SnailTrail: Generalizing Critical Paths for Online Analysis of Distributed Dataflows

Moritz Hoffmann, Andrea Lattuada, John Liagouris, Vasiliki Kalavri, Desislava Dimitrova, Sebastian Wicki, Zaheer Chothia, and Timothy Roscoe, ***ETH Zurich***

https://www.usenix.org/conference/nsdi18/presentation/hoffmann

This paper is included in the Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18).

April 9–11, 2018 • Renton, WA, USA

ISBN 978-1-939133-01-4

Open access to the Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation is sponsored by USENIX.

SnailTrail: Generalizing Critical Paths for Online Analysis of Distributed Dataflows*

strymon.systems.ethz.ch

Moritz Hoffmann, Andrea Lattuada, John Liagouris, Vasiliki Kalavri,
Desislava Dimitrova, Sebastian Wicki, Zaheer Chothia, Timothy Roscoe

Systems Group, Department of Computer Science, ETH Zürich

firstname.lastname@inf.ethz.ch

Abstract

We rigorously generalize critical path analysis (CPA) to long-running and streaming computations and present `SnailTrail`, a system built on Timely Dataflow, which applies our analysis to a range of popular distributed dataflow engines. Our technique uses the novel metric of *critical participation*, computed on time-based snapshots of execution traces, that provides immediate insights into specific parts of the computation. This allows `SnailTrail` to work *online* in real-time, rather than requiring complete offline traces as with traditional CPA. It is thus applicable to scenarios like model training in machine learning, and sensor stream processing.

`SnailTrail` assumes only a highly general model of dataflow computation (which we define) and we show it can be applied to systems as diverse as Spark, Flink, TensorFlow, and Timely Dataflow itself. We further show with examples from all four of these systems that `SnailTrail` is fast and scalable, and that critical participation can deliver performance analysis and insights not available using prior techniques.

1 Introduction

We present a generalization of Critical Path Analysis (CPA) to online performance characterization of long-running, distributed dataflow computations.

Existing tools which aggregate performance information from servers and software components into visual analysis and statistics [2, 30] can be useful in showing what each part of the system is doing at any point in time, but are less helpful in explaining which components in a complex distributed system need improvement to reduce end-to-end latency.

On the other hand, tools which capture detailed individual traces through a system, such as Splunk [9] and VMware LogInsight [3], can isolate specific instances of performance loss, but lack a "big picture" view of what really matters to performance over a long (possibly continuous) computation on a varying workload.

In this paper, we show that the design space for useful performance analysis of so-called "big data" systems is much richer than currently available tools would suggest.

Critical Path Analysis is a proven technique for gaining insight into the performance of a set of interacting processes [36], and we review the basic idea in Section 2. However, CPA is not directly applicable to long-running and streaming computations for two reasons. Firstly, it requires a complete execution trace to exist before analysis can start. In modern systems, such a trace may be very large or, in the case of stream processing, unbounded. Secondly, in a continuous computation, there exist many critical paths (as we show later on), which also change over time, and there is no established methodology for choosing one of them. It is therefore important to aggregate the paths both spatially (across the distributed computation) and temporally (as an evolving picture of the system's performance).

According to prior work [5, 37], the accuracy of CPA increases with the number of critical paths considered. However, existing approaches require full path materialization in order to aggregate information from multiple critical paths. Thus, they restrict analysis to k critical paths, where k is much smaller than the total number of paths in the trace. In open-ended computations where analysis is performed on trace snapshots and all paths are of equal length, materializing all paths is impractical, especially if the analysis needs to keep up with real time. For instance, in our experiments, the number of paths in a 10-sec snapshot of Spark traces is in the order of 10^{21}.

This paper's first contributions (in Section 3) are definitions of *Transient Critical Path*, a modification of classical critical path applicable to continuous unbounded computations, and *Critical Participation* (CP), a metric which captures the importance of an execution activity

*This work was partially supported by the Swiss National Science Foundation, Google Inc., and Amadeus SA.

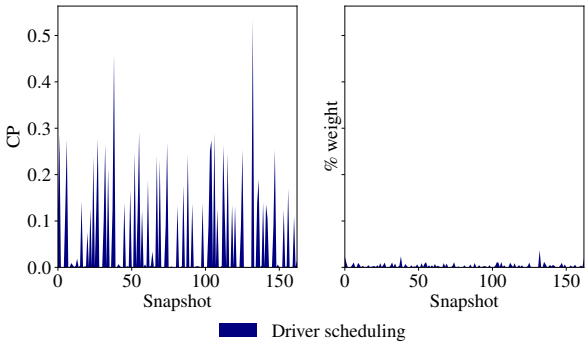


Figure 1: *CP*-based (left) and conventional profiling (right) summaries of Spark's driver activity on BDB [1] from [27] for 64 s snapshots. Spikes indicate coordination between workers and the driver.

in the transient critical paths of computation, and which can be used to generate new time-varying performance summaries. The CP metric can be computed *online* and aggregates information from *all* paths in a snapshot without the need to materialize any path.

Our next contribution (in Section 4) is a model for the execution of distributed dataflow programs sufficiently general to capture the execution (and logging) of commonly-used systems—Spark, Flink, TensorFlow, and Timely Dataflow—and detailed enough for us to define Transient Critical Paths and CP over each of these.

We then show (in Section 5) an algorithm to compute CP online, in real time, and describe `SnailTrail`, a system built (itself as a Timely Dataflow program) to do this on traces from the four dataflow systems listed above. In Section 7 we evaluate `SnailTrail`'s performance, demonstrate online critical path analysis using all four reference systems with a variety of applications and workloads, and show how CP is more informative than existing methods, such as conventional profiling and single-path critical path analysis (Sections 7.4 and 7.5).

Figure 1 gives a flavor of how CP compares with conventional profiling techniques. The key difference is that our approach highlights activities that contribute significantly to the performance of the system, while discarding processing time that lies outside the critical path.

We believe `SnailTrail` is the first system for online real-time critical path analysis of long-running and streaming computations.

2 Critical Path Analysis background

CPA has been successfully applied to high-performance parallel applications like MPI programs [11, 32], and the basic concepts also apply to the distributed dataflow systems we target in this paper. In this section we review classical CPA applied to batch computations as a prelude to our extension of CPA to online and continuous computations in the next section. Table 1 summarizes the notation we use in this section and the rest of this paper.

We view distributed computation as executed by individual system *workers* that perform *activities* (e.g. data transformations or communication). The *critical path* is defined as the sequence of activities with the longest duration throughout the execution. More formally:

Definition 1 Activity: *a logical operation performed at any level of the software stack, and associated with two timestamps* `[start,end]`, `start` ≤ `end`*, that denote the start and end of its execution with respect to a clock C.*

An activity can be either an operation performed by a worker (*worker activity*) or a message transfer between two workers (*communication activity*). Typically, worker activities correspond to the execution of some code, but can also be I/O operations performed by the worker (e.g. reads/writes to/from disk). Communication activities correspond to worker interactions, e.g. message passing.

Different systems have different concepts (threads, VMs, etc.) corresponding to workers. For consistency, we define workers as follows:

Definition 2 Worker: *a logical execution unit that performs an ordered sequence of activities with respect to a clock C.*

We require that no two activities of the same worker a_i:`[start`$_i$`,end`$_i$`]` and a_j:`[start`$_j$`,end`$_j$`]` (where $i \neq j$) can overlap in time, i.e. either `end`$_i \leq$ `start`$_j$ or `start`$_i \geq$ `end`$_j$.

