CrystalPerf: Learning to Characterize the Performance of Dataflow Computation through Code Analysis

Huangshi Tian, *HKUST;* Minchen Yu, *HKUST and Huawei Technologies Ltd.;*
Wei Wang, *HKUST*

https://www.usenix.org/conference/atc21/presentation/tian

This paper is included in the Proceedings of the
2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

CrystalPerf: Learning to Characterize the Performance of Dataflow Computation through Code Analysis

Huangshi Tian[1], Minchen Yu[1,2], Wei Wang[1]
[1] *HKUST,* [2] *Huawei Technologies Ltd.*
{htianaa,myuaj,weiwa}@cse.ust.hk

Abstract

Dataflow computation dominates the landscape of big data processing, in which a program is structured as a directed acyclic graph (DAG) of operations. As dataflow computation consumes extensive resources in clusters, making sense of its performance becomes critically important. This, however, can be difficult in practice due to the complexity of DAG execution. In this paper, we propose a new approach that *learns* to characterize the performance of dataflow computation based on *code analysis*. Unlike existing performance reasoning techniques, our approach requires *no code instrumentation* and *applies to a wide variety of dataflow frameworks*. Our key insight is that the source code of an operation contains learnable syntactic and semantic patterns that reveal how it uses resources. Our approach establishes a performance-resource model that, given a dataflow program, infers automatically how much time each operation has spent on each resource (e.g., CPU, network, disk) from past execution traces and the program source code, using machine learning techniques. We then use the model to predict the program runtime under varying resource configurations. We have implemented our solution as a CLI tool called CrystalPerf. Extensive evaluations in Spark, Flink, and TensorFlow show that CrystalPerf can predict job performance under configuration changes in multiple resources with high accuracy. Real-world case studies further demonstrate that CrystalPerf can accurately detect runtime bottlenecks of DAG jobs, simplifying performance debugging.

1 Introduction

Dataflow frameworks are deployed widely in the cloud for distributed machine learning (e.g., TensorFlow [2]), processing data streams (e.g., Flink [11]), and analyzing big data (e.g., Spark [81]). Dataflow frameworks provide a library of built-in operations (e.g., map, reduce, and tensor operators), which programmers use to compose a data processing pipeline, structured as a directed acyclic graph (DAG) of operations. Data flows between these operations and is thereby transformed. The framework schedules DAG executions as parallel tasks on a cluster of machines.

However, dataflow computation routinely faces various performance issues such as resource contention, inefficient configuration, and poor scalability. Due to the complexity of DAG executions, troubleshooting performance issues in dataflow computation usually demands painstaking efforts even for skilled programmers [19, 82, 84].

A leading factor that makes performance debugging difficult is the lack of handy toolchains that can provide useful *performance-resource information* with actionable advices. Existing tools provided by popular frameworks often generate an overwhelming amount of low-level execution traces, which inexperienced developers may find difficult to even make sense of them. Many system solutions have hence been developed recently to simplify performance reasoning, but they either are designed for a specific framework (e.g., [31, 54, 55]) or require modifying the framework's source code with intrusive instrumentation (e.g., [34, 54]). As the codebases of dataflow frameworks are evolving rapidly, the instrumentation code itself needs to be updated frequently, mandating even more, repeated labor.

In this paper, we present CrystalPerf[1], a new performance characterization tool that requires *no code instrumentation* while *generally applicable* to an array of batch dataflow frameworks. Central to our design is a *learning-based performance-resource model* that, for a dataflow job (program), infers from the execution traces and the program source code how much time each operation has spent on different resources (e.g., CPU, network, disk). The model can then predict the program runtime under varying configurations or identify abnormal resource uses in execution.

CrystalPerf builds the performance-resource model for a dataflow job in two stages. In the first stage, it constructs a DAG execution profile (§4.2) from the log traces generated by the built-in profilers, where each node of the DAG repre-

[1] Our tool serves as a *crystal sphere* where users can tell the performance of their dataflow jobs, either performance issues in the past execution or performance prediction in what-if scenarios.

sents an operation. For each operation, it extracts the execution details from the log traces, including the start and finish time, and the call trace of the executed functions. It then locates the source code of those functions in the framework's codebase for further analysis.

In the second stage, CrystalPerf infers the resource use of each operation, characterized by the *resource-time* which measures the time proportion the operation spends on a resource type (e.g., 30% runtime on CPU, 70% on network). However, many frameworks provide no such information in the log traces, nor can it be obtained using common profiling tools. To tackle this challenge, our key insight is that *how an operation uses resources can be largely characterized by analyzing its source code.*[2] For example, an operation containing many routines for data compression (data transfer) is likely CPU-bound (network-bound). In fact, many dataflow frameworks implement operations using low-level functions provided by standard open-source toolkits and libraries (e.g., Netty [17], Akka [4], Parquet [18]), where experienced developers can clearly identify the bottleneck resources by just reading the source code of those functions. Our approach imitates this process with *neural network classifiers* that learn the resource use by analyzing the syntactic and semantic patterns of the function code (§3.2). The classifiers are trained using the source code of low-level functions (e.g., network and I/O primitives) collected from popular open-source projects with clear indications of resource uses. The trained classifiers can then be used in any framework and language.

Combining the DAG execution profile and the inferred resource-time, CrystalPerf can easily detect the performance bottleneck or abnormal resource uses. CrystalPerf can also predict the job performance in what-if scenarios (e.g., "how would the runtime change if the network is upgraded from 1Gbps to 10Gbps?") by simulating the DAG execution under the assumed configuration (§5).

We have implemented CrystalPerf as a CLI tool (§6) with support of batch computations of Spark, TensorFlow, and limited cases of Flink [11]. To demonstrate its efficacy (§7), we use CrystalPerf to predict the runtime changes of dataflow jobs under varying configurations. Across six standard benchmarking workloads running in three frameworks, CrystalPerf predicts within an average deviation of 13.47%. Even compared with Monotasks [54], a meticulously architected Spark implementation for performance clarity, CrystalPerf makes runtime predictions with comparable accuracy. We highlight the framework generalizability and performance advantage of CrystalPerf over the existing dataflow characterization approaches in Figure 1. We further show through three real-world case studies that CrystalPerf can help troubleshoot various performance issues that are otherwise difficult to reason about for non-expert programmers.

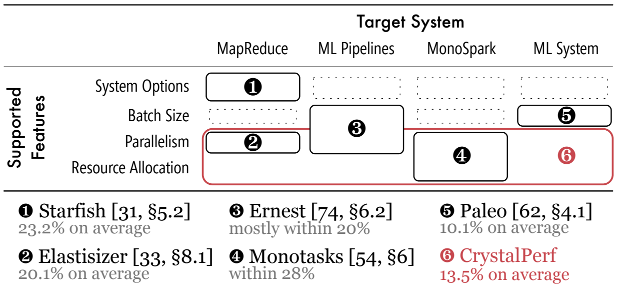


Figure 1: For performance reasoning, we compare CrystalPerf with existing works by answering "On *which features* can they predict the performance of *what systems* within *how much deviation*?"

2 Background and Motivation

In this section, we motivate the need of performance characterization for dataflow programs, where the key is to identify the performance-resource relationship (§2.1). We survey existing solutions and discuss their limitations, which are also the challenges we shall address (§2.2).

