Decentralized Task-Aware Scheduling for Data Center Networks

Fahad R. Dogar, Thomas Karagiannis, Hitesh Ballani, and Ant Rowstron
Microsoft Research

Abstract

Most data center applications perform rich and complex tasks (e.g., executing a search query or generating a user's wall). From a network perspective, these tasks typically comprise multiple flows, which traverse different parts of the network at potentially different times. Existing 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. Based on the network footprint of real applications, we motivate the use of a scheduling policy that dynamically adapts the level of multiplexing in the network. To apply task-aware scheduling to online applications with small (sub-second) tasks, we design and implement Baraat, a *decentralized* task-aware scheduling system. Through experiments with Memcached on a small testbed and large-scale simulations, we show that, compared to existing schemes, Baraat can reduce tail task completion times by 60% for data analytics workloads and 40% for search workloads.

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 cache [7], network bandwidth [11], and CPUs and network [6].

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 common problems associated with centralized scheduling (i.e., scalability, fault-tolerance, etc) while addressing the key challenge 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 relatively well-understood flow prioritization problem. While many flow prioritization mechanisms exist (e.g., priority queues, PDQ [16], D^3 [25], 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 [16, 12, 25]. 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 and data-analytics applications; the ns-2 simulator for conducting micro-benchmarks. We have also integrated the popular in-memory caching application, Memcached[1], with Baraat. Our results show that Baraat reduces tail task completion time by 60%, 30% and 40% for data-analytics, search and homogeneous workloads respectively compared to fair-sharing policies, and by over 70% compared to size-based policies. Besides the tail, Baraat further reduces average completion time for batched data parallel jobs by 30%-60% depending on the configuration.

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,

[1]http://memcached.org/

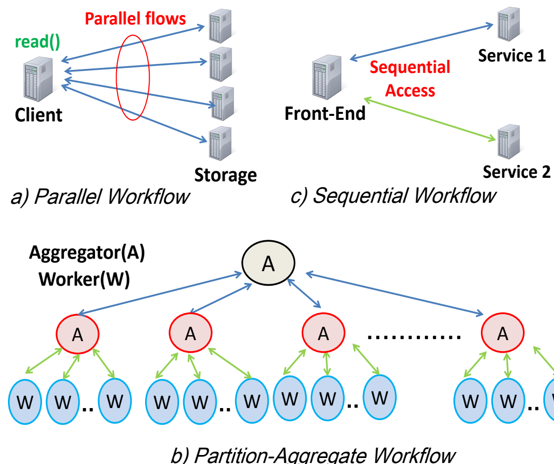

Figure 1: Common workflows.

these applications run on many servers that, in order to respond 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 wall 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.

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 typical prominent applications, namely web search and data analytics. Figure 2 (left) presents the normalized distribution of task sizes for the query-response

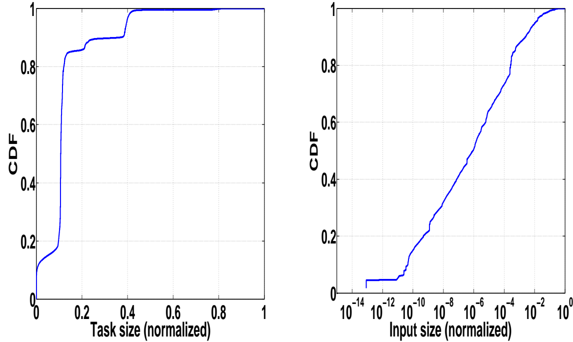

Figure 2: Normalized distribution of task sizes for search (left), data analytics (right) workflows.

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 [17]. 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, 7]. 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 a number of applications in various production data centers. Flows per task can range from a few tens, to hundreds or thousands, and as discussed, subsets of flows can be active at different times and across different parts of the network.

Implications for 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, 17], Cosmos [6], Facebook [19]) which typically have modest average data

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 [7]	30 (lower-bound)	Job contains 30 mappers/reducers at the median, 50000 at the maximum.
Cosmos [23]	55	70% of tasks involve 30-100 flows, 2% involve more than 150 flows

Table 1: Tasks in data centers comprise multiple flows.

