Decentralized Task-Aware Scheduling for Data Center Networks

Fahad R. Dogar, Thomas Karagiannis, Hitesh Ballani, and Antony Rowstron
Microsoft Research
{fdogar, thomkar, hiballan, antr}@microsoft.com

ABSTRACT

Many data center applications perform rich and complex tasks (e.g., executing a search query or generating a user's news-feed). From a network perspective, these tasks typically comprise multiple flows, which traverse different parts of the network at potentially different times. Most network resource allocation schemes, however, treat all these flows in isolation – rather than as part of a task – and therefore only optimize flow-level metrics.

In this paper, we show that task-aware network scheduling, which groups flows of a task and schedules them together, can reduce both the average as well as tail completion time for typical data center applications. To achieve these benefits in practice, we design and implement Baraat, a *decentralized* task-aware scheduling system. Baraat schedules tasks in a FIFO order but avoids head-of-line blocking by dynamically changing the level of multiplexing in the network. Through experiments with Memcached on a small testbed and large-scale simulations, we show that Baraat outperforms state-of-the-art decentralized schemes (e.g., pFabric) as well as centralized schedulers (e.g., Orchestra) for a wide range of workloads (e.g., search, analytics, etc).

Categories and Subject Descriptors: C.2.5 [Computer-Communication Networks]: Network Protocols
Keywords: datacenter; transport; scheduling; response time

1. INTRODUCTION

Today's data center applications perform rich and complex *tasks*, such as answering a search query or building a user's social news-feed. These tasks involve hundreds and thousands of components, *all* of which need to finish before a task is considered complete. This has motivated efforts to allocate data center resources in a "task-aware" fashion. Examples include task-aware allocation of caches [6], network bandwidth [11], and CPUs and network [7].

In recent work, Coflow [10] argues for tasks (or Coflows) as a first-order abstraction for the network data plane. This allows applications to expose their semantics to the network, and the network to optimize for application-level metrics. For example, allocating network bandwidth to tasks in a FIFO fashion, such that they are scheduled over the network one at a time, can improve the average task completion time as compared to per-flow fair sharing (e.g., TCP) [11]. While an exciting idea with important architectural ramifications, we still lack a good understanding of the performance implications of task-aware network scheduling in data centers– (i). How should tasks be scheduled across the network?, (ii). Can such scheduling only improve average performance?, and (iii). Can we realize these gains for small (sub-second) tasks common in data centers? In this paper, we answer these questions and make the following three contributions.

First, *we study policies regarding the order in which tasks should be scheduled.* We show that typical data center workloads include some fraction of heavy tasks (in terms of their network footprint), so obvious scheduling candidates like FIFO and size-based ordering perform poorly. We thus propose FIFO-LM or FIFO with *limited multiplexing*, a policy that schedules tasks based on their arrival order, but dynamically changes the level of multiplexing when heavy tasks are encountered. This ensures small tasks are not blocked behind heavy tasks that are, in turn, not starved.

Second, *we show that task-aware policies like FIFO-LM (and even FIFO) can reduce both the average and the tail task completion times.* They do so by smoothing bursty arrivals and ensuring that a task's completion is only impacted by tasks that arrive before it. For example, data center applications typically have multiple *stages* where a subsequent stage can only start when the previous stage finishes. In such scenarios, FIFO scheduling can smooth out a burst of tasks that arrive at the first stage. As a result, tasks observe less contention at the later stages, thereby improving the tail completion times.

Third, *we design Baraat, a decentralized task-aware scheduling system for data centers.* Baraat avoids the problems associated with centralized scheduling (i.e., scalability, fault-tolerance, etc) while addressing the challenges of decentralized scheduling i.e., making coordinated scheduling decisions while incurring low coordination overhead. To achieve this, Baraat uses a simple heuristic. Each task has a globally unique priority – *all* flows within the task use this priority, irrespective of *when* these flows start or *which* part of the network they traverse. This leads to consistent treatment for all flows of a task across time and space, and improves the chances that all flows of a task make progress together.

By generating flow priorities in a task-aware fashion, Baraat transforms the task-aware scheduling problem into the rel-

SIGCOMM'14, August 17–22, 2014, Chicago, IL, USA.

http://dx.doi.org/10.1145/2619239.2626322.

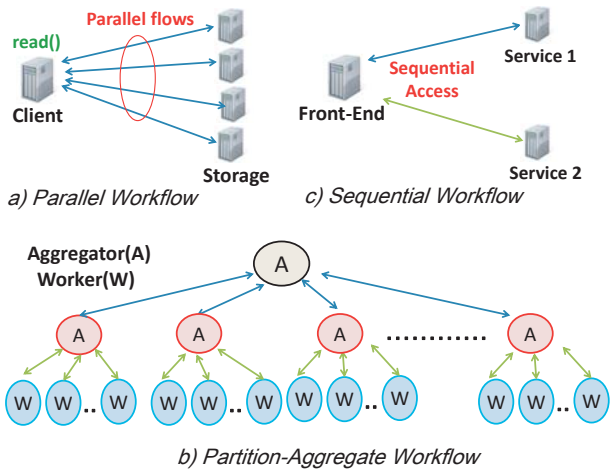

Figure 1: Common workflows.

atively well-understood flow prioritization problem. While many flow prioritization mechanisms exist (e.g., priority queues, PDQ [17], D^3 [28], pFabric [5]), we show that they do not meet all the requirements of supporting FIFO-LM. Thus, Baraat introduces Smart Priority Class (SPC), which combines the benefits of priority classes and explicit rate protocols [17, 13, 28]. It also deals with on-the-fly identification of heavy tasks and changes the level of multiplexing accordingly. Finally, like traditional priority queues, SPC supports work-conservation which ensures that Baraat does not adversely impact the utilization of non-network resources in the data center.

To demonstrate the feasibility and benefits of Baraat, we evaluate it on three platforms: a small-scale testbed for validating our proof-of-concept prototype; a flow based simulator for conducting large-scale experiments based on workloads from Bing [18] and data-analytics applications [9]; the ns-2 simulator for conducting micro-benchmarks. We have also integrated the popular in-memory caching application, Memcached [2], with Baraat. We compare Baraat against the state-of-the-art decentralized network scheduling scheme (i.e., pFabric [5]) as well as centralized schedulers which target MapReduce style workloads (e.g., Orchestra [11]). Our results show that for the Bing-like workload, Baraat reduces the 95^{th} percentile task completion time by 70% compared to pFabric and by 27% compared to Orchestra. For the data-analytics workload, Baraat reduces the 95^{th} percentile task completion time by 43% and 93% compared to pFabric and Orchestra, respectively.

2. A CASE FOR TASK-AWARENESS

Baraat's design is based on scheduling network resources at the unit of a task. To motivate the need for task-aware scheduling policies, we start by studying typical application workflows, which leads us to a formal definition of a task. We then examine task characteristics of real applications and show how flow-based scheduling policies fail to provide performance gains given such task characteristics.

2.1 Task-Oriented Applications

The distributed nature and scale of data center applications results in rich and complex workflows. Typically, these applications run on many servers that, in order to respond

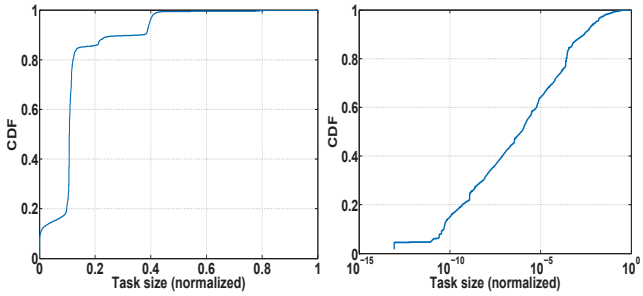

Figure 2: Normalized distribution of task sizes for search (left), data analytics (right) workflows. For data analytics we use the input size of a MapReduce job as a proxy for its size.

to a user request, process data and communicate across the internal network. Despite the diversity of such applications, the underlying workflows can be grouped into a few common categories which reflect their communication patterns (see Figure 1).