2.1 Performance Characterization

Performance characterization plays an indispensable role in the development of dataflow applications. Consider a programmer following the typical development workflow of coding, debugging, and optimizing. If the program runs much slower than expected, she may ask, "Is there any resource malfunctioning [28, 36]?" After fixing those issues, she may want to accelerate its execution, wondering, "What would happen if I put in use more resources [32, 38]?" If the performance remains unsatisfactory, she has to optimize the implementation, typically starting with the question, "What resource is the bottleneck during execution [34]?"

Understanding the performance of dataflow programs also allows cluster operators to better schedule DAG jobs and tune their configurations. State-of-the-art cluster schedulers need to predict how long each task of a DAG job would take if given a certain amount of resources [26, 27, 51, 71]; when deploying a user's job, the operator also wants to tune the optimal configuration for fast job execution [5, 33].

The key to performance characterization is to identify the *performance-resource relationship*. For example, bottleneck detection and performance debugging can be done by knowing how much time the job spends on each resource; what-if questions, runtime estimation and configuration tuning requires predicting how long a job (or its tasks) would take under a different resource configuration.

However, characterizing the performance-resource relationship for dataflow programs is difficult. During execution, different operations may bottleneck on different resources, e.g., data compression being CPU-intensive, file writing

[2] The source code includes both the program code *and* the associated documentation such as Javadoc.

disk-intensive. For some operations like map in Spark, the actual bottleneck resource depends critically on the input (the map functions applied to an RDD). To make the problem even more complicated, some operations like join and shuffle require cross-machine synchronization and data communication. In practice, even skilled programmers cannot avoid painstakingly reasoning about and troubleshooting performance issues [19, 82, 84].

2.2 Related Work and Challenges

Existing profiling tools provided by popular dataflow frameworks produce an overwhelming amount of low-level traces, in which the relevant performance information is easily drowned out. Take TensorFlow as an example. When training ResNet50, the profiling traces of a single iteration include around 9,500 tensors, 3,000 operators, and 5,600 memory operations [1]. In fact, almost no built-in profilers provide high-level actionable advices for performance debugging. Many performance characterization tools have hence been developed recently for dataflow computation. However, they cannot fully tackle the challenges as summarized below.

Challenge 1: No explicit resource-time information. Though Linux (and other Unix-like systems) provides built-in utilities for time accounting and device monitoring, profiling the program's resource-time (how long the program spent on each resource type) remains elusive. Linux's I/O monitoring mechanism is mainly used for throughput tracing. For example, the per-process statistics given in /proc [49] only report the total amounts of read/written characters/bytes or sent/received bytes/packets. Without knowing the effective bandwidth and the data exchange size in execution, the actual I/O time cannot be estimated correctly. The counter-based observability tool perf and the industrial toolchains built on it [63] share the same problem.

The missing resource-time information cannot be uncovered by the recently proposed profiling tools either. Notably, SnailTrail [34] finds the critical path in distributed dataflow, but focuses on coarse-grained activities instead of fine-grained resource use information; Ernest [74] builds the performance model for advanced analytics like machine learning, but only includes input size and cluster scale as parameters. The lack of resource-time information limits their ability in addressing the resource-related issues.

Challenge 2: Heterogeneous hardware behavior. Heterogeneous hardware further complicates the performance model as dataflow operations usually have different resource patterns (e.g., data compression being CPU-intensive, file writing disk-intensive). Two approaches have been proposed in the literature. *Whitebox* approaches such as Starfish [31] build a detailed performance model for MapReduce execution, which cannot be applied to other frameworks. Blackbox approaches, such as CherryPick [5] and Ernest [74], construct statistical machine learning models to correlate the resource allocations with the job performance, but it has to repeat the training process for each job type, making it time-consuming for ad-hoc or non-recurrent jobs.

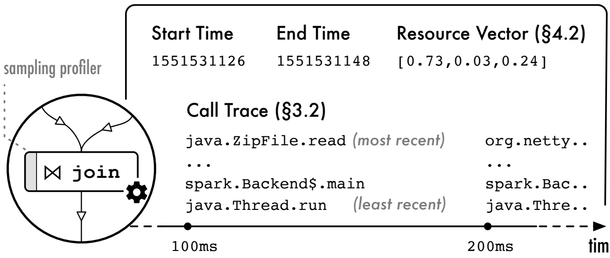


Figure 2: When a node (operation) is executing, attached is a sampling profiler for periodically sampling the call stacks. For each node, we build an *execution profile* to organize its resource-time information.

Challenge 3: Cross-framework support. As dataflow computation prevails in a wide range of frameworks and application domains [2, 11, 12, 42, 47, 48, 79, 81], a general solution with cross-framework support is highly desirable. However, many existing works are tailored to a certain framework, e.g., AROMA [43], ARIA [75], Elastisizer [33] and Starfish [31] building specialized models for MapReduce programs, and DAC [80] designing an algorithm for tuning Spark jobs. Their designs are built on the internal details of the target frameworks and cannot be ported to other systems.

Our Answer In the coming sections, we incrementally develop our solution CrystalPerf to address the three challenges. We start to show in §3 that the time an operation spends on each type of resource can be learned from the source code and its call trace (addressing Challenge 1), using neural network classifiers which, once trained, can be applied to any framework and language (addressing Challenge 3). We then assemble the operation-level resource-time information into a job-level execution profile in §4. In §5, we further construct a performance-resource model to capture the program behaviors with respect to the allocated hardware resources such as CPU, GPU, memory, I/O and network (addressing Challenge 2). Our solution is novel as it requires no code instrumentation and applies to a wide array of dataflow frameworks.

3 Mining Resource-Time from Traces

In this section, we characterize the resource-time of an operation by exploiting the latent information in the call traces (§3.1). The periodic samples of call stacks can tell how long each operation (and its called functions) runs. If we could further obtain the resource use information of each function, then the resource-time of that operation becomes available. We show that such information can be inferred from the function source code and documentation using neural network classifiers (§3.2–3.3). We validate the feasibility of this approach with a labeled and verified dataset (§3.4).

3.1 Obtaining Call Traces

Standard libraries provide built-in utilities to sample the call stacks, e.g., `backtrace` in GNU C Library and `AsyncGetCallTrace` in Java. CrystalPerf uses the sampling profilers [9] built on these utilities to collect the call traces of a dataflow job and its operation tasks periodically. Figure 2 shows a sample trace of a running `join` task in Spark.

In frameworks like TensorFlow and MXNet, CrystalPerf uses the built-in profiler as it records the start time and the duration of each operation. For JVM-based frameworks like Spark, Flink and Heron, CrystalPerf attaches a lightweight, non-intrusive sampling profiler to the JVM process, and aligns the trace with the time when an operation begins and finishes to extract the called functions. CrystalPerf then locates the implementation of those functions in the framework's codebase for further analysis. Note that using similar sampling profilers for C# [25], JavaScript [68] and Go [23], the call traces can also be obtained in other frameworks.

3.2 Inferring Resource Use Patterns

With the durations of each operation and its called functions, we derive the resource-time information by analyzing how those functions use resources. As those functions are usually *low-level routines* (e.g., network and I/O primitives), they have clear resource use patterns which can be identified by a skilled programmer by simply examining the source code and its documentation. Following this intuition, we design two neural network classifiers to *infer* the resource patterns from the code and documentation, respectively. This approach requires no instrumentation and is not limited to a specific framework. Figure 3 illustrates how it works. We first embed the source code and documentation into vectors so that they can be processed by neural networks.

