model*, which is however lacking in today's dataflow systems.

Our goal is to establish such a performance-resource model for dataflow computation in a non-intrusive manner by leveraging the runtime information (§2.1) that is commonly available in the prevalent frameworks. We first assemble this information into a framework-independent, graph-based execution profile (§2.2). For each operation node, we train two machine learning models to infer its resource usage from the source code and documentation (§2.3), through which the bottleneck can be easily identified by finding the resource that takes the longest time. Our approach can be further generalized for performance prediction (§2.4).

2.1 Availability Assumption

Our approach presumes the availability of the following three types of information in most dataflow systems.

Job DAG Dataflow systems organize the operations of a job as a DAG where each node corresponds to one *operation* and the directed edge represents the dependency between two operations. Most frameworks expose it through monitoring API or runtime logs. For instance, DAG can be extracted from the event logs in Spark, through REST API in Heron and from TensorBoard in TensorFlow.

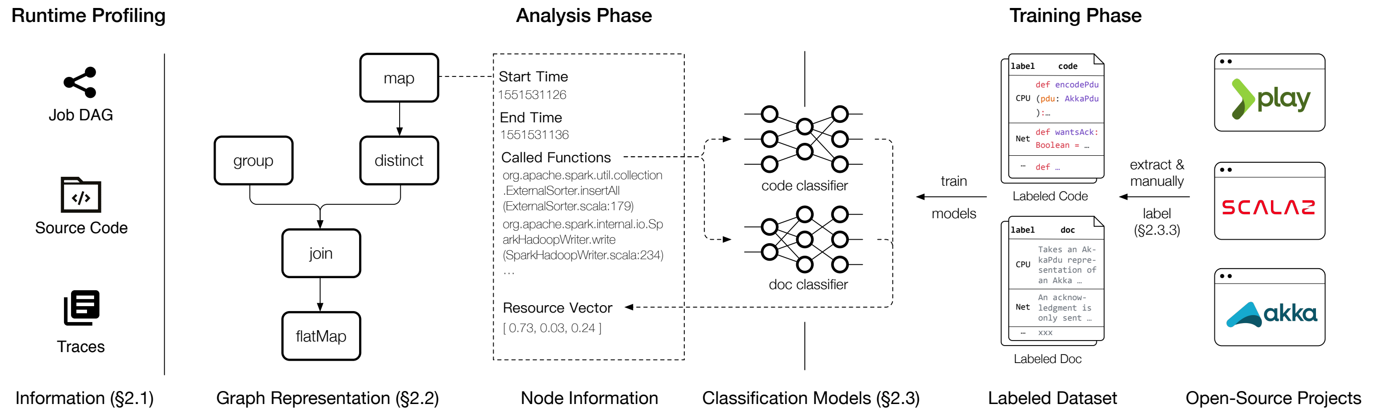

Figure 1: In a nutshell, our approach takes the DAG from runtime profiling and infers the resource usage of each operation based on its source code. For the inference, we design two machine learning models respectively for classifying code and documentation, and train them using source code extracted from open-source projects with manually labeled resource usage.

Source Code Given that many leading dataflow frameworks for big data analytics are open-source, we assume their source code is available and exploit it for analysis. This assumption holds true for TensorFlow, MXNet [12], Heron, Flink [10] and Spark. Moreover, those large scale projects typically have *function-level documentation* which provides useful information about resource usage.

Call Traces Such information is non-standardized, but we find it either accessible or derivable in existing systems. In essence, we intend to know *what code has been executed in each operation*. In TensorFlow and MXNet, the profiling result contains the operator names, and each operator is associated with a specific snippet of code which we can easily trace. However, as such association is lacking in Spark, we take an indirect measure by periodically recording the call stacks of JVM with a non-intrusive sampling profiler.[2] We then align the records with the time when an operation begins and ends to extract the called functions. The similar procedure is applicable to Heron and Flink.

2.2 Graph-based Execution Profile

We establish a performance-resource model for a dataflow job in two stages. In the first stage, we represent the job structure as a DAG, with necessary modification for certain framework. We further annotate each node (operation) with its execution details which will serve as the basis of resource-time inference in the second stage.

