Is advance knowledge of flow sizes a plausible assumption?

Vojislav Đukić[1], *Sangeetha Abdu Jyothi*[2], *Bojan Karlaš*[1],
Muhsen Owaida[1], *Ce Zhang*[1], *Ankit Singla*[1]
[1]ETH Zurich [2]University of Illinois at Urbana–Champaign

Abstract

Recent research has proposed several packet, flow, and coflow scheduling methods that could substantially improve data center network performance. Most of this work assumes advance knowledge of flow sizes. However, the lack of a clear path to obtaining such knowledge has also prompted some work on non-clairvoyant scheduling, albeit with more limited performance benefits.

We thus investigate whether flow sizes can be known in advance in practice, using both simple heuristics and learning methods. Our systematic and substantial efforts for estimating flow sizes indicate, unfortunately, that such knowledge is likely hard to obtain with high confidence across many settings of practical interest. Nevertheless, our prognosis is ultimately more positive: even simple heuristics can help estimate flow sizes for many flows, and this partial knowledge has utility in scheduling.

These results indicate that a presumed lack of advance knowledge of flow sizes is not necessarily prohibitive for highly efficient scheduling, and suggest further exploration in two directions: (a) scheduling under partial knowledge; and (b) evaluating the practical payoff and expense of obtaining more knowledge.

1 Introduction

Advance knowledge of future events in a dynamic system can often be leveraged to improve the system's performance and efficiency. In data center networks, such knowledge could potentially benefit many problems, including routing and flow scheduling, circuit switching, packet scheduling in switch queues, and transport protocols. Indeed, past work on each of the above topics has explored this, and in some cases, claimed significant improvements [34, 21, 5, 4, 30].

Nevertheless, little of this work has achieved deployment. Modern deployments largely use techniques that do not depend on knowing traffic features in advance, such as shortest path routing with randomization, and first-in-first-out queueing. A significant barrier to the adoption of traffic-aware scheduling is that in practice, traffic features can be difficult to ascertain in a timely fashion with adequate accuracy.

We focus on the plausibility and utility of obtaining *flow size* information *a priori* for use in packet, flow, and coflow scheduling in data centers. We explore this problem in the context of four scheduling techniques from past work: pFabric [5], pHost [21], FastPass [34], and Sincronia [4]. Each of these is a clairvoyant scheduler, with advance knowledge of the size of each flow at its start (but not necessarily the flow arrival times). For some problems, non-clairvoyant algorithms are also known, *e.g.,* PIAS [7] for packet scheduling, and Aalo [13] for coflow scheduling. While such techniques outperform FIFO and fair-sharing baselines, there is still a substantial performance gap compared to clairvoyant algorithms (§3.2). Further, it is unclear if similar non-clairvoyant methods can be developed for scheduling problems such as FastPass [34], where absolute flow sizes are needed, rather than just the relative priorities leveraged by PIAS and Aalo.

We thus examine a wide array of possibilities for estimating flow sizes in advance, including modifications to the application-system interface for explicit signaling by the application, as well as more broadly applicable application-agnostic methods, ranging from simple heuristics like reading buffer occupancy and monitoring system calls, to sophisticated machine learning. We analyze the complexity, accuracy, and timeliness of different approaches, and the utility of the (often imprecise) flow size information gleaned by these methods across our four example scheduling techniques.

We find that even simple heuristics effectively estimate flow sizes for a large fraction of flows in many settings. But accurate estimation for *all* flows is likely intractable: for many scenarios of practical interest, each of the estimation approaches under consideration has limitations that prevent accurate flow size estimation. Superficially, this can be seen as a negative result for clairvoyant schedulers. However, this does not necessarily restrict us to non-clairvoyant scheduling — as recent work shows [9, 40], partial knowledge of flow sizes, coupled with heuristics to tackle the unknown flow sizes, can often provide an effective compromise. We pose a novel question by intersecting this past work with our results on the effectiveness and complexity of different methods of flow size estimation: how does investment in increasing the coverage of flow size estimation pay off? To the best of our knowledge, no prior work has tackled this issue — as we invest greater effort in flow size estimation, how does scheduling performance change?

We show, empirically and analytically, that for packet scheduling at switches, a simple approach[1] incentivizes greater efforts in estimating flow sizes. While this may seem obvious, somewhat surprisingly, we find that adding more flow size estimates does not always improve performance — for coflow scheduling, an intuitive scheduling scheme for

[1]A simplification of Karuna [9] replacing discrete thresholds for priority queues with continuously degrading flow priority as more bytes are sent.

partial knowledge settings sometimes sees increased coflow completion times as more flow sizes are made known.

These first results call for more exploration in two directions: (a) schedulers explicitly designed for partial knowledge settings; and (b) the scheduling efficiency benefits of greater investments in learning flow sizes.

We believe this to be the first in-depth critical analysis of assumptions of (non-) clairvoyance in flow, coflow, and packet scheduling, together with considering the utility of partial knowledge, and various vectors for increasing this knowledge. We make the following contributions:

- We design FLUX, a framework for estimating flow sizes in advance, and tagging packets with this information.
- Using FLUX, we evaluate both simple heuristics using system calls and buffer sizes, as well as learning methods; identify which factors are effective in flow size estimation; and explain how these depend on applications.
- We implement FLUX in Linux[2], and demonstrate that even the learning method implemented incurs only small computational and latency overheads, which can be further reduced using specialized hardware.
- For each method of flow estimation, we also explore its limitations, concluding that for many practical scenarios, such estimation will remain challenging.
- We evaluate the utility of inferred (often imprecise) flow sizes across four scheduling techniques, finding significant benefits compared to flow-unaware scheduling.
- We analyze settings with some fraction of traffic permitting flow estimation, and show that the impact of increasing this fraction is not always positive.
- In a simplified model, we prove that for shortest remaining first packet scheduling [5], coupled with a simple heuristic for handling unknown flows, adding a flow's size cannot worsen its completion time.

2 Background & motivation

Many scheduling techniques for data center networks have been proposed, promising substantial performance gains:

- PDQ [22] and D^3 [43] schedule flows across the fabric, in a "shortest flow first" manner.
- pFabric [5] and EPN [29] schedule packets at switch queues using "least remaining flow" prioritization.
- pHost [21], Homa [30], and FastPass [34] schedule sets of packets across the network.
- Orchestra [14], Varys [15], Sincronia [4], and Baraat [18] schedule coflows (app-level aggregates).
- c-Through [41], Helios [20], and several followup proposals schedule flows along time-varying circuits.

[2]Code and traces here: https://github.com/vojislavdjukic/flux.

All of these proposals are clairvoyant schedulers, *i.e.,* they assume that the size of a flow is known when it starts. Some of this work has made this assumption explicit:

> *"In many data center applications flow sizes or deadlines are known at initiation time and can be conveyed to the network stack (e.g., through a socket API) ..."*
>
> — Alizadeh et al. [5], 2013

> *"The sender must specify the size of a message when presenting its first byte to the transport ..."*
>
> — Montazeri et al. [30], 2018

There is not, however, consensus on the availability of such information; work in the years intervening the above two proposals has argued the opposite. For instance:

> *"... for many applications, such information is difficult to obtain, and may even be unavailable."*
>
> — Bai et al. [7], 2015

Thus, we explore this question in depth: Is advance knowledge of flow sizes a plausible assumption? Further, what happens when only information for some flows is available? We design a framework for estimating flow size information and evaluate its utility for four example scheduling techniques in past work that depend on this information. We include here brief, simplified background on these techniques:

pFabric [5]: Each packet is tagged with a priority at the end-host, based on the remaining flow size. Switches then schedule packets in order of least remaining size. This results in near-optimal packet scheduling and can improve average flow completion time (FCT) by as much as 4× for certain workloads, compared to the oblivious FIFO scheme.

pHost [21]: pHost uses distributed scheduling, with the source sending to the destination a "Request To Send" message carrying the number of pending bytes in the flow. The destination clears transmission for the flow with the least bytes. pHost claims an average FCT reduction of 3×.

FastPass [34]: FastPass uses a centralized arbiter to schedule flows. When a host wants to send data, it asks the arbiter to assign it a data transmission time slot and path. The arbiter tries to make a decision based on the traffic demand (flow size) of all active flows. FastPass claims "near-zero queuing" on heavily loaded networks, cutting network RTT by 15×.

Sincronia [4]: Sincronia orders coflows using sizes of individual flows to find network bottlenecks and uses this ordering for priority scheduling. It achieves average coflow completion time (CCT) within 4× of the optimal.

If flow sizes were known *a priori*, such techniques could improve various performance metrics of interest by 3-15× compared to size-unaware ECMP-plus-FIFO scheduling. For some scheduling problems, where only relative flow priorities matter rather than absolute sizes, prior work has developed non-clairvoyant schedulers [7, 45, 13] that also beat the ECMP-FIFO baseline. But as we shall see later, their per-

formance improvements are often much more modest than clairvoyant schedulers. Thus, addressing the question of flow size estimation remains an interesting open problem.