All these workflows have a common theme. The "application task" being performed can typically be linked to a waiting user. Examples of such tasks include a read request to a storage server, a search query, the building of the user's social news-feed or even a data analytics job. Thus, we define *a task as the unit of work for an application that can be linked to a waiting user*. Further, *completion time of tasks* is a critical application metric as it directly impacts user satisfaction. In this paper, we aim to minimize task completion time focusing at both the average and the tail (95^{th} percentile and beyond).

As highlighted by the examples in Figure 1, a typical application task has another important characteristic: it generates multiple flows across the network. A task's flows may traverse different parts of the network and not all of them may be active at the same time. When *all* these flows finish, the task finishes and the user gets a response or a notification.

Task characterization. We use data from past studies to characterize two features of application tasks in today's data centers: 1) the task size and 2) the number of flows per task. Both are critical when considering task-aware scheduling for the network; the first influences the scheduling policy, while the latter governs when task-aware scheduling outperforms flow-based scheduling, as we will later discuss.

(1) A task's size is its network footprint, i.e. the sum of the sizes of network flows involved in the task. We examine two popular applications, namely web search and data analytics. Figure 2 (left) presents the normalized distribution of task sizes for the query-response workflow at Bing. For each query, the task size is the sum of flows sizes across all workers involved in the query. The figure reflects the analysis of roughly 47K queries based on datasets collected in [18]. While most tasks have the same size, approximately 15% of the tasks are significantly heavier than others. This is due to the variability in the number of the responses or iterations [4]. By contrast, Figure 2 (right) presents the distribution of the input size across MapReduce jobs at Facebook (based on the datasets used in [9]). This represents the task size distribution for a typical data analytics workload. The figure shows that the task sizes follow a heavy-tailed distribution, which agrees with previous observations [9, 8,

Application	Flows/task	Notes
Web search [4]	88 (lower-bound)	Each aggregator queries 43 workers. Number of flows per search query is much larger.
MapReduce [6]	30 (lower-bound)	Job contains 30 mappers/reducers at the median, 50000 at the maximum.
Cosmos [26]	55	70% of tasks involve 30-100 flows, 2% involve more than 150 flows

Table 1: Tasks in data centers comprise multiple flows.

6]. Similar distributions have been observed for the other phases of such jobs.

Overall, we find that the distribution of task sizes depends on the application. For some applications, all tasks can be similarly sized while others may have a heavy tailed distribution. In § 3.2, we show that heavy-tailed task distributions rule out some obvious scheduling candidates. Hence, *a general task-aware scheduling policy needs to be amenable to a wide-range of task size distributions, ranging from uniform to heavy-tailed.*

(2) For the number of flows per task, it is well accepted that most data center applications result in a complex communication pattern. Table 1 summarizes the number of flows per task for three production data center applications. Flows per task can range from a few tens to hundreds, and as discussed earlier, subsets of flows can be active at different times and across different parts of the network.

Implications for the data center network. Some of the above task characteristics (e.g., large number of concurrent flows) also contribute towards network congestion (and losses), which in turn, results in increased response times for the users. This has even been observed in production data centers (e.g., Bing [4, 18], Cosmos [7], Facebook [22]) which typically have modest average data center utilization. Thus, the network, and its resource allocation policy, play an important role in providing good performance to data center applications. In the following section, we show why today's flow-based resource allocation approaches are a misfit for typical task-oriented workloads.

2.2 Limitations of Flow-based Policies

Traditionally, allocation of network bandwidth has targeted per-flow fairness. Protocols like TCP and DCTCP [4] achieve fair-sharing by apportioning an equal amount of bandwidth to all the flows. This increases the completion time of flows and thus, the task completion time too. Because latency is the primary goal for many data center applications, recent proposals give up on per-flow fairness, and optimize *flow-level* metrics like meeting flow deadlines and minimizing flow completion time [28, 5]. For example, PDQ [17] and pFabric [5] can support a scheduling policy like shortest flow first (*SFF*), which minimizes flow completion times by assigning resources based on flow sizes. However, as we have shown, typical data center application tasks can have many flows, potentially of different sizes. SFF considers flows in isolation, so it will schedule the shorter flows of every task first, leaving longer flows to the end. This can hurt application performance by delaying completion of tasks.

We validate this through a simple simulation that compares fair sharing (e.g., TCP/DCTCP/RCP) with SFF in terms of task completion times, for a simple single stage partition-aggregate workflow scenario with 40 tasks comprising flows uniformly chosen from the range [5, 40]KB. Figure 3 shows SFF's improvement over fair-sharing as a function of the number of flows in a task. We also compare it with the performance of a straw-man task-aware scheme, where flows for the same task are grouped and scheduled together. If a task has just a single flow, SFF reduces the task completion time by almost 50%. However, as we increase the number of flows per task, the benefits reduce. Most tasks in data centers involve tens and hundreds of flows. The figure shows that in such settings, SFF performs similar to fair-sharing proposals. While this is a simple scenario, this observation extends to complex workflows as shown in our evaluation (§5). In contrast, the benefits are stable for the task-aware scheme.

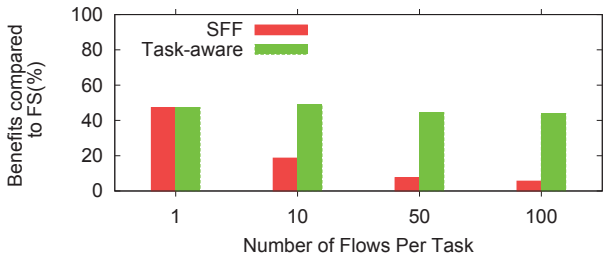

Figure 3: SFF fails to improve over fair sharing (in terms of task completion time) for realistic number of flows per task while a straw-man task-aware policy provides consistent benefits.

3. SCHEDULING POLICY

The scheduling policy determines the order in which tasks are scheduled across the network. Determining an ordering that minimizes task completion time is NP-hard; flow-shop scheduling [14, 25], a well known NP-hard problem in production systems, can be reduced to task-aware scheduling. Flow-shop scheduling is considered as one of the hardest NP-hard problems, with exact solutions not known for even small instances of the problem [15]. Thus, we need to consider heuristic scheduling policies.

The heuristic policy should meet two objectives. First, it should help reduce both the average as well as tail task completion time. Most scheduling policies optimize for one of these, but not both. Second, it should be amenable to decentralized implementation, i.e., it should facilitate scheduling decisions to be made locally (at the respective end-points and switches) without requiring any centralized coordination. While decentralization may appear at odds with the recent trend towards a centralized control plane [3], it is crucial for supporting applications with small (sub-second) tasks (e.g., search, social networking, etc) [23]. This class of applications is the most challenging to handle because of their flow dynamics, but also benefits the most from any improvement in task completion time due to their user-facing nature.

3.1 Task Serialization

The space of heuristics to allocate bandwidth in a task-aware fashion is large. Guided by flow-based policies that schedule flows one at a time [17], we consider serving tasks

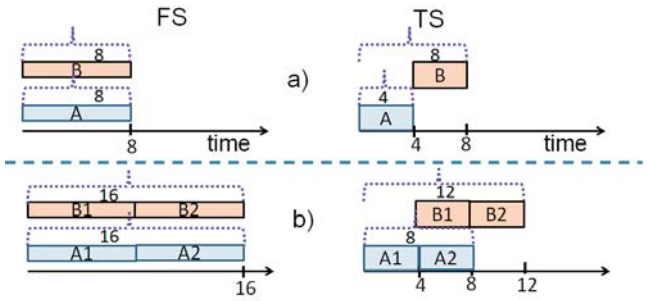

Figure 4: Distilling the Benefits of Task Serialization (TS) over Fair Sharing (FS) for single stage (a) and multi-stage transfers. In case of multiple stages, we assume that each stage has a different network bottleneck.

one at a time. This can help finish tasks faster by reducing the amount of contention in the network. Consequently, we define *task serialization* as the set of policies where an entire task is scheduled before moving to the next.