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. Transport 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 [25, 16, 5]. For example, PDQ [16] 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, tasks for typical data center applications can comprise hundreds of 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) 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 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

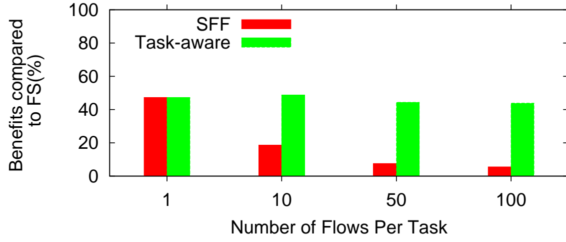

Figure 3: SFF fails to improve on fair sharing for realistic number of flows per task while a task-aware policy provides consistent benefits.

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.

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 [13, 22], 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 [14]. 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. 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.

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, we consider serving tasks 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).

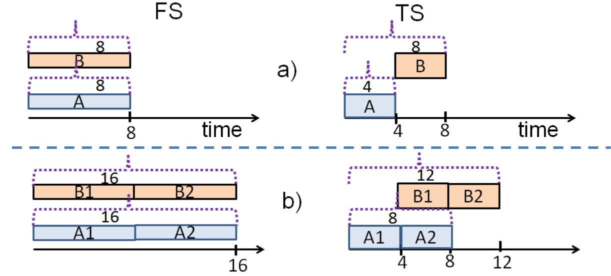

Figure 4: Distilling the Benefits of Task Serialization (TS) over Fair Sharing (FS).

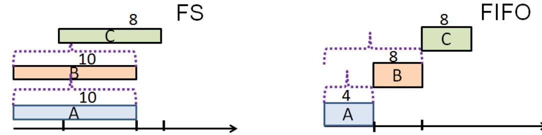

Figure 5: FIFO ordering can reduce tail completion times compared to fair sharing (FS).

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.

We now consider an application with two stages (Fig 4b), as in the partition-aggregate workflow of search. We consider a different network bottleneck for each of the two stages – for example, downlink to the mid-level aggregator in the first stage and downlink to the top-level aggregator in the second. 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.

Next, we consider specific policies that can achieve task serialization.

3.2 Task Serialization Policies

We begin with two obvious policies for task serialization: FIFO which schedules tasks in their arrival order and STF (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 [24]. 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[2], 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 bi-modal 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. For heavy-tailed distributions, our experimental evaluation with a number of heavy-tailed distributions such as Pareto or Log-normal with varying parameters (shape or mean respectively), shows that a threshold in the range of 80th-90th percentile provides the best results.

[2]typically referred to as limited processor sharing in scheduling theory [18].

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. This ensures that switches 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 metadata 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

	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	Yes	Yes	No	Partial	Yes
PDQ	Yes	No	No	Yes	Yes

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

a result, one of them generates odd *task-ids* (1, 3, 5,...) while the other generates even *task-ids* (2, 4, 6...). This approximates a FIFO ordering in a distributed scenario. These counters can be loosely synchronized and any inconsistency between them could be controlled and limited through existing techniques [26]. To deal with wrap-around, counters are periodically reset when the system load is low.

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 online service while *task-ids* greater than *n* could be used for the background service.

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 priortization mechanisms (e.g., priority queues, pFabric [5], PDQ [16], etc) at the switches and endpoints. While these mechanism 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 two properties, *strict priority* and *fair-sharing*, are 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. These two building blocks are effectively combined to support FIFO-LM through the third property – *handling heavy tasks*, which involves on-the-fly identification of heavy tasks and then changing 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 [16] does not support fair-sharing of bandwidth, so two flows having the same priority are scheduled in a serial fashion. Similarly, pFabric [5] does 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. Thus, we need additional functionality (e.g., explicit feedback from switches) to support work-conservation in multi-hop settings.

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

4.3 Smart Priority Class

SPC 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, SPC differs from traditional priority queues in two aspects: i) it employs an *explicit rate based protocol*: switches assign rates to each flow and end-hosts send at the assigned rate; ii) each switch has a *classifier* that maps flows to classes

and is responsible for handling heavy tasks. A key aspect of SPC design is that we mitigate two sources of overhead present in prior explicit rate protocols – the high flow switching overhead and the need to keep per-flow state at the switches.

Classifier: 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. The decision to map all flows of a task to the same class ensures that flows of the same task are active simultaneously, instead of being scheduled one-by-one [16], thereby reducing the overhead of flow switching.

The classifier also does *on-the-fly* identification of heavy tasks, hence tasks need not know their size upfront. The classifier keeps a running count of the size of each active task and uses the aggregate bytes reserved by flows of a task as a proxy for its current size. If the task size exceeds a pre-determined threshold, the task is marked as heavy. Subsequently, the heavy task and the task immediately next in priority to the heavy task share the same class. Finally, by just changing the way flows are mapped to classes, we can support other scheduling policies (e.g., fair sharing, flow level prioritization, etc).