3 Flow size estimation: design space

Before considering flow size estimation, it is necessary to define a flow. The primary goal of flow size estimation is to improve application performance using network scheduling. Where application messages directly translate to individual TCP connections, using TCP 5-tuples to define flows suffices. However, to avoid the overheads of connection setup and TCP slow start, it is common practice in many data centers to use persistent long-lived TCP connections (*e.g.,* between Web servers and caches) which carry multiple application messages over time. In these settings, it may be more appropriate to consider instead a series of packets between two hosts that are separated from other packet series by a suitable time gap. We note that this is an imprecise method, as system-level variability and workload effects can impair such identification of flows. For instance, multiple small cache responses sent out in quick succession could be mistakenly identified as one flow. This limitation applies to all application-oblivious methods — in some scenarios, mechanisms to identify packets that form an application-level message are inherently bound to be imprecise.

We next describe several approaches for obtaining flow sizes, and intuitively reason about their efficacy in various settings. Experimental evaluations of the quality of estimation and its impact on network scheduling are deferred to §6.

3.1 Exact sizes provided by application

Many applications can assess how much data they want to send, such as standard Web servers, RPC libraries, and file servers. Modifying such applications to notify the network of a message's size is thus plausible. This would require a new interface between applications and the network stack, and is clearly doable, but not trivial. The replacement must be interruptible, like `send` in Linux, and it's unclear how best to implement this – what happens when it gets interrupted after sending some bytes? When a new call is made to finish the transfer, how do we decide whether or not this is the same flow? Thus, this may also require introducing some notion of flow identifiers. While this can surely be done, we merely point out that it requires care.

Limitations: As discussed in some depth in prior work [7], there are several scenarios where the application itself is unaware of the final flow size when it starts sending out data, such as for chunked HTTP transfers, streaming applications, and database queries where partial results can be sent out before completion. Also, apart from introducing changes to the host network stack (which are not necessarily prohibitive for private data centers), this approach requires modifying a large number of applications to use the new API. In settings like the public cloud, this may not be feasible.

Still, this approach should not be casually dismissed; a few software packages dominate the vast majority of deployed applications, *e.g.,* a large fraction of Web servers use one of the three most popular server software packages, most data analytics tasks use one of a small number of leading frameworks, etc. Past work (*e.g.,* FlowProphet [42] and HadoopWatch [33]) has in fact explored the use framework-internals for gleaning flow sizes. Thus, this approach could make flow size information accessible to the network for many applications, provided the right APIs are developed.

3.2 Flow aging

A set of application-agnostic techniques have been proposed around the idea of using the number of bytes a flow has already sent as an estimator for its pending data. For instance, Least Attained Service [35] gives the highest priority to new flows and then decreases their priority as they send more data. Thus, flow priorities "age" over time. PIAS [7] explores a variant of this approach, coupling it with the use of a small number of discrete queues to fit commodity switches. Aalo [13] applies similar ideas to coflow scheduling.

Limitations: The most significant drawback is that this approach may not benefit scheduling techniques that require absolute flow sizes (as opposed to only relative priorities), such as Sincronia[3], FastPass and optical circuit scheduling. Even where applicable, the effectiveness of such methods depends on flow size distribution. For instance, LAS does not work well when there are a large number of flows of similar size. In the limiting case, if all flows are the same size, older flows nearer to completion are deprioritized, which is the opposite of the desired scheduling. More sophisticated methods based on multi-level feedback queues [7] still depend on estimating a *stable* underlying flow size distribution[4]. Further, even in favorable settings, with stable heavy-tailed flow size distributions, the performance of such application-agnostic techniques can be substantially lower than clairvoyant ones. For instance, recent work [30] reports $\sim 2\times$ difference in 99^{th} percentile slowdown between PIAS [7] and pFabric [5]. Similarly, Sincronia [4], the best-known clairvoyant coflow scheduler, claims a 2-8$\times$ advantage over Varys, and by extension, over CODA [45], the best-known non-clairvoyant coflow scheduler. (Note however, that the scheduling knowledge involved in CODA is not limited to flow sizes, but also classification of flows into coflows.)

3.3 TCP buffer occupancy

The occupancy of the TCP send buffer at the sending host can provide approximate information on flow sizes. When the buffer occupancy is small, the number of packets in the buffer may be the actual flow size of a small flow. When the

[3] Sincronia uses only relative priorities in the network, but for assigning these priorities, it computes the bottleneck port using sizes of all flows destined to each port. It is unclear if aging would be effective here.

[4] Alternatively, additional effort must be spent in continuously monitoring and following the changes in the underlying distribution [10].

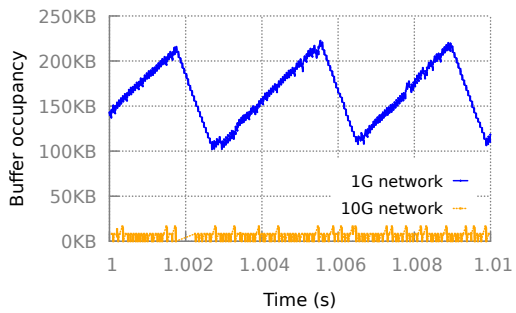


Figure 1: *Buffer occupancy while transferring a 1GB static file from the hard disk over 1G and 10G connections. We show a representative* 10 *ms segment of the trace starting at* 1 *second.*

buffer is fully occupied, *i.e.,* its draining rate is less than its filling rate, the flow may be categorized as a large flow. Mahout [17] and c-Through [41] used roughly this approach. ADS [31] also suggests (but does not evaluate) a similar mechanism, although it is unclear whether it uses system calls, or buffer occupancy, or both.

Limitations: Buffering reflects flow size only when the sender is network or destination limited. If the bottleneck is elsewhere, the buffer may not be filled even by a large flow. Consider a program that reads a large file from the disk and sends it over the network. The program reads data in chunks of a certain size (*e.g.,* 100 KB) and sends as follows:

```
while(...):
  read(file_desc, read_buffer, 100KB)
  write(socket_desc, read_buffer, 100KB)
```

Today's SSDs achieve a read throughput of ∼6 Gbps, while NIC bandwidths of 10-40 Gbps are common. This disparity implies that the buffer may remain sparsely populated most of the time. To illustrate this behavior, we ran a simple experiment. We transfer a 1GB static file served by a Web server over 1G and 10G connections (Fig.1). The file is stored on a regular 7200 RPM hard drive with the maximum read speed of ∼1 Gbps. We see that for the faster connection, the buffer is almost empty. For slow connections, the buffer indicates the lower bound of the flow size, but when the buffer occupancy starts decreasing, it is unclear if that means that no additional data will be added. This presents a serious obstacle for flow size estimation.

3.4 Monitoring system calls

The `write/send` system calls from an application to the kernel provide information on the amount of data the application wants to send. The flow size is typically greater or equal to the number of bytes in the first system call of a flow. It is also interesting to notice that many applications have a standard system call length. For instance, Apache Tomcat by default transfers data in chunks of 8 KB. If it wants to send less than 8 KB, it issues a single system call which reflects the exact flow size. For larger flows, multiple calls are, of course, necessary. Other applications behave similarly; MySQL uses chunks of 16 KB, Spark Standalone 100 KB, and YARN 262 KB. Thus, for identifying short flows, this is a reliable approach, and can directly enable algorithms like Homa [30]. Further, recent work from Facebook suggests that a substantial portion of flows is extremely small and most likely transferred over a single system call [46].

To test this approach, we run a simple experiment where we store 100,000 objects of sizes between 500 B to 1 KB using MySQL. Further, we execute three types of queries: fetch an object based on the key, fetch a range of 10 objects using a date index, and fetch 1000 objects (*e.g.,* to perform a join operation or backup). Since results for queries fetching 1 and 10 objects fit into the initial system call, we were, in fact, able to obtain their flow sizes accurately.

Limitations: The flow size information inferred from system calls may correspond to only a part of the flow rather than the whole flow, as in the above example for large queries. Increasing the size of the initial system call could work, but larger system calls require more buffer memory per connection. Thus, Web servers, databases, and other highly concurrent programs tend to keep system calls small.

3.5 Learning from past traces

We can also apply machine learning to infer flow size information from system traces. Ultimately, data sent out to the network trace causality to some data received, read from disk or memory, or generated through computation. Thus, traces of these activities may allow learning network flow sizes. Given that most jobs in data centers today are repetitive, there is a significant opportunity for such learning. For instance, in [25], the authors observed that more than 60% jobs in enterprise clusters are periodic with predictable resource usage patterns. Analysis of publicly available Google data center traces also confirms this finding: most of the resources are consumed by long-term workloads [37, 3, 36].

Unlike the simpler approaches above, the effectiveness and limitations of learning methods are hard to analyze without a serious attempt at building a learning system. A key challenge here is the short timescale: while past work [28, 44] has explored learning workload characteristics at timescales of minutes and hours, can we learn at the microsecond timescales necessary for flow size estimation? This represents a challenging leap across 8-10 *orders of magnitude* in timescales. We next detail our efforts towards building a learning system for flow size prediction.

