Network Scheduling Aware Task Placement in Datacenters

Ali Munir[1], Ting He[2], Ramya Raghavendra[2], Franck Le[2], Alex X. Liu[1]
[1]Michigan State University, [2]IBM Research

ABSTRACT

To improve the performance of data-intensive applications, existing datacenter schedulers optimize either the placement of tasks or the scheduling of network flows. The task scheduler strives to place tasks close to their input data (i.e., maximize data locality) to minimize network traffic, while assuming fair sharing of the network. The network scheduler strives to finish flows as quickly as possible based on their sources and destinations determined by the task scheduler, while the scheduling is based on flow properties (e.g., size, deadline, and correlation) and not bound to fair sharing. Inconsistent assumptions of the two schedulers can compromise the overall application performance. In this paper, we propose NEAT, a task scheduling framework that leverages information from the underlying network scheduler to make task placement decisions. The core of NEAT is a task completion time predictor that estimates the completion time of a task under given network condition and a given network scheduling policy. NEAT leverages the predicted task completion times to minimize the average completion time of active tasks. Evaluation using ns2 simulations and real-testbed shows that NEAT improves application performance by up to 3.7x for the suboptimal network scheduling policies and up to 30% for the optimal network scheduling policy.

Keywords

Cluster schedulers; Network schedulers; Compute placement; Cross-layer optimization

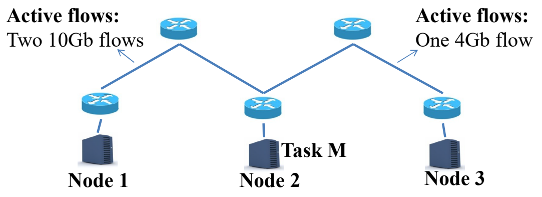
1. INTRODUCTION

Data transfer time has a significant impact on task completion times within a datacenter because most datacenter applications (such as MapReduce [1], Pregel [2], MPI [3], Dryad [4]) are data-intensive and they need to access data that are distributed throughout the network. For example, for MapReduce, 30% task completion time is spent on transferring data across the network [5]. For large commercial datacenters, the transfer of shuffle data is the dominant source of network traffic [6]. In Facebook [5] MapReduce clusters, 20% jobs are shuffle-heavy (i.e., generate a large amount of shuffle data), and in Yahoo! clusters [7] this goes up to 60%. The network congestion caused by shuffle data is a key factor that degrades the performance of MapReduce jobs [8]. Therefore, data transfer time in datacenter is critical for improving application performance.

Prior work on minimizing data transfer time falls into two categories: task placement ([5, 6, 9, 10, 11, 12]) and network scheduling ([13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]). The idea of task placement is to place compute tasks close to input data so that data locality is maximized and network traffic is minimized [5, 9, 10]. The idea of network scheduling is to schedule the flows or groups of flows (i.e., coflows) generated by tasks, at shared links, based on given flow properties (such as size, deadline, and correlation among flows) and given task placement to minimize flow completion times[13, 15, 16, 17, 21]. The limitation of prior task placement policies is that they design traffic matrix assuming fair sharing of network bandwidth and ignore the priorities assigned to different flows by the network scheduler; meanwhile, network schedulers schedule flows based on flow priorities (such as size or deadline) and the flow completion time of a low priority flow can increase if placed on a path sharing links with high priority flows. The limitation of network schedulers is that the source/destination of each flow is independently decided by the task schedulers and not necessarily optimal.

In this paper, we propose NEAT, a **N**etwork-sch**E**duling-**A**ware **T**ask placement framework that leverages *network scheduling policy* and *network state* in making task placement decisions for data-intensive applications. The placement by NEAT coupled with a properly selected network scheduling policy ensures that tasks are finished as quickly as possible. The intuition behind NEAT is that a good placement of data-intensive tasks should spread the flows to minimize the sharing of network links. When sharing is inevitable, however, the preferred placement strategy depends on how the network schedules flows, as illustrated by the following example.

CoNEXT '16, December 12-15, 2016, Irvine, CA, USA
© 2016 ACM. ISBN 978-1-4503-4292-6/16/12...$15.00
DOI: http://dx.doi.org/10.1145/2999572.2999588



Candidate hosts: Node 1 or 3, **New Flow (R) size**: 5 Gb,
Link capacity: 1 Gbps

Network Scheduling Policy	Placement of R	Completion time of R (s)	Increase in Total Completion time (s)
FCFS	Node 1	25	25
	Node 3	**9**	**9**
Fair	Node 1	15	25
	Node 3	**9**	**13**
SRPT	**Node 1**	**5**	**15**
	Node 3	9	9

Figure 1: Network scheduling aware placement

Motivating Example: Let us consider a simple scenario in Figure 1, where we want to place a task R that needs to read data from task M running on node 2 onto candidate hosts node 1 or node 3. At the current time, the network has one flow with remaining size of 4 Gb on path $2 \rightarrow 3$ and requires four more seconds (on 1 Gbps link) to finish if running alone. There are two other flows on path $2 \rightarrow 1$, each of remaining size 10 Gb, thus requiring ten more seconds if running alone. Suppose that the input data of candidate task R is of size 5 Gb and our goal is to finish the task as quickly as possible. Assuming First Come First Serve scheduling (FCFS) in the network, we will place task R on node 3, as it provides the smaller completion time of 9 seconds as compared to 25 seconds when placed on node 1, because flow R will not be scheduled until the existing flows finish data transfer. Assuming fair scheduling (Fair) in the network, we will place task R on node 3, as it requires 9 seconds to complete flow R on node 3 compared to 15 seconds on node 1. Assuming the network uses shortest remaining processing time (SRPT) scheduling, we will place task R on node 1, as it can finish in 5 seconds on node 1 by preempting both the existing flows. We assume flows with the same priority share the network fairly as in existing solutions [16, 17]. However, if the goal is to minimize the increase in total (i.e., sum) flow completion time of all the active flows in the network, then under SRPT, the scheduler will choose node 3 to minimize the increase in completion time of flows in the network (i.e., completion time of new flow plus increased completion times of existing flows). We see that the optimal task placement in a network can vary depending on the network scheduling policy and the network performance metric.