Code Embedding The key to code analysis is the *lexical and syntactic information*. The former includes the identifiers of variables, functions, classes and types; the latter refers to the abstract syntax tree (AST). CrystalPerf embeds them with a recursive autoencoder [67] (Figure 3). Following the embedding approach in [76], CrystalPerf parses the function code into an AST and augment it to a binary tree with some artificial nodes. CrystalPerf then constructs a vector representation for each AST node using the autoencoder. It starts by computing a word embedding for each identifier on a leaf node and recursively computes the embedding vector of a parent node form its children's. Formally, let s_1 and s_2 be the two sibling nodes. The autoencoder computes their parents vector as $p = W[s_1 : s_2] + b$, where W and b are model parameters refined by training, and [:] means concatenation. The vector of the root node gives the code embedding.

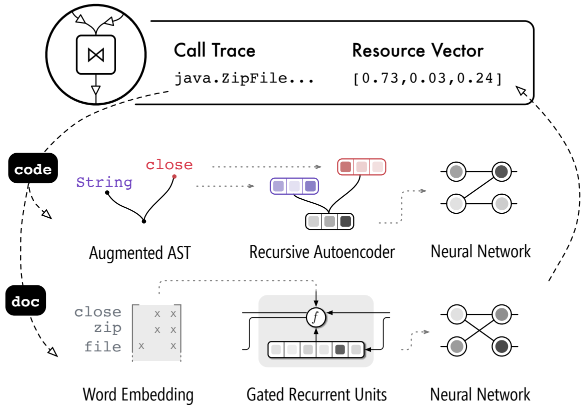


Figure 3: We design two classifiers that can infer the dominant resources for a function from the source code and documentation respectively.

Documentation Embedding A function may have an associated documentation explaining its performed task and usage, written in natural language. To embed a function documentation, CrystalPerf removes the contained punctuations and hyperlinks and encodes the words into vectors with Flair [3], a language model with state-of-the-art performance. CrystalPerf then encodes the whole document with gated recurrent unit (GRU) [14] which can take an arbitrarily long input sequence. GRU has internal states and works in a recursive manner: it takes an input element, updates state, produces output and repeats. CrystalPerf sequentially feeds the word embeddings to GRU and takes the state from the last iteration as the document embedding.

Inference Procedure With the code and documentation embeddings calculated, CrystalPerf infers the function's resource uses with two pre-trained neural network models (see §3.3). Both models output a *resource vector*, where each element is the probability of the function using a certain type of resource (e.g., 0.7 in CPU and 0.3 in network), given by the code or documentation analysis. CrystalPerf takes the average of the two predictions as the inferred resource vector[3]. Our design currently supports three resource types and the output is in the form of `<CPU, Disk, Network>`. We next describe the neural network design and training.

3.3 Model Design and Training

Critical to our approach are the two neural network models that learn the resource vector from the function code and documentation. This problem resembles document classifi-

[3] §B in the supplemental material [70] provides the justification for averaging.

Table 1: The prediction quality of the classifier on the collected dataset. The higher-than-random accuracies imply the learnability of resource information from the source code.

	Accuracy	F1 Score
CPU	$77.12\% \pm 7.39\%$	0.869 ± 0.046
Disk	$82.46\% \pm 5.63\%$	0.903 ± 0.034
Network	$82.90\% \pm 5.26\%$	0.906 ± 0.032
Average	$80.83\% \pm 4.60\%$	0.893 ± 0.028

Table 2: Top 5 words that make the classifier predict a certain category. Most of them appear in *both code and documentation*, except that the underlined ones only in code.

Category	Words
CPU	engine, entry, stream, key, certificate
Disk	file, error, tar, info, name
Network	socket, sock, send, result, address

cation [78,83], in which a neural network model is trained to predict the probability of a document belonging to a certain class. In our design, we adopt the single-layer neural network (i.e., linear model) because it supports variable lengths in both input and output, allowing us to adjust the dimension of embeddings and incorporate new resources.

To train the two classification models, we collect source code from open-source projects with clean designs and comprehensive documentation. In each project, we choose the source files that have apparent resource uses, such as TCP and UDP services (network-bound), compression and hashing (CPU-bound), file operations (I/O-bound). We extract the selected functions together with their documentation and label their resource patterns. In total, we have collected around 5,000 low-level functions from five popular open-source libraries, including FS2 [22], Twitter Util [72], Play Framework [59], Scalaz [65] and Scala Standard Library.[4] It took two PhD students 5 hours to manually label those functions.[5] The labeled resource vectors are in the form of `<CPU, Disk, Network>`, with the dominant resource set to 1 and the others 0 (one-hot vector). Although some functions use more than one resource, only labeling the dominant one is still feasible as our training dataset only includes low-level libraries where most functions rely on just one type of resource. Admittedly, such labeling inevitably involves subjectiveness and imprecision, but our sensitivity analysis (§7.4) shows that CrystalPerf is robust against the labeling errors.

We have implemented the two classifiers with PyTorch [57].[6] The classification models, once trained, are *not restricted* to certain frameworks or programming languages. First, our model design accepts arbitrary words as its input because the word embedding techniques, such as Flair [3], work on the level of characters and allow unseen words beyond the training corpus. Second, regardless of the frameworks, their documentation is all written in natural language, enabling the reuse of the documentation classifier. Third, many keywords and syntaxes (e.g., condition and repetition) are commonly used in different programming languages, a common ground for our code classifier to exploit. Our evaluation confirms the generalizability of our design: the models trained with Scala code can make accurate predictions in frameworks written in Java and C++ (§7.1).

3.4 Model Validation

We design a model validation test to verify that both the code and documentation contain useful resource information that can be learned correctly by our classifiers. We collect a set of functions from Python standard library and OpenJDK, *manually verify* their resource uses with `strace`, train our classifiers over them, and compute the prediction accuracy. As shown in Table 1, the trained models can identify the dominant resource of a code snippet with accuracy over 77%. The F1 scores[7] are higher than 0.86, suggesting no possible skewness towards any classes. These results show that our classifiers can accurately identify resource uses from the source code. We believe this also holds true for other codebases as we randomly select the source files from standard libraries.

We next inspect what features learned by our classifiers make major contributions to the prediction results. We turn to LIME [64], a popular framework that can explain what words are the key to making a certain prediction. Table 2 lists the most frequent words that appear in the explanations. As one may expect, the computation of "key" or "certificate" is CPU-intensive and "file" operations involve disk I/O. They indicate that our models have indeed learned meaningful patterns for resource inference. We further compare the predictions made separately by the code and documentation classifiers, and have two major findings: (1) the code and the documentation classifiers have roughly the same level of prediction ability, and (2) they make more accurate predictions on *longer* and *more diverse* code and documentation. We defer the detailed analysis to supplemental materials (§A).

Now that we have the function-level resource information,

[4] The collected dataset are released at [69].

[5] 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 codes tend to appear in the same module, so it is feasible to "batch process" them.

[6] The documentation classifier is optimized with stochastic gradient descent (SGD). Its learning rate is set to 0.1 and annealing rate 0.5. For the more complicated code classifier, we choose a more stable optimization algorithm, Limited-memory BFGS, with its step size as 1×10^{-6}.

[7] F1 score is the harmonic mean of the precision and recall, whose value is within 0 and 1. The larger the value is, the more accurate and the less biased the classifier is.

it remains a challenge to build a job-level profile on top of it, which we address in the next section.

4 Profiling Resource Usage of Job

The previous section infers resource-time for a single function, yet a performance-resource model requires the information about each operation (which may call multiple functions) and the whole job. In this section, we piece together the function resource-time into a resource vector for each operation (§4.1), which unifies the varied resource patterns and captures the time proportion spent on each resource. We then assemble the operation information into a job-level DAG execution profile (§4.2), which forms the basis of performance debugging and prediction.