4 Learning flow sizes

We explore the design of a learning-based approach for flow size estimation, addressing the following questions:

- Which methods can we use for flow size prediction?
- What prediction accuracy is achievable?

Learning task: We would like to learn flow sizes for outgoing flows in advance, using system traces. When a flow

f starts, are recent measurements of network, disk, memory I/O, CPU utilization, etc. predictive of f's size?

We first present a superficial "black box" treatment of this question, going directly from training standard learning methods over traces we collected to the best accuracy we obtained. §5 shall delve into the details of how we made this approach work, and give intuition into its success. §6 will discuss how the accuracy results translate to network performance improvements, while §7 discusses its limitations.

4.1 Workloads

To explore what inputs are predictive of flow sizes, it is essential to gather job execution traces with as much detail as possible, across many instances of jobs with variable inputs and configurations. Unfortunately, publicly available data center traces do not contain enough information. Facebook's traces [39], by sampling 1 per 30000 packets, provide no visibility at the flow granularity. Google's traces [38] completely omit network data, focusing on CPU, memory utilization, etc. We thus collect traces for (a) a large range of synthetic workloads; and (b) machine learning applications running on our university clusters.

Our traces comprise 5 applications: PageRank, K-Means, and Stochastic Gradient Descent (SGD) implemented on Spark; training deep neural networks using TensorFlow; and a Web workload. The SGD and Tensorflow traces are from instrumented applications running on our university cluster.

Each of SGD, KMeans, and PageRank runs on a Spark cluster of 8 machines, each machine with 2 CPU sockets (4 cores each) and 24 GB DRAM. For SGD, the input sizes vary from 2-25 GB, with significant variation across the hyperparameters. We also impose large input variations for KMeans and PageRank: for PageRank, we randomly generate new graphs with 1-15 million nodes; and for KMeans, we generate datasets with 20-50 million points, while also varying K. We also vary the number of workers per job from 8 to 64.

The Tensorflow trace consists of one 25 minute long execution of distributed training of AlexNet [26] on the ImageNet dataset on 40 GPU machines.

For the synthetic Web workload, we use Apache Tomcat 7.0 to host a full Wikipedia mirror and fetch random pages.

Fig. 2 summarizes these workloads. Fig. 2(a) shows the job execution times across different executions for each algorithm. Execution times for KMeans, PageRank, and SGD vary by factors of as much as 2.6×, 1.5×, and 24.5× respectively. Thus, there is substantial variation across executions. The main source of the variations across traces is the change in the size of the input data and the number of iterations. However, for many runs there were more resources in the cluster than required by the job, leading traces to further incorporate the influence of Spark's scheduling decisions.

We also aggregate flow statistics across all jobs and executions for each application to give a sense of the traffic; Fig. 2(b) shows the flow size distributions, and Fig. 2(c) shows the arrival rates (aggregated across the workers). The TensorFlow workload consists of many short flows with an average arrival rate of 8273 flows/second.

4.2 Machine learning models

We evaluate several ML models, but with only modest efforts to optimize these, because our goal is not to identify the best model or hyperparameters, but to show that a variety of methods could work (as we show with results on improvements in scheduling in §6) with reasonable effort , modulo the limitations of learning in this context, as discussed in §7.

Recurrent Neural Network with LSTM layers: All our traces are time series. Given the natural dependency between data points in the trace, we test a network that can keep state and learn these dependencies while processing the trace sequentially. For this purpose, we use an LSTM model [24]. Using Keras [12] and Tensorflow [2], we test varied LSTM models with different numbers of layers. The best configuration in our setting uses a single layer with 64 LSTM units.

Gradient Boosting Decision Trees: In many of our traces, simple conditionals reveal information about the flow size. For instance, if the first system call size is below a certain value, that can often reveal the flow's size. Thus, we train GBDT models of different sizes (*i.e.,* numbers of trees) and find that using 50 trees (with maximum depth of 10 per tree) gives fast yet accurate results.

Feed-Forward Neural Network: The dependency between flow size and other system-level features should not strictly depend on the ML model we choose. Thus, we test a standard FFNN model [23] with various configurations for the number of layers and neurons, implemented using Keras [12]. We find that 2 layers (and the ReLU activation function) with 5 neurons each yield the best performance.

Results: We split the traces into 3 fixed sets – training, validation, and test. Table 1 compares the three tested models. We use the coefficient of determination (R^2) to measure accuracy. R^2 is very useful because it can be easily compared across different models: $R^2 = 1$ if the model produces perfect predictions, and $R^2 = 0$ if the model makes a prediction of zero value, always predicting the mean. GBDT and FFNN achieve comparable accuracy (Table 1), with the high values of R^2 implying highly accurate flow size predictions. For two workloads, LSTM gave inferior results and did not seem to capture the dependencies, particularly across traces where the underlying executions were very different (*e.g.,* test-set for SGD). With greater effort, for instance, specializing the model to these traces, it may be possible to overcome LSTM's apparent deficiency. However, we wanted to use the same training and inference approach across traces.

GBDT's accuracy, fast convergence, and fast inference motivate its choice for FLUX. The tradeoff is that the model updates in batch mode (not online); this should suffice, unless applications change at sub-second timescales.

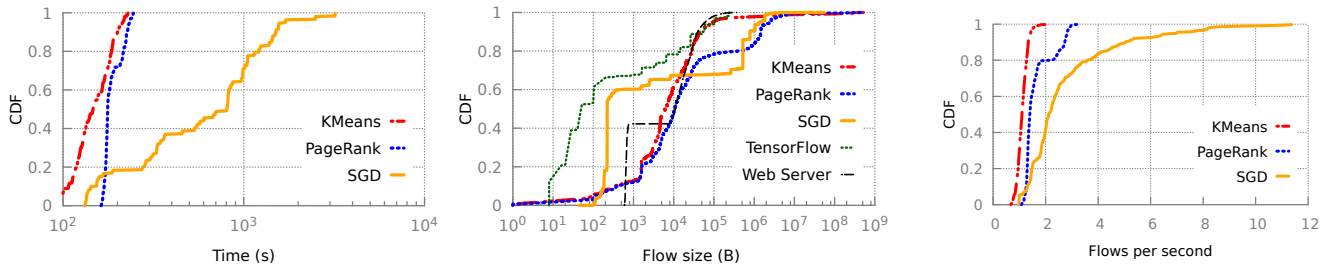


Figure 2: *Workload diversity: (a) Execution time varies substantially across executions of the same job. We also show the distribution of (b) flow sizes and (c) flow arrival rate across our workloads.*

	GBDT	FFNN	LSTM
Web server	94 | 96	92 | 94	73 | 74
TensorFlow	97 | 97	95 | 95	94 | 94
PageRank	85 | 83	84 | 84	83 | 83
Kmeans	88 | 90	88 | 95	88 | 93
SGD	58 | 79	54 | 72	46 | 0

Table 1: *Prediction accuracy (shown for validation-set | test-set) across models and workloads in terms of R^2 percentage.*

5 Opening the black box

What explains the high accuracy of our ML approach? We discuss the predictive power of various system-level measurements, and detail refinements that led from poor initial results to these high-accuracy predictions.

5.1 The treachery of time

We first tried what we considered a natural model for the data of our interest: time series. To generate time series data, during the execution of each workload, we sampled CPU and memory utilization, and disk and memory I/O, every 20 ms, and recorded headers of all incoming and outgoing packets. We then attempted to predict the next few time-steps for network traffic. However, this gave poor results due to low-level system effects that can have a significant impact on timing.

An alternative representation with a *flow-centric view* treats a job as a series of flows, with several attributes recorded per flow (Table 2). This is effective for flow size prediction, as it does not suffer from minor timing variations, and captures the relationship between (for instance) system calls and the volume of outgoing traffic. In addition, the measurements themselves serve as a "clock", one that is more robust to system scheduling artifacts.

5.2 Why these features?

New flows are created either by reading data from the disk or memory, processing previously received flows, doing some computation to create data, etc. Thus, features that characterize each of these causal factors could help estimate flow size. For each type of system measurement, we track the total number of operations or bytes from the beginning of program execution. This enables the learning algorithm to know how many operations or bytes were processed between the last and the new flow. In our experiments, when we predict the flow size, we use features from Table 2 for last 5 flows. Thus, we try to catch dependencies between consecutive flows as well as resources that have been consumed.