NEAT Overview: NEAT leverages these insights in its design and provides a task placement framework to minimize the average flow completion time (AFCT) in the network. NEAT employs a task performance predictor that predicts the completion time of a given task under an arbitrary network scheduling policy. NEAT is quite pluggable in terms of better resource or application models and predictors. NEAT makes task placement decision in two steps: first, it predicts task performance by hypothetically placing a task on each candidate node and analyzing its performance based on the network scheduling policy and the network state. Next, it selects a node for the task based on the predicted task completion time and a node state, that characterizes the sizes of existing flows. The proposed task placement mechanism focuses on network performance in task placement decisions and only uses node properties (e.g., CPU, memory) to determine whether a node is a candidate host. We leave joint placement to future work. To realize this idea, NEAT addresses two challenges:

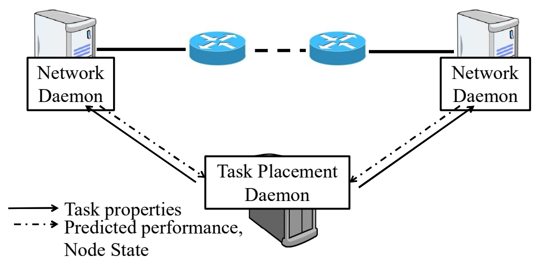


Figure 2: NEAT Architecture

The first technical challenge in designing NEAT is to *accurately predict the performance of a task for a given network state and scheduling policy.* We use the completion time of data transfer, i.e., the *flow completion time (FCT)* or the *coflow completion time (CCT)*, to measure the overall task performance. Compared to network-centric performance metrics such as delay, jitter, and available bandwidth, completion time is task-centric and better models network conditions in terms of their impact on task performance (§6).

The second technical challenge is to *efficiently collect network state information and make placement decisions.* As illustrated in Figure 2, NEAT uses a distributed control framework with two components: a local controller (network daemon) that maintains a *compressed* state of all active flows (or coflows) on each node to predict task completion times, and a global controller (task placement daemon) that gathers the predicted task completion times from local controllers to make task placement decisions. Here, a "node" refers to an endhost (e.g., server) capable of running tasks. Using the flow state information, the local controllers predict the FCT/CCT on edge links (i.e., links directly connected to nodes), based on which the global controller uses a greedy algorithm to pick the node whose edge link has the smallest FCT/CCT. To reduce communications with the local controllers, the global controller also maintains a *state* for each node, defined as the remaining size of the smallest active flow on that node, such that only nodes with preferred states (no less than the size of the current flow) are contacted.

NEAT does not require any changes to the network. However, it requires task information (e.g., size) from applications, similar to prior works [16, 17, 19], to accurately predict task completion time. For recurrent workloads, this information can be inferred by monitoring task execution [12].

NEAT's performance is upper-bounded by the underlying network (flow/coflow) scheduling policy. When the network scheduling policy is suboptimal (e.g., Fair, LAS), when our goal is FCT, there is plenty of room for improvement and NEAT is able to significantly improve the task performance;

when the network scheduling policy is near optimal (SRPT), the room for improvement is reduced, NEAT still achieves notable improvement for such scenarios.

NEAT Evaluation: We evaluate NEAT using both a trace-driven simulator and a 10-machine testbed based on a variety of production workloads [1, 16] and network scheduling policies such as DCTCP [24], L^2DCT [15], PASE [17], and Varys [21]. We evaluate two other state-of-art task placement policies: *minLoad* that always selects a node with the minimum load, measured by the total size of flows scheduled on that node, and *minDist* that always selects a node closest to the input data. NEAT performs 3.7x better than alternative strategies when the underlying network scheduling policy is Fair (DCTCP [24]), 3x better when the policy is least attained service (LAS) (L^2DCT [15]), and 30% better when the policy is SRPT (PASE [17]). In both cases, NEAT improves performance by being aware of the network scheduling policy. In particular, since Fair is the underlying policy in most commercial datacenters, NEAT can significantly improve the performance of data-intensive applications in majority of cases.

Summary of Contributions:

- We propose a unified task completion time prediction model, that can accurately predict task performance for a given network scheduling policy.
- We propose a distributed framework, NEAT, to gather network state and make task placement decisions in an online fashion.
- We evaluate NEAT across a range of network scheduling policies and network conditions using simulations and real testbed experiment.

2. LIMITATIONS OF PRIOR ART

The overall performance of data-intensive workloads depends on both (1) scheduling of compute tasks, and (2) scheduling of the resulting network flows. However, existing solutions address either the task placement problem or the network scheduling problem in isolation, leaving significant room for further improvement. Below, we first summarize related work and then provide a comparative study to highlight limitations of existing schemes.

2.1 Related Work

The related work can be divided in two categories: Task placement (or scheduling) and network scheduling.