Explicit Rate Protocol: Similar to existing explicit rate protocols [12, 16, 25], switches assign a rate to each flow and senders sent data at that rate.[3] However, instead of keeping per-flow state at the switches, we only maintain aggregate, *per-task* counters. Given the typical large number of flows per task, this can provide an order of magnitude or more reduction in the 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 through a combination of two techniques: First, switches provide *richer feedback* to sources. Switches inform senders about two types of rates. 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 bottlenck share. Second, our mechanism puts *increased responsibility on end-points*. Instead of just asking for their maximum demand, the end-host demands intelligently, keeping in view the feedback from the switches. While the sender initially conveys its maximum demand, it lowers it for the next RTT, if it is bottlenecked at some switch. This allows the non-bottlenecked links to free up the unused bandwidth and use it for some low priority flow.

[3] We assume that end-hosts and switches are protocol compliant, a reasonable assumption for production data center environments.

We now describe the details of our explicit rate protocol, focusing on the key operations that end-hosts and switches need to perform.

Algorithm 1 Sender – Generating SRQ

1: MinNR - minimum NR returned by SRX
2: $Demand_{t+1} \leftarrow min(NIC_Rate, DataInBuffer \times RTT)$ //if flow already setup
3: **if** $MinNR < Demand_t$ **then**
4: $\quad Demand_{t+1} \leftarrow min(Demand_{t+1}, MinNR + \delta)$
5: **end if**

Algorithm 2 Switch - SRQ Processing

1: Return Previous Allocation and Demand
2: $Class = Classifier(TaskID)$
3: $ClassAvlBW = C - Demand(HigherPrioClasses)$
4: $AvailShare = ClassAvlBW - Demand(MyClass)$
5: **if** $AvailShare > CurrentDemand$ **then**
6: $\quad NominalRate(NR) \leftarrow CurrentDemand$
7: **else**
8: $\quad NR \leftarrow ClassAvlBW / NumFlows(MyClass)$
9: **end if**
10: **if** $(C - Allocation) > NR$ **then**
11: $\quad ActualRate(AR) \leftarrow NR$
12: **else**
13: $\quad AR \leftarrow (C - Allocation)$
14: **end if**
15: Update Packet with AR and NR
16: Update local info – Demand(MyClass), BytesReserved(TaskID), and Allocation

4.3.1 End-host Operations

Every round-trip-time (RTT), the sender transmits a scheduling request message (SRQ), either as a standalone packet or piggy-backed onto a data packet. The most important part of SRQ is the *demand*, which conveys the sender's desired rate for the next RTT.

Algorithm 1 outlines the key steps followed in generating an SRQ. 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). In addition to the demand, the SRQ also contains other information, including the *task-id*, previous demand, and allocations made by the switches in the previous round. We later explain how this information is used by the switches.

Based on the response (SRX), the sender identifies the bottleneck rates: it transmits data at AR and uses NR to determine how much it should demand in the next

RTT. If the flow is bottlenecked on a network link, 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.

4.3.2 Switch Operations

We explain how switches process SRQ; the key steps are outlined in Algorithm 2. Each switch locally maintains three counters for each task: i) total demand, ii) total bytes reserved so far (this acts as a proxy for the size of the task), iii) number of flows in the task. In addition, the switch maintains a single aggregate counter for each link that keeps track of the bandwidth allocations that have already been made.

As noted earlier, the SRQ response consists of two pieces of information: The first is NR, which is the bandwidth share of the flow, based on its relative priority vis-a-vis the other flows traversing the switch. To calculate NR of a new flow with class k, a switch needs to know two things: i) demands of flows belonging to *higher priority classes* i.e., those with priority $> k$; these flows have strictly higher priority, so we subtract their demand from the link capacity (C), giving us *ClassAvlBw*, the amount of bandwidth available for class k (Step 2), and ii) demands of flows of the *same task* i.e., task id k; these flows have the same priority and thus share *ClassAvlBw* with the new flow.

Even if a flow's share is positive, the actual rate at which it can send data may be lower because the switch may have already reserved bandwidth for a lower priority flow *before* the arrival of the current flow. In this case, the switch needs to first preempt that flow, take back the bandwidth and then assign it to the higher priority flow. Thus, each switch also informs the sender about AR, which is the actual rate the switch can support in the next RTT. It is equal to or lower than the flow's NR. The switch adds these two rates to the SRQ before sending it to the next hop.