4.1 Resource Vector

We characterize the resource uses of an operation with a *resource vector*, where each component is the *proportion* of runtime it spent on one resource during execution. For example, the resource vector $\vec{R} = (p_{\text{cpu}}, p_{\text{disk}}, p_{\text{net}}) = (0.15, 0.03, 0.82)$ indicates that the operation mainly uses CPU, disk, and network, with each accounting for 15%, 3% and 82% of the operation execution time. We define resource vectors with *relative* time proportions instead of the absolute scale because the former can be more conveniently handled in machine learning. As the proportions add up to one, our definition implicitly assumes *sequential resource use* during execution. Though this may not be exactly the case (e.g., pipelining in Spark), it serves as a good approximation in many frameworks because operations are typically low-level functions performing simple tasks (e.g., math operation in TensorFlow, primitive operation in Flink). Furthermore, for those operations spending the vast majority of time on one resource, which is usually the case, neglecting concurrent uses on the other resources has a negligible impact to runtime prediction. One piece of empirical evidence is Monotasks [54], which rearchitects Spark operators to avoid resource pipelining and merely incurs 9% increase in runtime. Our evaluation also confirms the validity of the sequential resource use model (§7.1).

CrystalPerf infers the resource vector of each operation from its call traces. A stack typically has some system function in the bottom (e.g. `Thread.run`) and the most recently called function at the top. Since the topmost function can be some library or native routine, we scan downwards until we find a function whose source code is available (i.e., some function in the dataflow framework). We then apply the classifiers introduced in §3.2 to infer its resource use, and take the output as its resource vector. As the profiler samples at fixed intervals, we simply assume that the functions in a given stack have remained execution in that interval. We take the average of all the inferred vectors and use it as the resource vector of the operation in that interval.

4.2 DAG-Based Execution Profile

With the resource vector of operations, we establish the execution profile for a dataflow job in two stages. In the first stage, we organize a DAG to represent the job execution, with necessary modification for certain frameworks (details following next). In the second stage, we annotate each node (operation) with its execution details and the resource-time inferred from its call traces.

Dataflow frameworks expose DAG information through monitoring APIs or runtime logs. In most dataflow frameworks, the job DAG provided in the log trace (e.g., TensorFlow computation graphs and Flink dataflows) can be directly used to represent the internal computation structure, i.e., the operations and their dependencies. Yet a special treatment is needed for Spark, in which a job is decomposed into multiple stages, each containing multiple parallel tasks [81]. We therefore turn to a *hierarchical* DAG representation where a node itself can be a smaller DAG. After constructing the DAG, we associate the profiled execution details to each node, including the start and end time of the operation, the functions it called during execution, the source code of those functions, and the inferred resource vector of the operation. Figure 2 shows a sample profile attached to a `join` operation in Spark.

The DAG execution profile forms the basis for debugging performance issues. To identify the execution bottleneck, CrystalPerf follows the operation execution on the critical path and sums up the time spent on different resources. The one that takes the longest time is identified as the *bottleneck resource*. The execution profile can also be used to troubleshoot configuration problems by displaying the resource-time breakdown of each operation. For example, in one of our case studies in §7.5, a machine learning job has spent a significant portion of time on I/O. Even though I/O is not the major bottleneck compared with computation, its overuse still suggests a misconfiguration problem for the training job.

5 Performance-Resource Model

The DAG execution profile established in the previous section can be used to characterize the job behaviors in the past execution. In this section, we take a step forward by predicting how the job would perform in what-if scenarios with different resource configurations. We categorize hardware resources into three classes following the von Neumann architecture and model their impacts to the job performance respectively.

Computing Devices such as CPU, GPU or other emerging processors [21, 39] can affect job performance in two ways. (1) Changing the number of devices (e.g., CPU cores) may

Algorithm 1 Predict job runtime under target computing devices.

```
   – $N, N'$: original and target number of computing devices
   – $s, s'$: original and target computing speed
1: function PREDICTRUNTIME($s, N, s', N'$)
2:     Replay tasks in DAG on $N'$ devices.
3:     $w \leftarrow$ slowdown of the first-wave tasks due to warm-up
4:     Scale first-wave task runtime by $w$          ▷ warm-up effect
5:     for each task in replayed DAG do
6:         $v_{\text{cpu}} \leftarrow$ CPU component of resource vector
7:         $t_{\text{cpu}} \leftarrow v_{\text{cpu}} \times$ task runtime
8:         Scale $t_{\text{cpu}}$ by $s/s'$
9:     Traverse the DAG and compute the updated job runtime
```

Algorithm 2 Predict job runtime under target I/O devices.

```
   – $B, B'$: original and target I/O bandwidth
   – $S_B$: buffer size
   – $o$: overhead of processing buffer for one time
1: function PREDICTRUNTIME($B, B'$)
2:     for each task in job DAG do
3:         $v_{\text{I/O}} \leftarrow$ I/O component of resource vector
4:         $t_{\text{I/O}} \leftarrow v_{\text{I/O}} \times$ task runtime
5:         $D \leftarrow t_{\text{I/O}} \cdot B$                 ▷ estimated data size
6:         $t_p \leftarrow (D/S_B) \cdot o$             ▷ processing overhead
7:         $t_c \leftarrow t_{\text{I/O}} - t_p$               ▷ communication time
8:         Scale $t_c$ by $B/B'$.
9:     Traverse the DAG and compute the updated job runtime.
```

change the degree of parallelism of tasks. (2) Utilizing devices with different processing speeds may change the task runtime. Taking both factors into account, Algorithm 1 estimates the job runtime under a parallelism change from N to N' and a speed change from s to s'. The first part (Line 2-4) reschedules the parallel tasks over N' devices and *simulates* their executions to estimate the runtime.[8] During the simulation, we distinguish the sequential execution from the parallel, following the insight of Amdahl's law [6]. As a concrete example, some Spark tasks are executed in a single thread despite the existence of multiple cores, so we keep it sequential in the simulation. We also consider the *warm-up effect*. That is, the first wave of tasks take longer time to complete than the rest, due to various forms of cache (e.g., the instruction cache of CPU, the data cache in distributed storage systems [10, 24] the code cache in JVM [46], etc.). Therefore, if a non-first-wave task is rescheduled to the beginning, we adjust it by a warm-up factor w and vice versa. After the adjustment, the second part (Line 5-8) of the algorithm rescales the compute time of each task by $^s/_{s'}$ as CPU time is inversely proportional to the speed [30].[9]

Memory plays a passive role in job execution. Once a job is allocated sufficient memory, more allocation will not make it run faster. Conversely, when a job runs short of memory, frequent garbage collections or paging prolongs its execution. We characterize the relationship between memory and runtime with a *reversed roofline model* [77]—a linearly decreasing function followed by a constant lower limit, where 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) and the slope is the ratio between the I/O speed of in-memory and on-disk data access. The intuition behind is that the memory is sufficient as long as it can accommodate all data and the delay is primarily caused by data transfer between memory and disk. We admit that other factors, such as garbage collection, may also affect the runtime, but our evaluation shows that such a simplified model is sufficient to predict the runtime accurately under a wide range of memory changes (§7). We leave it as a future work to model more complicated memory effects on dataflow jobs.