Task Scheduling: State-of-art task schedulers follow the principle of "maximizing data locality" to minimize data transfer over the network. For the first stage of jobs (e.g., Map), techniques such as delay scheduling [5] and flow-based scheduling [9] try to place tasks on the machines or racks where most of their input data are located. Mantri instead optimizes the data locality while placing reducers [10]. However, when the data is spread throughout the cluster (e.g., in distributed file systems like HDFS [25]), the subsequent job stages (e.g., shuffle phase) have to read data using cross-rack links, and often suffer from heavy network congestion [26]. To alleviate the problem, techniques like ShuffleWatcher [6] attempt to localize the Map tasks of a job to one or a few racks, and thus reduce cross-rack shuffling. However, such techniques end up increasing the cross-rack transfer of the input data. A similar idea was exploited in Sinbad [26] to place the endpoints of flows to avoid congested links, but Sinbad is for a special type of flows (replicated file system writes) and is agnostic to the underlying network scheduling policy. When the input data of a job is known beforehand, techniques like Corral [12] can be used to pack the input data to a few racks such that all computation can be performed on local data. Similarly, many fairness based cluster scheduling frameworks [27, 28, 29, 30, 31] or straggler mitigation techniques [32, 10, 33, 34] have been proposed. However, these works assume fair sharing in the network and does not consider effect of underlying network scheduling policies. Another category of works [35, 36] does joint network and task execution scheduling, however these works assume that they have knowledge about task arrival times and make offline task scheduling decisions.

Network Scheduling: Generally, data-intensive jobs spend a substantial fraction of their execution time on data transfer [5], and consequently, their performance critically depends on the performance of network scheduling. In recent years, a variety of network scheduling systems such as DCTCP [24], L^2DCT [15], PASE [17], Varys [21], and Baraat [20] have been proposed. Many of these protocols provide near-optimal data transfer time, but the improvement to the overall task performance is limited as they do not optimize the placement of sources/destinations of the flows.

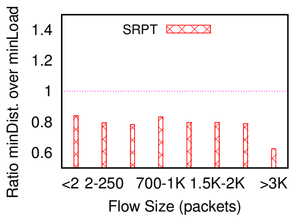
NEAT addresses these limitations in two ways: it places the destinations of flows (via task placement) while taking into account the underlying network scheduling policy to minimize the overall data transfer time, and it uses a performance predictor that captures network state and essence of the network scheduling policies.

2.2 Comparative Study

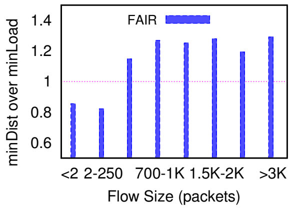
Task placement policies perform differently when used with different network scheduling policies. To illustrate this, we compare the performance of two task scheduling policies under two different network scheduling policies. For task scheduling policies, we consider: minimum-distance placement (minDist), which places each task to minimize its distance to the input data, and minimum-load placement (minLoad), which places each task on the server with the minimum traffic load. For network scheduling policy, we consider: DCTCP [24], which approximates the Fair sharing policy, and PASE [17], which approximates the shortest remaining processing time (SRPT) policy in the network.

We simulate a topology of 160 machines, connected using a folded CLOS topology as in [17]. We configure the links to have 200μsec RTT and 1 Gbps (edge) or 10 Gbps (core) bandwidth. We evaluate each combination of the above task/network scheduling policies under flow demands generated according to the data-mining workload [16]. Figure 3(a) and Figure 3(b) show the ratio between the FCT of minDist and the FCT of minLoad for various flow sizes under the

Fair, and SRPT network scheduling policy, respectively. A bar with coordinates (x, y) indicates that flows of sizes x have an FCT ratio of y, where y<1 indicates that minDist outperforms minLoad and y>1 indicates the opposite.



(a) SRPT (PASE)



(b) Fair (DCTCP)

Figure 3: Comparison of minDist and minLoad placement under SRPT and Fair network scheduling policy

The results show that when the network follows SRPT policy (Figure 3(a)), minDist placement works better (ratio less than 1). In contrast, when the network follows a Fair policy (Figure 3(b)), minLoad placement works better for the majority of the flow sizes. The results can be explained by the following observations: when the network follows SRPT policy (Figure 3(a)), short flows preempt long flows, and receive full link bandwidth. Short flows therefore achieve near-optimal completion time. We verified it in the FCT logs but omit the details because of page limitations. After the short flows complete, more bandwidth becomes available to the long flows, allowing them to finish quickly. The minDist placement outperforms minLoad placement because it minimizes the total network load, defined as sum of the product of the size and the hop count for each flow. In contrast, when the network follows the Fair policy (Figure 3(b)), short flows no longer preempt long flows. The minLoad placement works better – especially for long flows – because it ensures that long flows avoid being placed on nodes with existing long flows. However, Figure 3(b) shows that short flows may suffer longer FCTs under the minLoad placement compared to the minDist placement. This usually happens when newly arriving long flows are placed on nodes with ongoing short flows, thus increasing the FCTs of short flows. In conclusion, this experiment demonstrates that the performance of task placement depends on the underlying network scheduling policy. Therefore, it is important to design a task placement framework that takes into account the underlying network scheduling policy.

3. SYSTEM OVERVIEW

NEAT, is a **N**etwork-sch**E**duling-**A**ware **T**ask placement framework that considers the *network scheduling policy* and the *network state* while making task placement decisions. NEAT considers placing both flows and coflows in the network via placing the tasks on the nodes that act as destinations of these flows (coflows). The placement framework, coupled with the underlying network scheduling policy (e.g., Fair, FCFS, LAS, SRPT), ensures that network resources are properly allocated among tasks to minimize their average completion time.