Finally, switches play a key role in supporting preemption. While calculating NR, the switch ignores demands of lower priority flows, implicitly considering them as preemptable. Of course, a flow that gets preempted needs to be informed so it can stop sending data. This flow switching can take 1-2 RTTs (typical task setup overhead in Baraat). Finally, the protocol adjusts to 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 [12, 25].

4.4 Implementation

We have built a proof-of-concept switch and end-host implementation and have deployed it on a 25 node testbed. We have also integrated Baraat with Memcached application. While we focused on the prioritization mechanism in the previous sections, our implementation supports complete transport functionality that is required for reliable end-to-end communication. 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*. In addition to the rate control protocol discussed earlier, end-hosts also implement 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.

Our switch implementation is also efficient. On a server-grade PC, we can saturate four links at full duplex line rate. To keep per-SRQ overhead low in switches, we use integer arithmetic for rate calculations. Overall, the average SRQ 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 SRQ/SRX 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 sending the SRQ message 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

	Avg	Min	95th perc.	99th perc.
FS	40ms	11ms	72ms	120ms
Baraat	29ms	11ms	41ms	68ms
Improvement	27%	0	43%	43.3%

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

data center simulator. In all experiments, we use flow-level fair sharing (FS) of the network as a baseline since it represents the operation of transport protocols like TCP and DCTCP used in today's data centers. Further, completion time of tasks is the primary metric for comparison. In summary, we find–

Testbed experiments. For data retrievals with Memcached, Baraat reduces tail task completion time by 43% compared to fair-sharing. The testbed experiments are also used to cross-validate the correctness of our protocol implementation in the ns-2 and large-scale simulators.

Large-scale simulations. We evaluate Baraat at data-center scale and compare its performance against various flow and task aware policies based on the workloads in §2. We show that Baraat reduces tail task completion time by 60%, 30% and 40% for data-analytics, search and homogeneous workloads respectively compared to fair-sharing policies, and by over 70% compared to size-based policies. We also analyze Baraat's performance across three different workflows – partition-aggregate, storage retrieval and data parallel.

ns-2 micro-benchmarks. We have used ns-2 to benchmark Baraat's protocol performance under various scenarios. With the help of controlled experiments, we have validated the correctness of our protocol and verified that it achieves the desired properties (e.g., work-conservation and preemption). We have also evaluated the impact of short flows and tiny tasks on Baraat's performance.

5.1 Testbed experiments

For the testbed experiments, we model a storage retrieval scenario whereby a client reads data from multiple storage servers in parallel. This represents a parallel workflow. To achieve this, we arrange the testbed nodes in five racks, each with four nodes.

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 backend. 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 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. A new request is queued if the client is busy serving another request. Each request (or task) corresponds to a multi-get that involves fetching data from one or more memcached servers.

We compare Baraat's performance against FS. For FS, we use an optimized version of RCP [12].[4] Table 3 shows results of 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 FS. We observe more gains at high percentiles where Baraat provides around 43% improvement over FS. Finally, the minimum task completion time is the same for both FS and Baraat, which verifies that both perform the same when there is just a single task in the system.

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. 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 6 compares the performance of Baraat against FS 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 FS. The first bar in each set shows testbed results. For a single task, Baraat and FS perform the same. However, as the number of concurrent tasks increases, Baraat starts to outperform FS. 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. We repeated the same experiment in the ns-2 and large-scale simulators. Figure 6 also

[4]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.

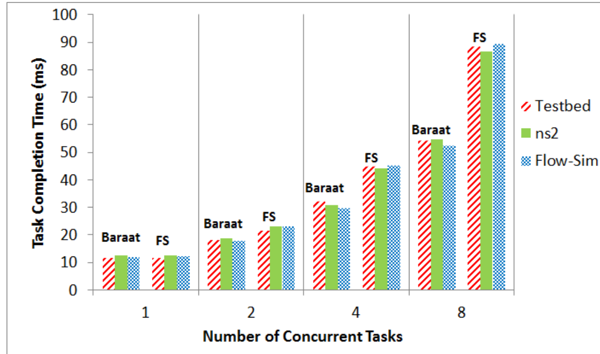

Figure 6: Baraat's performance against FS for a parallel workflow scenario across all experimental platforms. At 8 concurrent tasks, average task completion time reduces by roughly 40% with Baraat. Absolute task completion times are similar across platforms, thus cross-validating the ns-2 and flow-based simulations.