Feature	Description
Start time, t_f	Start time of f relative to job start time
Flow gap	Time since the end of the previous flow
First Call	Size of the first system call t_f
Network In	Data received until t_f
Network Out	Data sent until t_f
Network In(d)	Data received at flow's dest. d until t_f
Network Out(d)	Data sent by this host to d until t_f
CPU	CPU cycles used until t_f
Disk I/O	Total disk I/O until t_f
Memory I/O	Total memory I/O until t_f
Previous flows	Flow sizes for last k flows

Table 2: *Features of a flow f. All network, memory, disk and CPU activity is cumulative until this flow's start at t_f.*

As expected, the most predictive features vary across applications. For instance, some applications do not produce a lot of disk traffic, while others rely completely on the disk. The GBDT model provides a natural way of assessing feature importance: it uses a set of decision trees, with each attribute's contribution measured in terms of "splits", *i.e.,* in what percentage of branch conditions in the decision tree the attribute appears. Fig. 3 shows these splits for the aggregated flow-centric traces from a Spark environment. Interestingly, for these traces, we find that similar accuracy can be obtained by a model constrained to *not* use memory, disk, and CPU monitoring, relying only on network data and timing of flows. Web queries, on the other hand, have a different set of critical features where 66% of all splits use disk I/O.

Finding the *best* model and feature-set may require manual work, but effective solutions could be obtained automatically by comparing the accuracy of the largest model to the accuracy of candidate models limited to using only the first model's most salient features.

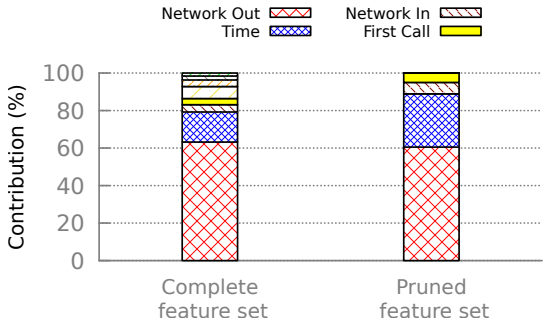


Figure 3: *Feature contributions for* 2 *models for the Spark workloads, measured using GBDT splits. The figure omits labels for the less important features: memory and CPU utilization, and disk and memory I/O. Both models provide the same prediction accuracy. If we exclude any of the top* 3 *contributors, the accuracy decreases.*

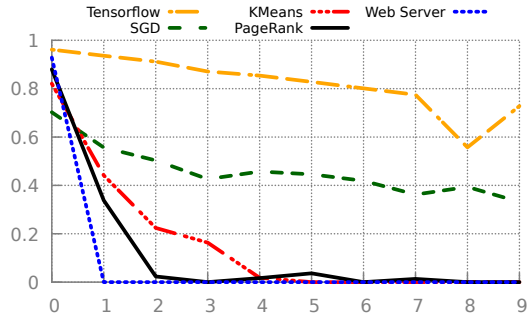


Figure 4: *Prediction accuracy declines for more distant flows.* $x = 0$ *is the current flow, for which packets are starting to be sent out, while* $x = 1$ *is the next flow after this one, and so on.*

5.3 Model accuracy

The accuracy of predictions obtained using learning depends on three main factors, which we discuss next.

How far the predicted future is: It is critical to make predictions within a time budget. Since most flows in data centers are small, this budget is commensurately small: if the inference takes too long, we might either block on the inference and slow down the flow, or allow packets to flow without having the result of the inference and without tagging them appropriately, resulting in sub-optimal performance.

There are two possibilities for overcoming this issue: (a) when a new flow starts, make a prediction in an extremely small time budget by engineering down the inference time; and (b) when a new flow starts, start inference for the *next* outgoing flow, or even more generally, for some future flow. This choice represents a trade-off: we can either get high-accuracy inference by incorporating the maximum information available for inference, but incurring a data path latency to do so (or use results late, as they become available); or get lower accuracy due to missing some relevant information from needing to predict a farther future.

Fig.4 shows the dependence of prediction accuracy on this "future distance", starting from trying to predict a flow's size immediately when it starts, through predicting the next several flows. For TensorFlow, predicting several flows into the future is possible with high accuracy, because flows are predictive of future flows. But as expected, for the Web server workload, it is only possible to accurately predict the flow starting now, because two consecutive flows share no relationship (because we are requesting random objects from a Wikipedia mirror) – in essence, each "job" is of size one.

Model size: Larger models often yield higher accuracy at the cost of more memory and computation, and consequently, and more crucially, higher latency for inference.

While details of the impact of model accuracy on scheduling performance are deferred to §6, we use flow completion times instead of R^2 to compare model sizes. For an example trace (pFabric scheduling for PageRank), when predicting the current (next) flow's size, the average FCT using a smaller model with 20 trees is worse by 9% (10%) than the larger model with 50 trees. Interestingly, the larger model achieves better results than the smaller one, even when the larger model is impaired by having to predict the *next* flow, while the smaller model predicts the current flow.

Training dataset size: Obviously, the learning approach depends on having seen enough training data, but this "convergence time" varies across workloads. For the Web workload, the model only needs to observe ∼50 requests to achieve $R^2 > 0.5$. To achieve nearly its maximum prediction accuracy, the model needs to observe ∼500 requests. For a popular Web server, this is on the order of a few seconds. (Of course, our model for a Web server is extremely simple.)

The model for TensorFlow needs to see ∼3000 flows to reach peak accuracy, but given its flow arrival rate of more than 8200 flows per second, convergence time is sub-second. This is negligible compared to the job duration itself (∼25 min). The iterative nature of neural network training, with similar traffic across iterations, allows accurate prediction within a few iterations of monitoring a never-seen-before job. The Spark workloads show the highest variability, and this is reflected in their convergence time. Here we need traces from multiple executions of the same job type to achieve high accuracy; 10 executions suffice for each of our 3 test job types[5]. Fortunately, data processing frameworks are often run repetitively with many instances of the same job [25, 16] since the workloads often involve tasks like making daily reports, code builds, backups or re-indexing data structures.

Note that good results can be achieved even across repeat executions with very different underlying data and run configurations — the job instances over which we train and test exhibit such variations, as discussed in §4.1.

5.4 Fast-enough, deployable learning?

With data center round-trip times on the order of 10 μs, our objective is to achieve inference in a fraction of this time.

[5]Each execution yields n traces, when n machines execute the job.

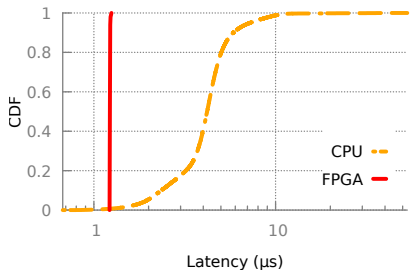


Figure 5: *Inference latency across 100 measurements for GBDT with 50 trees implemented in-kernel on a CPU and using an FPGA.*

CPU implementation: To efficiently implement GBDT, we use *treelite* [1], which takes an XGBoost model as input, and transforms it into a single C function, which is a long sequence of simple *if-else* statements. This approach incurs a minor slowdown when the model is updated (to generate C code), but improves inference performance by an order of magnitude in comparison to the original GBDT implementation in the XGBoost library [11]. This enables us to make inferences in 5 μs within the typical case.

Accelerator implementation: The latency caused by the CPU implementation is small, but still comparable to data center RTTs. We investigate if offloading inference to specialized hardware could improve latency. Such logic could be built into NIC hardware, or deployed on the FPGAs already in use in some data centers [8].

We extend recent work on deploying GBDT on FPGA [32] to obtain our hardware implementation. Fig. 5 compares the inference latency for a 50-tree model on a CPU vs. using an FPGA. The mean latency is 4.3μs on the CPU, and 1.23μs on the FPGA. In each case, this includes the end-to-end time elapsed from when a new flow's packet arrives in the kernel to when it has the result for packet tagging. The FPGA also eliminates the variance in software performance.

Thus, inference latency can be driven down to a fraction of the typical RTTs. Appendix A provides greater detail on our implementation for interested readers.

6 Improving network scheduling

Assessing accuracy of ML-based method in terms of mean error and R^2 is useful, but unsatisfactory — we ultimately want to understand the impact of errors on scheduling that uses the estimates. We thus quantify the performance of both flow-level (FastPass, pFabric, and pHost) and coflow-level (Sincronia) schedulers with varying degrees of inaccuracy in flow sizes. Throughout this evaluation, we use the same traces used for our validation and testing results in §4.2.

6.1 Flow-level scheduling

We use the YAPS simulator [27]. We use the leaf-spine topology used in pFabric [5], with 4 spines, 9 racks, and 144 servers, with all network links being 10 Gbps. To measure the effect of inaccurate predictions on flow completion times (FCT), we replay the network traces collected from our cluster in YAPS. Each experiment uses traces[6] from one of the 5 job types. We run all our tests at 60% network utilization, mirroring the original pFabric and pHost papers.

We compare network performance across the following flow estimators: (0) "Perfect", an ideal predictor with zero error. (1) "Mean", whereby every flow size is predicted to be the mean. (2) "GBDT", the gradient-boosting decision tree learning approach with 50 trees. (3) Specifically for pFabric, we also evaluate the 0-knowledge LAS policy – "Aging" (§3.2). Today's commonly deployed approach – FIFO scheduling at switches and ECMP forwarding – is also evaluated as a baseline ("Oblivious").