Graph Representation In many frameworks, we can directly use the job DAG (e.g., TensorFlow computation graphs and Heron topologies) to represent its internal computation structure, i.e., the operations and their dependencies. A special treatment is needed for Spark, in which a job is decomposed into multiple stages with *wide dependency*, each containing multiple parallel tasks [49]. A completely stage-based graph may be overly coarse while a task-based graph may complicate the analysis. To strike a balance, we allow *hierarchical* DAG where a node may contain another DAG (e.g., a Spark stage consisting of many parallel tasks with narrow dependency).

Node Information After constructing the DAG, we extend the content of each node with detailed execution information, including the start/end time of the operation, the functions that it calls during the execution, the source code related to those functions, etc. Figure 1 shows an example representation of a Spark job and a sample piece of auxiliary information attached to a "map" node.

2.3 Resource Usage Inference

With the source code collected for each operation (including both function code and its associated documentation), we infer its resource usage in the second stage. Specifically, given a snippet of code, we measure its resource usage as a *resource vector*, where each component is the probability of using a type of resource during execution. For example, a resource vector taking the form of $\vec{R} = (p_{cpu}, p_{net}, p_{disk}) = (0.2, 0.5, 0.3)$ implies that the program mainly uses three types of resources, i.e., CPU, network and disk. They respectively account for 20%, 50% and 30% of the execution time.

Our prediction approach is based on the fact that the implementations of many operations are heavily built on functions in standard libraries and open-source toolkits. These functions are usually *low-level* and involves a single type of resource, e.g., algebra library uses CPU; socket operation uses

[2] Sampling profilers [8] periodically sample the execution of JVM and record the call stacks. They are widely used in both academia [14, 43] and industry [20].

network. We thus design machine learning models to learn the code patterns correlated with resource usage, which can be used to predict the resource vector of an operation based on code analysis (§2.3.1). To train those models, we collect source code from several open-source projects with apparent resource usage (§2.3.2).

2.3.1 Inference Method. Inferring resource usage of a code snippet is equivalent to calculating its probability of using a certain type of resource. This problem resembles document classification [32, 48, 51], a well-studied topic that predicts the probability of a document (code snippet) belonging to a certain class. We therefore borrow their techniques and design the following procedure.

General Procedure As source code comes in unstructured text format, we first convert them to a mathematical representation (i.e., a vector) for the convenience of analysis and calculation. In data mining, this process is termed as "embedding" [33, 37] because it embeds the text into a vector space. We embed both code and their associated documentation for inference, which we will elaborate in the coming subsection.

With the embeddings calculated, we use one-layer neural networks [9] to infer resource vectors. The neural network has the advantage of supporting variable length in both input and output, so we can easily adjust the dimension of embedding or incorporate a new type of resource. After feeding the embeddings into the network, we interpret the output as resource vectors. For the vectors generated from code and documentation embeddings, we take their average as the final resource vector for an operation.

Code Embedding Critical to code analysis is *lexical* and *syntactic* information [4]. The former mainly refers to identifiers of variables, functions, classes, types, etc; the latter refers to the abstract syntax trees (AST). To combine them, we adopt the structure of recursive autoencoder [42]. For a snippet of code, we first parse it into an AST and augment it to a binary tree by inserting some artificial nodes [46]. Then the autoencoder will compute a vector representation for each node of the AST. It starts by assigning a random embedding vector to each identifier, so the leaf nodes in the AST now have vector representations attached to them. For the intermediate nodes, the autoencoder computes their vectors in a bottom-up order. It recursively computes the vector representation of parent nodes based on their children's. More concretely, if two sibling nodes are represented as c_1 and c_2, their parent's vector p is computed as $p = W[c_1 : c_2] + b$, where W and b are model parameters, and $[:]$ denotes concatenation. The vector of the root node is taken as the embedding of the code snippet. The model parameters and embedding vectors are iteratively refined through training (§2.3.2).