Central to CPA is the *Program Activity Graph* (PAG):

Definition 3 Program Activity Graph: *A PAG* $G = (V, E)$ *is a directed labeled acyclic graph where:*

- *V is the set of vertices. A vertex* $v \in V$ *represents an event corresponding to the start or end of an activity. Each vertex v has a* timestamp $v[t]$ *and a* worker id $v[w]$.
- $E \equiv E_w \cup E_c \subset V \times V$, $E_w \cap E_c = \emptyset$*, is the set of directed edges. An edge* $e = (v_i, v_j) \in E$ *represents an activity* a:`[start,end]`*, where* $v_i[t]$ = `start` *and* $v_j[t]$ = `end`*. An edge e has a type* $e[p]$ *and a weight* $e[w]$ *indicating the activity duration in time units, so that* $e[w] = v_j[t]$ - $v_i[t]$ = `end` - `start` ≥ 0*. An edge* $e \in E_w$ *denotes a worker activity whereas an edge* $e \in E_c$ *denotes a communication activity.*

The direction of an edge $e = (v_1, v_2) \in E$ from node $v_1 \in V$ to node $v_2 \in V$ denotes a `happened-before` relationship between the nodes [24]. The critical path is then defined as the *longest path* in the program activity graph:

Symbol	Description
a:[start,end]	Activity a with start and end timestamps
G	Activity graph
$G_{[t_s,t_e]}$	Snapshot of activity graph G in the time interval $[t_s, t_e]$
$\prod_{t_s}^{t_e}(e)$	Projection of edge e on the time interval $[t_s, t_e]$
$v[w]$	Worker id of vertex v
$v[t]$	Timestamp t of vertex v
$e[w]$	Weight w of edge e
$e[p]$	Type p of edge e
$\|\|\vec{p}\|\|$	Total weight of edges in path $\vec{p}$
E_w	Set of worker activities
E_c	Set of communication activities
$c(e)$	transient path centrality of edge e
CP_e	critical participation of edge e

Table 1: Notation used throughout this paper

Definition 4 Critical Path: *Given a program activity graph $G = (V, E)$, the critical path is a path $\vec{p} \in G$ such that $\nexists \vec{p'} \in G : \|\vec{p'}\| > \|\vec{p}\|$, where $\|\vec{p}\| = \sum_{\forall e \in \vec{p}} e[w]$ and $\|\vec{p'}\| = \sum_{\forall e \in \vec{p'}} e[w]$ is the sum of all edge weights in $\vec{p}$ and $\vec{p'}$ respectively.*

3 Online Critical Path Analysis

Offline processing in traditional CPA is not feasible for long-running or continuous computations like streaming applications or machine learning model training. In these cases, neither the program activity graph nor the critical path can be defined as in Section 2.

Instead, we define online CPA on PAG *snapshots*, performing it on user-defined *time windows*: slices of the PAG that contain activities within a specified time interval. This enables not only performance analysis of running applications, but also targeting specific parts of the computation like the model training phase in a TensorFlow program or a specific time window in a Flink stream.

To achieve this, we show here how to define a time-based program activity graph snapshot and a *transient critical path* on this graph. We then define the *critical participation* performance metric, and we provide the intuition behind it in Section 3.3.

3.1 Transient Critical Paths

To retrieve a snapshot of the PAG, we first assign activities to time windows. Given an edge in a graph, we call its corresponding edge in a snapshot an *edge projection*:

Definition 5 Edge Projection: *Let $e = (v_i, v_j)$ be an edge of an activity graph $G = (V, E)$, where $e \in E$ and $v_i, v_j \in V$. Let also $[t_s, t_e]$, $t_s \leq t_e$, be a time interval with respect to a clock C. Let u_s be a copy of v_i with $u_s[t] = t_s$ and u_e a copy of v_j with $u_e[t] = t_e$. The projection of e on $[t_s, t_e]$ is an edge of the same type as e and is defined only whenever $\left[v_i[t], v_j[t]\right]$ overlaps with $[t_s, t_e]$ as follows:*

$$\prod_{t_s}^{t_e}(e) = \left(\arg\max_{[t]}(v_i, u_s), \arg\min_{[t]}(v_j, u_e)\right)$$

Activities entirely within the time interval $[t_s, t_e]$ are unchanged by the projection, whereas activities that straddle the boundaries are truncated to fit the interval. We can now define a snapshot as follows:

Definition 6 PAG Snapshot: *Let $G = (V, E)$ be a program activity graph, and $[t_s, t_e]$, $t_s \leq t_e$, be a time interval with respect to a clock C. The snapshot of G in $[t_s, t_e]$ is a directed labeled acyclic graph $G_{[t_s,t_e]} = (V', E')$ that is constructed by projecting all edges of G on $[t_s, t_e]$.*

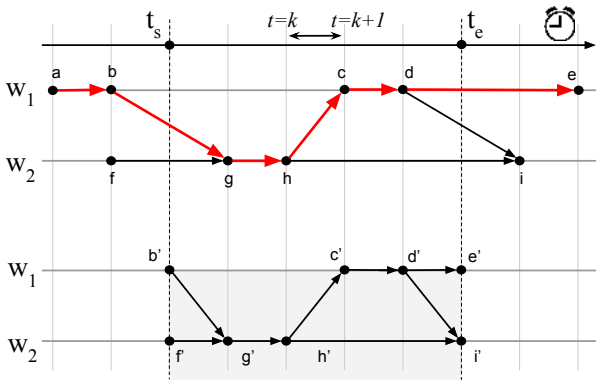
The snapshot $G_{[t_s,t_e]}$ is that part of the PAG which can be observed in the time window $[t_s, t_e]$. Figure 2a shows this applied to the activity timelines of two worker threads, w_1 and w_2, with time flowing left to right. The complete PAG is shown at the top with the critical path in red. Below is the projection of the PAG into the interval $[t_s, t_e]$. The activities straddling the window (e.g. $\prod_{t_s}^{t_e}(b, g) = (b', g')$) are projected to fit in the snapshot.

The key observation is that we cannot define a single critical path in a PAG snapshot since there exist *multiple* longest paths with the same total weight: $t_e - t_s$. All paths starting at t_s and ending at t_e are *potentially* parts of the evolving global critical path. For this reason, we define the notion of *transient critical path*:

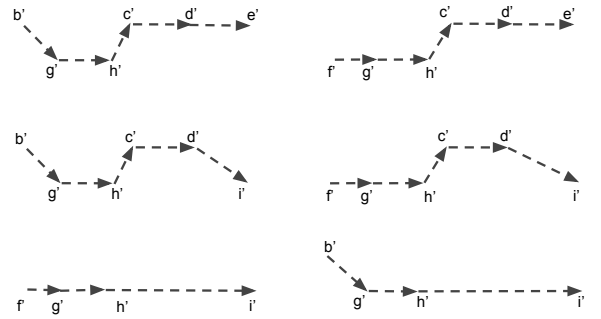
Definition 7 Transient Critical Path: *Let $G_{[t_s,t_e]} = (V, E)$ be the snapshot of an activity graph G in the time interval $[t_s, t_e]$. We define the set of paths $\mathcal{P}$ on $G_{[t_s,t_e]}$ as $\mathcal{P} \equiv \{\vec{p} \subseteq E \mid \nexists\, \vec{p'} : \|\vec{p'}\| > \|\vec{p}\|\}$, where $\vec{p}$ denotes a path in $G_{[t_s,t_e]}$, and $\|\vec{p}\|$ denotes the total weight of all edges in $\vec{p}$, i.e., $\|\vec{p}\| = \sum_{\forall e \in \vec{p}} e[w]$.*

Any path $\vec{p} \in \mathcal{P}$ is a transient critical path of the activity graph G in the time interval $[t_s, t_e]$.

Figure 2b shows all six transient critical paths for the snapshot in Figure 2a. Since each could potentially participate in the evolving global critical path, we need a metric that can aggregate information from all paths and rank activities according to their impact on computation performance. In offline CPA such a ranking is trivial since there is only one critical path for the entire computation.



(a) Program activity timelines of a distributed execution with two workers. The vertical lines divide the timeline into intervals of one time unit. The critical path is highlighted in red in the top timeline. The bottom timeline shows the PAG snapshot into the time interval $[t_s, t_e]$.



(b) Transient critical paths for the graph snapshot of Figure 2a.

Figure 2: A program activity graph, its snapshot in the interval $[t_s, t_e]$, and its transient critical paths.

Since all transient paths can potentially be part of the evolving global critical path, an activity that appears on many transient paths is more likely to be critical and should be ranked high. In Figure 2b, edge (d', i') appears in two paths, while edge (g', h') belongs to all six. The performance metric we define next incorporates this information and ranks activities based on their potential contribution to the global critical path.

3.2 Critical Participation (CP metric)

Given the duration of an activity $e[w]$ and the total length $\|\vec{p}\|$ of the critical path $\vec{p}$, the *participation* of e to $\vec{p}$ is defined as:

$$q_e = \frac{e[w]}{\|\vec{p}\|} \in [0,1] \tag{1}$$

and is easily computed for all activities in a single $\vec{p}$ pass.

We correspondingly define average *critical participation* (CP) of an activity e in a transient critical path as:

$$CP_e = \frac{\sum_{i=1}^{i=N} q_e^i}{N} \in [0,1] \tag{2}$$

where q_e^i is the participation of e to the i-th transient critical path (given by Eq. 1), and N is the total number of transient critical paths in the graph snapshot.