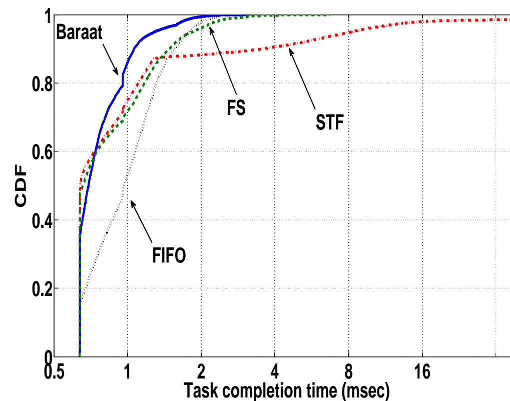

Figure 7: Aggregate CDF of task completion times for a Bing-like workload (*x-axis* in log scale.)

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.

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 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.

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, 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 and a selectivity of 3%, 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 examine the effectiveness of different scheduling policies under various distributions of task sizes, and then analyze the expected benefits of Baraat across a number of application workflows and parameters.

5.2.1 Evaluation of policies

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, 25, 16]).

We compare performance of Baraat (i.e., FIFO-LM) against four other scheduling policies – two flow-based policies (FS and SFF) and two task-aware policies (FIFO and STF). We model one stage where workers generate responses to aggregators. We report results for the execution of 10,000 tasks and for an 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.

Table 4 summarizes the results for the median, 95th and the 99th percentile of the task completion times of Baraat relative to all other policies for the three workloads. *In all cases, Baraat significantly improves task completion times compared to all other policies, especially towards the tail of the distributions, and as distributions become more heavy-tailed.*

For Bing-like workloads (Figure 7), all policies are comparable till roughly the 70th percentile at which point size-based policies (i.e., SFF & STF) start penalizing heavier tasks, leading a number of tasks to starvation. For data-analytics workloads exhibiting heavy-tailed distributions, FIFO's performance suffers from head-of-line blocking. In this case, size-based policies do result in reduction of completion time compared to FS, especially beyond the median up to the 95th percentile. However, even in this case, Baraat's FIFO-LM policy results in improved performance of roughly 60% relative to FS and

Policy	Bing			Data-analytics			Uniform		
	median	95th perc.	99th perc.	median	95th perc.	99th perc.	median	95th perc.	99th perc.
FS	1.05	0.7	0.66	0.75	0.42	0.38	0.93	0.63	0.6
SFF	0.96	0.3	0.24	0.96	0.57	0.39	0.62	0.34	0.25
STF	1.08	0.16	0.03	1	0.63	0.34	1	0.99	0.94
FIF0	0.72	0.73	0.84	0.06	0.07	0.16	1	1	1

Table 4: Task completion times with Baraat relative to other policies.

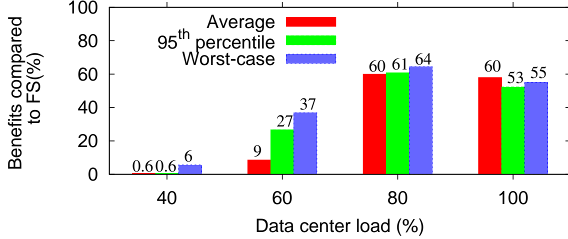

Figure 8: Reduction in task completion time for the partition-aggregate workflow.

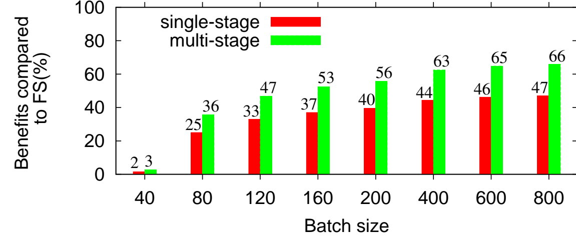

Figure 9: Reduction in mean task completion time for data-parallel jobs.

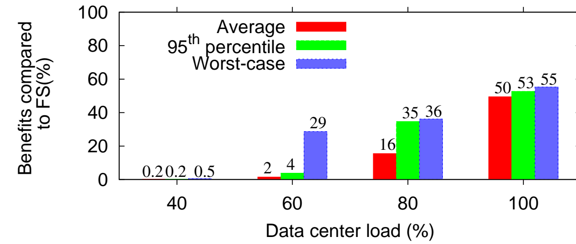

Figure 10: Reduction in task completion time compared for storage retrieval scenario.