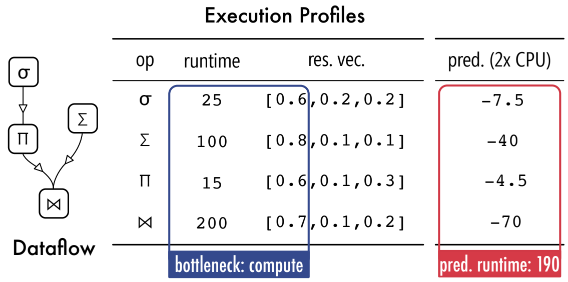
I/O and Network Devices We adopt a buffered I/O model [56] where the data is first placed inside a buffer and then transferred. While other I/O variants do exist, the buffered I/O model remains the most common practice used in Linux API [50] and Java Standard Library [37]. In Algorithm 2, we divide the I/O time into the *buffer processing time* and the *data communication time*. We assume that data are sequentially placed into a buffer and processing a buffer incurs a fixed overhead. The total processing overhead is hence estimated as the number of buffers times the overhead (Line 6). As for the parameters, we set the overhead o the same as the measurement result reported in [58], and buffer size S_B the default values in Linux. The communication time is then rescaled by $^B/_{B'}$ to simulate the change in bandwidth.[10]

Putting It All Together CrystalPerf combines the execution profile with resource-runtime model to characterize job performance. Figure 4 illustrates the main use cases. For debugging, CrystalPerf generates a detailed report of the time spent on each resource to help users understand the potential issues of the program. For prediction, we consider the scenario where a user profiles a job under a given configuration and wonders how the job completion time would change with a different configuration. CrystalPerf makes the prediction by constructing the DAG execution profile and computing the operation runtime in the new configuration following the aforementioned device models. In both use cases, CrystalPerf requires no code instrumentation nor restricts to a certain framework.

[8] Currently, the replay simulation does not consider the task scheduling but follows the original task order. It is feasible for users to provide a customized scheduler as a plugin (§6).

[9] Approximating speed as a constant follows the precedents in Paleo [62] and Starfish [31].

[10] Following prior works [31, 62], we assume constant bandwidth in our modelling.



op	runtime	res. vec.	pred. (2x CPU)
σ	25	[0.6,0.2,0.2]	-7.5
Σ	100	[0.8,0.1,0.1]	-40
Π	15	[0.6,0.1,0.3]	-4.5
⋈	200	[0.7,0.1,0.2]	-70
	bottleneck: compute		pred. runtime: 190

Figure 4: Given an executed job (left), CrystalPerf can either ① analyze resource usage or ② predict its job runtime under another resource configuration (right).

6 Implementation

We have implemented CrystalPerf as a CLI tool in around 4,000 lines of Python and Scala code, which is open-sourced at [69]. Our implementation currently supports three frameworks, Spark, Flink, and TensorFlow.

Pluggable Architecture As a framework-independent tool, CrystalPerf sets to provide a general support for various frameworks, even the future ones. So it has to accommodate different logging conventions, trace formats, DAG representations, etc. We provide plugin mechanisms wherever possible to maximize the usability and generalizability. Our implementation consists of three modules for parsing runtime logs and traces, annotating DAG with resource vectors, and predicting performance based on the given models. The parser accepts plugins that can update with the changing log formats. The annotator supports plugins for locating source code with different directory structure or naming convention. The predictor allows plugins for customized performance models, such as Starfish [31] and Paleo [62]. In our implementation, the supported frameworks (i.e., Spark, Flink and TensorFlow) share the same base modules but all have their specific plugins.

Usage Instructions To use CrystalPerf, users need to enable logging and profiling when executing dataflow jobs. For most JVM-based systems, this can be done by just adding a configuration line and attaching a `jar` package of an off-the-self profiler.[11] Users simply specify the location of the logs and traces in the CLI, and let CrystalPerf extract the runtime information and construct the DAG profile automatically. CrystalPerf provides two modes. (1) For debugging, CrystalPerf outputs the time spent on each resource. (2) To predict the performance under a certain resource configuration, users have to pass that configuration to the CLI, together with the original one. CrystalPerf applies the performance models and outputs the predicted runtime.

7 Evaluation

In this section, we evaluate CrystalPerf in various scenarios. The highlights of our evaluation are summarized as follows:

- CrystalPerf can predict the job runtime of three frameworks under multiple types of resource variations within an average deviation of 13.47% (§7.1).
- CrystalPerf can predict the performance of a Spark benchmark with accuracy comparable to Monotasks [54], the heavily intrusive state-of-the-art operating on a version of Spark that simplifies performance reasoning (§7.2).
- The neural network classifiers are resistant to the inaccuracy in the labels of training dataset and generalizable to different frameworks (§7.4).
- CrystalPerf can effectively help users identify bottleneck resources and address performance issues (§7.5).

7.1 Performance Prediction

Methodology We evaluate CrystalPerf against six workloads in Spark, Flink, and TensorFlow with different resource variations, as summarized in Table 3. For each workload, we first run it under a baseline configuration and measure its runtime. We then change the resource configuration and use CrystalPerf to predict the new runtime. To measure the accuracy, we rerun the job on a real cluster with the same configuration as in prediction. We report the predicted and the actual runtime in the experimental results. We use *deviation* as a metric to measure accuracy, defined as $|p-a|/a$, where a is the actual runtime and p the prediction.

Spark Figure 5 shows the runtime predictions given by CrystalPerf for two TPC-H queries (see Table 3) under various configurations, where the second bar group in the central graph shows the baseline execution. The average deviation of the predictions is 13.49% ± 8.57%, demonstrating the effectiveness of CrystalPerf for Spark applications. Note that such deviations are evenly distributed around the actual runtime, an indication that our approach shows no systematic over- or underestimation but mainly suffers from random errors.

Flink For stream processing, we choose two widely used benchmarks [34, 73]. As CrystalPerf does not natively support long-running jobs, we define the *latency* of a streaming application as its runtime because it measures the time of a data batch (i.e., a data buffer in Flink) traversing through the DAG. Therefore, throughout the experimentation of Flink, we run the application for 10 minutes and report the average latency as its runtime. The resource variation includes the change of CPU share and network bandwidth, where CPU share determines the fraction of CPU time granted to a given program.

Figure 6 presents the prediction results of Flink applications, where the average deviation is 12.70% ± 10.11%.

[11] For frameworks such as TensorFlow and MXNet, the log traces already contain the operator names, and there is no need to specify a profiler.

Table 3: Summary of the workloads used in performance prediction (§7.1).