A straightforward way to compute CP_e is to materialize all N transient paths and compute the participation of each activity in every path. However, path materialization is not viable in an online setting because a single graph snapshot might contain too many paths to maintain. Instead, we exploit the fact that the CP of an activity actually depends on the total number of transient paths this activity belongs to. Hence, we define the *transient path centrality* as follows:

Definition 8 Transient Path Centrality: *Let* $\mathcal{P} = \{\vec{p}_1, \vec{p}_2, ..., \vec{p}_N\}$ *be the set of* N *transient paths of snapshot* $G_{[t_s,t_e]}$ *with length* $\|\vec{p}\| = t_e - t_s$. *The transient path centrality of an edge* $e \in G_{[t_s,t_e]}$ *is defined as*

$$c(e) = \sum_{i=1}^{N} c_i(e), \text{ where } c_i(e) = \begin{cases} 0 : e \notin \vec{p}_i \\ 1 : e \in \vec{p}_i \end{cases}$$

The following holds:

$$CP_e = \frac{\sum_{i=1}^{i=N} q_e^i}{N} = \frac{c(e)}{N} \cdot \frac{e[w]}{\|\vec{p}\|} \tag{3}$$

Eq. 3[1] indicates that the computation of CP_e can be reduced to the computation of $c(e)$, which requires no path materialization and can be performed in parallel for all edges in $G_{[t_s,t_e]}$. Section 5 provides an algorithm for transient path centrality and CP without materialization. Note that we can normalize by the number of paths N and their length $\|\vec{p}\|$ because of Definition 7 guaranteeing that all paths have the same length.

We can now compute the transient path centrality and critical participation for the example in Figure 2. For instance, $c(d', i') = 2$ and $c(g', h') = 6$. Respectively, since $t_e - t_s = 5$ and $N = 6$, $CP_{(d',i')} = 0.066$ and $CP_{(g',h')} = 0.2$.

The CP of Eq. 2 can be generalized for activities of a specific type c as:

$$\sum_{\forall e: e[p]=c} CP_e \tag{4}$$

and the following holds[1]:

$$\sum_{\forall c \in G} \sum_{\forall e: e[p]=c} CP_e = 1 \tag{5}$$

Intuitively, Eq. 5 states that the estimated contribution of an activity type, e.g., serialization, to the critical path of the computation is *normalized* over the contribution of all other activity types in the same snapshot.

[1] We provide proofs of Eqs. 3 and 5 in the Appendix A.3.

3.3 Comparison with existing methods

Figure 3 illustrates by example a comparison of CP-based performance analysis with two existing methods: conventional profiling and traditional critical path analysis.

Conventional profiling summaries aggregate activity durations by type or by worker timeline. Such summaries provide information on how much time (i) a program spends on a certain activity type (e.g. serialization) or (ii) a worker spends executing an activity type as compared to other workers. Since conventional profiling summaries rely solely on durations and do not capture execution dependencies, they cannot reveal bottlenecks and execution barriers. Conventional profiling in the execution of Figure 3 would rank activities (a,b) and (c,d) high since they both have a duration of 3 time units, larger than all other activities. However, optimizing those activities cannot result into any performance benefit for the parallel computation as they are both followed by a waiting state (denoted with a dashed line).

On the other hand, CPA captures execution dependencies and can accurately pinpoint activities which influence performance. However, traditional CPA is not directly applicable in a continuous computation as the critical path is not known by just inspecting a snapshot of the execution traces. In a snapshot like the one of Figure 3, all paths starting at s_i and finishing at e_i have equal length in time units, thus traditional CPA would choose one of them at random. We have highlighted such a path in Figure 3 in red color. Although this randomly selected path does not contain the activities (a,b) and (c,d), whose optimization would certainly not improve the latency of the computation, it misses several important activities, such as (x,u) and (v,z), whose optimization would do so.

The CP metric overcomes the limitations of both conventional profiling and traditional CPA by ranking activities based on their potential contribution to the evolving critical path of the computation, which in turn reflects potential benefits from optimization.

Given a snapshot and no knowledge of the execution timelines outside of it, any path between the s_i and e_i points in Figure 3 is *equally probable* to be part of the critical path. CP is a *fairer* metric compared to existing methods in that it aggregates an activity's contribution over *all* transient critical paths and normalizes by the number of paths and the activity's duration. The more paths an activity contributes to, the higher the probability it is a part of the evolving critical path and, hence, the higher its CP metric is. In Figure 3, activities (a,b) and (c,d) do not contribute to any path and thus have zero transient path centrality and CP values. On the other hand, activities (x,u), (u,v), and (v,z) will be ranked as top-three by CP, since they participate in six, nine, and six transient critical paths respectively.

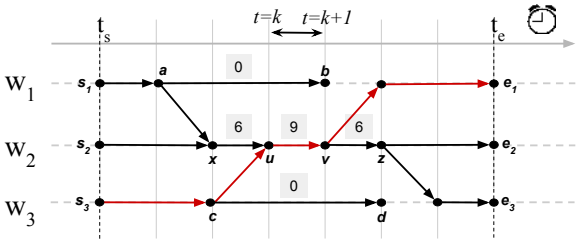


Figure 3: A program activity graph snapshot with three workers. The vertical lines divide the timeline into intervals of one time unit. A randomly chosen critical path is highlighted in red. Edge annotations correspond to transient path centrality (Definition 8).

In Section 7.4 we empirically compare CP-based performance summaries to conventional profiling and traditional CPA, and demonstrate how the results of the latter can be misleading. Further, in Section 7.5, we show how CP can detect and help optimize execution bottlenecks like the one represented by activity (u,v) in Figure 3.

4 Applicability to dataflow systems

Here we show the applicability of our applicability to a range of modern dataflow systems. We provide details on the model assumptions and the instrumentation requirements in the Appendix.

Spark, Flink, TensorFlow, and Timely are superficially different, but actually similar with regard to CPA: all execute dataflow programs expressed as directed graphs whose vertices are operators (e.g. map, reduce) and whose edges denote data dependencies. During runtime, a logical dataflow graph is executed by one or more workers, which can be threads or processes in a machine or a cluster. Each worker has a copy of the graph and processes a partition of the input data in parallel with other workers.

4.1 Activity types

We define a small set of *activity types* we use to classify both the activity of a worker at any given point in time, and communication of data between workers/operators. We consider the following types of *worker* activities:

Data Processing: The worker is computing on data in an operator, which usually has a unique ID. We also include low-level (de)compression operations.

Scheduling: Deciding which operator a worker will execute. In Spark and Flink, scheduling is done by special workers (the Driver and the JobManager).

Barrier Processing: The worker is processing information which coordinates the computation (e.g distributed progress tracking in Timely or watermarks in Flink).

Buffer Management: The worker is managing buffers between operators (e.g. Flink's FIFO queues) or buffering data moving to/from disk (e.g. Spark). The activity may include copying data into/out of buffers, locking, recycling buffers (e.g. Flink) and dynamically allocating them (e.g. Timely).

Serialization: Data is being (un)marshaled, an operation common to all dataflow systems when messages are sent between processes and/or machines.

Waiting: The worker is waiting on some message (data or control) from another worker, and is therefore either *spinning* (as in Timely) or *blocked* on an RPC (as in TensorFlow). Waiting in our model is always a consequence of other, concurrent, activities [21], and so is a key element of critical path analysis: a worker does not produce anything *useful* while waiting, and so *waiting activities can never be on the critical path*.

I/O: The worker is waiting on an external (uninstrumented) system, (e.g. Spark waiting for HDFS, or Flink spilling large state to disk). I/O activities have no special meaning, but capture cases where performance of the reference system is limited by an external system.

Unknown: Anything else: gaps in trace records and any worker activity not captured by the instrumentation. A large number of unknown activities usually indicates inadequate instrumentation [21].

In contrast, interaction between workers is modeled as a *communication* activity, which captures either: (i) **application data exchange** over a communication channel, or (ii) **control messages** conveying metadata about worker state or progress and exchanged between pairs of workers (as in Timely) or through a master (as in Spark, Flink).

4.2 Instrumenting specific systems

We applied our approach to Spark, TensorFlow, Flink, and Timely Dataflow, mapping each to our taxonomy of activities. In some cases we used existing instrumentation, whereas in others we added our own. Space precludes a full discussion of either the structure of these systems or their instrumentation; we provide only brief summaries here and we give more details in [21].

Timely Dataflow [26] required us to add explicit instrumentation, and was the first system we addressed (in part because `SnailTrail` is written in Timely). Timely's progress tracking corresponds to our "barrier" activity, discrete (de)serialization is performed on both data records and control messages, and Timely's cooperative scheduling means that any otherwise unclassified worker activity corresponds to "scheduling".

Apache Flink [10] adopts (unlike Timely) a master-slave architecture for coordination. We treat Flink's

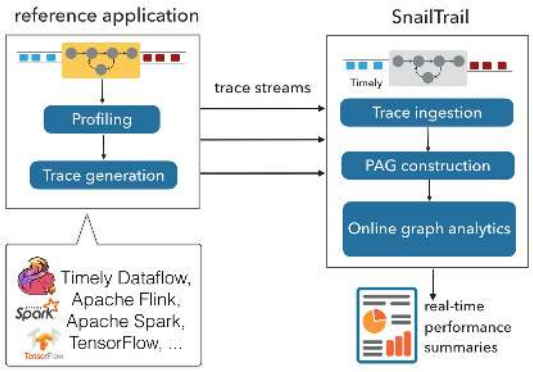


Figure 4: `SnailTrail` overview.