36% over size-based policies at the 95th percentile. For uniform workloads, Baraat and STF perform similarly with the exception of the tail. With an STF policy, the worst-case completion time is inflated by 50% relative to FS, whereas Baraat reduces worst-case completion time by 48% relative to FS. Note that Baraat and FIFO collapse to the same policy in this case due to the absence of heavy tasks. Overall, these results highlight that 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. To compare performance across workflows, we look at homogeneous workloads with flow sizes uniformly distributed (as in the previous section). We simulate the arrival and execution of 48,000 requests.

Figure 8 plots the reduction in the task completion time with Baraat compared to fair share 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 95th percentile by 60% and 61% respectively. In all cases, the confidence intervals for the values provided are less than 10% within the mean, and are not plotted for clarity.

For the storage retrieval scenario (Figure 10), the worst case completion time reduces by 36% compared to fair-sharing at 80% load (35% and 16% reduction at 95th percentile and the average respectively). The reduced benefit results from the fact that tasks here involve only a single stage.

Figure 9 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 web search scenario at 80% load.

Adding computation. While our paper focuses on network performance, we now consider tasks featuring both network transfers and computation. Intuitively, bene-

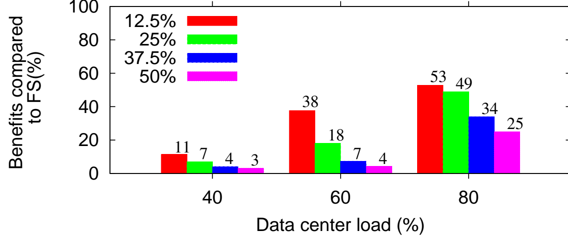

Figure 11: Reduction in worse-case task completion time with Baraat compared to fair-sharing when considering computation. Computation time is expressed as percentage of the overall task completion time.

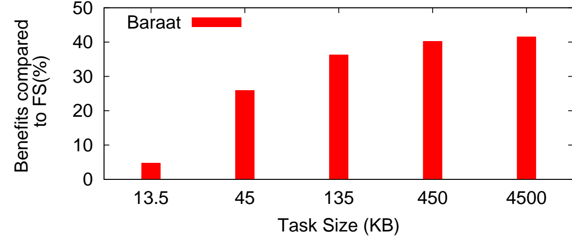

Figure 12: Benefits of Baraat extend to small tasks. Benefits increase with the task size and become stable to roughly 40% above task sizes of 135KB.

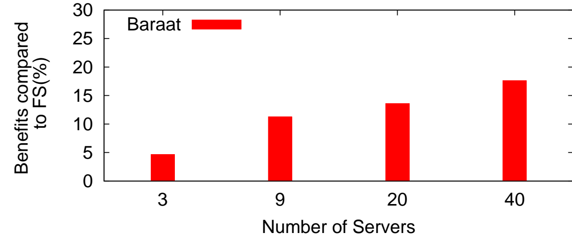

Figure 13: Benefits for tiny flows for 3 packets. Benefits increase as the number of flows per task increases.

fits of task-based network scheduling depend on whether network or computation is the overall bottleneck of the task. We extend the simulator to model computation for worker machines; it is modeled as an exponentially distributed wait time before a worker flow is started. Figure 11 presents the corresponding results, by varying the percentage a task spends on computation when there is no network contention. As expected, as this percentage increases, the benefits of Baraat drop since task completion time mostly depends on computation. 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%.

Over-subscription and selectivity. Increasing over-subscription and selectivity have similar effects; increasing over-subscription implies less available cross-rack bandwidth, and increasing selectivity more network-heavy cross-rack transfers. Hence, in both cases, the main network bottleneck is shifted to top-level aggregator stage. Thus, the benefits of Baraat start approximating benefits of single-stage tasks; significant gains are observed for the average case but the gains for the worst case reduce as the over-subscription or selectivity increases. As a reference point, increasing the network over-subscription to a ratio of 8:1 results in average gains of 60% and 28% for the worst case; similarly, increasingly the selectivity to 10% results in gains of 54% and 21% respectively.

5.3 ns-2 Micro-benchmarks

We use ns-2 to benchmark various aspects of Baraat's performance.

Benefits extend to smaller tasks and tiny flows. To quantify the benefits of Baraat for smaller tasks, we repeat our testbed experiments with varying task sizes. To achieve this, we change the size of the response generated by each of the three servers. We consider 8 concurrent tasks and a RTT of 100μsec. Figure 12 shows the reduction in average task completion time with Baraat over FS. As expected, the benefits increase as the task size increases because the task switching overhead is amortized. Benefits become significant beyond task sizes of 45KB. Beyond a task size of 400KB, the overhead becomes negligible and the benefits are stable. In contrast, for a small task of 13.5KB, the overhead is significant and Baraat provides little benefits. Note that the task size is the sum of all flows per task; hence a task size of 13.5KB implies a task with three flows of only 3 packets.