Through simple examples, we illustrate the benefits of task serialization (TS) over fair sharing (FS). The first example illustrates the most obvious benefit of TS (Fig 4a). There are two tasks, A and B, which arrive at the same time (t = 0) bottlenecked at the same resources. FS assigns equal bandwidth to both the tasks, increasing their completion times. In contrast, TS allocates all resources to A, finishes it, and then schedules B. Compared to FS, A's completion time is reduced by half, but B's completion time remains the same.

Orchestra [11] confirms the above benefits of task serialization in improving the average task completion time for batched workloads. However, the question that still remains unanswered is: *can task serialization also provide gains in tail task completion time?*

One scenario where task serialization can improve the tail task completion time is shown in Fig 4b. There is an application with two stages, as in the partition-aggregate workflow of search. Each stage has a different network bottleneck, which is the case for an application like search, where downlink to the mid-level aggregator is the bottleneck in the first stage and downlink to the top-level aggregator is the bottleneck in the second stage.

There are two tasks, A and B, which arrive in the system at the same time (t = 0). With FS, both tasks get the same amount of resources and thus make similar progress: they finish the first stage at the same time, then move together to the second stage, and finally finish at the same time. TS, in contrast, enables efficient pipelining of these tasks. Task A gets the full bandwidth in the first stage, finishes early, and then moves to the second stage. In parallel, B makes progress in the first stage. By the time B reaches the second stage, A is already finished. This reduces the completion times of both the tasks. In the next section, we show how choosing the right ordering of tasks (scheduling policy), can also result in improving the tail task completion time, even for single stage workloads.

3.2 Task Serialization Policies

We begin with two obvious policies for task serialization: FIFO which schedules tasks in their arrival order and STF

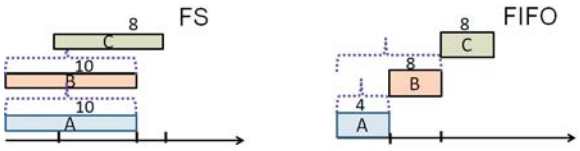

Figure 5: FIFO ordering can reduce tail completion times compared to fair sharing (FS).

(shortest task first) that schedules tasks based on their size. STF can provide good average performance but can lead to high tail latency, or even starvation, for large sized tasks. Moreover, it requires knowledge about task sizes up front, which is impractical for many applications.

FIFO is attractive for many reasons. In addition to being simple to implement, FIFO also limits the maximum time a task has to wait, as a task's waiting time depends only on the tasks that arrive *before* it. This is illustrated in Figure 5 which compares a FIFO policy with fair sharing (FS). While tasks A and B arrive at t = 0, task C arrives later (t = 4). With FS, C's arrival reduces the bandwidth share of existing tasks as all three tasks need to share the resources. This increases the completion times of both A and B and they both take 10 units of time to finish. In contrast, with TS, C's arrival does not affect existing tasks and none of the tasks take more than 8 units of time to finish. *This example illustrates that in an online setting, even for single stage workflows, a FIFO task serialization policy can reduce both the average and tail task completion times compared to FS.*

In fact under simple settings, FIFO is proven to be optimal for minimizing the tail completion time, if task sizes follow a light tailed distribution. i.e., task sizes are fairly homogeneous and do not follow a heavy-tailed distribution [27]. However, if task sizes are heavy-tailed, FIFO may result in blocking small tasks behind a heavy task. As discussed earlier in §2.1, data center applications do have such heavy tasks. For such applications, we need a policy that can separate out these "elephants" from the small tasks.

3.3 FIFO-LM

We propose to use FIFO-LM[1], which processes tasks in a FIFO order, but can dynamically vary the number of tasks that are multiplexed at a given time. If the degree of multiplexing is one, it performs exactly the same as FIFO. If the degree of multiplexing is ∞, it works similar to fair sharing. This policy is attractive because it can perform like FIFO for the majority of tasks (the small ones), but when a large task arrives, we can increase the level of multiplexing and allow small tasks to make progress as well.

An important question is how to determine that a task is heavy i.e., how big is a heavy task. We assume that the data center has knowledge about task size distribution based on historically collected data. Based on this history, we need to identify a threshold (in terms of task size) beyond which we characterize a task as heavy. For applications with bimodal task size distribution or resembling the Bing workload in Figure 2, identifying this threshold is relatively straightforward. As soon as the task size enters the second mode, we classify it as heavy and increase the level of multiplexing.

[1]typically referred to as limited processor sharing in scheduling theory [21].

For heavy-tailed distributions, our experimental evaluation with a number of heavy-tailed distributions such as Pareto or Lognormal with varying parameters (shape or mean respectively), shows that a threshold in the range of 80^{th}-90^{th} percentile provides the best results.

4. BARAAT

Baraat is a decentralized task-aware scheduling system for data center networks. It aims to achieve FIFO-LM scheduling in a decentralized fashion, without any explicit coordination between network switches.

In Baraat, each task is assigned a globally unique identifier (*task-id*) based on its arrival (or start) time. Tasks with lower ids have a higher priority over ones with a higher id. Network flows carry the identifier of the task they belong to and inherit its priority. Based on this identifier switches can make *consistent* decisions without any coordination. If two switches observe flows of two different tasks, both make the same decision in terms of flow prioritization (consistency over space). If a switch observes flows of two tasks at different times, it makes the same decision (consistency over time). Such consistent resource allocation increases the likelihood that flows of a task get "similar" treatment across the network and hence, tasks actually progress in a serial fashion. Finally, switches *locally* decide when to increase the level of multiplexing through on-the-fly identification of heavy tasks.

In the next section, we discuss how the task priorities are generated. We then discuss how switches act on these priorities and why existing mechanisms are insufficient. Finally, we present the Smart Priority Class mechanism, and discuss how it meets our desired prioritization goals.

4.1 Generating Task Identifiers

Baraat uses monotonically increasing counter(s) to keep track of incoming tasks. We only need a single counter when all incoming tasks arrive through a common point. Examples of such common points include the load balancer (for user-facing applications like web search), the job scheduler (for data parallel and HPC applications), the meta-data manager (for storage applications), and so on.

The counter is incremented on a task's arrival and is used as the task's *task-id*. We use multiple counters when tasks arrive through multiple load balancers. Each counter has a unique starting value and an increment value, i, which represents the number of counters in the system. For example, if there are two counters, they can use starting values of 1 and 2 respectively, with i = 2. As a result, one of them generates odd *task-ids* (1, 3, 5,...) while the other generates even *task-ids* (2, 4, 6...). We highlight such a setting, which approximates a FIFO ordering in a distributed scenario, in one of our testbed experiments in §5. These counters can be loosely synchronized and any inconsistency between them could be controlled and limited through existing techniques [29].

The generation of task identifiers should also account for background services (e.g., index update) that are part of most production data centers. Tasks of such services often involve long flows which can negatively impact tasks of online services, if not properly handled. In Baraat, we assign strictly lower priority to such background tasks by assigning them *task-ids* that do not overlap with the range of *task-ids* reserved for the high priority online service. For example, *task-ids* less than n could be reserved for the online service while *task-ids* greater than n could be used for the background service.

	Strict Priority	Fair Sharing	Heavy Task Support	Work Conservation	Preemption
DCTCP	No	Yes	No	Yes	No
RCP	No	Yes	No	Yes	No
D^3	Partial	Yes	No	Yes	No
pFabric / Priority Queues	Yes	Yes	No	Partial	Yes
PDQ	Yes	No	No	Yes	Yes

Table 2: Desired properties and whether they are supported in existing mechanisms.

Propagation of task identifiers. A flow needs to carry the identifier for its parent task. Thus, all physical servers involved in a task need to know its *task-id*. Applications can propagate this identifier along the task workflow; for example, for a web-search query, aggregators querying workers inform them of the *task-id* which can then be used for the response flows from the workers back to the aggregators.