`JobManager`, `TaskManager`s, and `Task`s all as workers, and Flink's runtime has clear activities corresponding to buffer management and serialization. Scheduling is performed in the `JobManager`, barrier processing corresponds to the watermark mechanism, and control messages correspond to communication between the `JobManager` and `TaskManager`s.

TensorFlow [4] has its own instrumentation based on "Timeline" objects, which we reuse unchanged. While enough to generate meaningful results, it also shows how even a well-considered logging system can easily omit information vital for sophisticated performance analysis.

Spark [38] also has native instrumentation which we use to model both the Spark driver and executors as workers. The logs provide information on the lineage of Resilient Distributed Datasets (RDDs) facilitating construction of the PAG. Since executor scheduling is not instrumented, we assume greedily that a task is started on the most recently used thread, which aligns with Spark's observed behavior.

5 `SnailTrail` system implementation

CP is implemented in `SnailTrail`, itself a data-parallel streaming application written in Rust using Timely Dataflow (Figure 4). It reads streams of activity traces via sockets, files, or message queues from a reference application and outputs a stream of performance summaries. `SnailTrail` operates in four pipeline stages: it (i) ingests logs, (ii) slices the stream(s) into windows $[t_s, t_e]$ and constructs PAG snapshots, (iii) computes the *CP* of the snapshots, and (iv) outputs the summaries we show in Section 6.

Traces are sent to `SnailTrail` which ingests a stream S of performance events corresponding to vertices in the activity graph. The snapshots are constructed using Algorithm 1. First, `SnailTrail` extracts from S the events in

the time window $[t_s, t_e]$ (line **1**). These are then grouped by the worker that recorded them (line **2**). Each group corresponds to a worker timeline in Figure 2a. Then, `SnailTrail` sorts the events in each timeline by time (line **4**), and scans each timeline in turn to create the set of edges E_w (line **6**) that correspond to worker activities (cf. Section 4.1). Meanwhile, communication activities are partially initialized based on `send` and `receive` at each worker (line **7**). Then (line **8**), partial edges are grouped by the attributes $\left(w_{id}^{src}, w_{id}^{dst}, c_{id}\right)$; note that w_{id}^{src} is the sender worker id, w_{id}^{dst} is receiver id, and c_{id} is generated to uniquely identify a message. These pairs of partial edges are concatenated to create the final communication edges in E_c, and the output is the union of sets E_w and E_c (line **9**).

Algorithm 1: Graph Snapshot Construction

Input : A stream S of logs and a window $[t_s, t_e]$;
Output : The graph snapshot $G_{[t_s,t_e]}$;

1 let $S_{[t_s,t_e]}$ be the logged events from S in $[t_s, t_e]$;
2 group events in $S_{[t_s,t_e]}$ by worker;
3 **for** *each worker timeline in* $S_{[t_s,t_e]}$ **do**
4 sort events by time;
5 scan events and generate:
6 (a) the set E_w of edges for worker activities;
7 (b) a set E_h of half edges for `send` and `receive` events;
8 group half edges in E_h by $\left(w_{id}^{src}, w_{id}^{dst}, c_{id}\right)$ and create the set E_c of edges for communication activities;
9 **return** $E_w \cup E_c$

Algorithm 1 requires two shuffles of the incoming log stream: one on worker id (before line **2**), and a second on the triple $\left(w_{id}^{src}, w_{id}^{dst}, c_{id}\right)$ (before line **8**). The most expensive step is sorting the timeline (line **4**), requiring $O(|T| \cdot \log|T|)$ time, where $|T|$ is the number of events in the timeline. Parallelism is limited by the number of workers in the reference system (usually many more than `SnailTrail`) and the density of the graph. We emphasize that edges in the PAG represent real `happened-before` dependencies given by the instrumentation. More details in the way edges are created in lines **6-7** are given in the Appendix along with a discussion on clock alignment.

For each graph snapshot, the CP metric is computed using Algorithm 2. `SnailTrail` collects 'start' and 'end' nodes (lines **1-2**) as seeds to traverse $G_{[t_s,t_e]}$. V_s (resp. V_e) includes the node(s) with the minimum (resp. maximum) timestamp $v[t]$ in $G_{[t_s,t_e]}$. Typically, $|V_s| = |V_e| = \ell$, where ℓ is the number of timelines, and so all nodes in V_s have timestamp t_s whereas all nodes in V_e have timestamp t_e.

Algorithm 2 computes the transient path centrality $c(e)$ of Eq. 3 for all edges in $G_{[t_s,t_e]}$. Observe that $c(e) = c_1 \cdot c_2$, where c_1 is the number of paths from the source of e to any node in V_s, and c_2 is the number of paths from the destination of e to any node in V_e. The algorithm thus performs two simple traversals of $G_{[t_s,t_e]}$ in parallel, computing c_1 and c_2 for each edge (lines **3-4**). Each traversal outputs pairs (e, c_i) and these are finally grouped by e to give CP values (lines **6-7**).

Note that, while traversing $G_{[t_s,t_e]}$, we visit each edge in $G_{[t_s,t_e]}$ *only once* by propagating the *final* value c_1 (resp. c_2) from each edge to all its adjacent edges. This reduces the intermediate results of the computation significantly. We compute the CP according to Equation 3, which does not require path materialization.

Algorithm 2 requires two partitions of $G_{[t_s,t_e]}$: one on source, and one on destination ids. Worst-case time complexity is $O(d)$, where d is the diameter of $G_{[t_s,t_e]}$ in number of edges, i.e., the maximum number of edges in any transient critical path.

Algorithm 2: Critical Participation (CP Metric)

Input : An activity graph snapshot $G_{[t_s,t_e]} = (V, E)$;
Output : A set $S = \{(e, \text{CP}) \mid e \in G_{[t_s,t_e]}\}$ of CP values;

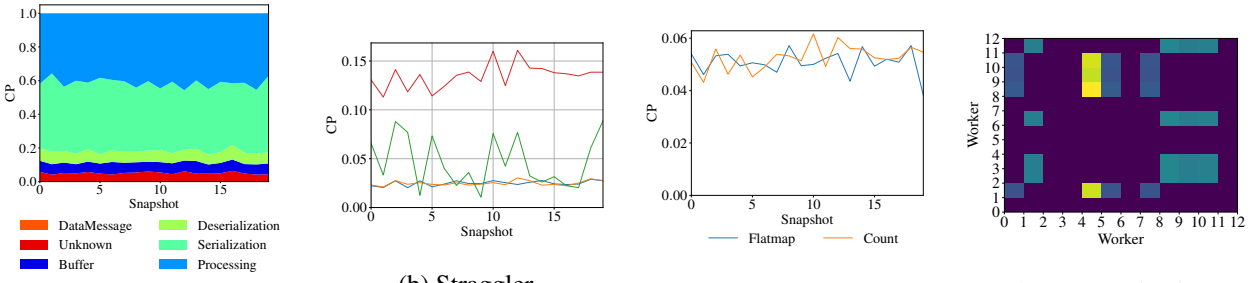
1 let $V_s \equiv \{v \in V \mid \nexists v' \in V : v'[t] < v[t]\}$; `//start nodes`
2 let $V_e \equiv \{v \in V \mid \nexists v' \in V : v'[t] > v[t]\}$; `//end nodes`
`//Both traversals are performed in parallel`
3 traverse $G_{[t_s,t_e]}$ starting from V_s, and count the total number of times each edge is visited, let c_1;
4 traverse $G_{[t_s,t_e]}$ backwards, starting from V_e, and count the total number of times each edge is visited, let c_2;
5 $S = \emptyset$;
6 **for** *each edge* $e \in E$ **do**
7 $S = S \cup \{(e, \frac{c_1 \cdot c_2 \cdot e[w]}{N \cdot (t_e - t_s)})\}$
8 **return** S

Performance summaries are constructed by user-defined groupings on the edge attributes and summing CP values over each group.

`SnailTrail`'s accuracy depends on the quality of the instrumentation. A more complete set of dependencies increases the accuracy of the CP metric. We leave a worst-case error bound analysis for future work.

6 CP-based performance summaries

The CP metric provides an indication of an activity's contribution to the evolving critical path. `SnailTrail` can be configured to generate different types of performance summaries using the CP metric. Each summary type targets a specific aspect of an application's performance and is designed to reveal a certain type of bottleneck. In particular, `SnailTrail` provides four performance summaries which can answer four types of questions: (i) *Which activity type is on the critical path?* (ii) *Is there data skew?* (iii) *Is there computation skew?* (iv) *Is there communication*



(a) Activity (b) Straggler (c) Operator (d) Communication

Figure 5: Examples of `SnailTrail` summary types for the Dhalion [18] benchmark on Flink with 1s snapshots.

skew? The performance summaries not only indicate potential bottlenecks, but also provide immediate actionable feedback on which activities to optimize, which workers are overloaded, which dataflow operator to re-scale, and how to minimize network communication.