While we do expect applications to generate such tiny flows, most tasks will have dozens of such flows as highlighted in Table 1. Figure 13 presents such a scenario of tiny flows against the number of flows per task. With a larger number of flows per task, the benefits of Baraat compared to FS increase. Even though the individual flows are small (3 packets each), the overall task size is big enough to amortize the overhead.

Finally, if tasks are too small, we can aggregate multiple tasks and map them onto the same class. This amortizes the overhead of switching. Note that as we increase the aggregation level the scheduling granularity moves from task serialization to fair-sharing. This is shown in Figure 14 where we consider a scenario with small tasks (13.5KB each) and how aggregation helps in improving performance. As we map more tasks to the same class, the switching overhead gets amortized. We observe maximum performance gains (compared to FS) when 8 tasks

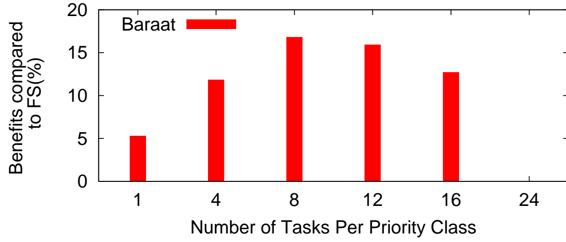

Figure 14: Aggregating tiny tasks into a single class.

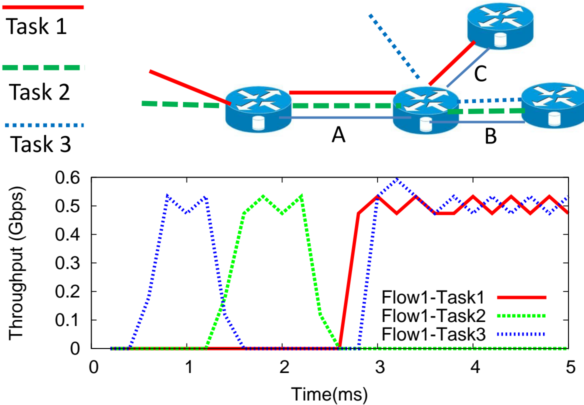

Figure 15: Verifying that Baraat ensures the properties of work-conservation and preemption.

are aggregated. Too much aggregation leads to increased multiplexing and we approach the performance of fair sharing.

Preemption and work conservation. We now validate that Baraat satisfies the two basic properties of work-conservation and preemption. We use a simple example with three tasks traversing different links. Each of the three tasks has two flows and both flows traverse the same set of links. For clarity, we report the throughput of one flow for each task (throughput of the other flow is similar). Figure 15 shows the setup and the results.

Initially, only flows of Task 3 are present in the system, with each of its two flows getting roughly half of the link bandwidth. As soon as flows of Task 2 arrive, both of Task 3's flows are preempted. This validates that flows belonging to higher priority tasks can preempt lower priority flows and can quickly reclaim bandwidth – roughly 200μsec in the experiment (twice the RTT). Finally, when Task 1 arrives, it preempts flows of Task 2 on their common link. Since Task 2 cannot make progress, work-conservation specifies that Task 3 should utilize the bandwidth as it does not share any link with Task 1. Indeed, the figure shows that Task 2's flows retract their demands, allowing Task 3's flows to grab the bandwidth. Hence, because of work conservation, Tasks 1 and 3 can make progress in parallel.

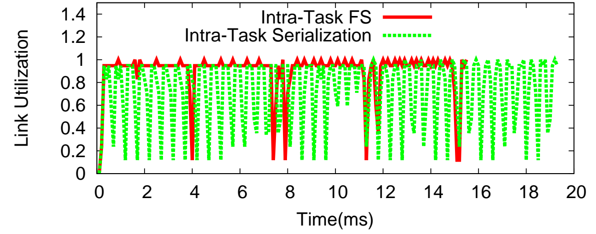

Figure 16: Intra-task fair sharing leads to improved link utilization compared to intra-task serialization

Benefits of Intra-Task fair sharing. Finally, we show that intra-task fair sharing results in more flows being simultaneously active, and thus, lowers the switching overhead compared to per-flow serialization [16]. We consider four concurrent tasks, each comprising of 10 flows. With intra-task serialization, only a single flow is active at a given time. This leads to frequent switching between flows and causes link utilization to drop (Figure 16). In comparison, higher level of multiplexing with intra-task fair sharing improves link utilization, enabling the tasks to finish faster.