Framework	Workload	# Instances	Instance Type	Varied Resources
Spark v2.4.3	TPC-H [60] query 6 (short) and 9 (long) with scale factor 100	16	AWS `m5.xlarge` (4 cores, 16GB mem, 10Gbps)	# cores, memory, network bandwidth
Flink v1.7.2	Yahoo Streaming Benchmark [13] and Dhalion Benchmark [20]	11 (1 master and 10 workers)	AWS `m5.xlarge`	CPU share, network bandwidth
TensorFlow v1.13	ResNet [29] and VGG [66] on a flower image dataset	8 (1 master, 4 workers and 3 servers)	GCP `n1-16-standard` (16 CPUs, 60GB mem, Tesla P100 GPU)	computing devices, # cores, memory

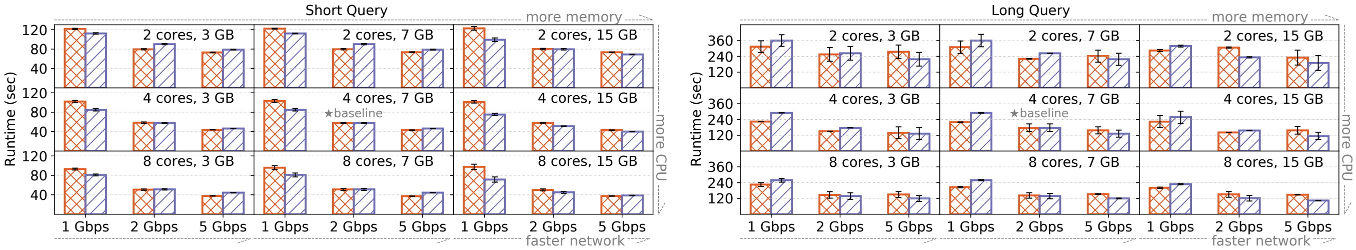


Figure 5: Given the baseline execution (located at the center) of two TPC-H queries, CrystalPerf predicts their runtime under the rest 26 resource configurations. Each bar pair shows one prediction and its corresponding actual runtime (▨: Prediction, ☒: Actual).

Again, the deviations are evenly distributed instead of biasing towards over- or underestimation. Moreover, as our classifiers are trained with Scala code (§3.3) whereas Flink is mainly written in Java, the results demonstrate the generalizability of the classification models to other programming languages. Such desirable property results from the fact that programmers tend to use meaningful names [35] and hence those resource-related terminologies are used in different languages.

TensorFlow As for distributed machine learning, we use a managed version of TensorFlow (v1.13) provided by AI Platform [15] in Google Cloud. The training programs adopt the parameter server architecture [45] with asynchronous parallel (ASP) scheme. Because the runtime characteristics of parallel machine learning are highly repetitive across iterations, we report the *average iteration time* as the job completion time. As the iterations in ASP may not be temporally aligned, we take the average of each worker's iteration time. The resource variation in learning applications include upgrading CPU to GPU, using a different number of virtual CPUs and changing the memory size. Figure 7 shows the prediction results given by CrystalPerf. The average deviation is 14.22% ± 11.77%. This again confirms the satisfactory prediction accuracy of our approach and the generalizability of our classification models.

7.2 Comparison with Monotasks

As summarized in Figure 1, Monotasks [54] is the most closely related to our work, so we make a comparison to it by predicting the performance of Big Data Benchmark (BDB) [7]. Note that Monotasks has devised its own variant of Spark for performance clarity called *MonoSpark*, which decomposes the pipelined usage of multiple resources so that each compute task only uses one resource. Following the description in §6.2 in [54], we reproduce the experiment in the same settings: first run the BDB workloads on MonoSpark with two disks and let CrystalPerf predict the runtime under the configuration of one disk. As most of MonoSpark's implementation relies on Spark (including logging), CrystalPerf can reuse its Spark interface.

Figure 8 plots the actual and the predicted runtimes of all queries in the benchmark. CrystalPerf achieves an average deviation of 12.53% ± 5.10%, while Monotasks reports most errors within 9% and an exception as high as 28% (cf. Figure 12 in [54]). It is worth mentioning that Monotasks uses a whitebox performance model that is specifically tailored to the decomposed design of MonoSpark. In comparison, CrystalPerf requires no such tailoring but still can capture the resource pattern of MonoSpark. This once again confirms that (1) CrystalPerf has comparable prediction ability with the intrusive state of the art, and (2) its effectiveness is not restricted to a particular framework.

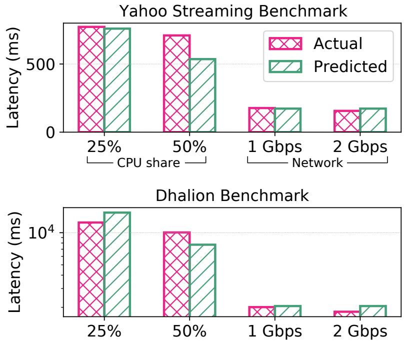


Figure 6: Performance prediction of Flink workloads. The baseline machine uses 100% CPU share and 5Gbps network.

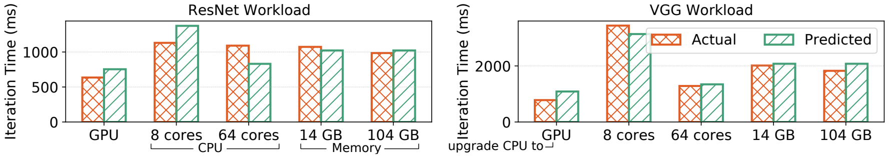


Figure 7: Performance prediction of TensorFlow workloads. The baseline machine uses 16 virtual CPU cores and 60GB memory.

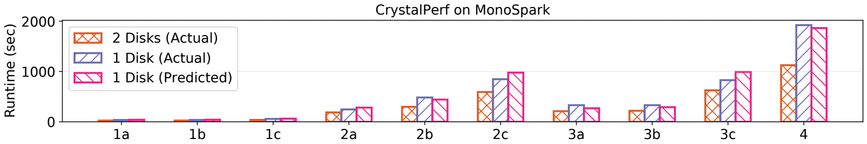


Figure 8: Performance prediction of Big Data Benchmark (BDB) on MonoSpark.

7.3 Microbenchmarking Performance Models

Having shown the predicting ability of CrystalPerf, we need to confirm that the constructed performance models match the real execution. We scrutinize the Dhalion Benchmark running with 100% CPU share (baseline) and 25% (prediction). We choose this case because Flink does not have complicated pipelining so that we could easily check the real resource usage based on the profiling traces. It is worth mentioning that the traces also include the warm-up period (i.e., Java JIT compilation), so we exclude them from the traces and only take samples after the system enters a stable state. The benchmark mainly has two operators, `FlatMap` and `Window`. From the latency measurements, we find that they respectively incur 1583.63ms and 2322.50ms latency in the 100% case, and 2961.75ms and 7181.63ms in 25%.

Resource Vector We first verify the CPU entries in the resource vectors of both operators, because they are the key components in prediction. CrystalPerf infers that `FlatMap` and `Window` respectively spend 15.13% and 85.05% of the operation time on computation, with an average percentage of 56.70%.[12] We manually examine the profiling traces of the baseline case and find the most dominant computation functions are `StreamFlatMap.processElement` and `WindowOperator.processElement`. They appear in 65.46% of the call stack samples, which is close to the inferred percentage of CPU usage.

Performance Model Following our resource-time models (§5), CrystalPerf predicts that `FlatMap` and `Window` will take 2296.26ms and 8244.88ms, respectively, if running with 25% CPU share. Again, they are within a tolerable error range when compared with the measured durations. Furthermore, we observe that the occurrences of `NioEventLoop.run`, a network-related function, increases from 3.20% of the samples to 4.10%. This shows how the slowdown of CPU could prolong the processing time of network I/O, confirming our intuition used in modeling resource changes.

7.4 Classifier Sensitivity and Generalizability

Sensitivity to Labeling Quality Recall in §6 that the resource vectors of the low-level functions we collected in the training dataset are manually labeled, and it is difficult to check their accuracy due to the lack of the ground truth. Henceforth, we instead evaluate the robustness of CrystalPerf with respect to the labeling quality. We reuse the training dataset and perturb the labels of 10%/20% data samples. For instance, for the data samples originally labeled as CPU-dominant, we change 1.67% of them to disk and another 1.67% to network, and repeat the process for other two resources to reach the 10% fraction. As shown in Figure 9, the resource classifiers used in CrystalPerf are robust against the labeling errors.