Documentation Embedding For each function, we view its related documentation as a natural language document. After removing punctuations, each document consists of a sequence of words. We first utilize the state-of-the-art technique of word embedding, Flair [2], to convert words to vector representations. Then we use a well-established network architecture called gated recurrent unit (GRU) [13] for sentence encoding. GRU is a specialized recurrent neural network that can take an arbitrarily long sequence. It has internal states and works in a recursive way: take an input element; update state; produce output and repeat. We feed the word embeddings to the network according to their order in the document and take the state from the last iteration as the overall embedding for the document.

2.3.2 Model Training. In order to train the neural networks used in code (documentation) embedding and resource inference, we collect code with clear resource usage from several open-source projects. The selection criteria include clean design, comprehensive documentation, sustained development activities and wide adoption. Within each project, we manually choose the source files that have apparent resource usage, such as TCP and UDP services (network-dominant), compression and hashing (computation-dominant), file operations (disk-dominant). We admit that code selection unavoidably involves subjectiveness, so we leave it as a future work to validate how different code selection may affect the characterization performance.

We extract the selected functions together with their documentation and manually label their resource usage.[3] For code (documentation) classification model, each data sample contains the code (documentation) of a function and a label of bottleneck resource. During training, we will treat the label as a resource vector where the probability of the bottleneck resource is set to 1 and all others 0. Such approximation is viable as we only choose low-level libraries where most functions only use one type of resource. In each training iteration, we compute the gradient based on the difference between the predicted resource vector and the ground truth (i.e., the labeled vector). We then back propagate [40] the gradient information to update the model parameters. We omit the mathematical details due to the space constraint and refer the interested reader to [13, 46].

Note that the models trained on certain programming languages can be *reused* by the other frameworks or languages, as language models have a transferable [11, 27] nature, and the program code statistically resembles natural language [25, 44]. This allows our system users to employ pre-trained models for performance characterization, without retraining them for a particular framework.

Why it works? The validity of our approach critically depends on the assumption that *the code and documentation correctly reflect the resource usage.* We discover several pieces

[3]Manual labeling is feasible as long as the project follows modern design guidelines (e.g., modularity, high cohesion, low coupling). For instance, in a well modularized project, functionally similar code tend to appear in the same module, so it is easy to "batch process" them.

of evidence in current practice of coding and documenting. Most obviously, different resources typically have their specific terminologies (e.g., transmitting *packets* over network, reading *blocks* from disk) that are easily distinguishable. The documentation written in natural language frequently mentions those terminologies, thus revealing the resource information. This is also the case for the identifier names in source code [25]. Moreover, the AST is distinguishable as well for certain resources. For instance, the matrix operation library [5] is awash with `for` loops and mathematical operators, suggesting the heavy usage of CPU. Despite the empirical justification provided above, we leave it as a future work to verify the assumptions with quantifiable metrics and refutable proofs.

2.4 Performance Characterization

With the resource vector inferred for each operation, we further have to model *how operation runtime changes with regard to resource allocation* so as to predict or debug job performance. Since different resources affect performance in a different manner, we categorize them into three classes according to the von Neumann architecture and model them separately.

Computing Devices CPU, GPU or TPU mainly influence job execution in two respects. (1) Utilizing devices with different computing speeds (e.g., upgrading CPU to GPU) may change the durations of computation tasks. Suppose the number of low level instructions remains the same, switching from original device with computing speed s_c (e.g., 5MHz, 10 MFLOPS) to a new one s'_c will cause the original runtime t to become $t' = t \cdot s_c / s'_c$. (2) Altering the quantity of devices (e.g., CPU cores) may cause the parallelism of tasks to change. For instance, Spark will schedule more tasks concurrently if given more cores; TensorFlow will perform more calculations; Flink will arrange more task slots. Therefore, given such resource variation during prediction, we replay the DAG execution by rescheduling the parallel tasks and update the job runtime.

Memory It plays a passive role in job execution for it never performs active operations. As long as given enough memory, a job won't run any faster even if we allocate more memory to it. However, if a job runs short of it, frequent garbage collection or communication with secondary storage is likely to elongate the execution. In light of such behavior, we adopt a "reversed" roofline model [47], i.e., a linear decreasing function followed by a constant lower limit, to model the relationship between memory and runtime. The turning point is chosen by summing up the data size used in the program, e.g., RDD blocks in Spark, tensors in TensorFlow, network buffers in Flink; the slope is set as the ratio between the I/O speed of in-memory and on-disk data access. The intuition behind is that memory is sufficient as long as it can accommodate all data, so the delay is primarily caused by exchanging data between memory and disk.