4.2 Prioritization Mechanism - Requirements

Baraat's task-aware assignment of flow priorities, in the form of *task-ids*, opens up the opportunity to use existing flow prioritization mechanisms (e.g., priority queues, pFabric [5], PDQ [17], etc) at the switches and end-points. While these mechanisms provide several attractive properties, they do not meet all the requirements of supporting FIFO-LM. Table 2 lists the desired properties and whether they are supported in existing mechanisms.

The first three properties, *strict priority*, *fair-sharing* and *heavy task support*, are the basic building blocks for FIFO-LM: we should be able to strictly prioritize flows of one task over another; likewise, if the need arises (e.g., heavy task in the system), we should be able to do fair-sharing of bandwidth amongst a set of flows. Finally, the system should on-the-fly identify heavy tasks and then change the level of multiplexing accordingly.

The last two properties, *work-conservation* and *preemption*, are important for system efficiency. Work conservation ensures that a lower priority task is scheduled if the highest priority task is unable to saturate the network – for example, when the highest priority task is too small to saturate the link or if it is bottlenecked at a subsequent link. Finally, preemption allows a higher priority task to grab back resources assigned to a lower priority task. Thus, preemption *complements* work conservation – the latter lets lower priority tasks make progress when there is spare capacity, while the former allows higher priority tasks to grab back the resources if they need to. These two properties also prove crucial in supporting background services; such services can continue to make progress whenever there are available resources while high priority tasks can always preempt them.

Limitations of existing mechanisms. As the table highlights, no existing mechanism supports all these five properties. Support for handling heavy tasks is obviously missing as none of these mechanisms targets a policy like FIFO-LM. PDQ [17] does not support fair-sharing of bandwidth, so two flows having the same priority are scheduled in a serial fashion. Similarly, pFabric [5] and priority queues

do not support work-conservation in a multi-hop setting because end-hosts always send at the maximum rate, so flows continue to send data even if they are bottlenecked at a subsequent hop. In such scenarios, work-conservation would mean that these flows back-off and let a lower priority flow, which is not bottlenecked at a subsequent hop, send data. Protocols like PDQ avoid this problem with the help of explicit feedback from the switches, but they have other limitations, as we discussed earlier.

These limitations of existing mechanisms motivate Smart Priority Class (SPC), which we describe next.

4.3 Smart Priority Class

We propose Smart Priority Class (SPC), which is *logically* similar to priority queues used in switches: flows mapped to a higher priority class get strict preference over those mapped to a lower priority class, and flows mapped to the same class share bandwidth according to max-min fairness. However, compared to traditional priority queues, SPC has two additional *smarts*:

- To provide work-conservation in multi-hop settings, SPC supports *explicit feedback* from switches. We leverage prior work on explicit rate control protocols to communicate this feedback between the switches and the end-hosts [17, 13]. However, unlike prior work, we only keep aggregate, per-task counters at the switches, instead of per-flow state.
- SPC supports *dynamic mapping* of flows to priority queues — this is required to support FIFO-LM as a flow's mapping may change during its lifetime, if a heavy task is identified in the system. We implement a light-weight *classifier* in each switch, which uses the per-task counters to identify heavy tasks and maps tasks to priority classes accordingly.

Figure 6a provides an overview of SPC functionality, which comprises support at the switches and an end-host transport.[2] The figure shows the key steps in sending a block of data from a sender to a receiver: i) Applications pass the *task-id* and data to the SPC transport, which computes the flow *demand* based on the size of data. ii) The *task-id* and demand are passed to the next-hop switch in a packet header. iii) Each switch adds its feedback to the packet header. The feedback is calculated based on the flow's priority class and aggregate task information stored locally at each switch (such as the total demand of the task and the number of flows in the task). iv) The receiver piggybacks the consolidated feedback on the acknowledgment packets that are sent back to the sender. v) Finally, each sender uses this feedback to decide the rate at which it should send data. The process is repeated once every round trip time, until the sender has no more data to send. Other senders who are part of the task also follow the same steps without explicitly coordinating with each another.

While Baraat requires changes to both end-hosts and switches, we believe that such changes are worthwhile given the performance benefits of Baraat. Further, the changes required at the switches are practically feasible, as demonstrated by prior proposals [28, 17] and our software-switch implementation. Specifically, in terms of switch state, PDQ [17] shows that switches only need 10KB memory to keep per-flow state (Baraat stores even less information i.e. per-task only). In §6, we also discuss how recent work may further simplify introducing Baraat-like functionality in future data centers.

We now elaborate on the most important parts of SPC—the classifier and how explicit feedback is calculated. Finally, we describe the rate control transport protocol that is used between end-hosts and switches.

4.3.1 Classifier

As shown in Figure 6b, the classifier maps flows to appropriate priority classes. It maintains a mapping between *task-id* and priority classes, so flows are first mapped to tasks (based on their *task-id*) and then to the relevant priority class. By default, the classifier maintains a *one-to-one* mapping between tasks and priority classes. The highest priority task maps to the highest priority class and so on. This achieves the standard FIFO scheduling where tasks are scheduled one by one based on their priority (*task-id*).

To support FIFO-LM, the classifier also does *on-the-fly* identification of heavy tasks (hence tasks need not know their size *upfront*). The classifier queries per-task counters to check whether a task is heavy or not. We maintain a counter that keeps track of the total bytes reserved by flows of a task, which we use as proxy for the task's size. If the task size exceeds a pre-determined threshold, the task is marked as heavy.[3] Subsequently, the heavy task and the task immediately next in priority to the heavy task share the same class. For example, as shown in the figure, if task 1 is identified as heavy, it shares the top priority class with task 2. This enables small tasks to make progress, so when task 2 finishes, task 3 is moved to the top priority queue.

In addition to supporting FIFO-LM, our classifier design provides two key advantages. First, it maps all flows of a task to the *same* class, which ensures that flows of the same task are active simultaneously, instead of being scheduled one-by-one (e.g., as in [17]), thereby reducing the overhead of flow switching. Second, it decouples classification from feedback computation and rate control protocol, which makes it easier to support other scheduling policies in the system. By just changing the way flows are mapped to classes, we can support policies like fair sharing, flow level prioritization, etc.

4.3.2 Explicit Feedback

After mapping a flow to its appropriate class, the next step is to compute the feedback, which corresponds to the rate that the switch can support for the given flow. As shown in Figure 6b, feedback is computed based on the flow's priority class and aggregate task counters.

The feedback provided to each flow roughly corresponds to the rate that the flow would get with standard priority queues. Thus, the entire link capacity is allocated to the highest priority class; any leftover is given to the next class, and so on. Whether the highest priority class is able to use the entire link capacity depends on its total demand, which in turn depends on how many tasks are mapped to

[2]We assume that end-hosts and switches are protocol compliant, a reasonable assumption for production data center environments. Further, compared to FIFO or sized based scheduling, FIFO-LM limits the impact of non-confirming sources (if any) by increasing the level of multiplexing.

[3]A heavy task on one switch may not be identified as heavy in some other part of the network (where it will not be causing head-of-line blocking).

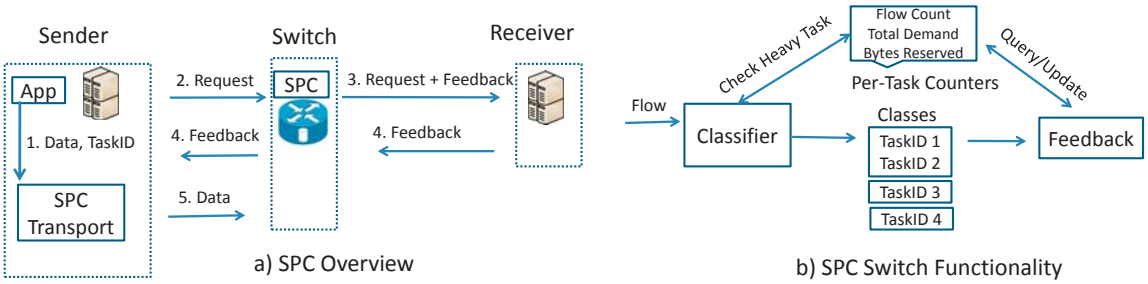

Figure 6: Overview of SPC and its switch functionality.

this class, and the aggregate demand of each of these tasks (stored in the per-task counters)[4]. If the total demand of the highest priority class is more than the available capacity[5], we assign a max-min fair bandwidth allocation. To this end, we maintain another per-task counter, which keeps track of the number of flows in the task.