Fig. 6 shows the average FCT across all 5 workloads, 3 flow-level scheduling techniques, and these flow estimators. Note that the Aging result is shown only for pFabric, because it can be easily modified to use LAS.

Oblivious often results in mean FCT more than 2× that of Perfect, *e.g.,* compared to FastPass across all workloads, and compared to pFabric in Fig. 6(a) and 6(c); the largest gap is as large as 11.1×, vs. FastPass in Fig. 6(a). GBDT achieves mean FCT close to Perfect across all cases, with the largest gap being 1.21×, vs. FastPass in Fig. 6(e). Compared to Oblivious, improvements with GBDT range from 1.1-11.1× across our experiments.

Understanding the performance of these schemes requires a closer look across the entire flow size distribution. Fig. 7 (left) shows the distribution for one example – pFabric scheduling over an SGD trace, *i.e.,* details behind the mean FCTs for pFabric in Fig. 6(a). Note that the logarithmic x-axis in Fig. 7 visually suppresses significant differences. Aging indeed achieves good results for the short flows for the SGD trace, but for longer flows, which share the same priority for a long time, its performance is worse than Oblivious, resulting in a larger mean FCT (Fig. 6(a)). The TensorFlow workload, with most flows being short, presents a difficult scenario for Aging – as noted in §3.2, for such workloads, Aging's behavior is the opposite of desirable (Fig. 7 (right)).

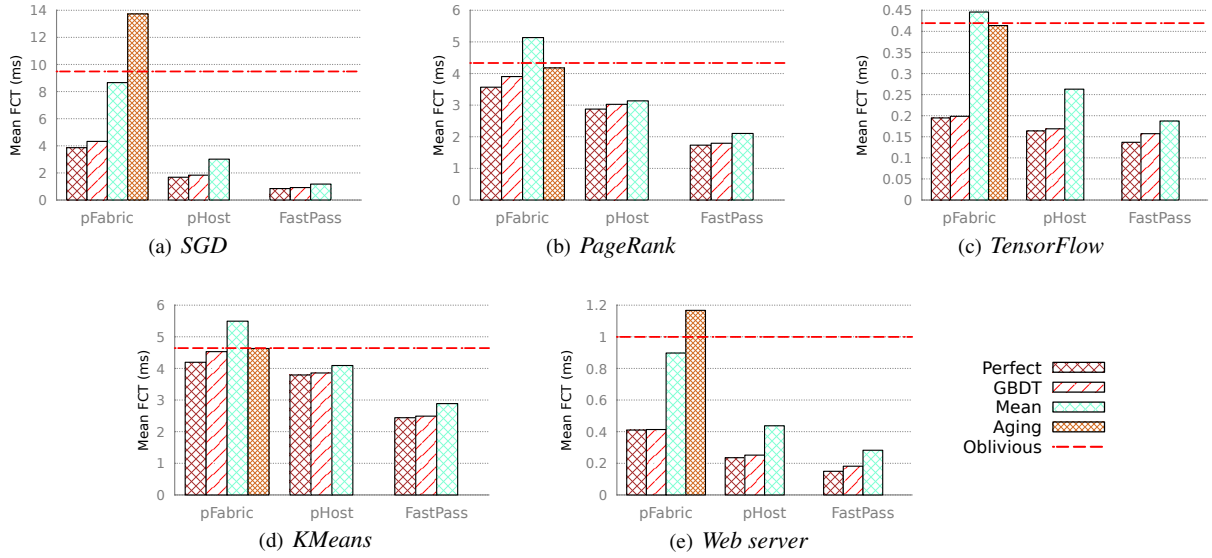
In contrast to Aging, GBDT's performance is similar to Perfect across the flow size spectrum for both workloads.

6.2 Coflow scheduling

We evaluate Sincronia, a recent proposal that leverages flow size information to provide near-optimal coflow completion time (CCT), with our imprecise flow size estimates.

We generate coflows from our traces by picking r consecutive flows grouped together to create a coflow. For each coflow, r is chosen uniformly at random from $\{1,2,3,\ldots,20\}$. For each of our five traces, we run 200 coflows with Sincronia's offline simulator at 60% network load. We execute experiments for Perfect and GBDT,

[6]Note that, unfortunately, we cannot provide results for flow size distributions often used in data center research because we do not have the traces to produce the distribution of estimation error for them.



(a) *SGD* (b) *PageRank* (c) *TensorFlow*

(d) *KMeans* (e) *Web server*

Figure 6: *Mean FCT across 4 scheduling techniques, 5 workloads, and several flow size estimators.*

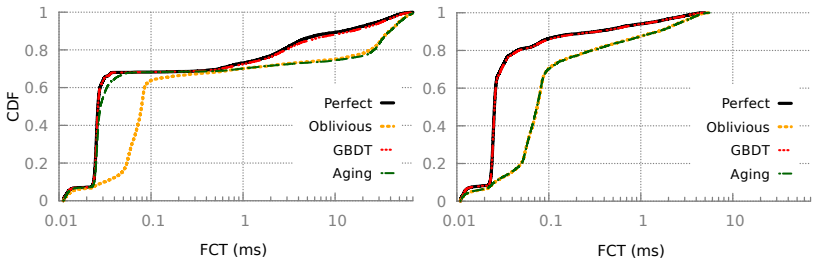


Figure 7: *FCTs for pFabric for the SGD (left) and TensorFlow (right) workloads. Due to the log-scale, small visual differences are significant. On the right plot, Perfect and GBDT are visually indistinguishable, and so are Aging and Oblivious.*

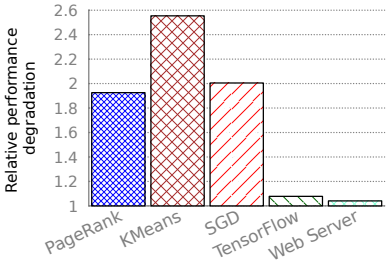


Figure 8: *Relative performance degradation for Sincronia expressed as the ratio between mean CCT with imperfect estimates and perfect knowledge.*

and record mean CCT. To measure the effect of inaccurate predictions, we define relative performance degradation as *GBDT-CCT / Perfect-CCT*.

Fig. 8 shows that the performance degradation for coflow scheduling for PageRank, KMeans and SGD, is substantialy higher than for flow scheduling algorithms. That is because errors in estimates for individual flow sizes compound with coflows. This also explains why workloads with very high accuracy for individual flow sizes, such as Web server and TensorFlow, are only exposed to modest degradation.

7 Limitations of learning

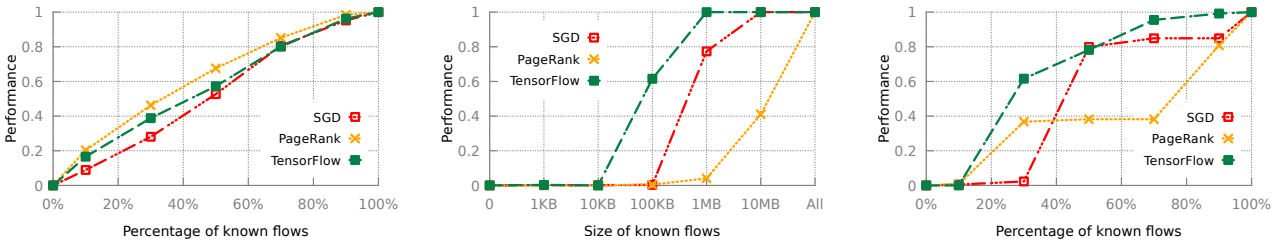
It should be clear that the learning approach is not a panacea. There are several scenarios where it falls short. First and foremost, the prediction context should be clear, *i.e.,* the learning module has to identify the program that is responsible for sending a flow and monitor all features of interest for that flow, as described in §5.2. For Spark, the prediction context assumes knowing start time of a job as well as its ID. This is not unreasonable, as noted in §5.4.

However, for Web servers, we would have to tie disk and memory reads to particular requests. To demonstrate the effect of missing context, we run Apache Tomcat, serving concurrent clients, so that it is not obvious how to match HTTP requests with corresponding disk reads and responses. In this case, disk reads become almost useless as an indicator, and we can only rely on system calls. The prediction accuracy can be made arbitrarily bad by tweaking the experiment parameters, so we omit a concrete accuracy number.

One possibility for obtaining such context is to apply *vertical context injection* [6], which is deployed in Google's data centers; it tags system calls with application information for easier monitoring and debugging.

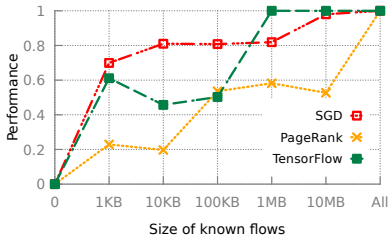
Further, for the execution of one-shot jobs without repetitive internal structure, there is clearly no learning potential. Likewise, for jobs where large, non-deterministic data volumes are generated (*e.g.,* computationally) for transmission, and there is little repetition across executions, it is unlikely that this approach can succeed.