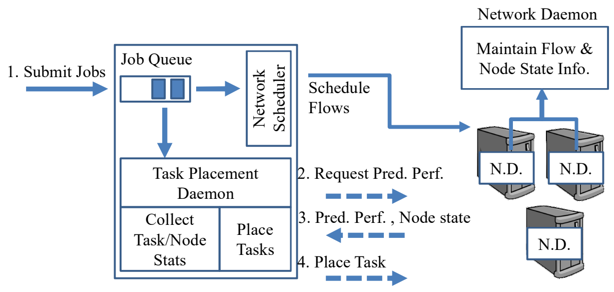


Figure 4: NEAT components

NEAT uses a distributed architecture to make task placement decisions, as shown in Figure 4. It leverages the master-slave architecture of existing task scheduling systems [28, 31] to achieve its functionality in a distributed fashion. NEAT has two components, i) a global task placement daemon and ii) a per-node network daemon. The two components exchange information to perform network-aware task scheduling.

Network Daemon: NEAT network daemon (similar to node manager in Hadoop) runs on all the nodes (i.e., end-hosts) in the network and uses the completion time of data transfer (FCT for flow, CCT for coflow) as a metric for predicting task performance. The prediction is based on the network state, e.g., (residual) sizes of all active flows, and the network scheduling policy. Therefore, network daemon performs two functions: (i) maintaining network information of all the active tasks on the node (such as the number of flows and their residual sizes), and ii) predicting the performance of a new task when requested by the task placement daemon, based on the analysis in section 4. NEAT considers only the edge links for task performance prediction. Therefore, the task performance prediction is based on the assumption that only the edge links are bottleneck in the network and core is congestion free. As a result NEAT needs to maintain flow information at the end-hosts only and does not require any feedback from the network.

Task Placement Daemon: NEAT task placement daemon makes placement decisions in two steps: First, it queries the network daemons on nodes that are valid hosts (determined by node state, CPU and memory) of a given task for their predicted task performance. Next, it chooses the best node or subset of nodes for placing the task using the predicted task completion times and node states (i.e., the smallest residual flow size of active tasks on each node). It also caches the node states locally to facilitate future placement decisions, as discussed below. The node states cache is updated whenever a new task is placed on the node or an existing task is completed.

NEAT Workflow: As illustrated in Figure 4, when the task placement daemon receives a request to place a new task (step 1), it identifies a set of preferred hosts based on local copies of node states and contacts the respective network daemons to get predicted performance of the new task (step 2). Each network daemon maintains the local network state (such as the number and the sizes of flows starting/ending at the node) and uses this state together with knowledge of the network scheduling policy to predict the performance of the new task. The task placement daemon then gathers this information (step 3) and places the task on the best of the preferred hosts based on the predicted performance (step 4).

Discussions: It is worthwhile to clarify a few points. First of all, NEAT is a task placement framework and as such, it does not impose any changes in the network switches or network scheduling mechanism. The network congestion is managed by the underlying network scheduling policy. However, NEAT helps to mitigate the network congestion by properly distributing traffic loads across the network. Moreover, in the current design, NEAT abstracts network as a single-switch topology, as assumed in earlier works [12, 16]. The single-switch assumption means that only edge links are the bottleneck and thus for task completion time (FCT/CCT) prediction, it only considers edge links. However, such restriction is not fundamental to NEAT, and we discuss in Section 7 how it can be extended to other topologies.

In the next two sections, we discuss task performance prediction and NEAT design in more detail.

4. TASK PERFORMANCE PREDICTOR

NEAT has a pluggable task performance predictor that can predict the completion time of a given task under an arbitrary network scheduling policy. An interesting insight of our analysis is that: for flow-based scheduling, it suffices to predict FCT according to the Fair policy under mild conditions even if the underlying policy is not Fair; for coflow-based scheduling, however, this is generally not true.

From a networking perspective, placing a task onto a server is equivalent to placing a flow onto a path (or a coflow onto a set of paths). We now define the objective function for flow placement; the objective function for coflow placement is analogous. For flow placement, our goal is to place each flow onto the path that minimizes the sum (and hence average) FCT over all active flows. Consider a flow f placed on path p_f with size s_f (bits). Let $\text{FCT}(f,l)$ denote the completion time of flow f on link $l \in p_f$ (i.e., time taken for the flow to finish transmission over link l). Now consider placing a new flow f_0 onto path p_{f_0}. Let $\Delta\text{FCT}(f,l)$ denote the increase in the completion time of f on link l due to the scheduling of f_0. Let F be the set of cross-flows in the network (i.e., flows that share at least one link with f_0). Then the increase in the sum FCT over all active flows due to the placement of f_0 equals

$$\max_{l\in p_{f_0}} \text{FCT}(f_0,l) + \sum_{f\in F}\Big(\max_{l\in p_f}(\text{FCT}(f,l)+\Delta\text{FCT}(f,l)) - \max_{l\in p_f}\text{FCT}(f,l)\Big), \quad (1)$$

and the optimal online placement is to place f_0 such that (1) is minimized (note that (1) depends on p_{f_0}).

The original objective (1) requires calculation of the FCT before and after placing f_0 for every flow $f \in F$ and every hop l traversed by f. To simplify computation, we exchange the order of summation and maximization to obtain an alternative objective function

$$\max_{l\in p_{f_0}} \text{FCT}(f_0,l) + \max_{l\in p_{f_0}} \sum_{f\in F_l} \Delta\text{FCT}(f,l)$$
$$\geq \max_{l\in p_{f_0}} \left(\text{FCT}(f_0,l) + \sum_{f\in F_l} \Delta\text{FCT}(f,l)\right), \quad (2)$$

where F_l is the set of flows traversing link l. Note that (2) is only an approximation of (1) and thus minimizing (2) is not guaranteed to minimize the sum FCT. However, as we show later, the alternative objective (2) has a nice property that it is invariant, up to a constant scaling, to the network scheduling policy under certain conditions (see Propositions 4.1 and 4.2), and the placement that minimizes (2) achieves superior performance in minimizing the FCT for a variety of flows and network scheduling policies.