Algorithm 1 Explicit Feedback Calculation

Require: TaskID, Demand for flow, D, and the allocation for the previous interval.
Output: AR, NR.
Link capacity is C; A reflects current allocations across all flows; D_k is the sum of flow demands belonging to class k; F_k is the number of flows belonging to class k.

1: Return Previous Allocation and Demand by subtracting from relevant counters
2: $k = Classifier(TaskID)$
3: $ClassAvlBW = max(0, C - \sum_{i=1}^{k-1} D_i)$
4: $AvailShare = max(0, ClassAvlBW - D_k)$
5: **if** $AvailShare \geq D$ **then**
6: $NR \leftarrow D$
7: **else**
8: $NR \leftarrow ClassAvlBW/F_k$
9: **end if**
10: $AR \leftarrow min(NR, C - A)$
11: Update Packet with AR and NR
12: Update counters and A

Algorithm 1 lists all the steps followed in computing feedback for a flow. A key aspect of our design is that unlike prior explicit feedback protocols that maintain per-flow state (e.g., PDQ [17]), we only use aggregate, per-task counters to compute feedback. Given the typical large number of flows per task, this can provide an order of magnitude or more reduction in the amount of state kept at the switches. However, without per-flow state, providing work-conservation becomes challenging, as switches no longer keep track of the bottleneck link of each flow.

We address this challenge by providing *two* rates to sources through the switch feedback. An *actual rate* (AR) at which senders should send data in the next RTT and a *nominal rate* (NR), which is the maximum share of the flow based on its priority. NR might differ from AR due to flow dynamics — the switch might have already assigned bandwidth to a lower priority flow which needs to be preempted before NR is available. NR essentially allows senders to identify their current nominal bottleneck share, which they use to adjust their demand, thus allowing other switches along the path to free up unused bandwidth and allocate it to lower-priority flows (i.e., work conservation). In the next section, we describe how end-host transport precisely calculates this demand.

[4]total demand corresponds to the *current* aggregate demand of a task while bytes-reserved is the sum of *all* prior reservations for the task.

[5]We adjust for over and under utilization of a link by using the notion of virtual capacity, which is increased or decreased depending on link utilization and queuing [13, 28].

4.3.3 *End-host Transport*

As shown in Figure 6a, SPC end-host transport logic leverages prior work on explicit rate based protocols [13, 17, 28]. The key difference is in the way senders calculate their *demand*. In SPC, demand calculation is more *accurate* and reflects not only how much data the sender can send (as is the case in prior work), but also how much data the bottleneck link on the path can support. A more accurate demand ensures that non-bottleneck switches only reserve what is required by the flow, which leads to work conservation in multi-hop settings, without requiring per-flow state at the switches.

Algorithm 2 Sender – Calculating Demand

1: MinNR - minimum NR returned in the previous RTT
2: $Demand_{t+1} \leftarrow min(NIC\_Rate, DataInBuffer/RTT)$ //if flow already setup
3: **if** $MinNR < Demand_t$ **then**
4: $Demand_{t+1} \leftarrow min(Demand_{t+1}, MinNR + \delta)$
5: **end if**

Algorithm 2 provide details on how senders calculate demand. The initial demand is set to the sender's NIC rate (e.g., 1Gbps) or lower if the sender has only a small amount of data to send (Step 2). Based on the earlier feedback from switches (the NR and AR vectors), the sender identifies the bottleneck rates: it transmits data at the minimum of the rates specified in the AR vector as discussed earlier, and uses the NR vector to determine how much it should demand in the next RTT. If the flow is bottlenecked on a network link (i.e., the minimum of the NR vector is less than the previously requested demand), the sender lowers its demand for the next RTT and sets it equal to $NR + \delta$. Lowering the demand allows other links to only allocate the necessary bandwidth that will actually be used by the flow, using the rest for lower priority flows (i.e., work conservation). Adding a small value (δ) ensures that whenever the

bottleneck link frees up, the sender recognizes this and is able to again increase its demand to the maximum level.

In addition to rate control, the SPC transport also implements other transport functionality, such as reliability and flow control. Note that due to the explicit nature of our protocol, loss should be rare, but end-hosts still need to provide reliability. Our reliability mechanism is similar to TCP. Each data packet has a sequence number, receivers send acknowledgments, and senders keep timers and retransmit, if they do not receive a timely acknowledgment.

4.4 Implementation

We have built a proof-of-concept switch and end-host implementation and have deployed it on a 25 node testbed. Both the switch and end-host are implemented on server-grade machines. We have also integrated Baraat with Memcached application. Both the end-host and switch implementations run in user-space and leverage zero-copy support between the kernel and user-space to keep the overhead low. At end-hosts, applications use an extended Sockets-like API to convey *task-id* information to the transport protocol. This information is passed when a new socket is created. The application also ensures that all flows per task use the same *task-id*. Our switch implementation is also efficient. On a server-grade PC, we can saturate four links at full duplex line rate. To keep SPC header processing overhead low in switches, we use integer arithmetic for rate calculations. Overall, the average SPC processing time was indistinguishable from normal packet forwarding. Thus, we believe that it will be feasible to implement Baraat's functionality in commodity switches.

Header: The SPC header requires 26 bytes. Each *task-id* is specified in 4 bytes. We encode rates as $Bytes/\mu s$. This allows us to a use a single byte to specify a rate – for example, 1Gbps has a value of 128. We use a scale factor byte that can be used to encode higher ranges. Most of the header space is dedicated for feedback from the switches. Each switch's response takes 2 bytes (one for NR and one for AR). Based on typical diameter of data center networks, the header allocates 12 bytes for the feedback, allowing a maximum of 6 switches to provide feedback. The sender returns its previous ARs assigned by each switch using 6 bytes. We also need an additional byte to keep track of the switch index – each switch increments it before forwarding the packet and uses 2 bytes to specify the current and previous demands.

5. EVALUATION

We evaluate Baraat across three platforms: our small scale testbed, an ns-2 implementation and a large-scale data center simulator. Our data center simulator allows us to implement a range of schemes, including the state-of-the-art flow-based protocol (i.e., pFabric), a prior task-aware centralized scheduler (i.e., Orchestra), idealized schemes (e.g., centralized scheduler with complete task size information), and compare their performance with Baraat. We use the ns-2 simulator to verify the basic properties of Baraat (e.g., work conservation, preemption, etc) and to quantify the overheads. Finally, our testbed allows us to evaluate Baraat with a real application (Memcached) and to cross validate the results of our simulator platforms. Here, we only present the key results from our testbed and large scale simulator experiments. Detailed results, including ns-2 micro-benchmarks, are available in the accompanying technical report [12].

	Avg	95^{th} perc.	99^{th} perc.
RCP	40ms	72ms	120ms
Baraat	29ms	41ms	68ms
Improvement	27%	43%	43.3%

Table 3: Performance comparison of Baraat against RCP in a Memcached usage scenario.

In our evaluation, we consider three varied workloads: search, data analytics, and applications with homogeneous network footprint. We also analyze Baraat's performance across three different workflows – partition-aggregate, storage retrieval and data parallel. In all our experiments, the primary metric for comparison is the *task completion time*. We consider both the average as well as the tail (95^{th} percentile and beyond) task completion time.

Below is a summary of our key results.

- Against decentralized schemes: Baraat reduces the 95^{th} percentile of the task completion time by 70%, 43% and 66% compared to the best known decentralized scheme (pFabric) for search, data-analytics and uniform workloads respectively.