I/O Devices NIC cards, hard drives or even I/O buses have data transmission as their primary functionality, and the bandwidth largely determines the time spent. We hence choose a linear relationship to approximate their operation runtime. Suppose the amount of data is held constant, changing a device with bandwidth B to a new one with B' will let the runtime t become $t' = t \cdot B / B'$.

Putting It All Together We combine the execution profile with resource-runtime model to characterize job performance. For debugging, we simply output the bottleneck resource that takes the most of time during execution, which can help users understand the potential bottleneck of the program. For prediction, we consider the scenario where a user profiles the execution of a job under a certain resource configuration and wonders how the completion time would change with more or less resources. After converting the trace into an annotated DAG representation, we simulate its execution under the given resource configuration and determine the runtime of each operation with the aforementioned device-specific models. We then sum up the adjusted runtime and output it as our prediction.

It is noteworthy that such estimation is *approximate* in its nature because our simplified models only capture the commonality among resources. The approximation is nonetheless acceptable in our approach because the users are mostly interested in knowing the general trend of runtime with varying resource allocation instead of the exact job durations. Provided that they are aware of which resource is the bottleneck, they can take action to alleviate or optimize it. Similarly, when cluster schedulers decide which job should be allocated with more resources, it suffices to make a rough numerical comparison on the estimated runtime.

3 PRELIMINARY EVALUATION AND FUTURE WORK

We have prototyped our approach as a command line tool in Python. For resource inference, we have implemented the models for code and documentation embedding (§2.3.1) with PyTorch [36]. To train those models, we have extracted around 5,000 low-level functions from five popular open-source libraries, including FS2 [19], Twitter Util [45], Play Framework [38], Scala Standard Library, and Scalaz [41]. The resource usage of those functions are manually labeled by a domain expert in several hours.

Performance Diagnosing To evaluate how our approach can streamline performance diagnosing, we apply our tool to a real case [7] reported in Stack Overflow, where a programmer implemented a Spark program to count a dataset in HDFS, yet found it taking much longer time than expectation. Someone else mentioned Spark UI for debugging, but it has led to more confusion as the programmer replied: *"In fact, after looking at the information in the Spark UI, I leave with more questions than answers ..."*

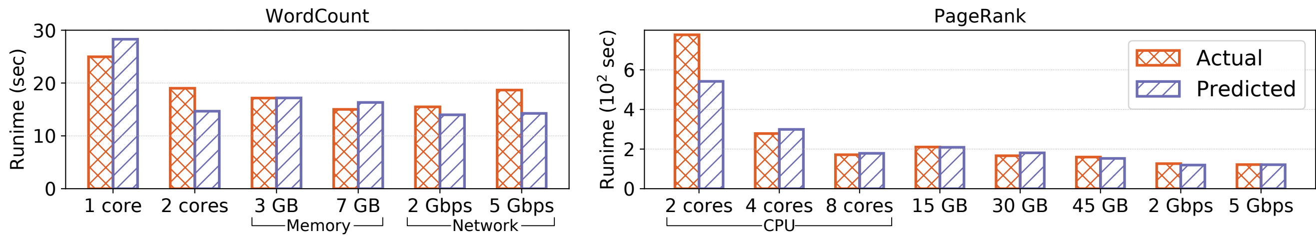

Figure 2: We run two Spark applications and collect their traces. We then vary certain resource configurations (e.g., reducing memory to 7 GB) and use our tool to predict the job completion time. Each bar group corresponds to one execution with one resource variation.

We reproduce this case with the same configuration as mentioned in the original post. After applying our tool, we find that, within each task, a large portion of time is spent in CPU-related function calls (i.e., decompressing and decoding data from raw data blocks). So we simply double the number of cores and, as a result, the runtime drops from 25.89s to 14.76s, proving that CPU is the bottleneck.