For simplicity of analysis, we do not consider the impact of future task arrivals or task completion on the prediction. We choose this design in favor of applicability (not requiring prediction capability) and stability (not moving existing flows) of NEAT design (see remark § 5.1.1).

4.1 FCT Prediction for Flow Scheduling

Given a placement of flow f_0 and its cross-flows, the FCT of f_0 is determined by the scheduling policy employed by the network. We therefore study FCT prediction under several widely-adopted scheduling policies, including FCFS, Fair, LAS, and SFF. For ease of presentation, we fix a (candidate) placement and consider the prediction for a given link l with bandwidth B_l. The key idea of NEAT predictor is to compute the total number of bytes that will be transmitted across the bottleneck link by the time the current flow finishes transmission, which is then divided by the link bandwidth to predict the completion time of the current flow. For example, to predict the completion time of a flow f when placed onto a path p, we compute for each link l on p the total traffic volume V_l from f and coexisting flows that will cross l upon the completion of f, and then the maximum of V_l/B_l (B_l is the bandwidth of l) over all the links $l \in p$ gives the predicted completion time of f. This approach applies to any work-conserving scheduling policy (i.e., a link is never idle when there is pending traffic on that link), while different policies differ in the computation of V_l.

4.1.1 FCT under FCFS Scheduling

Under FCFS scheduling, flows are served in the order they arrive, and FCT of the new flow can be predicted as:

$$\text{FCT}^{\text{FCFS}}(f_0,l) = \frac{1}{B_l}(s_{f_0} + \sum_{f\in F_l} s_f), \quad (3)$$

and $\Delta\text{FCT}^{\text{FCFS}}(f,l) \equiv 0$ for all $f \in F_l$ since the new flow does not affect the completion of existing flows. Hence, the two objectives in (1) and (2) coincide in this case, both equal to $\max_{l\in p_{f_0}} \text{FCT}^{\text{FCFS}}(f_0,l)$.

4.1.2 FCT under Fair or LAS Scheduling

Under a scheduling policy that performs fair sharing, all flows will receive equal service until completion, i.e., by the time flow f_0 completes transmission over link l, each existing flow $f \in F_l$ will have transmitted $\min(s_f,\ s_{f_0})$ bytes over l. The FCT of f_0 can thus be predicted as:

$$\mathrm{FCT}^{\mathrm{FAIR}}(f_0, l) = \frac{1}{B_l}\left(s_{f_0} + \sum_{f \in F_l} \min(s_f,\ s_{f_0})\right). \quad (4)$$

Meanwhile, the fair sharing rule implies that for each flow $f \in F_l$ of size $s_f < s_{f_0}$, scheduling f_0 increases its FCT by imposing an additional traffic load of s_f on link l (during the lifetime of f); for each flow $f \in F_l$ of size $s_f \geq s_{f_0}$, f_0 will finish earlier, causing an additional load of s_{f_0}. Thus, the change in FCT of existing flows is:

$$\Delta\mathrm{FCT}^{\mathrm{FAIR}}(f, l) = \frac{\min(s_f,\ s_{f_0})}{B_l}. \quad (5)$$

Substituting (4) and (5) into (1) gives the overall increase in the sum FCT.

The alternative objective (2) accepts a more compact form in this case. Specifically, for given l,

$$\begin{aligned}&\mathrm{FCT}^{\mathrm{FAIR}}(f_0, l) + \sum_{f \in F_l} \Delta\mathrm{FCT}^{\mathrm{FAIR}}(f, l)\\ &= \frac{s_{f_0}}{B_l} + \frac{2}{B_l}\sum_{f \in F_l} \min(s_f,\ s_{f_0}) \approx 2\mathrm{FCT}^{\mathrm{FAIR}}(f_0, l), \quad (6)\end{aligned}$$

where the approximation is accurate when s_{f_0} is small relative to $\sum_{f \in F_l} \min(s_f,\ s_{f_0})$.

Remark: Since LAS scheduling (with preemption) is equivalent to fair sharing, the above result also applies to LAS scheduling.

4.1.3 FCT under SRPT Scheduling

If the network employs SRPT scheduling, then only flows of remaining sizes smaller than or equal to the current flow will be served before the current flow finishes (assuming FCFS rule is applied to break ties among flow sizes), and larger flows will be preempted. Hence, the FCT of the new flow f_0 under SRPT can be predicted as:

$$\mathrm{FCT}^{\mathrm{SRPT}}(f_0, l) = \frac{1}{B_l}\left(s_{f_0} + \sum_{f \in F_l : s_f \leq s_{f_0}} s_f\right). \quad (7)$$

Meanwhile, each flow of size $s_f > s_{f_0}$ will be delayed by s_{f_0}/B_l due to the placement of f_0, while flows of size $s_f \leq s_{f_0}$ will not be affected, implying:

$$\Delta\mathrm{FCT}^{\mathrm{SRPT}}(f, l) = \frac{s_{f_0}}{B_l}\mathbb{1}_{s_f > s_{f_0}}, \quad (8)$$

where $\mathbb{1}.$ is the indicator function. Applying (7) and (8) to (1) gives the first objective. For second objective, we have

$$\begin{aligned}&\mathrm{FCT}^{\mathrm{SRPT}}(f_0, l) + \sum_{f \in F_l} \Delta\mathrm{FCT}^{\mathrm{SRPT}}(f, l)\\ &= \frac{1}{B_l}\left(s_{f_0} + \sum_{f \in F_l} \min(s_f,\ s_{f_0})\right) = \mathrm{FCT}^{\mathrm{FAIR}}(f_0, l). \quad (9)\end{aligned}$$

4.1.4 Invariance Condition

Although the predicted FCT varies for different network scheduling policies, we show that the optimal placement can be invariant under certain conditions. Specifically, we have shown that under SRPT scheduling, the alternative objective (2) reduces to the fair-sharing FCT of the newly arrived flow (see (9)), and under Fair/LAS scheduling, this objective is approximately twice of the fair-sharing FCT (see (6)). Since constant-factor scaling does not affect the selection of placement, these results imply that we can use the same objective function to place flows no matter whether the network performs Fair, LAS, or SRPT scheduling.