Thus, for many workloads of practical interest, despite our best efforts, this approach will also be limited. We next discuss scenarios where learning or other heuristics can only estimate flow sizes for some fraction of the traffic.



(a) *pFabric, % of flows known* (b) *pFabric, flows known by size* (c) *Sincronia, % of flows known*

Figure 9: *As more flow sizes are known, performance generally improves ...*



(d) *Sincronia, flows known by size*

Figure 9: *(continued) ... but for Sincronia, as larger and larger flows become known, performance sometimes degrades.*

8 More knowledge ⇒ better performance?

While our exploration across several heuristics and an ML-based approach is promising, it is also clear that we will simply not get accurate flow size estimates for *all* applications. Thus, we advocate a pragmatic, two-fold approach: (a) Schedulers should tread a middle-ground – rather than giving up entirely on flow size estimation and operating in a non-clairvoyant manner, using estimates when they are available. (b) We should assess whether it is worth spending effort to expand the set of flows for which sizes can be estimated. We explore such pragmatism for flow- and coflow-level scheduling with pFabric and Sincronia.

To the best of our knowledge, past work has only touched on the first of these ideas. SOAP [40] and Karuna [9] have explored settings with a fixed proportion of flows of known and unknown sizes. Karuna combines pFabric's shortest remaining first (SRF) approach for known flows with Aging[7] for unknown flows. We refer to this policy as SRF-age, but consider a version with infinitely many priorities.

Instead of settings where we have size estimates for a fixed subset of flows, our interest is in examining what happens when we can invest in estimating a larger fraction of flows. In the following, we refer to flows with available sizes as "known" and other flows as "unknown".

Knowing x% of all flows: If $x = 0$, SRG-age devolves to Aging, and if $x = 100$, it becomes Perfect (pFabric with full knowledge). We define performance with an arbitrary x% of flows known as the following normalization, where FCT_P is mean FCT with policy P:

$$Perf(x) = \frac{FCT_{SRF-age(x)} - FCT_{Aging}}{FCT_{Perfect} - FCT_{Aging}}$$

Fig. 9(a,b,c) shows the results on this normalized metric for a sample of workloads from §6. As more flow sizes become known, performance improves.

Knowing all flows of size up to x bytes: Given that some approaches, like using the initial system call, are more effective at estimating smaller flows, it is worth asking how much benefit knowledge of small flows gives. To evaluate this, we modify SRF-age as follows: We assign priorities to known flows following standard SRF, but for flows larger than x, we use $max(age, x)$. This reflects our confidence that any unknown flow is larger than x bytes.

Fig. 9(b) shows that just knowing small flows will not improve performance drastically in terms of mean flow completion time, because they finish near the highest priority even in case of zero knowledge. However, their performance can improve if other, *larger* flows are known, and do not compete with small flows at the same high priority.

Coflow scheduling: Using Sincronia, we also explore the effects of having partial knowledge on coflow scheduling. We generate coflows in the same manner as in §6.2. We run Sincronia offline with 1000 coflows. Unfortunately, not having a packet-level implementation of Sincronia, coupled with the lack of a known or intuitive translation of Aging, limits our analysis. For unknown flows we thus assume that they are of the mean flow size of the whole trace, and refer to this policy as Sinc-mean.

We normalize performance with partial knowledge in the same manner as for pFabric, except using coflow completion times (CCT). The results with x% of flow sizes known and all flows smaller than x bytes known are shown in Fig. 9(c) and Fig. 9(d) respectively.

In some cases, knowing large fractions of flows does not improve CCT substantially. For instance, for the PageRank workload, knowing 70% of flows still gives more than 60% worse results than with perfect knowledge (Fig. 9(c)). This is due to unknown flows within a coflow acting like stragglers.

Fig. 9(d), oddly, indicates that sometimes adding knowledge decreases performance. We explain this with an exam-

[7] We are simplifying here; Karuna actually uses a multi-level feedback queue, with queue thresholds set based on the flow size distribution.

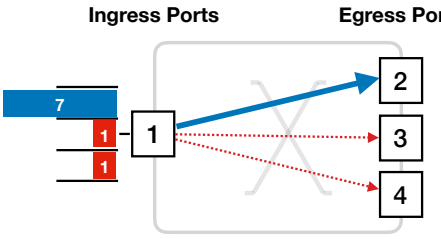


Figure 10: *In this scenario, Sinc-mean scheduling policy leads to priority inversion and performance degradation when knowledge about certain flows is added to the system.*

ple scenario, following a brief (simplified) overview of Sincronia. Sincronia finds a bottleneck port, *i.e.,* one with the largest number of bytes accumulative across flows; and then assigns the lowest priority to the largest coflow on that link. Flows within a coflow share priority.

Now consider a scenario with two coflows, with all their flows going from the same ingress port to different egress ports as shown in Fig. 10. Coflow c_1 contains only one flow with 7 packets, and coflow c_2 contains two flows of 1 packet each. The mean *flow* size is thus 3 packets. Regardless of which flows are un/known, with Sinc-mean, the ingress port would correctly be identified as the bottleneck. If all flows are unknown, Sinc-mean would consider all of them to be of the size 3. Sinc-mean would give c_1 higher priority, because its total estimated coflow size is 3 (compared to 6 for c_2), and, thus, finish c_1 first, within 7 time units. Now instead, say we had disclosed the size of c_1's single constituent flow. This leads Sinc-mean to detect c_1 as the larger coflow (with size 7 for c_1 vs. an estimated 6 for c_2) and give higher priority to c_2. In this case, c_1 finishes after c_2 with a coflow completion time of 9 time units. Thus, for c_1, making its size known results in worse performance under Sinc-mean.

When does more knowledge help? Ideally, we would like the assurance that investing in learning about more flows only improves performance. Otherwise, there are limited incentives for data center operators and/or users to change their applications to expose flow size information or to deploy methods to estimate it.

This property clearly does not hold for Sinc-mean. It is as yet unclear to us how Aging could be incorporated into Sincronia, and whether a partial-knowledge variant can be developed that does not have the quirk of (sometimes) deteriorating when given additional knowledge. However, for the much simpler pFabric/SRF, we can prove a positive result in this direction, showing that for SRF-age, making a certain flow's size known can never deteriorate its performance, at least when interpreted in a worst-case manner.

Our simplified model assumes that all flows go through one link with unlimited output queuing. This output buffer queues packets in flow priority order. This implies that across different flows, packets leave the queue in priority order, but *within* flow packets leave in the same order as they arrive. At every timestep, either a packet leaves the queue, or a new flow arrives. When flows arrive, all their packets are immediately added to this priority queue in their respective positions, with priority ties being broken randomly. To tackle this randomness, we define *worst-case scheduling* for a particular flow f_x as the schedule where any and all ties for f_x's packets break against f_x.

For some flows, their flow sizes are known, and for others, they are not. For flows with unknown sizes, each packet uses the flow's age so far as its priority. The first packet of such a flow has the priority set to zero (highest), with successive packets seeing increments in priority value (*i.e.,* decreasing priority with more packets sent). (For brevity, we omit the distinction between packets and bytes and assume all packets are the same size.) In line with SRF, for known flows, the priority value for their last packet is zero (highest). If the size of a flow f is known, we denote it with f^k; otherwise with f^u. We define priorities such that if $P(p)$ and $P(q)$ are the priorities of packets p and q, then $P(p) > P(q)$ implies p has higher priority, and is scheduled before q.

Theorem 8.1. *All else fixed, with SRF-age, learning the flow size of a particular flow f_x cannot deteriorate its worst-case completion time, i.e., $FCT(f_x^k) \leq FCT(f_x^u)$.*

Proof. To prove the result, we shall show that every packet of any other flow that is scheduled before the end of f_x^k would have also been scheduled before the end of f_x^u, assuming worst-case scheduling for either. It is easy to see that this would imply that the FCT for f_x^k in a worst-case schedule cannot be worse than the FCT of f_x^u.

Suppose a packet r of some other flow is scheduled before a packet p_x^k in f_x^k, given worst-case scheduling for f_x^k. This scheduling implies r has priority higher than or equal to p_x^k, *i.e.,* $P(r) \geq P(p_x^k)$.

Now, say the last packet of f_x^u is l_x^u. Notice that this last packet of f_x^u must have priority lower than or equal to *all* packets of f_x^k, including p_x^k, *i.e.,* $P(l_x^u) \leq P(p_x^k)$. This follows from the definition of SRF-age. If the size of a flow f is $|f|$ packets, then the last packet of f^u (per aging) has priority value $|f|-1$. The n^{th} packet of f^k has priority value $|f|-n$.