Runtime Prediction We next evaluate how our approach can be used to predict the job runtime under different resource configurations (§2.4). We experiment with two Spark applications, `WordCount` and `PageRank`, on three-node EC2 [6] clusters with instance type `m5.xlarge` (4 cores, 16 GB memory, 10 Gbps network) and `m5.4xlarge` (16 cores, 64 GB memory, 10 Gbps network), respectively. We run two applications with all the available resources and collect the traces. Then we use our tool to predict the runtime under the condition of adjusting the amount of resources, e.g., reducing memory from 16 GB to 7 GB. To check the accuracy of prediction, we rerun the job on real cluster with the same resource configuration as in prediction. Figure 2 shows the actual and predicted runtimes for two applications, with an average prediction error of 10.06%. The effective overhead of our approach comes from the sampling profiler, which we set to sample once per 100ms. We measure WordCount jobs with and without profiler and their runtimes differ within 2%, lower than the overhead of instrumentation in SnailTrail [26] (10%).

Future Work The inaccuracy of current prediction partially stems from the neglect of *pipelined operations*. We plan to address it with our key insight that each data block is processed *sequentially* even among multiple threads or workers. In Spark, data blocks have to be read first, then computed, but not both at the same time; tensors in TensorFlow cannot participate in multiple calculations simultaneously; buffered data are transferred sequentially among long-running operators in Flink. Therefore, if we further decompose an operation into *sub-operations* on the level of data blocks, then the pipelining vanishes and our approach becomes directly applicable. The sub-operations can be inferred from the thread-level stack traces provided by sampling profilers.

Our approach is also applicable to streaming systems, though they even lack the notion of "job completion time" given the *unbounded* data streams. We observe that, under the hood, streaming data are chunked into blocks and then processed, so we narrow down our attention onto one data block and focus on how long it takes to traverse the whole processing logic. Such duration is similar to the job duration in batch processing frameworks and naturally fits into our approach. However, as the data size may vary for each block, we will record it and normalize the runtime against it to get a size-agnostic performance indicator.

Finally, a possible obstacle to extending our approach to deep learning systems is that they may have multiple implementations for the same operation. For example, in TensorFlow or MXNet, a mathematical operator may have both CPU and GPU implementations, even TPU or FPGA. We may leverage the log information to infer which resource the operator is actually running on.

4 CONCLUSION

In this paper, we proposed a framework-independent and non-intrusive approach to performance characterization for dataflow computation. Our approach constructs a DAG execution profile of a job and infers the resource usage for each operation using classification models trained over manually labeled functions from open-source projects. Preliminary evaluation shows that our approach can detect program bottlenecks and predict job runtime under varying resource configurations.

ACKNOWLEDGMENT

This work was supported in part by RGC grant 26213818. Huangshi Tian and Qizhen Weng were supported in part by the Hong Kong PhD Fellowship Scheme.

REFERENCES

[1] Martín Abadi, Paul Barham, et al. 2016. Tensorflow: a system for large-scale machine learning.. In *OSDI*.
[2] Alan Akbik, Duncan Blythe, and Roland Vollgraf. 2018. Contextual string embeddings for sequence labeling. In *COLING*.
[3] akka. 2019. Apache Akka. https://akka.io. (2019).
[4] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* (2018).

[5] Edward Anderson, Zhaojun Bai, Jack Dongarra, et al. 1990. LAPACK: A portable linear algebra library for high-performance computers. In *SC*.

[6] Amazon AWS. 2018. Amazon EC2. (2018). https://aws.amazon.com/ec2/

[7] Berne. 2017. Spark count() taking long to run. https://bit.ly/2J7E8ma. (2017).

[8] Walter Binder. 2006. Portable and accurate sampling profiling for Java. *Software: Practice and Experience* (2006).

[9] Christopher M Bishop. 2006. *Pattern recognition and machine learning*.

[10] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* (2015).

[11] Lingzhen Chen and Alessandro Moschitti. 2019. Transfer Learning for Sequence Labeling Using Source Model and Target Data. In *AAAI*.

[12] Tianqi Chen, Mu Li, et al. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *Neural Information Processing Systems, Workshop on Machine Learning Systems* (2015).

[13] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, et al. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *EMNLP* (2014).