PROPOSITION 4.1. *If the network performs Fair, LAS, or SRPT flow scheduling and each flow is small relative to the total load on each link, then the optimal placement that minimizes (2) is always the one that minimizes bottleneck fair-sharing FCT of the newly arrived flow as predicted by (4).*

Remark: Surprisingly, we have shown that when the network schedules flows by LAS or SRPT, we should place flows to minimize their predicted FCTs under Fair scheduling in order to optimize the objective (2).

4.2 CCT Prediction for Coflow Scheduling

We now revise the above formulas for the case that the network schedules traffic at the level of coflow. We study the following policies applied to coflow scheduling, including FCFS [20], Fair, LAS [23], and an extension of SRPT called *permutation scheduling* [21]. For a coflow c, let s_c denote its total size (in number of bytes), and $s_{c,l}$ denote the sum size of the individual flows in c that traverse link l. We show that the idea of computing FCT can be generalized to predict the performance of coflows.

We make the following assumptions: (i) all flows of a coflow have the same priority, and (ii) all flows of a coflow complete simultaneously. The first assumption is based on a principle adopted by state-of-art coflow schedulers [20, 21] that flows within a coflow should progress together; the second assumption is based on a state-of-art coflow rate adaptation mechanism [21], which slows down all but the slowest flows in a coflow to match the completion time of the slowest flow so as to conserve bandwidth without increasing the CCT. In particular, assumption (ii) allows us to apply the same objective functions (1, 2) in placing coflows, with f (f_0) replaced by c (c_0), FCT replaced by CCT, and p_f replaced by p_c (denoting the set of links traversed by any flow of a coflow c). The computation of $\mathrm{CCT}(c, l)$ and $\Delta\mathrm{CCT}(c, l)$ is, however, different from their single-flow counterparts, and will be detailed below. We redefine F_l to be the set of coflows with at least one constituent flow traversing link l.

4.2.1 CCT under FCFS Scheduling

Under FCFS scheduling, each link will serve traffic from existing coflows before serving the newly arrived coflow, thus

$$\mathrm{CCT}^{\mathrm{FCFS}}(c_0,l)=\frac{1}{B_l}(s_{c_0,l}+\sum_{c\in F_l}s_{c,l}), \tag{10}$$

and $\Delta\mathrm{CCT}^{\mathrm{FCFS}}(c,l)\equiv 0$ for all $c\in F_l$.

4.2.2 CCT under Fair or LAS Scheduling

Under fair sharing or LAS scheduling, all coflows of size smaller than c_0 will have finished and all coflows of size larger than c_0 will have transmitted s_{c_0} bytes when coflow c_0 completes. Under assumption (ii), flows within a coflow make progress proportionally to their sizes, and thus when a coflow traversing link l finishes b bytes over all its constituent flows, $bs_{c,l}/s_c$ bytes must be transmitted over link l. Therefore, each coflow $c\in F_l$ introduces a load of $\min(s_c,\ s_{c_0})s_{c,l}/s_c$ on link l during the lifetime of coflow c_0 and vice versa. The above arguments imply that

$$\mathrm{CCT}^{\mathrm{FAIR}}(c_0,l)=\frac{1}{B_l}\left(s_{c_0,l}+\sum_{\substack{c\in F_l\\ s_c\leq s_{c_0}}}s_{c,l}+\sum_{\substack{c\in F_l\\ s_c>s_{c_0}}}\frac{s_{c_0}s_{c,l}}{s_c}\right) \tag{11}$$

for the newly arrived coflow c_0, and

$$\Delta\mathrm{CCT}^{\mathrm{FAIR}}(c,l)=\frac{s_{c_0,l}}{B_l s_{c_0}}\min(s_c,\ s_{c_0}) \tag{12}$$

for each existing coflow $c\in F_l$. From the above, we see that

$$\sum_{c\in F_l}\Delta\mathrm{CCT}^{\mathrm{FAIR}}(c,l)=\frac{1}{B_l}\left(\sum_{\substack{c\in F_l\\ s_c\leq s_{c_0}}}\frac{s_c s_{c_0,l}}{s_{c_0}}+\sum_{\substack{c\in F_l\\ s_c>s_{c_0}}}s_{c_0,l}\right), \tag{13}$$

which is different from (11) unless $s_{c,l}/s_c=s_{c_0,l}/s_{c_0}$ for all $c\in F_l$. It means that compared to the single-flow Fair/LAS scheduling, where minimizing the objective (2) reduces to minimizing the FCT of the newly arrived flow, we have to explicitly consider the impact on existing coflows through (13) (in addition to (11)) under coflow Fair/LAS scheduling.

4.2.3 CCT under Permutation Scheduling

Permutation scheduling [21] serves the coflows sequentially and includes many scheduling policies as special cases (e.g., FCFS and all variations of SRPT). Given a permutation $\pi=(\pi_c)_{c\in F_l\cup\{c_0\}}$ of all the coflows sharing link l, where π_c is the order of scheduling coflow c, the CCT of the newly arrived coflow c_0 equals

$$\mathrm{CCT}^{\pi}(c_0,l)=\frac{1}{B_l}\sum_{\pi_c\leq\pi_{c_0}}s_{c,l}, \tag{14}$$

and the increase in CCT for each existing coflow $c\in F_l$ equals

$$\Delta\mathrm{CCT}^{\pi}(c,l)=\frac{s_{c_0,l}}{B_l}\ {}_{\pi_c>\pi_{c_0}}, \tag{15}$$

which can be used to evaluate the first objective (1).