Figure 5 shows examples of the four summary types for the Dhalion [18] benchmark on Flink with 1s snapshots. In the rest of this section, we describe each summary type in detail and we discuss how to use them in practical scenarios to improve an application's performance.

Activity summary. *Is the fault-tolerance mechanism in the critical path when taking frequent checkpoints? Is coordination among parallel workers an overhead when increasing the application's parallelism?* An activity summary can answer this sort of questions about an application's performance. This summary plots the proportional CP value of selected *activity types* with respect to the other activity types in a given snapshot. Activity reveal bottlenecks inherent to the system or its configuration. Having a ranking of activity types based on their critical participation essentially gives us an indication on which activities have the higher potential for optimization benefit. For example, if we find that serialization is on the critical path, we might want to try a different serialization library. The activity summary ranking can also help us choose good configurations for our application, like how to adjust the checkpoint interval or the parallelism. The activity summary of Figure 5 shows that serialization and processing have the higher potential for optimization. Activity summaries can be configured to plot selected activities only, as in Figure 1 where we only show the Spark driver's scheduling.

Straggler summary. *Is there data skew? If so, which worker is the straggler?* `SnailTrail` can answer these questions with a straggler summary, which plots the critical participation of a worker's *timeline* in a certain snapshot. The straggler summary relies on the observation that if a worker is a straggler then many transient critical paths pass through its timeline. Hence, we can compare how how critical a worker's activities are as compared to the other workers in the computation and reveal computation imbalance. This ranking can serve as input to a work-stealing algorithm or guide a data re-distribution technique. The straggler summary of Figure 5 clearly shows one straggler worker in the Flink job. In Section 7.5, we look closer into detecting skew with `SnailTrail`.

Operator summary. *Will re-scaling my dataflow improve performance? And if yes, which operator in the dataflow to re-scale?* An operator summary plots the critical participation of each operator's processing activity in a snapshot, normalized by the number of parallel workers executing the operator. This summary reveals bottlenecks in the dataflow caused by resource underprovising and serves as a good indicator for scaling decisions. Traditional profiling methods fail to detect that an operator might be limiting the end-to-end throughput of a dataflow even if its parallel tasks are perfectly balanced. Such bottlenecks are hard to detect by looking at traditional metrics such as queue sizes, throughput, and backpressure. The operator summary of Figure 5 shows that both operators have similar critical participation, thus the parallelism of the job is properly configured. In Section 7.5, we present a detailed use-case where operator summaries guide scaling decisions for streaming applications.

Communication summary. *Is there communication skew? And if yes, which communication channels to optimize?* A communication summary plots the critical participation of communication activities between each pair of workers within a given snapshot. Contrary to traditional communication summaries, this CP-based summary does not rely on communication frequency or absolute message sizes. Instead, it ranks communication edges by their critical importance: the more often a communication edge belongs to a transient critical path, the higher it will be ranked by the summary. Communication summaries can be used to minimize network delays and optimize distributed task placement. If we find that a pair of workers' communication is commonly on the critical path, it

is probably a good idea to physically deploy these two workers on the same machine. For example, the communication summary of Figure 5 indicates that colocating worker 5 with workers 11-13 could benefit performance.

7 Evaluation

To show generality, we evaluate SnailTrail analyzing four different reference systems: Timely Dataflow (version 0.1.15), Apache Flink (1.2.0), Apache Spark (2.1.0), and TensorFlow (1.0.1). Our evaluation is divided into four categories. First, in Section 7.2 we show the instrumentation SnailTrail needs does not cause significant impact on the performance of reference systems. Second, in Section 7.3 we investigate SnailTrail's performance and show it can deliver results in real time with high throughput and low latency. Third, we compare the quality of SnailTrail's analysis and the utility of the *CP* metric with both conventional profiling and traditional critical path analysis (Section 7.4). Finally, we present use cases for SnailTrail with analysis results (Section 7.5).

7.1 Experimental setting

SnailTrail uses the latest Rust version of Timely Dataflow [25] compiled with Rust 1.17.0. In all experiments, SnailTrail ran on an Intel Xeon E5-4640 2.40 GHz machine with 32 cores (64 threads) and 512G RAM running Debian 7.8 ("wheezy"), and was configured to produce results by ingesting execution traces from a reference system on a different cluster.

Benchmarks. We compare SnailTrail to existing approaches with several traces generated by Flink, Spark, and TensorFlow using the following benchmarks. For Flink, we use the Yahoo Streaming Benchmark (YSB) [12] and the WordCount benchmark of Dhalion [18]. For Spark, we use YSB and, for TensorFlow, we use the AlexNet [23] program on ImageNet [29]. To evaluate SnailTrail performance we use Flink (configured with 48 parallel tasks) running a real-world *sessionization* program on a 10min window of operational logs from a large industrial datacenter. This generates a trace with a median number of 30K events per second (around 7.5M events for a 256s snapshot, the largest we used). We also show the instrumentation overhead in Flink, with the same sessionization experiment, and Timely, using a PageRank computation with 16 parallel workers on a random graph.

7.2 Instrumentation Overhead

SnailTrail relies on tracing functionality in the reference system, and this incurs performance overhead. To evaluate the overhead of the instrumentation we added, we implemented a streaming analytic job, *sessionization*, in Flink and an iterative graph computation, PageRank, in Timely, and measured performance with tracing enabled and disabled. For TensorFlow and Spark we use their existing, and somewhat incomplete, tracing facilities.

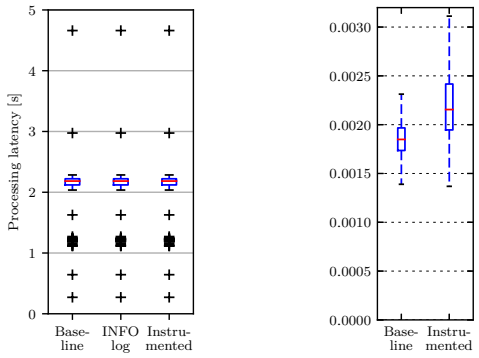


Figure 6: Latency with and without instrumentation for Flink (left) and Timely (right)

Figure 6 shows box-and-whisker plots of processing latency for Flink and Timely implementations. Individual bars correspond to the cases where logging is completely turned off (*baseline*), the default logging level (*info*), and our detailed tracing (*instrumented*).

Flink shows a statistically significant difference of 9.7 % (±1.43 %) additional mean latency, or 203ms (±29.9 µs) in absolute terms, at 95% confidence. This overhead is negligible, given that Flink typically runs with logging enabled in production deployments.

For Timely, there is a statistically significant difference of 13.9 % (±5.5 %) increase in the mean latency, or 319 µs (±126.2 µs) in absolute terms, at 95% confidence.

Experiments with Spark and TensorFlow showed no discernible overhead for collecting the traces required by SnailTrail. Overall, we argue that performance penalties around 10% are an acceptable tradeoff for greater insight, and could be additionally amortized in some cases.

7.3 SnailTrail Performance

We evaluate SnailTrail's performance to demonstrate that (i) it always operates online and thus provides feedback to the running reference applications in real-time and (ii) its analysis scales to large deployments of reference applications without violating this online requirement.

Latency. We require SnailTrail to be capable of constructing the PAG and computing the CP metric for a snapshot of size *x secs* in less than *x secs*. The number of events in a snapshot depends on (i) the snapshot duration and (ii) the instrumentation granularity of the reference system. For this experiment, we vary the number of events

snapshot size	1	2	4	8	16	32	64	128	256
latency	0.06	0.14	0.29	0.62	1.40	2.93	5.91	13.16	24.84
#events	0.03	0.06	0.12	0.24	0.48	0.94	1.91	3.76	7.5

Table 2: SnailTrail's median latency per snapshot (s) for the online analysis of different snapshot intervals (s). The last row shows the median number of events (millions) per snapshot.

snapshot size	1	2	4	8	16	32	64	128	256
throughput	1.2	1.2	1.2	1.1	1.1	1.0	0.8	0.5	0.4
latency	0.7	1.4	3.2	7.1	10.1	10.2	16.8	24.9	30.8

Table 3: SnailTrail's maximum achieved throughput (millions of processed events per second) and corresponding latency per snapshot (s) for the online analysis of different snapshot intervals (s).

in the snapshot by increasing its duration from 1s to 256s (in powers of 2) and we run SnailTrail on the Flink sessionization job trace, which is the densest one we have. Note that the public Spark traces from real-world cloud deployments [27] are not as dense as the ones generated by the Flink streaming computations we run.