6 Discussion

The notion of task serialization underlying Baraat is both affected by and has implications for various aspects of data center network design. We briefly discuss two issues here.

Multi-pathing. Today's data centers typically use multi-rooted tree topologies that offer multiple paths between servers. The path diversity is even greater for more recent designs like fat-tree [2] and hypercube topologies [1]. However, existing mechanisms to spread traffic across multiple paths, like ECMP and Hedera [3], retain flow-to-path affinity. Consequently, packets for a given flow are always forwarded along a single path. This means that Baraat can work across today's multipath network topologies. Further, the fact that Baraat involves explicit feedback between servers and switches means it is well positioned to fully capitalize on network path diversity. Senders can split their network demand across multiple paths by sending SRQ packets along them. Since per-flow fairness is an explicit non goal, the logic for splitting demand for a sender can be simpler than existing multi-pathing proposals [21].

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

7 Related Work

Baraat is related to, and benefits from, a large body of prior work. We broadly categorize them into:

Cluster Schedulers and Resource Managers. There is a huge body of work on *centralized* cluster schedulers and resource managers [15, 20, 11]. Most of these proposals focus on scheduling jobs on machines while we focus on scheduling flows (or tasks) over the network. Like Baraat, Orchestra [11] explicitly deals with network scheduling and how task-awareness could provide benefits for MapReduce jobs. Baraat differs from Orchestra in two important directions. First, scheduling decisions are made in a decentralized fashion rather than through a centralized controller. Second, Baraat uses FIFO-LM which has not been considered in prior work.

Straggler Mitigation Techniques. Many prior proposals attempt to improve task completion times through various straggler mitigation techniques (e.g., re-issuing the request) [6, 17]. 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.

Flow-based Network Resource Management. Most existing schemes for network resource allocation target flow level goals [25, 16, 4, 12], either fair sharing or some form of prioritization. As we show in this paper, such schemes are not suitable for optimizing task-level metrics. However, the design of our prioritization mechanism does leverage insights and techniques used in flow-based schemes, especially ones that use explicit rate protocols [12, 16].

Task-Aware Network Abstractions. As noted earlier, CoFlow [10] makes a case for task-awareness in data center networks and proposes a new abstraction. However, for specific task-aware scheduling policies, it relies on prior work (e.g., Orchestra).

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. In parallel, Baraat ensures work-conservation so that utilization and system throughput remain high. 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 across three platforms show that Baraat can significantly reduce the average as well as tail task completion time.

References

[1] H. Abu-Libdeh, P. Costa, A. Rowstron, G. O'Shea, and A. Donnelly. Symbiotic routing in future data centers. In *ACM SIGCOMM*, 2010.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proc. of ACM SIGCOMM*, 2008.

[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 *ACM SIGCOMM*, pages 63–74, 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, 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 *OSDI*, volume 10, page 24, 2010.

[7] 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.

[8] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Scale-up vs scale-out for hadoop: Time to rethink? In *Proceedings of the ACM Symposium on Cloud Computing*, 2013.

[9] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The case for evaluating mapreduce performance using workload suites. In *Proc. of MASCOTS*, pages 390–399, 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] N. Dukkipati. *Rate Control Protocol (RCP): Congestion control to make flows complete quickly*. PhD thesis, Stanford University, 2007.

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

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

[15] 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 *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pages 22–22. USENIX Association, 2011.

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

[17] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. In *Proceedings of the ACM SIGCOMM*, pages 219–230, 2013.

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

[19] 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 *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, pages 385–398. USENIX Association, 2013.

[20] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Scalable scheduling for sub-second parallel jobs. Technical report, Tech. Rep. UCB/EECS-2013-29, EECS Department, University of California, Berkeley, 2013.

[21] C. Raiciu, C. Pluntke, S. Barre, A. Greenhalgh, D. Wischik, and M. Handley. Data center networking with multipath tcp. In *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks*, page 10. ACM, 2010.

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

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

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

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

[26] H. Yu, A. Vahdat, et al. Efficient numerical error bounding for replicated network services. In *Proceedings of the Twenty-Sixth International Conference on Very Large Data Bases*, pages 123–133. Citeseer, 2000.