For the second objective (2), we have

$$\mathrm{CCT}^{\pi}(c_0,l)+\sum_{c\in F_l}\Delta\mathrm{CCT}^{\pi}(c,l)$$
$$=\frac{1}{B_l}\left(\sum_{\pi_c\leq\pi_{c_0}}s_{c,l}+s_{c_0,l}|\{c\in F_l:\ \pi_c>\pi_{c_0}\}|\right). \tag{16}$$

In particular, for a counterpart of SRPT called *smallesT-Coflow-First (TCF)* [21], (16) becomes

$$\mathrm{CCT}^{\mathrm{TCF}}(c_0,l)+\sum_{c\in F_l}\Delta\mathrm{CCT}^{\mathrm{TCF}}(c,l)$$
$$=\frac{1}{B_l}\left(s_{c_0,l}+\sum_{\substack{c\in F_l\\ s_c\leq s_{c_0}}}s_{c,l}+\sum_{\substack{c\in F_l\\ s_c>s_{c_0}}}s_{c_0,l}\right), \tag{17}$$

which is similar to the CCT under fair sharing (11) if $s_{c,l}/s_c=s_{c_0,l}/s_{c_0}$ for all $c\in F_l$, $s_c>s_{c_0}$.

4.2.4 Invariance Condition

The above analysis shows that in the special case of $s_{c,l}/s_c=s_{c_0,l}/s_{c_0}$ for all $c\in F_l$, i.e., all coflows split traffic among traversed links in the same way, the objective (2) satisfies $\mathrm{CCT}^{\mathrm{FAIR}}(c_0,l)+\sum_{c\in F_l}\Delta\mathrm{CCT}^{\mathrm{FAIR}}(c,l)\approx 2\mathrm{CCT}^{\mathrm{FAIR}}(c_0,l)$ under Fair/LAS scheduling (assuming $s_{c_0,l}\ll\sum_{c\in F_l\,s_c\leq s_{c_0}}s_{c,l}+\sum_{c\in F_l\,s_c>s_{c_0}}s_{c_0,l}$), and $\mathrm{CCT}^{\mathrm{TCF}}(c_0,l)+\sum_{c\in F_l}\Delta\mathrm{CCT}^{\mathrm{TCF}}(c,l)=\mathrm{CCT}^{\mathrm{FAIR}}(c_0,l)$ under TCF scheduling. By arguments similar to Proposition 4.1, we have the following statement.

PROPOSITION 4.2. *If the network performs Fair, LAS, or TCF coflow scheduling, each coflow imposes a small load on each link (relative to its total load), and $s_{c,l}/s_c$ are identical among all coflows $c\in F_l$ for each link l, then the optimal placement that minimizes (2) is always the one that minimizes the bottleneck fair-sharing CCT of the newly arrived coflow as predicted by (11).*

Remark: We note that compared to the case of flow scheduling (Proposition 4.1), the invariance property for coflow scheduling only holds in very special cases when all the coflows split traffic in the same way. It means that in contrast to flow placement, there is rarely a uniformly optimal placement for coflows under different coflow scheduling policies, and it is crucial to customize coflow placement according to the underlying coflow scheduling policy in the network.

Algorithm 1: Task Placement Algorithm

```
   Input: $< predFCTNode, nodeState >$
   Output: $< assignNode >$
1  /*predFCTNode = predicted task completion time of the node.
     nodeState = minSize of flows scheduled on a node */
2  Initialize: $assignNode = 1, found = 0.$;
3  for $node := 1$ to $M$ do
4      /* identify candidate hosts */;
5      /* if node is empty or the nodeState is larger than task size*/;
6      if ($predFCTNode == null$) || ($nodeState >= taskSize$ )
         then
7          $NodeSet += node$;
8          $found == 1$;
9  /* If no host available, choose the hosts one-hop away */;
10 if ($found != 1$) then
11     /* Among the hosts, 1-hop away */;
12     $NodeSet += node$;
13 /* Select the best host that gives minimum FCT */;
14 for $nodes := 1$ to $NodeSet$ do
15     if $predFCTNode <= minFCT$ then
16         $assignNode = node$;
17 return $assignNode$;
```

5. NEAT DESIGN

NEAT targets data-intensive applications such as MapReduce or directed acyclic graph DAG-based systems (such as Spark, Tez etc.). Such applications run in multiple stages and NEAT aims at placing tasks in each stage with the goal of minimizing the average completion time of all the active tasks in the system. For example, a MapReduce job has two stages, Map (where input data is processed) and Reduce (where results of Map stage are summarized) and each stage may have a data shuffle part and a data processing part. Both data shuffle and data processing contribute to the total completion time of a job. Similarly, DAG-based applications can be considered an extension of MapReduce applications, which can have multiple Map/Reduce phases [12]. Therefore, in this work for simplicity, we use MapReduce as the base model and then extend it to DAG for multi-stage jobs.

5.1 NEAT Task Placement

In this section, we discuss how a new task is placed in NEAT framework and it's applications to MapReduce and DAG-based applications.

5.1.1 Flow Placement

NEAT places a new task in two steps: First, it identifies a set of candidate hosts for task placement (see algo 1 lines 4-14) and next, it selects a best host among these for task execution (see algo 1 lines 15-18).