- Against centralized schedulers: In a hypothetical scenario where centralized schedulers like Orchestra are made to handle search-like workloads, we show that Baraat's FIFO-LM policy reduces tail (95^{th} percentile) task completion time by 27% compared to Orchestra and by 84% compared to a size-aware scheduler. For data-analytics workloads, tail completion time reduces by 93% and 37% over Orchestra and size-aware schedulers respectively.

5.1 Testbed experiments

We evaluate Baraat on our testbed and demonstrate its use with Memcached. Our testbed has five racks with four nodes each. The racks are connected through a two level tree topology: each rack has a top-of-rack (TOR) switch and the TOR switches are connected through a root switch. All end-hosts and switches are Dell Precision T3500 servers with a quad core Intel Xeon 2.27GHz processor, 4GB RAM and 1 Gbps interfaces.

In addition to Baraat, we have also implemented an optimized version of RCP [13] on our testbed.[6] For the testbed experiments, we compare Baraat against RCP for a storage retrieval scenario whereby a client reads data from multiple storage servers in parallel. This represents a parallel workflow.

Online Data Retrieval with Memcached. Our Memcached setup mimics a typical web-service scenario. We have one rack dedicated to the front-end nodes (i.e., Memcached clients) while the four other racks are used as the Memcached caching back-end. The front-end comprises of four clients; each client maintains a separate counter that is used to assign a *task-id* to incoming requests. Each counter is

[6]We have introduced a number of optimizations to account for data center environments, such as information about the exact number of active flows at the router (RCP uses algorithms to approximate this). With our RCP implementation sources know exactly the rate they should transmit at, whereas probe-based protocols like TCP/DCTCP need to discover it. Hence, our RCP implementation can be considered as an upper-bound for fair-share protocols.

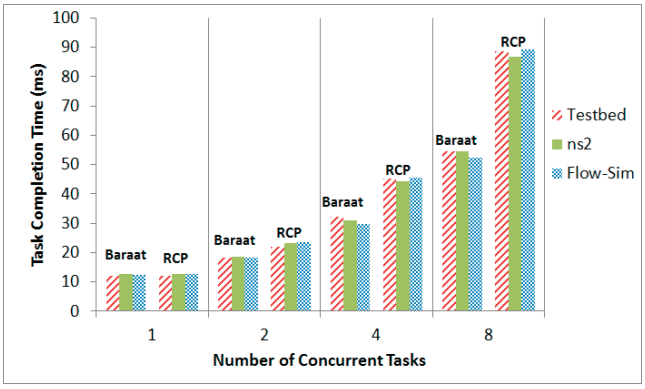

Figure 7: Baraat's performance against RCP for a parallel workflow scenario across all experimental platforms.

initialized to a unique value and is incremented by four for every incoming request. This models a scenario where requests arrive through multiple load-balancers (see §4.1).

For the experiment, we consider an online scenario where each client independently receives requests based on a Poisson arrival process. Each request (or task) corresponds to a multi-get that involves fetching data from three randomly chosen memcached servers. Table 3 compares Baraat with RCP for an experiment with 1000 requests, task size of 800KB, and an average client load of 50%. In this case, Baraat reduces average task completion time by 27% compared to RCP. We observe more gains at high percentiles where Baraat provides around 43% improvement over RCP.

Batched Requests. We now evaluate the impact of varying the number of concurrent tasks in the system and also use this experiment to cross-validate our testbed results (without Memcached) with the simulation platforms. This experiment is inspired by similar workloads considered in prior work to show incast in storage retrieval scenarios [24]. For this experiment, one node acts as a client while the other three nodes in the rack act as storage servers. All data is served from memory. For the request, the client retrieves 400 KB chunks from each of the three servers. The request finishes when data is received from all servers.

Figure 7 compares the performance of Baraat against RCP as we vary the number of concurrent tasks (i.e., read requests) in the system. Our results ignore the overhead of requesting the data which is the same for both Baraat and RCP. The first bar in each set shows testbed results. For a single task, Baraat and RCP perform the same. However, as the number of concurrent tasks increases, Baraat starts to outperform RCP. For 8 concurrent tasks, Baraat reduces the average task completion time by almost 40%. The experiment also shows that our implementation is able to saturate the network link — a single task takes approximately 12msec to complete, which is equal to the sum of the task transmission time ($\frac{1.2MB}{1Gbps}$) and the protocol overhead (2 RTTs of 1msec in our testbed).

Cross-validation and comparison with optimal. We repeated the same experiment in the ns-2 and large-scale simulators. Figure 7 also shows that the results are similar across the three platforms; absolute task completion times across our testbed and simulation platforms differ at most by 5%. This establishes the fidelity of our simulators which we use for more detailed evaluation in the following sections. We also used simple brute-force approach for this experiment and found that Baraat's schedules are optimal. In general, this is a strong NP-hard problem [15], so computing optimal for even modest-sized experiments is infeasible.

5.2 Large-scale performance

To evaluate Baraat at large scale, we developed a simulator that coarsely models a typical data center. The simulator uses a three-level tree topology with no path diversity, where racks of 40 machines with 1Gbps links are connected to a Top-of-Rack (ToR) switch and then to an aggregation switch. By varying the connectivity and the bandwidth of the links between the switches, we vary the over-subscription of the physical network. We model a data center with 36,000 physical servers organized in 30 pods, each comprising 30 racks.

Each simulated task involves workers and one or more layers of aggregators. The simulator can thus model different task workflows and task arrival patterns. We evaluate Baraat's performance under three different workloads. The first two workloads are based on the Bing and Facebook traces discussed earlier (§2) while the third one models a more homogeneous application with flow sizes that are uniformly distributed across [2KB, 50KB] (as suggested in prior work [4, 28, 17]). For single-stage workloads, the aggregator queries all other nodes in the rack, so each task comprises 40 flows. For two-stage workloads, the top-level aggregator queries 30 mid-level aggregators located in separate racks, which in turn query 40 workers each, resulting in 1200 flows per task. Top-level aggregators are located in a separate rack, one per pod. We use network over-subscription of 2:1 at the ToR switch and a selectivity of 3% (data input-to-output ratio for aggregator nodes), which is consistent with observations of live systems for data aggregation tasks [9]. We examine other configurations towards the end of the section.

Over the following sections, we first compare Baraat's performance against various decentralized and centralized solutions. We then dive into Baraat's performance for various workflows and explore a series of parameters that affect performance.

5.2.1 *Comparison against Existing Solutions*

We first compare Baraat against decentralized solutions – such solutions can handle scenarios involving short flows but are not task-aware. We then compare against centralized solutions – while these solutions are task-aware, they are not amenable to handling short flow scenarios.

Comparison against Decentralized Solutions. We consider two flow-based schemes: i) pFabric [5], which is the state-of-the-art transport protocol with the best reported performance. Our simulator models the best-case performance of pFabric because it implements the shortest flow first scheduling policy with no protocol overhead; and ii) RCP [13], which represents the best case performance of any fair sharing scheme (i.e., TCP/DCTCP, etc). Fair sharing schemes present a good baseline for comparison and can also surprisingly do quite well under certain scenarios. Like Baraat, both these schemes are decentralized, and hence suitable for scenarios involving small tasks (e.g., search).

Figure 8 highlights the reduction in tail task completion time (95^{th} percentile) with Baraat relative to pFabric and

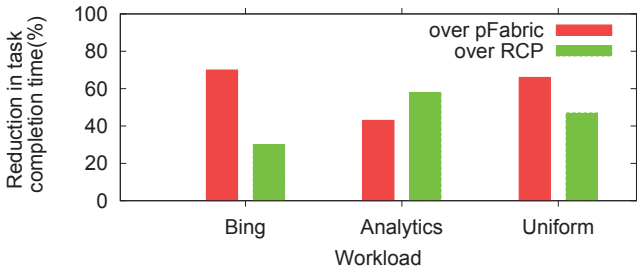

Figure 8: Reduction in tail task completion time with Baraat against decentralized schemes.