Putting the above two inequalities together yields $P(r) \geq P(l_x^u)$. Thus, r would also be scheduled before l_x^u (which is the end of f_x^u), at least in the worst-case schedule for f_x^u. □

A few remarks about this result are in order:

- The theorem and the proof specify worst-case tie-breaking for the flow under consideration. It is easy to produce counterexamples to the theorem statement without the worst-case addendum.
- The definition of SRF-age is central to the result, and it is easy to produce counterexamples for an analogous statement for SRF-mean.
- For systems with a limited number of priority queues, like Karuna or PIAS, the theorem still holds if both known and unknown flows share the same priority thresholds.

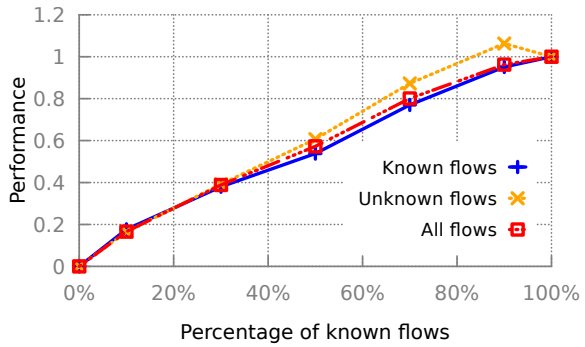


Figure 11: *For the TensorFlow workload, with TCP, unknown flows finish faster under SRF-age.*

- The result can appear counter-intuitive; after all, large unknown flows benefit from high priority in the beginning, which they wouldn't if they were known. While this is true, unknown flows keep slowing down with aging, while known flows keep speeding up with SRF. The proof formalizes this idea.
- Our model, like past work, assumes that scheduling does not change packet inputs to the queue. This is *not true* for TCP flows entering a finite queue.

The impact of the last issue above has not been explored deeply in prior work on packet scheduling. To illustrate its impact in practice, we take a closer look at the TensorFlow trace in Fig. 9(a), separating out the FCTs for known and unknown flows. As Fig. 11 shows, unknown flows finish somewhat faster. This apparent deviation from our theorem's result stems from our simple model which ignores TCP dynamics, assuming instead that all packets of a flow are available for scheduling at its arrival time. The TensorFlow workload comprises nearly 90% flows of sizes smaller than 100 KB. For unknown flows of this type, Aging results in higher priority in the beginning, allowing TCP's exponential slow-start to grow such flows faster than flows with known sizes. Incorporating TCP dynamics into our model to potentially bound the disadvantage that known flows can suffer will require substantial additional effort, which is left to future work. While this discrepancy and its impact on scheduling results should be examined in greater detail, this does not take away from our results on incremental benefits from having greater knowledge with SRF-age scheduling overall.

Mean completion time across all co/flows: Although we have shown that SRF-age cannot deteriorate the performance of a particular flow when its size is made known, it is easy to produce examples where it hurts mean flow completion time across the set of all flows[8]. While our empirical results show improved mean FCT with SRF-age (and an overall trend for improvement even for mean CCT with Sinc-mean), a fuller analysis of this issue is left to future work.

[8]This is also true of Sinc-mean for mean coflow completion time.

9 Related work

We have described relevant past work in context throughout, discussing past efforts on using various types of heuristics for estimating flow sizes in §3, learning-based efforts that operate at slower timescales in §3.5, and work on non-clairvoyant methods that do not use flow sizes in §2 and §3.2.

CODA [45] merits mention as a coflow scheduler that acknowledges imprecision in scheduling inputs, and explicitly handles such imprecision. However, CODA's focus is on clustering flows into coflows, rather than on flow size information, for which it also relies on PIAS-like techniques.

We also discussed Karuna [9] and SOAP [40] in §8, but as the closest prior efforts considering scheduling in a mixed setting with flow sizes (or deadlines, job sizes, etc.) available and not available, we highlight our contributions in comparison to these here. Both Karuna and SOAP only explore scheduling in a mixed setting with a fixed set of known and unknown flows, while our work (a) systematically examines ways of expanding the set of flows for which size estimates are available; (b) evaluates the utility of imprecise estimates across multiple scheduling approaches; and (c) takes first steps towards assessing how the incremental addition of knowledge about flows impacts scheduling, with interesting results for both flow and coflow scheduling.

10 Conclusion

While clairvoyant scheduling promises large performance benefits across a variety of network scheduling problems, its assumption of advance knowledge of flow sizes is, at best, optimistic. Our analysis of how such information may be obtained reveals several settings where even our best efforts are bound to fail. Superficially, this would suggest focusing on non-clairvoyant scheduling, but we argue that such absolutism is unnecessary – we should be using flow size information where available, and examining whether estimating it for more flows yields additional improvements in scheduling. Along these lines, we present several heuristics and a practically implementable learning-based approach to expand the scenarios where flow size knowledge is available. We further show empirically and analytically, that incrementally adding such knowledge is helpful for SRF packet scheduling. For coflow scheduling, we find that small errors in flow size estimation get compounded, leaving a sizable performance gap compared with fully clairvoyant coflow scheduling. We also find that for at least some intuitive policies for scheduling with partial information, additional information can *deteriorate* scheduling, thus necessitating deeper examination of this issue in future work.

Acknowledgments

We are greateful to Kai Chen, Mosharaf Chowdhury, Rachit Agarwal, and the anonymous NSDI reviewers, for their feedback; and to George Porter for shepherding our paper.

References

[1] Treelite : toolbox for decision tree deployment. http://treelite.readthedocs.io/en/latest/, 2017.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A system for large-scale machine learning. In *USENIX OSDI*, 2016.

[3] O. A. Abdul-Rahman and K. Aida. Towards understanding the usage behavior of Google cloud users: the mice and elephants phenomenon. In *IEEE CloudCom*, 2014.

[4] S. Agarwal, S. Rajakrishnan, A. Narayan, R. Agarwal, D. Shmoys, and A. Vahdat. Sincronia: near-optimal network design for coflows. In *ACM SIGCOMM*, 2018.

[5] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM*, 2013.

[6] D. Ardelean, A. Diwan, and C. Erdman. Performance analysis of cloud applications. In *USENIX NSDI*, 2018.

[7] W. Bai, K. Chen, H. Wang, L. Chen, D. Han, and C. Tian. Information-agnostic flow scheduling for commodity data centers. In *USENIX NSDI*, 2015.

[8] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *IEEE/ACM MICRO*, 2016.

[9] L. Chen, K. Chen, W. Bai, and M. Alizadeh. Scheduling mix-flows in commodity datacenters with Karuna. In *ACM SIGCOMM*, 2016.

[10] L. Chen, J. Lingys, K. Chen, and F. Liu. Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *ACM SIGCOMM*, 2018.

[11] T. Chen and C. Guestrin. XGBoost: A scalable tree boosting system. In *ACM SIGKDD*, 2016.

[12] F. Chollet et al. Keras. https://github.com/keras-team/keras, 2015.

[13] M. Chowdhury and I. Stoica. Efficient coflow scheduling without prior knowledge. In *ACM SIGCOMM*, 2015.

[14] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with Orchestra. In *ACM SIGCOMM*, 2011.

[15] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with Varys. In *ACM SIGCOMM*, 2014.

[16] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *ACM SOSP*, 2017.

[17] A. R. Curtis, W. Kim, and P. Yalagandula. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *IEEE INFOCOM*, 2011.

[18] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized task-aware scheduling for data center networks. In *ACM SIGCOMM*, 2014.

[19] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan. High performance network virtualization with SR-IOV. In *IEEE HPCA*, 2010.

[20] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. In *ACM SIGCOMM*, 2010.

[21] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker. pHost: Distributed Near-optimal Datacenter Transport over Commodity Network Fabric. In *ACM CoNEXT*, 2015.

[22] C.-Y. Hong, M. Caesar, and P. B. Godfrey. Finishing flows quickly with preemptive scheduling. In *ACM SIGCOMM*, 2012.

[23] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Netw.*, 2(5), 1989.

[24] L. C. Jain and L. R. Medsker. *Recurrent Neural Networks: Design and Applications*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1999.

[25] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, Í. Goiri, S. Krishnan, J. Kulkarni, et al. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *USENIX OSDI*, 2016.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[27] G. Kumar, A. Narayan, and P. Gao. YAPS network simulator. https://github.com/NetSys/simulator, 2015.

[28] K. LaCurts, J. C. Mogul, H. Balakrishnan, and Y. Turner. Cicada: Introducing predictive guarantees for cloud networks. In *USENIX HotCloud*, 2014.

[29] Y. Lu, G. Chen, L. Luo, K. Tan, Y. Xiong, X. Wang, and E. Chen. One more queue is enough: Minimizing flow completion time with explicit priority notification. In *IEEE INFOCOM*, 2017.

[30] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *ACM SIGCOMM*, 2018.

[31] A. Mushtaq, R. Mittal, J. McCauley, M. Alizadeh, S. Ratnasamy, and S. Shenker. Datacenter congestion control: Identifying what is essential and making it practical. https://people.eecs.berkeley.edu/~radhika/adsrpt.pdf, 2017.

[32] M. Owaida, H. Zhang, C. Zhang, and G. Alonso. Scalable inference of decision tree ensembles: Flexible design for CPU-FPGA platforms. In *FPL*, 2017.