Generalizability Apart from robustness, generalizability is another desirable property because the users don't want to retrain the classifiers for each single framework. Although §7.1 confirms that CrystalPerf can work on different systems with the same classifiers, here we zoom in on the predictions they make on various programming languages. For the test set, we collect 105 low-level functions from Spark (Scala), 194 from Flink (Java), 302 from TensorFlow (Python), and label them in the same way as in §3.3. The prediction results shown in Table 4 indicates that our classifiers, trained on low-level Scala code, predict similarly well across three frameworks, which partially explains the generalizability of CrystalPerf. One noteworthy detail is that sometimes TensorFlow has even higher accuracy than Spark. This is because the tested TensorFlow functions include some example code where they are lengthily documented and thus appear more

[12] It is computed as a weighted average: $(15.13\% \times 1583.63 + 85.05\% \times 2322.50) \div (1583.63 + 2322.50) = 56.70\%$

Table 4: The classifiers trained with Scala code perform similarly on the chosen systems, implying the generalizability of our approach. (Accuracy/F1 Score)

	Code Cls.	Doc. Cls.
Spark	81.0%/.797	76.5%/.673
Flink	73.3%/.664	74.8%/.589
TensorFlow	79.9%/.752	77.8%/.652

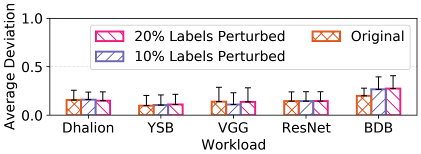


Figure 9: We perturb a portion of the labels used in training two classification models and rerun the prediction experiments on previous workloads. The error bars are *clipped* to show only the upper halves.

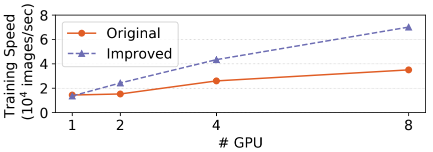


Figure 10: When training models, the user reported the poor multi-GPU scalability in the *original* setting. With CrystalPerf pinpointing the bottleneck as I/O, the user could significantly *improve* it by adopting optimized I/O library.

informative to the classifier model.

7.5 CrystalPerf in Action: Case Studies

We further conduct three real-world case studies to demonstrate how CrystalPerf could help users identify resource bottleneck and address performance issues with ease.

Diagnosing Slow Operation A user implemented a Spark program to process a dataset stored in HDFS [8]. However, the program took much longer time then she had expected. We investigated the case by reproducing it with the same configuration as original. We launched a six-node cluster on AWS EC2 with `m5.4xlarge` instances (16 cores, 64 GB memory, 10 Gbps network), generated dummy data with Sqoop and MySQL, and allocated 5 Spark executors with 3 cores and 15 GB memory each to run the job.

We analyzed the runtime logs using CrystalPerf, and it suggests that CPU is the bottleneck resource. We looked into the DAG execution profile and discovered that most CPU entries in the task resource vectors were around 0.785, indicating that CPU had taken the majority of runtime. For further validation, we inspected the profiling traces and the two most frequently called functions were `Text.decode` and `BuiltinGzipDecompressor.decompress`, which appeared in 41.18% and 35.29% of sample stacks. Knowing the CPU dominance of the program, the user could simply allocate more CPUs to accelerate it. According to our measurement, doubling the number of CPU cores reduces the job runtime from 25.89s to 14.76s.

Diagnosing Slow Iteration We then turn to an issue [52] from TensorFlow community, where a user trained a neural network over the MNIST [44] dataset. The user already deployed GPUs but still found the iteration time much longer than she had expected. We reproduced the problem on AWS EC2 using a `p2.16xlarge` instance (16 NVIDIA K80 GPUs, 64 vCPUs, and 732 GB of memory). After taking in the runtime traces, CrystalPerf suggests that I/O sustains for the longest in the application. We verified the conclusion by manually examining the profiling traces and observed that an operator named `QueueDequeueManyV2` (managing queued data in I/O) occupied the majority of iteration time. Such I/O dominance is rare in model training because computation is usually the most time-consuming part, which indicates that the data transfer into and from GPU could be problematic. So we changed GPU to CPU and reran the program. This time, the iteration time dropped from 2 seconds to 80 milliseconds, and the `QueueDequeueManyV2` no longer dominated the traces.

Diagnosing Poor Scalability Next we turn to a question [61] raised in StackOverflow, where a user trained a neural network over the CIFAR-10 [41] dataset. When she increased the number of GPUs, she noticed that the scalability of TensorFlow in the multi-GPU setting was rather poor. We reproduced the user's problem on AWS EC2, using `p3.16xlarge` instances (8 Tesla V100 GPUs, 64 vCPUs and 488 GB of memory). After analyzing the traces, CrystalPerf reports that the program is actually I/O bound. Again, we verified this conclusion by checking the traces and noticing that operation `MEMCPYHtoD` (copying data from CPU to GPU) took a significant portion of time. As a remedy, we enabled NCCL [53], an optimized library for inter-GPU I/O, and reran the program. Figure 10 shows the increased scalability from the *original* setting to *improved*, demonstrating the effectiveness of CrystalPerf in addressing performance issues.

8 Discussion

Profiling Overhead As runtime logs are enabled by default in many frameworks, the overhead of CrystalPerf mainly comes from the sampling profiler, which we set to sample once per 100ms. We measure Spark applications with and without the profiler and their runtimes differ within 2%, lower than the overhead of instrumentation in SnailTrail [34] (10%). Furthermore, if the framework provides the called functions (e.g., the built-in profiler in TensorFlow), CrystalPerf does not even require profiling.

Extending to New Frameworks In §6 we have introduced the pluggable architecture of our CLI tool, where multiple frameworks can share some common modules such as resource inference and basic performance models. When extending it to a new framework, the user has to provide the plugins that parse the generated runtime logs and construct the execution profile. From our experience, it does not involve much labor work. For instance, after implementing the prototype for Spark and TensorFlow, we extend CrystalPerf to Flink with only 172 lines of code.

Advantage over Existing Works As a major improvement, CrystalPerf models dataflow jobs as general DAGs instead of domain-specific structures. For instance, Paleo [62], a performance model for deep learning, is derived from the layer operations of GPUs; Ernest [74] models machine learning pipelines with the common communication patterns specific to machine learning; the stage-by-stage model in Starfish [31] exactly matches the execution of MapReduce jobs. Whereas their model structures limit their applicability scope, CrystalPerf targets a broader array of dataflow computations (e.g., data analytics, stream processing, ML) and exploits commonly available information such as the source code, the execution trace, and the job DAG.

Limitations in Streaming Scenarios Although CrystalPerf is applicable to Flink (§7.1), its ability is restricted by several properties of streaming applications. First, the operators in stream processing can scale up or down as the incoming data fluctuates [40]. Such dynamic parallelism changes the underlying computation graph and thus requires CrystalPerf to update its model accordingly. Second, the processing latency varies with the arrival rate of data [16] as a larger data batch takes a longer time to process. CrystalPerf does not include the rate information for generality. If the users would like to monitor the performance more precisely, we recommend them to record the input rate of data streams and augment the analysis given by CrystalPerf.