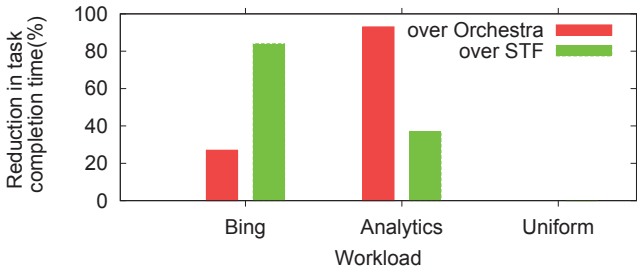

Figure 9: Reduction in tail task completion time with Baraat against centralized schemes.

RCP for the three workloads. The results reflect the execution of 10,000 tasks for 80% data center load, which captures the average load of bottlenecked network links. We examine how load and other parameters affect results in the following section. For all workloads, Baraat significantly improves task completion times compared to the other two decentralized solutions: The 95^{th} percentile task completion time reduces by 70%, 43% and 66% over pFabric for Bing, data-analytics and uniform workloads respectively (30%, 58% and 47% over RCP).

Comparison against Centralized Solutions. While making centralized solutions work for short tasks is still an open problem, we consider the hypothetical scenario in which this is feasible and evaluate their performance. We compare Baraat against two centralized, task-aware solutions: Orchestra [11], which schedules tasks in a FIFO order, and ii) Shortest Task First (STF), which determines the ordering of tasks based on their sizes (assuming an idealized scenario where the centralized scheduler has the whole task size information upfront).

We use the previous experiment settings and again compare performance at the 95^{th} percentile task completion time. Figure 9 shows that even though Baraat is decentralized, it is able to outperform centralized schedulers – compared to Orchestra, it provides 27% and 93% reduction in task completion completion for the Bing and data-analytics workloads, respectively. Similar gains of 84% and 37% are achieved for the same workloads in comparison with STF. For uniform workloads, all centralized solutions perform the same, as they all schedule tasks in a task-aware fashion and for this workload all tasks are similar in size.

Detailed Analysis. To understand the gains of Baraat against both centralized and decentralized solutions, we now examine the results in more detail. Table 4 reports the median, 95^{th} percentile, and 99^{th} percentile task completion time reduction with Baraat compared to all other schemes.

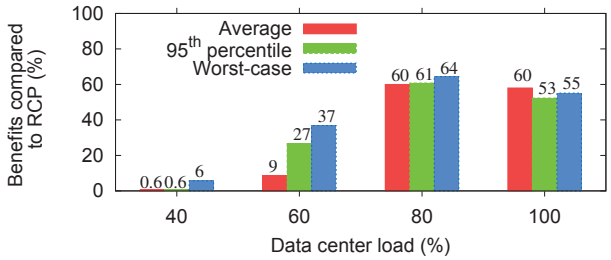

Figure 10: Reduction in task completion time for the partition-aggregate workflow.

For the Bing-like workload, we observe that Baraat outperforms Orchestra at both the median and higher percentiles. This is because of head-of-line blocking with Orchestra's FIFO based scheduling. Other schemes (pFabric, RCP) perform close to Baraat at the median (or even slightly better in the case of RCP) because they allow small tasks to finish quicker while Baraat may give only a fixed fraction of link bandwidth to small tasks (i.e., limited multiplexing). However, the difference in performance is small and at higher percentiles Baraat provides significant gains over these schemes.

For data analytics workloads exhibiting heavy-tailed distributions (Table 4), Orchestra again suffers from head-of-line blocking. In this case, size-based policies (pFabric, STF) do result in reduction of completion time compared to fair-sharing policies like RCP, especially beyond the median up to the 95^{th} percentile. However, even in this case, Baraat's FIFO-LM policy results in improved performance of roughly 60% relative to RCP and 36% over size-based policies at the 95^{th} percentile.

For uniform workloads, Baraat and task-aware policies (Orchestra and STF) have similar performance. Note that due to the absence of heavy tasks in this workload, Baraat and Orchestra collapse to the same policy. However, compared to pFabric, Baraat provides reduction of 66% and 75% at the 95^{th} and 99^{th} percentiles, respectively.

In summary, our results highlight that compared to existing solutions, Baraat can reduce the task completion time both at the median and at the tail, and for a wide range of workloads (uniform, bi-modal, heavy-tailed, etc).

5.2.2 *Varying Workflows*

We now look at Baraat's performance under the different workflows described in Figure 1 in §2. In particular, we examine three workflows – (i) a two-level partition aggregate workflow where requests arrive in an online fashion, (ii) the storage retrieval scenario used for our testbed experiments where tasks have parallel workflows and request arrival is online, and (iii) a data-parallel application where tasks have a parallel workflow and there is a batch of jobs to execute. For this set of experiments, our focus is on the different workflows, so we keep the workload static (i.e., homogeneous workload described in the previous section) and use RCP as the baseline for comparison.

Figure 10 plots the reduction in the task completion time with Baraat compared to RCP for the partition-aggregate workflow. As expected, the benefits increase with the load – at 80% load, the worst case task completion time reduces by 64%, while the average and 95^{th} percentile by 60% and 61% respectively. In all cases, the confidence intervals for the values provided are less than 10% within the mean, and

	Bing			Data-analytics			Uniform		
Policy	median	95^{th} perc.	99^{th} perc.	median	95^{th} perc.	99^{th} perc.	median	95^{th} perc.	99^{th} perc.
RCP	-5%	30%	34%	25%	58%	62%	7%	37%	40%
pFabric	4%	70%	76%	4%	43%	61%	38%	66%	75%
STF	-8%	84%	97%	0	37%	64%	0	1%	6%
Orchestra	28%	27%	16%	94%	93%	84%	0	0	0

Table 4: Reduction in task completion time with Baraat relative to other policies.

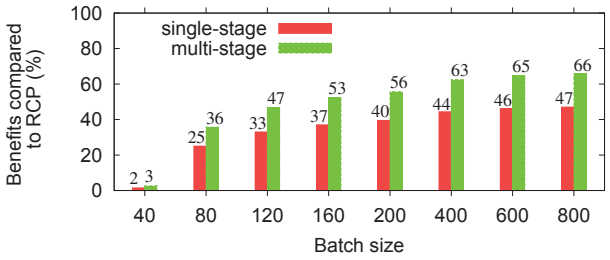

Figure 11: Reduction in mean task completion time for data-parallel jobs.

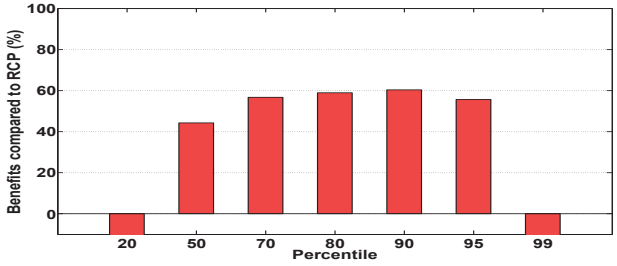

Figure 12: Performance of Baraat when varying the heavy-task identification threshold for the data analytics workload.

are not plotted for clarity. For the storage retrieval scenario, the worst case completion time reduces by 36% compared to RCP at 80% load (35% and 16% reduction at 95^{th} percentile and the average respectively). The reduced benefit results from the fact that tasks here involve a single stage.

Figure 11 presents Baraat's benefits for the scenario involving a batch of data-parallel jobs. For batch sizes of 400 jobs, average task completion time is reduced by 44% and 63% for single-stage and multi-stage jobs respectively. As discussed in §2, batch execution scenarios involving single-stage jobs only provide benefits at the average. For multiple stages, worst case completion time also drops beyond batch sizes of 40; for batch sizes of 400, worst case completion time reduces by 32%.

5.2.3 *Varying parameters*

We now examine how varying the experiment parameters affect performance. We will focus on the partition-aggregate workflow at 80% load. We present only an overview of our observations, which are based on Baraat's performance in comparison with RCP. Detailed results are available in the technical report [12].