[14] Daniel J Dean, Hiep Nguyen, Xiaohui Gu, Hui Zhang, et al. 2014. Perfscope: Practical online server performance bug inference in production cloud computing infrastructures. In *SoCC*.

[15] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*.

[16] Netty Developers. 2019. Netty Project. https://github.com/netty/netty. (2019).

[17] Parquet Developers. 2019. Parquet MR. https://github.com/netty/netty. (2019).

[18] Danyel Fisher, Rob DeLine, Mary Czerwinski, and Steven Drucker. 2012. Interactions with big data analytics. *interactions* (2012).

[19] functional-streams-for scala. 2019. FS2: Functional Streams for Scala. https://github.com/functional-streams-for-scala/fs2. (2019).

[20] Brendan Gregg. 2016. The Flame Graph. *Commun. ACM* (2016).

[21] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, et al. 2018. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *FAST*.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*.

[23] Herodotos Herodotou and Shivnath Babu. 2011. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. In *VLDB*.

[24] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, et al. 2011. Starfish: A Self-tuning System for Big Data Analytics.. In *CIDR*.

[25] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *ICSE*.

[26] Moritz Hoffmann, Andrea Lattuada, John Liagouris, Vasiliki Kalavri, Desislava Dimitrova, Sebastian Wicki, Zaheer Chothia, and Timothy Roscoe. 2018. Snailtrail: Generalizing critical paths for online analysis of distributed dataflows. In *NSDI*.

[27] Jeremy Howard and Sebastian Ruder. 2018. Universal language model fine-tuning for text classification. In *ACL*.

[28] Peng Huang, Chuanxiong Guo, et al. 2017. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *HotOS*.

[29] Michael Isard, Mihai Budiu, et al. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*. ACM.

[30] Yurong Jiang, Lenin Ravindranath Sivalingam, Suman Nath, and Ramesh Govindan. 2016. WebPerf: Evaluating what-if scenarios for cloud-hosted web applications. In *SIGCOMM*.

[31] Sanjeev Kulkarni, Nikunj Bhagat, et al. 2015. Twitter heron: Stream processing at scale. In *SIGMOD*.

[32] Larry M Manevitz and Malik Yousef. 2001. One-class SVMs for document classification. *Journal of machine Learning research* (2001).

[33] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *NeurIPS*.

[34] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. 2017. Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks. In *SOSP*.

[35] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making sense of performance in data analytics frameworks. In *NSDI*.

[36] Adam Paszke, Sam Gross, Soumith Chintala, et al. 2017. Automatic differentiation in pytorch. *Neural Information Processing Systems, Autodiff Workshop* (2017).

[37] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *KDD*.

[38] playframework. 2019. Play Framework. https://www.playframework.com. (2019).

[39] Ladislav Rampasek and Anna Goldenberg. 2016. Tensorflow: Biology's gateway to deep learning? *Cell systems* (2016).

[40] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. 1988. Learning representations by back-propagating errors. *Cognitive modeling* (1988).

[41] scalaz. 2019. Scalaz. https://scalaz.github.io/7/. (2019).

[42] Richard Socher, Jeffrey Pennington, Eric H Huang, Andrew Y Ng, and Christopher D Manning. 2011. Semi-supervised recursive autoencoders for predicting sentiment distributions. In *EMNLP*.

[43] Nathan Tallent and John Mellor-Crummey. 2009. Effective performance measurement and analysis of multithreaded applications. In *PPoPP*.

[44] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In *FSE*.

[45] twitter. 2019. Twitter Util. https://github.com/twitter/util. (2019).

[46] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *ASE*.

[47] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* (2009).

[48] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. 2016. Hierarchical attention networks for document classification. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

[49] Matei Zaharia, Mosharaf Chowdhury, et al. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*.

[50] Ce Zhang, Wentao Wu, and Tian Li. 2017. An overreaction to the broken machine learning abstraction: The ease. ml vision. In *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics*.

[51] Xiang Zhang, Junbo Zhao, and Yann LeCun. 2015. Character-level convolutional networks for text classification. In *NeurIPS*.

[52] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An Empirical Study on TensorFlow Program Bugs. In *ISSTA*.