9 Conclusion

In this paper, we have presented CrystalPerf, an instrumentation-free, framework-independent approach to performance debugging and reasoning for dataflow computations. For each job, CrystalPerf constructs a DAG execution profile, and infers the resource usage of each operation node from its code and documentation with two machine learning classifiers. We have implemented CrystalPerf as a CLI tool supporting three mainstream frameworks and evaluated it with multiple workloads and use cases. The results show that CrystalPerf can accurately predict the job runtimes under various resource changes and effectively address performance issues.

Acknowledgment

We would like to thank our shepherd, Vasiliki Kalavri, and the anonymous reviewers for their valuable feedback that helps improve the quality of this work. This research was supported in part by RGC GRF grant 16213120 and ECS grant 26213818. Huangshi Tian and Minchen Yu were supported in part by the Hong Kong PhD Fellowship Scheme and the Huawei PhD Fellowship Scheme, respectively.

References

[1] Tensorflow benchmarks. https://github.com/tensorflow/benchmarks, 2020.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.

[3] Alan Akbik, Duncan Blythe, and Roland Vollgraf. Contextual string embeddings for sequence labeling. In *COLING*, 2018.

[4] akka. Apache Akka. https://akka.io, 2019.

[5] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, et al. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *NSDI*, 2017.

[6] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967.

[7] UC Berkeley AMPLab. Big data benchmark, 2014. https://amplab.cs.berkeley.edu/benchmark/.

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

[9] Walter Binder. Portable and accurate sampling profiling for java. *Software: Practice and Experience*, 2006.

[10] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, et al. Windows azure storage: A highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[11] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, et al. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2015.

[12] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, et al. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *Neural Information Processing Systems, Workshop on Machine Learning Systems*, 2015.

[13] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, et al. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, 2016.

[14] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, et al. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *EMNLP*, 2014.

[15] Google Cloud. Ai platform, 2019. https://cloud.google.com/ai-platform/.

[16] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. Adaptive stream processing using dynamic batch sizing. In *SoCC*, 2014.

[17] Netty Developers. Netty project. https://github.com/netty/netty, 2019.

[18] Parquet Developers. Parquet mr. https://github.com/apache/parquet-mr, 2019.

[19] Danyel Fisher, Rob DeLine, Mary Czerwinski, and Steven Drucker. Interactions with big data analytics. *INTERACTIONS*, 19(3), 2012.

[20] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. Dhalion: self-regulating stream processing in heron. *VLDB*, 2017.

[21] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, et al. A configurable cloud-scale dnn processor for real-time ai. In *ISCA*, 2018.

[22] functional-streams-for scala. Fs2: Functional streams for scala. https://github.com/functional-streams-for-scala/fs2, 2019.

[23] Felix Geisendörfer. fgprof - the full go profiler. https://github.com/felixge/fgprof, 2020.

[24] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP*, 2003.

[25] Corvios GmbH. Nprofiler. https://www.nprofiler.com/, 2015.

[26] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *OSDI*, 2016.

[27] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *OSDI*, 2016.

[28] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, et al. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Trans. Storage*, 14(3), 2018.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[30] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

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

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

[33] Herodotos Herodotou, Fei Dong, and Shivnath Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *SoCC*, 2011.

[34] Moritz Hoffmann, Andrea Lattuada, John Liagouris, Vasiliki Kalavri, et al. Snailtrail: Generalizing critical paths for online analysis of distributed dataflows. In *NSDI*, 2018.

[35] Einar W. Høst and Bjarte M. Østvold. Debugging method names. In *ECOOP*, 2009.

[36] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, et al. Gray failure: The achilles heel of cloud-scale systems. In *HotOS*, 2017.

[37] API Specification Java Platform, Standard Edition 7. Class standardsocketoptions. https://docs.oracle.com/javase/7/docs/api/java/net/StandardSocketOptions.html, 2020.

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

[39] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, et al. In-datacenter performance analysis of a tensor processing unit. In *ISCA*, 2017.

[40] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, et al. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *OSDI*, 2018.

[41] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[42] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, et al. Twitter heron: Stream processing at scale. In *SIGMOD*, 2015.

[43] Palden Lama and Xiaobo Zhou. Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud. In *ICAC*, 2012.

[44] Yann LeCun. The mnist database of handwritten digits, 1998. http://yann.lecun.com/exdb/mnist/.

[45] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, et al. Scaling distributed machine learning with the parameter server. In *OSDI*, 2014.

[46] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. Don't get caught in the cold, warm-up your JVM: Understand and eliminate JVM warm-up overhead in data-parallel systems. In *OSDI*, 2016.

[47] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, et al. Distributed graphlab: A framework for machine learning and data mining in the cloud. In *VLDB*, 2012.

[48] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, et al. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.

[49] Linux Programmer's Manual. Proc filesystem. http://man7.org/linux/man-pages/man5/proc.5.html, 2020.

[50] Linux Programmer's Manual. Tcp protocol. http://man7.org/linux/man-pages/man7/tcp.7.html, 2020.

[51] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *SIGCOMM*. 2019.

[52] nryant. reading_data/fully_connected_reader.py very slow relative to fully_connected_feed.py. https://bit.ly/2kiT2dD, 2016.

[53] Nvidia. Nccl library, 2016. https://github.com/NVIDIA/nccl.

[54] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *SOSP*, 2017.

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

[56] Vivek S Pai, Peter Druschel, and Willy Zwaenepoel. Io-lite: A unified i/o buffering and caching system. In *OSDI*, 1999.

[57] Adam Paszke, S. Gross, Francisco Massa, A. Lerer, James Bradbury, et al. Pytorch: An imperative style, high-performance deep learning library. 2019.

[58] Simon Peter, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system as control plane. In *OSDI*, 2013.

[59] playframework. Play framework. https://www.playframework.com, 2019.

[60] Meikel Poess and Chris Floyd. New tpc benchmarks for decision support and web commerce. *ACM Sigmod Record*, 2000.

[61] A. J. Polk. Scaling performance across multi gpus. https://bit.ly/2kCMwif, 2016.

[62] Hang Qi, Evan R Sparks, and Ameet Talwalkar. Paleo: A performance model for deep neural networks. In *ICLR*, 2017.

[63] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, 30(4):65–79, 2010.

[64] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Why should i trust you?: Explaining the predictions of any classifier. In *KDD*, 2016.

[65] scalaz. Scalaz. https://scalaz.github.io/7/, 2019.

[66] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

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

[68] V8 Developer Team. Using v8s sample-based profiler. https://v8.dev/docs/profile, 2020.

[69] Huangshi Tian. Crystalperf. https://github.com/All-less/crystalperf, 2021.

[70] Huangshi Tian, Minchen Yu, and Wei Wang. Supplemental materials to crystalperf. https://www.cse.ust.hk/~weiwa/papers/crystalperf-supp.pdf, 2021.

[71] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, et al. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *EuroSys*, 2016.

[72] twitter. Twitter util. https://github.com/twitter/util, 2019.

[73] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, et al. Drizzle: Fast and adaptable stream processing at scale. In *SOSP*, 2017.

[74] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *NSDI*, 2016.

[75] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Aria: Automatic resource inference and allocation for mapreduce environments. In *ICAC*, 2011.

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

[77] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communication of the ACM*, 2009.

[78] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. Hierarchical attention networks for document classification. In *NAACL*, 2016.

[79] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, et al. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[80] Zhibin Yu, Zhendong Bei, and Xuehai Qian. Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing. In *ASPLOS*, 2018.

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

[82] Ce Zhang, Wentao Wu, and Tian Li. 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*, 2017.

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

[84] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. An empirical study on tensorflow program bugs. In *ISSTA*, 2018.