We show median latency and number of events per snapshot in Table 2; SnailTrail is always capable of operating online and its latency increases almost linearly with the snapshot duration. Specifically, it can process 1s of input logs in 6ms and 256s of input logs in under 25s.

Throughput. To evaluate SnailTrail's throughput, we interleave the processing of multiple snapshots to increase the number of events sent to the system. Table 3 shows the maximum achieved throughput (number of processed events per second) while respecting the online requirement and the corresponding latency for processing an input snapshot, including PAG construction and CP computation. For 1s snapshots, SnailTrail can process 1.2 million events per second; a throughput *two orders* of magnitude larger than the event rate we observed in all log files we have, including the Spark traces from [27]. SnailTrail comfortably keeps up with all tested workloads: the time to process a snapshot is always smaller than the snapshot's duration. Throughput decreases when increasing the snapshot size since the PAG gets bigger.

7.4 Comparison with existing methods

We examine how useful the *CP*-based summaries produced by SnailTrail are in practice, as compared to the weight-based summaries produced by conventional profiling, where activities are simply ranked by their total

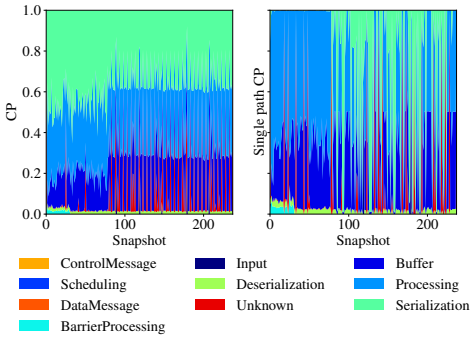


Figure 7: *CP*-based (left) and single-path (right) summaries for Flink on YSB (1s snapshots).

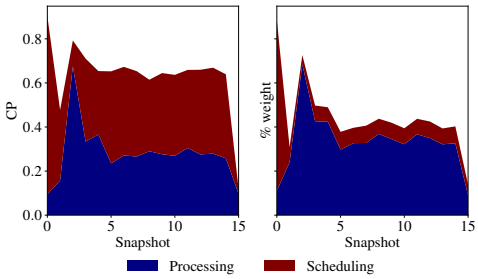


Figure 8: *CP*-based (left) and conventional profiling (right) summaries for Spark on YSB [12] (8s snapshots).

duration, and the single-path summaries, where *CP* is computed on a single transient critical path (in this experiment selected at random). We show examples of such summaries in Figures, 7, 8, and 9 for Flink, Spark, and TensorFlow, along with the configuration of each system.

First note that single-path summaries correspond to a straight-forward application of traditional CPA on trace snapshots where only a single path is chosen at random. The plot on the right of Figure 7 exhibits high variation because different transient critical paths may consist of completely different activities, even within the same graph snapshot. In contrast, *CP* is a *fairer* metric that avoids this misleading critical activity "switching" by aggregating information from all transient critical paths in a snapshot.

Conventional profiling summaries are different from *CP*-based summaries in that they do not account for overlapping activities, thus, they overestimate the participation of activities in the critical path (e.g., the processing activity in the right plot of Figure 8), resulting in activity durations that may even exceed the total duration of the snapshot. The *CP*-based summary of Figure 8 overcomes this problem and highlights the overhead of global coordination in micro-batch systems (driver's scheduling activity), a known result also pointed out in Drizzle [34].

SnailTrail is also different to traditional profiling in its ability to focus on different parts of a long-running

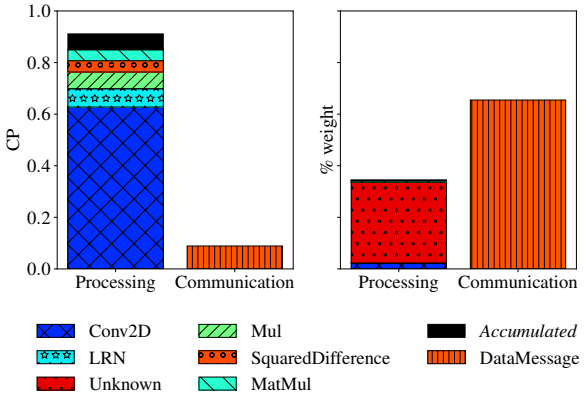


Figure 9: *CP*-based (left) and conventional profiling (right) summaries for the accuracy phase of AlexNet on TensorFlow (16 threads).

computation. This feature is particularly useful in machine learning, where program phases have diverse performance characteristics. As an example, Figure 9 shows *CP*-based and conventional summaries for the accuracy phase of the AlexNet image processing application on TensorFlow with 16 workers. We plot processing and communication as separate bars for convenience and we further break down processing into the different operators appearing in this computation phase. The conventional summary overestimates the participation of communication and underestimates the importance of the Conv2D operator, which is the most critical one according to the *CP*-based summary. Processing in the conventional summary is dominated by the unknown activity type due to limited instrumentation in TensorFlow (see [21]).

7.5 `SnailTrail` in practice

We select Apache Flink as the representative streaming system and demonstrate `SnailTrail` in action. We describe two use-cases and give examples of how the CP-based summaries can be used to understand and improve application performance of long-running computations.

Detecting skew. To demonstrate straggler summaries in action, we use the benchmark of [18]. The benchmark contains a WordCount application and a data generator. The data generator can be configured with a skewness percentage. We experiment with 30%, 50%, and 80% skewness. We configure the parallelism to be equal to 4 for all operators and we generate straggler summaries and conventional summaries shown in Figure 10. For small skew percentage, the conventional summaries fail to detect any imbalance and essentially indicate uniform load across workers. For higher skew percentages (50-80%) they indeed reveal a skew problem, yet they are unable to indicate a single worker as the straggler. Instead, they attribute the imbalance problem to several workers. On the other hand, the CP-based straggler summaries consistently and accurately detect the straggler worker, even for low skew percentage.

Optimizing operator parallelism. We now demonsrate how `SnailTrail` can guide scaling decisions for streaming applications. We use Dhalion's [18] benchmark again and initially under-provision the flatmap stage. We configure four parallel workers for the source, two parallel workers for the flatmap, and four parallel workers for the count operator. Figure 11 (left) shows the operator and conventional profiling summaries for this configuration. We see that the operator summary detects that the flatmap workers are bottlenecks. On the other hand, the conventional summary shows a negligible difference between the parallel workers' processing. In addition, we gather metrics from Flink's web interface. Using those, we can observe backpressure, yet we have no indication of the cause. We next decrease the source's input rate, by changing its parallelism to one worker. Note that slowing down the source is a common system reaction to backpressure. Figure 11 (middle) shows the operator and conventional profiling summaries after this change. Notice how slowing down the source does not solve the problem and how the operator summary still provides more accurate information than the conventional one. The operator summary essentially indicates that the flatmap operator has a high CP value and needs to be re-scaled. Figure 11 (right) shows the summaries after applying a parallelism of four to all operators. Checking Flink's web interface again we see that backpressure disappears.

8 Related Work

There exists abundant literature on performance analysis, characterization, and debugging of distributed systems, although we know of no prior work to perform online critical path analysis for long running computations, or applicable across a broad range of execution models. We distinguish three main areas of related work:

Critical Path Analysis: Yang *et al.* [36] first applied CPA to distributed and parallel applications, defined the PAG, gave a distributed algorithm for CPA, and showed its benefits over traditional profiling. CPA and related techniques have since been used to analyze distributed programs like MPI applications [32, 8] and web services [13], in all cases using offline traces. Algorithms to compute the k longest (near-critical) paths in a computation are given in [6].

The first *online* method for computing critical path profiles seems to be [22], where performance traces are piggybacked on data messages exchanged by processes at

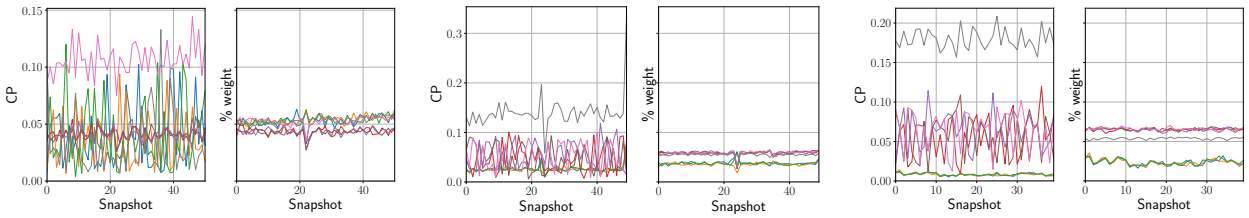


Figure 10: Straggler and conventional profiling summaries for the benchmark of [18] on Flink and different skewness percentage. The data generator has been configured with 30% (left), 50% (middle), and 80% (right) skeweness.