[33] Y. Peng, K. Chen, G. Wang, W. Bai, Z. Ma, and L. Gu. Hadoopwatch: A first step towards comprehensive traffic forecasting in cloud computing. In *IEEE INFOCOM*, 2014.

[34] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized zero-queue datacenter network. In *ACM SIGCOMM*, 2014.

[35] I. A. Rai, G. Urvoy-Keller, M. K. Vernon, and E. W. Biersack. Performance analysis of LAS-based scheduling disciplines in a packet switched network. In *ACM SIGMETRICS*, 2004.

[36] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *ACM SoCC*, 2012.

[37] C. Reiss, J. Wilkes, and J. L. Hellerstein. Google cluster-usage traces: format + schema. Technical report, Google Inc., Mountain View, CA, USA, Nov. 2011. Revised 2014-11-17 for version 2.1. Posted at https://github.com/google/cluster-data.

[38] C. Reiss, J. Wilkes, and J. L. Hellerstein. Google cluster-usage traces: format + schema. *Google Inc., White Paper*, pages 1–14, 2011.

[39] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM*, 2015.

[40] Z. Scully, M. Harchol-Balter, and A. Scheller-Wolf. SOAP: One clean analysis of all age-based scheduling policies. In *ACM SIGMETRICS*, 2018.

[41] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. E. Ng, M. Kozuch, and M. Ryan. c-through: Part-time optics in data centers. In *ACM SIGCOMM*, 2010.

[42] H. Wang, L. Chen, K. Chen, Z. Li, Y. Zhang, H. Guan, Z. Qi, D. Li, and Y. Geng. FlowProphet: Generic and accurate traffic prediction for data-parallel cluster computing. In *IEEE ICDCS*, 2015.

[43] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron. Better never than late: Meeting deadlines in datacenter networks. In *ACM SIGCOMM*, 2011.

[44] D. Xie, N. Ding, Y. C. Hu, and R. Kompella. The only constant is change: Incorporating time-varying network reservations in data centers. In *ACM SIGCOMM*, 2012.

[45] H. Zhang, L. Chen, B. Yi, K. Chen, M. Chowdhury, and Y. Geng. CODA: Toward automatically identifying and scheduling coflows in the dark. In *ACM SIGCOMM*, 2016.

[46] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy. High-resolution measurement of data center microbursts. In *ACM IMC*, 2017.

A Deployment architecture

Different deployment options expose different flexibility-performance tradeoffs. We first discuss FLUX's placement in the app-kernel interface in a controlled environment, and then usage in virtualized environments. For illustration, without loss of generality, we consider the pFabric use case, tagging packets with the remaining flow size.

A.1 Where does FLUX operate?

There are three possibilities for partitioning FLUX's components across user- and kernel-space, shown in Fig. 13.

The data collector must have visibility of app I/O calls, so it must be implemented as either a library to intercepts these, or within the kernel. The packet tagger must sit in the kernel to efficiently manipulate packets. Model training operates off-datapath, and can sit essentially anywhere. The key question is: where is inference implemented?

Fig. 13(a) – as a separate process: Inference is a standalone process, serving requests from the syscall interceptor whenever a new flow starts. This approach makes changing or updating the prediction model trivial. Since the model is a single C function, it can be compiled to a shared library, and loaded dynamically by the inference module.

When an app issues a *send*, the data collector requests inference; a response for which is sent to the tagger. The inference path is much longer than the data path, and packets will arrive for tagging before the inference. This blocking time distribution is shown in Fig. 12(a) across 1000 flow starts. The median (95$^{\text{th}}$ percentile) latency is 67.4μs (720μs).

Fig. 13(b) – as an interposed library: To reduce inter-process communication, inference runs in the same library as the collector. The *send* is intercepted and then issued to the kernel *after* inference finishes, with the predicted flow size. The application perceives this process as a long system call. The median (95$^{\text{th}}$ percentile) latency is 4.97μs (16.7μs).

Fig. 13(c) – as a kernel module: All components except learning can be implemented in the kernel. This kernel module itself runs inference for new flows, and communication with the tagger uses shared kernel memory. The latency comprises entirely of inference, and is shown in Fig. 12(b) for different model sizes, with an average of 4.3μs for the 50-tree model. This approach is quite inflexible: given that different apps could need different inference models, a new model must be inserted in the kernel for each new app.

A.2 Virtualization and offload

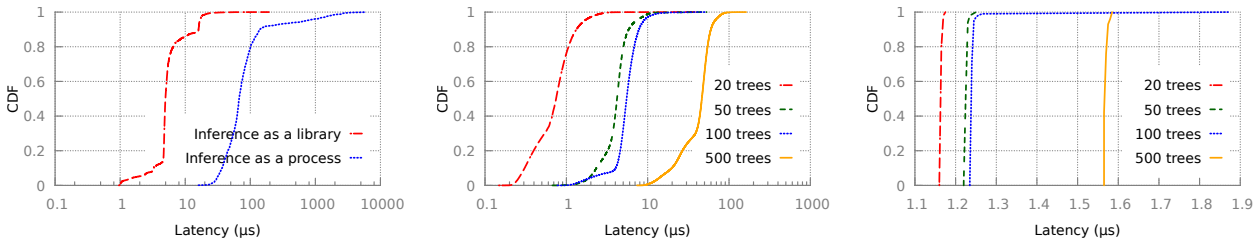
Virtualization: For containers, operation similar to a non-virtualized environment works. For a guest OS, FLUX must be interposed in the guest-host interface. Ultimately, the guest is an "application" on the host, and the network interface is similar, so this does much in terms of performance and accuracy, except in cases where some networking functionality is additionally offloaded to hardware.

Hardware offload: Parts of the network stack may be implemented in hardware, or VMs may interact directly with hardware, such as with SR-IOV [19]. Nevertheless, these environments still must expose a similar *send*, *rcv* API to underlying layers, which FLUX can intercept. However, in these settings, FLUX must be implemented as part of a smart NIC. **This is fundamental to *any* method of using such packet tagging** for network scheduling, because the hypervisor may not touch individual packets at all.

RDMA stacks: RDMA has a significantly different API than TCP. However, even for RDMA networking stacks, the API exposes similar information about sent and received data, which FLUX can exploit.

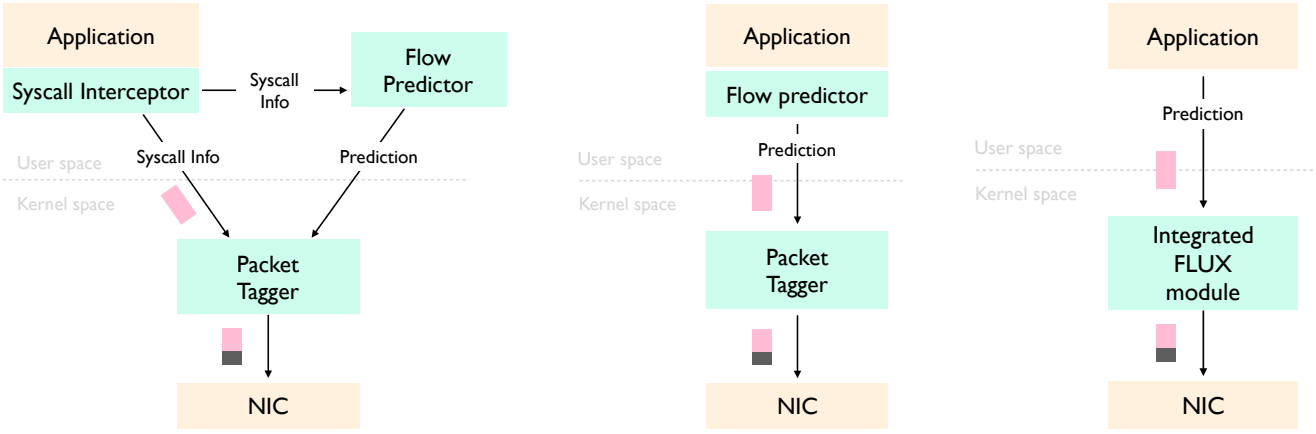
A.3 Inference speed

Fig. 12 shows how inference latency depends on the size of GBDT models on CPU and on FPGA. For the 50-tree model, with which we obtain reasonable accuracy on our traces, the FPGA can achieve latency under 1.3μs. Such low latency is possible because the FPGA is connected to the CPU through the Intel's QPI interconnect which guarantees very short FPGA-CPU round trip. On the other hand, a PCIe-attached FPGA exhibits a slower round trip (on the order of 2.5μs).



(a) (b) (c)

Figure 12: *Inference latency: (a)* 50 *tree GBDT implemented in a library vs. in a process; as a function of model size (b) in-kernel on a CPU; and (c) using an FPGA.*



(a) *as a process* (b) *as a library* (c) *as a kernel module*

Figure 13: *Three possibilities for partitioning* FLUX*'s components across user- and kernel-space.*