Adding computation. While our paper focuses on network performance, we now consider tasks featuring both network transfers and computation. We extend the simulator to model computation for worker machines as an exponentially distributed wait time. As expected, as computation time increases as a fraction of the total task completion time, the benefits of Baraat drop. However, overall Baraat still provides significant benefits. For example, at 80% load and when computation comprises 50% of the task, the worst case completion time reduces by 25% and the average completion time reduces by 14% (50% reduction when computation comprises 25% of the task completion time).

Heavy task identification. As discussed in §4, classification is threshold-based in Baraat. For heavy-tailed distributions (e.g., Pareto or Log-normal), we found that a threshold in the range between the 80^{th} and the 90^{th} percentile provides the best performance. Figure 12 highlights this for the data analytics workload. Note that while Baraat shows robust performance under a wide range of thresholds, a significant mis-estimation of the threshold can severely affect performance: if the threshold is too low, the policy will converge to fair-sharing (RCP) and if the threshold is too high, it will converge to Orchestra.

6. DISCUSSION

The notion of task-aware scheduling underlying Baraat is both affected by and has implications for various aspects of data center network design. We briefly discuss some important issues here.

Changes to the Network Fabric. While Baraat's mechanism (SPC) requires light-weight changes to the network switches, recent work may make it feasible to support task-aware scheduling in a more non-intrusive manner. For example, PASE [20] supports flow-based scheduling using existing data plane functionality (i.e., priority queues and ECN) but cannot support other Baraat requirements such as heavy task identification. Similarly, proposals for programmable switches[19] may make it easier to introduce Baraat-like functionality in future.

Incremental Deployment. This paper focused on the scenario where applications and network elements are both Baraat compliant. We believe that there are potential benefits of incrementally rolling-out the system. From an applications perspective, Baraat can be incrementally deployed by considering non-Baraat traffic as background traffic on a link. From a network perspective, Baraat can be deployed on a subset of switches (e.g., close to the aggregators) to gain partial benefits. We leave the exploration of these benefits for future work.

Non-network resources. Baraat reduces network contention through task serialization. However, it still retains pipelined use of other data center resources. Consider a web search example scenario where an aggregator receives responses from a few workers. Today, either the CPU or the network link at the aggregator will be the bottleneck resource. Baraat is work conserving, so it will ensure the fewest number of simultaneously active tasks that can ensure that either the aggregator's network link is fully utilized or the CPU at the aggregator is the bottleneck. Thus, Baraat does not adversely impact the utilization of non-network resources. While additional gains can be had from coordi-

nated task-aware scheduling across multiple resources, we leave this to future work.

7. RELATED WORK

We briefly discuss work that is most relevant to Baraat.

Task-Aware Schedulers and Network Abstractions. Proposals that most closely relate to Baraat include Orchestra [11] and CoFlow [10] which also argue for bringing task-awareness in data centers. Orchestra focuses on how task-awareness could provide benefits for MapReduce style workloads. Baraat differs from Orchestra in three ways. First, Baraat makes the scheduling decisions in a *decentralized* fashion rather than through a centralized controller. Second, Baraat uses FIFO-LM which has not been considered in prior network scheduling proposals including Orchestra. Third, while Orchestra focuses on improvement in the average task completion time for batched workload, we show that we can also improve the tail completion time, for dynamic scenarios and multi-stage workloads.

CoFlow [10] focuses on a new abstraction that can capture rich task semantics, which is orthogonal to Baraat's focus on scheduling policy and the underlying mechanism. However, beyond the abstraction, CoFlow does not propose any new scheduling policy or mechanism to achieve task-awareness.

Cluster Schedulers and Resource Managers. There is a large body of work on *centralized* cluster schedulers and resource managers [16, 23, 11, 1]. Some of these proposals use policies similar to FIFO-LM to balance fairness and performance in their systems. For example, the Hadoop fair scheduler [1] allows limited multiplexing, although the limit is set by the user/administrator. However, the above proposals focus on scheduling jobs on machines and not on decentralized scheduling of flows (or tasks) over the network, which requires new mechanisms.

Straggler Mitigation Techniques. Many prior proposals attempt to improve task completion times through various straggler mitigation techniques (e.g., re-issuing the request) [7, 18]. These techniques are orthogonal to our work as they focus on non-scheduling delays, such as delays caused by slow machines or failures, while we focus on the delays due to the resource sharing policy.

8. CONCLUSIONS

Baraat is a decentralized system for task-aware network scheduling. It provides a consistent treatment to all flows of a task, both across space and time. This allows active flows of the task to be loosely synchronized and make progress at the same time. By changing the level of multiplexing, Baraat effectively deals with the presence of heavy tasks and thus provides benefits for a wide range of workloads. Our experiments show that Baraat can significantly reduce the average as well as tail task completion time.

Acknowledgements: We thank our shepherd, Amin Vahdat, and the SIGCOMM reviewers for their feedback. We are also grateful to the Kwiken [18] team, especially Ishai Menache and Virajith Jalaparti, for sharing their Bing traces.

9. REFERENCES

[1] The Hadoop Fair Scheduler. `https://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html`.

[2] Memcached. `http://memcached.org/`.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proc. of NSDI*, pages 19–19, 2010.

[4] M. Alizadeh, A. G. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proc. of SIGCOMM*, 2010.

[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] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: coordinated memory caching for parallel jobs. In *Proc. of NSDI*, 2012.

[7] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proc. of OSDI*, 2010.

[8] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Scale-up vs scale-out for hadoop: Time to rethink? In *Proc. of SOCC*, 2013.

[9] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The case for evaluating mapreduce performance using workload suites. In *Proc. of MASCOTS*, 2011.

[10] M. Chowdhury and I. Stoica. Coflow: An application layer abstraction for cluster networking. In *ACM Hotnets*, 2012.

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

[12] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized Task-aware Scheduling for Data Center Networks. Technical Report MSR-TR-2013-96, Microsoft Research, September 2013. `http://research.microsoft.com/apps/pubs/?id=201494`.

[13] N. Dukkipati. *Rate Control Protocol (RCP): Congestion control to make flows complete quickly*. PhD thesis, Stanford University, 2007.

[14] M. Garey and D. Johnson. Computers and intractability. 1979.

[15] L. Hall. Approximability of flow shop scheduling. *Mathematical Programming*, 82(1):175–190, 1998.

[16] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proc. of NSDI*, pages 22–22. USENIX Association, 2011.

[17] C. Hong, M. Caesar, and P. Godfrey. Finishing flows quickly with preemptive scheduling. *ACM SIGCOMM Computer Communication Review*, 42(4):127–138, 2012.

[18] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. In *Proc. of SIGCOMM*, 2013.

[19] V. Jeyakumar, M. Alizadeh, C. Kim, and D. Mazières. Tiny packet programs for low-latency network control and monitoring. In *Proceedings of Hotnets*, HotNets-XII. ACM, 2013.

[20] A. Munir, G. Baig, S. Irteza, I. Qazi, I. Liu, and F. Dogar. Friends, not foes — synthesizing existing transport strategies for data center networks. In *Proc. of SIGCOMM*, 2014.

[21] J. Nair, A. Wierman, and B. Zwart. Tail-robust scheduling via limited processor sharing. *Performance Evaluation*, 67(11):978–995, 2010.

[22] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at facebook. In *Proc. of NSDI*, 2013.

[23] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proc. of SOSP*, 2013.

[24] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan. Measurement and analysis of tcp throughput collapse in cluster-based storage systems. In *FAST*, volume 8, pages 1–14, 2008.

[25] H. Röck. The three-machine no-wait flow shop is np-complete. *Journal of the ACM*, 31(2):336–345, 1984.

[26] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the data center network. In *Proc. of NSDI*, 2011.

[27] A. Wierman and B. Zwart. Is tail-optimal scheduling possible? *Operations Research*, 60(5):1249–1257, 2012.

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

[29] H. Yu, A. Vahdat, et al. Efficient numerical error bounding for replicated network services. In *Proc. of VLDB*, 2000.