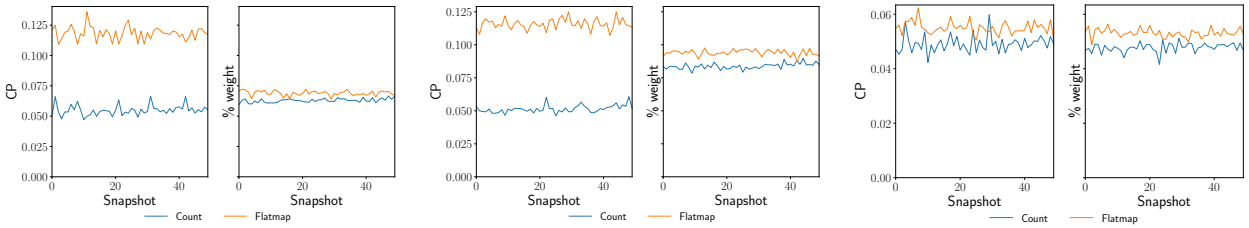


Figure 11: Operator and conventional profiling summaries for the benchmark of [18] on Flink and different configurations of operator parallelism. The source, flatmap, and count operators are configured with parallelism 4-2-4 (left), 1-2-4 (middle), and 4-4-4 (right).

runtime. However, the proposed algorithm is too expensive to construct the full PAG and is thus limited to a small number of user-selected activities. A nice feature of [22] is combining online CPA with dynamic instrumentation to selectively enable trace points on demand. [31] extends the analysis of [22] to the full software stack, and [17] uses this information for adaptive scheduling. Sonata [20] pinpoints critical activities in the spirit of CPA. It supports offline analysis of MapReduce jobs through identifying correlations between tasks, resources and job phases.

Dataflow Performance Analysis: [28] employs *blocked time* analysis to dataflow, a 'what-if' approach quantifying performance improvement assuming a resource is infinitely fast. Blocked time analysis is performed offline and assums staged batch execution. It can only identify bottlenecks due to network and disk and does not provide insights into the interdependence of parallel tasks and operators. An alternative approach in Storm [33]) is based on the *Actor Model* [7] rather than CPA. HiTune [16] and Theia [19] focus on Hadoop profiling; in particular, on cluster resource utilzation and task progress monitoring.

Distributed Systems Profiling: A comprehensive overview of prior work in distributed profiling is [39], which also introduces Stitch, a tool for profiling multi-level software stacks using traces. Like `SnailTrail`, Stitch requires no domain knowledge of the reference system, but its *Flow Reconstruction Principle* assumes logged events are sufficient to reconstruct the execution flow. `SnailTrail` in contrast does not assume this, and indeed yields insights for the better instrumentation of dataflow systems. VScope [35] targets online anomaly detection and root-cause analysis in large clusters. Finally, we note that capturing dependencies between activities in dataflows is similar to *causal profiling* in `Coz` [15]. `Coz` does not focus on distributed dataflows, but does work non-intrusively without instrumentation, and may be applicable to `SnailTrail`.

9 Conclusion

Online critical path analysis represents a new level of sophistication for performance analysis of distributed systems, and `SnailTrail` shows its applicability to a range of different engines and applications. Looking forward, `SnailTrail`'s online operation suggests uses beyond providing real-time information to system administrators: `SnailTrail`'s performance summaries could serve as immediate feedback for applications to perform automatic reconfiguration, dynamic scaling, or adaptive scheduling.

The code in `SnailTrail` has been released as open source[2].

Acknowledgments

We thank Ralf Sager for working on some intitial ideas of this paper, Frank McSherry and the anonymous NSDI reviewers for their comments, and Raluca Ada Popa for shepherding the paper. Vasiliki Kalavri is supported by an ETH Postdoctoral Fellowship.

[2]https://github.com/strymon-system/snailtrail

References

[1] BigData benchmark. https://amplab.cs.berkeley.edu/benchmark/. (accessed: September 2017).

[2] Nagios. https://www.nagios.org. (accessed: September 2017).

[3] VMware LogInsight. http://www.vmware.com/products/vrealize-log-insight.html. (accessed: September 2017).

[4] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA* (2016).

[5] Alexander, C., Reese, D., and Harden, J. C. Near-critical path analysis of program activity graphs. In *International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems* (1994).

[6] Alexander, C. A., Reese, D. S., Harden, J. C., and Brightwell, R. B. Near-critical path analysis: A tool for parallel program optimization. In *Southern Symposium on Computing* (1998).

[7] Bedini, I., Sakr, S., Theeten, B., Sala, A., and Cogan, P. Modeling performance of a parallel streaming engine: Bridging theory and costs. In *ICPE* (2013).

[8] Böhme, D., de Supinski, B. R., Geimer, M., Schulz, M., and Wolf, F. Scalable critical-path based performance analysis. In *IEEE International Parallel and Distributed Processing Symposium* (2012).

[9] Carasso, D. *Exploring Splunk*. Evolved Technologist Press, 2012.

[10] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. Apache Flink: Stream and batch processing in a single engine. *Data Engineering 38*, 4 (2015).

[11] Chen, J., and Clapp, R. M. Critical-path candidates: scalable performance modeling for MPI workloads. In *IEEE International Symposium on Performance Analysis of Systems and Software* (2015).

[12] Chintapalli, S., Dagit, D., Evans, B., Farivar, R., Graves, T., Holderbaugh, M., Liu, Z., Nusbaum, K., Patil, K., Peng, B., and Poulosky, P. Benchmarking streaming computation engines: Storm, Flink and Spark Streaming. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016* (2016), pp. 1789–1792.

[13] Chow, M., Meisner, D., Flinn, J., Peek, D., and Wenisch, T. F. The mystery machine: End-to-end performance analysis of large-scale internet services. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI'14, USENIX Association, pp. 217–231.

[14] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 251–264.

[15] Curtsinger, C., and Berger, E. D. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 184–197.

[16] Dai, J., Huang, J., Huang, S., Huang, B., and Liu, Y. Hitune: Dataflow-based performance analysis for big data cloud. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2011), USENIXATC'11, USENIX Association, pp. 7–7.

[17] Dooley, I., and Kalé, L. V. Detecting and using critical paths at runtime in message driven parallel programs. In *IEEE International Symposium on Parallel and Distributed Processing* (2010).

[18] Floratou, A., Agrawal, A., Graham, B., Rao, S., and Ramasamy, K. Dhalion: Self-regulating stream processing in heron. *Proc. VLDB Endow. 10*, 12 (Aug. 2017), 1825–1836.

[19] Garduno, E., Kavulya, S. P., Tan, J., Gandhi, R., and Narasimhan, P. Theia: Visual signatures for problem diagnosis in large hadoop clusters. In *Proceedings of the 26th International Conference on Large Installation System Administration: Strategies, Tools, and Techniques* (Berkeley, CA, USA, 2012), lisa'12, USENIX Association, pp. 33–42.

[20] Guo, Q., Li, Y., Liu, T., Wang, K., Chen, G., Bao, X., and Tang, W. Correlation-based performance analysis for full-system mapreduce optimization. In *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA* (2013), pp. 753–761.

[21] Hoffmann, M., Lattuada, A., Liagouris, J., Kalavri, V., Dimitrova, D., Wicki, S., Chothia, Z., and Roscoe, T. Snailtrail: Generalizing critical paths for online analysis of distributed dataflows. Tech. rep., ETH Zurich, 2018.

[22] Hollingsworth, J. K. An online computation of critical path profiling. In *SIGMETRICS Symposium on Parallel and Distributed Tools* (1996).

[23] Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[24] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.

[25] McSherry, F. A modular implementation of timely dataflow in Rust (accessed: April 2017). https://github.com/frankmcsherry/timely-dataflow.

[26] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 439–455.

[27] Ousterhout, K. Spark performance analysis (accessed: April 2017). https://kayousterhout.github.io/trace-analysis/.

[28] Ousterhout, K., Rasti, R., Ratnasamy, S., Shenker, S., and Chun, B.-G. Making sense of performance in data analytics frameworks. In *NSDI* (2015).

[29] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV) 115*, 3 (2015), 211–252.

[30] Sacerdoti, F. D., Katz, M. J., Massie, M. L., and Culler, D. E. Wide area cluster monitoring with ganglia. In *2003 IEEE International Conference on Cluster Computing (CLUSTER 2003), 1-4 December 2003, Kowloon, Hong Kong, China* (2003), p. 289.

[31] Saidi, A. G., Binkert, N. L., Reinhardt, S. K., and Mudge, T. N. Full-system critical path analysis. In *IEEE International Symposium on Performance Analysis of Systems and Software* (2008).

[32] Schulz, M. Extracting critical path graphs from MPI applications. *IEEE International Conference on Cluster Computing* (2005).

[33] Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J. M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., Bhagat, N., Mittal, S., and Ryaboy, D. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2014), SIGMOD '14, ACM, pp. 147–156.

[34] Venkataraman, S., Panda, A., Ousterhout, K., Ghodsi, A., Franklin, M. J., Recht, B., and Stoica, I. Drizzle: Fast and Adaptable Stream Processing at Scale. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017).

[35] Wang, C., Rayan, I. A., Eisenhauer, G., Schwan, K., Talwar, V., Wolf, M., and Huneycutt, C. Vscope: Middleware for troubleshooting time-sensitive data center applications. In *Proceedings of the 13th International Middleware Conference* (New York, NY, USA, 2012), Middleware '12, Springer-Verlag New York, Inc., pp. 121–141.

[36] Yang, C.-Q., and Miller, B. P. Critical path analysis for the execution of parallel and distributed programs. In *IEEE International Conference on Distributed Computing Systems* (1988).

[37] Yen, S. H., Du, D. H., and Ghanta, S. Efficient algorithms for extracting the k most critical paths in timing analysis. In *Proceedings of the 26th ACM/IEEE Design Automation Conference* (1989), DAC '89, pp. 649–654.

[38] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012), USENIX Association, pp. 2–2.

[39] Zhao, X., Rodrigues, K., Luo, Y., Yuan, D., and Stumm, M. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *Proceedings of the 12th USENIX*

Conference on Operating Systems Design and Implementation (Berkeley, CA, USA, 2016), OSDI'16, USENIX Association, pp. 603–618.

A Appendix

A.1 Model assumptions

We support both synchronous and asynchronous execution in shared-nothing and shared-memory architectures. Most dataflow systems use asynchronous computations on shared-nothing clusters, but sometimes synchronous computation is supported (e.g. in TensorFlow), and system workers can share state (e.g. in Timely). Specifically, our model is consistent with respect to critical path analysis under two assumptions:

Assumption 1 (Message-based Interaction). *Every interaction between operators in the dataflow must occur via message exchange, even if executed by the same worker.*

Note this assumption does *not* preclude shared-memory systems. Operators in the reference dataflow may share state as long as any modification to this state is appropriately instrumented to trigger a 'virtual' message exchange between the workers sharing that state. We use this approach in instrumenting shared state in Timely Dataflow, for example.

Assumption 2 (Waiting State Termination). *Every waiting activity in a worker's timeline is terminated by an incoming message, either from the same or a different worker.*

In other words, a worker in a waiting state cannot start performing activities unprompted without receiving a message. In the activity graph, a waiting edge's end node must correspond to that of a communication activity, i.e., a receive.

A.2 Instrumentation requirements

An activity may consist of sub-operations spanning multiple levels of the stack from user code to OS and network protocols. A given system can be instrumented at different levels of granularity, depending on the use-case: a multi-layered activity tracking approach enables more detailed performance analysis but introduces higher overhead. We allow this choice, but require that any instrumentation of the reference system satisfy two properties, without which the transient critical paths are ill-defined. The first states that any event having prior events must be caused by an activity earlier in time, i.e. any "out-of-the-blue" events in $(t_s, t_e]$ indicate insufficient instrumentation:

Property 1 (Minimum in-degree) *Let $G_{[t_s,t_e]} = (V,E)$ be the snapshot of activity graph G in time interval $[t_s, t_e]$. Let also $V_s \equiv \{v \in V \mid \nexists v' \in V : v'[t] < v[t]\}$ be a set of vertices in $G_{[t_s,t_e]}$. A vertex $v \in V \setminus V_s$ has in-degree at least one.*

The second states that at no point do all system workers perform waiting activities while no communication activity is occurring. Such behavior would imply deadlock, and so any such points in the activity graph of a non-blocked computation indicates insufficient instrumentation:

Property 2 (Communication Existence) *Let $G_{[t_s,t_e]} = (V,E)$ be the snapshot of an activity graph G in $[t_s,t_e]$, and $\tau \in [t_s,t_e]$ be a point in time. Let $S \equiv \{e = (v_i, v_j) \in E_w \subseteq E \mid e[p] = Waiting,\ v_i[t] \leq \tau \leq v_j[t]\}$. If $|S| = N_\tau$, where N_τ is the number of active workers of the reference system at time τ, then $\exists e' = (v_k, v_m) \in E_c \subseteq E$ for which $v_k[t] \leq \tau \leq v_m[t]$.*

These two properties can also checked efficiently online to inform users when the ingested activity logs are incomplete. For example, instrumentation (or associated log preprocessing) can guarantee that no waiting activities are created as long as the corresponding communication activity, which caused the waiting activity to end, has not been observed.

A.3 Proofs for Equations of Section 3.2

First, we provide a proof for Eq. 3:

$$CP_e = \frac{\sum_{i=1}^{i=N} q_e^i}{N} = \frac{c(e) \cdot e[w]}{N(t_e - t_s)} \in [0,1]$$

Recall that e is an activity edge in the PAG snapshot, N is the total number of transient critical paths in the snapshot, q_e^i is ratio of the activity's duration to the total duration of the i-th transient critical path (the ratio is 0 if the activity edge is not part of the i-th path), $0 \leq c(e) \leq N$ is the number of transient critical paths the activity e belongs to, $e[w]$ is the weight of the activity e, i.e., its duration, and $[t_s, t_e]$ is the snapshot window size.

Without loss of generality, we assume that the transient critical paths $\vec{p}_i$ the activity edge e belongs to are numbered from $i = 1$ to $i = c(e)$. Then:

$$CP_e = \frac{\sum_{i=1}^{i=N} q_e^i}{N} = \frac{\sum_{i=1}^{i=N} \frac{e[w]}{\|\vec{p}_i\|}}{N} = \frac{\sum_{i=1}^{i=c(e)} \frac{e[w]}{\|\vec{p}_i\|}}{N} + 0 = \frac{\sum_{i=1}^{i=c(e)} \frac{e[w]}{\|\vec{p}_i\|}}{N}$$

All transient critical paths in the snapshot have the same length $\|\vec{p}_i\|$ (in time units), which is equal to the duration of the snapshot $t_e - t_s$. Hence:

$$CP_e = \frac{\sum_{i=1}^{i=c(e)} \frac{e[w]}{t_e - t_s}}{N} = \frac{\sum_{i=1}^{i=c(e)} e[w]}{N \cdot (t_e - t_s)} = \frac{c(e) \cdot e[w]}{N \cdot (t_e - t_s)}$$

Now we provide the proof for Eq. 5:

$$\sum_{\forall c \in G} \sum_{\forall e : e[p]=c} CP_e = 1$$

Recall that c denotes an activity type, e.g., serialization, and $e[p]$ is the type of the activity edge e in the snapshot $G_{[t_s,t_e]}$. We have:

$$\sum_{\forall c \in G} \sum_{\forall e : e[p]=c} CP_e = \sum_{\forall e \in G} CP_e = \sum_{\forall e \in G} \frac{\sum_{i=1}^{i=N} q_e^i}{N} =$$

$$= \sum_{\forall e \in G} \frac{\sum_{i=1}^{i=c(e)} \frac{e[w]}{\|\vec{p}_i^{\,e}\|} + 0}{N} = \sum_{\forall e \in G} \frac{\sum_{i=1}^{i=c(e)} \frac{e[w]}{\|\vec{p}_i^{\,e}\|}}{N} =$$

$$= \frac{\sum_{\forall e \in G} \sum_{i=1}^{i=c(e)} \frac{e[w]}{t_e - t_s}}{N} = \frac{\sum_{\forall e \in G} \sum_{i=1}^{i=c(e)} e[w]}{N \cdot (t_e - t_s)} = \frac{N \cdot (t_e - t_s)}{N \cdot (t_e - t_s)} = 1$$

since $\sum_{\forall e \in G} \sum_{i=1}^{i=c(e)} e[w]$ denotes the sum of the weights (durations) of all activity edges that comprise all N transient critical paths in the snapshot, which is equal to $N \cdot (t_e - t_s)$.

A.4 Clock alignment

Computing critical paths only needs logical time, i.e. the happens-before relationship between events. In practice we are using wall-clock time as a stand-in for Lamport timestamps [24] to establish partial ordering of events. Performance statistics such as summaries, however, do require real time.

A practical system for critical path analysis must therefore address issues of *clock drift* (where clocks on different nodes run at different rates) and *clock skew* (where two clocks differ in their values at a particular time).

Clock *drift* only affects activities running on the same thread with durations greater than the drift. Even a drift of 10 seconds/day translates to 0.1ms inaccuracy for activities taking around a second, which is probably tolerable. Clock *skew* is not an issue for activities timestamped by the same thread, but might be for communication activities.

In `SnailTrail`, we assume that the trend toward strong clock synchronization in datacenters [14] means that clock skew is not, in practice, a significant problem for our analysis. If it were to become an issue, we would have to consider adding Lamport clocks and other mechanisms for detecting and correcting for clock skew.