1) *Identification of preferred hosts:* For a given task, NEAT divides nodes into *preferred* hosts and *non-preferred* hosts based on the the size of the current task and the node state (defined as the smallest remaining flow size on each node). If the task generates a flow of size s, then a node is a preferred host if it is idle or all the flows currently scheduled on this node are no smaller than s. Using preferred hosts provides various benefits. First, it reduces the communication overhead among the network daemons and task placement daemons. For each new task, the task daemon contacts only a subset of hosts to get predicted task completion times. Thus, it improves the time taken to search the right candidate such that the selection time is linear in the number of candidate hosts. Secondly, using preferred hosts compliments the benefits of different network scheduling policies. For example, under the Fair or LAS policy, it ensures that long flows are not placed with existing short flows (although short flows can be placed with existing long flows under LAS scheduling), thus improving separation between short and long flows and reducing the switch queuing delays experienced by the short flows. Similarly, under the SRPT policy, it makes sure that the new task can start data transfer immediately. The preferred hosts are further compared in the next step. If there is no preferred host, such that every node has at least one flow smaller than the flow of the current task, then this preference is ignored, i.e., all the nodes are considered preferred hosts. NEAT, in case of large set of preferred hosts, can also consider input data placement to limit the number of preferred hosts for each task.

2) *Selection of a best host:* Given a set of preferred hosts, the next step is to select a host that achieves the minimum task completion time. The *completion time* of a task depends on: i) the time required to transfer input data, and ii) the time required to process the data. The data transfer time is captured by the predicted FCT/CCT, which is obtained from the network daemon (see Section 3). The data processing time depends on the available node resources (e.g., CPU and memory), which can be obtained from an existing per-node resource manager (e.g., Hadoop node manager). In the current design, NEAT compares the available node resources of each preferred host with requirements imposed by the application (e.g., minimum CPU and memory) to identify candidate hosts, and then selects the one that gives the minimum data transfer time (see algo 1 lines 15-18). However, this design can be easily extended with a processing time predictor to minimize the total task completion time; we leave the details to future work.

Remark: NEAT is an online scheduler that greedily optimizes the performance of each new task. We choose this design because: (i) it significantly simplifies the computation per placement, and (ii) it approximates the optimal placement under certain conditions as specified in Proposition 4.1 and 4.2. Note that NEAT only places each task once at the beginning of its lifetime, and does not move any task during its execution.

5.1.2 Coflow Placement

Compared to flow placement, coflow placement poses additional challenges. Ideally, we want to jointly place all the flows in a coflow to minimize the completion time of the slowest flow. However, in the case of one-to-many or many-to-many coflows, joint placement of all the flows in a coflow has exponential complexity due to the exponentially many possible solutions. To solve this problem, we use a sequential heuristic, where we first place the largest flow within the coflow using the flow placement algorithm, and then place the second largest flow using the same algorithm based on the updated network state, and so on. This sequential heuristic used for coflow placement should have quadratic complexity. The reason for placing flows in descending order of their sizes is that larger flows are more likely to be the critical flows determining the completion time of the coflow and hence should be placed on nodes that have more available resources. We leave evaluation of other coflow placement algorithms as a future work. Note that many-to-one coflows do not have the complexity problem and thus can be placed optimally.

5.1.3 Application to MapReduce Scheduling

NEAT considers each MapReduce job as a concatenation of two tasks, one for the Map stage and one for the Reduce stage. Since there are generally multiple Map "tasks" in a job[1], this requires NEAT to place two coflows, one many-to-many coflow for reading input data into Map tasks and one many-to-one coflow for shuffling intermediate results to Reduce task (or many-to-many if there are multiple Reduce tasks). In both the cases, NEAT strives to achieve data locality if possible (as a node with the input data will have zero FCT/CCT), and minimizes data transfer time otherwise. For example, for placing Map tasks, it can leverage the data placement policy of the cluster filesystem (such as HDFS in Apache Yarn). In HDFS, data is usually placed on multiple locations for fault tolerance and to provide better data locality. During Map task placement, NEAT can either place task on the nodes that has the input data available or choose the nodes using it's placement heuristic.

5.1.4 Application to DAG Scheduling

DAG-style applications can be modelled as an extension of MapReduce applications, except that a DAG may have multiple shuffle stages. NEAT considers each stage of a DAG job as a separate task and places flows (coflows) within a task using the procedure discussed in sections 5.1.1 and 5.1.2. The placement algorithm used by NEAT encourages placing tasks of same job on nodes close to the input data.

5.2 NEAT Optimizations

NEAT needs the state of all the active flows in the network to make optimal task placement decisions, which can incur significant overhead at both the network daemons and the task placement daemon. To reduce the overhead, NEAT introduces two optimizations: (i) instead of keeping individual states of all the active flows, each network daemon only maintains a compressed state with a constant size, (ii) instead of contacting all the network daemons, the task placement daemon determines a subset of nodes as candidate hosts based on local information and only contacts the network daemons at the candidate hosts.

Compressed Flow State: Keeping the original flow state requires space that grows linearly with the number of active flows, which limits scalability of the network daemon (with respect to network load). NEAT addresses this issue by approximating the FCT/CCT prediction using a compressed flow state. Each network daemon compresses information of its local flows (coflows) by quantizing the flow sizes into a fixed number of bins and maintaining summary statistics (e.g., total size and number of flows) in each bin. Compared with the original flow state that keeps track of individual flows, this compressed state has a size that is determined by the number of bins, regardless of the number of flows in the network.

We refer to the set of active flows F_l on link l as the *flow state* of link l. The flow states of all the links form the overall flow state F of the network. Note that under FCFS, we can easily compress F_l by storing only the total load ($\sum_{c\in F_l} s_f$ for flow scheduling and $\sum_{c\in F_l} s_{c,l}$ for coflow scheduling), which suffices for FCT/CCT prediction. We thus focus on the other scheduling policies in the sequel.

For FCT prediction, our idea is to divide flows on each link into a finite number of bins $n = 1, \ldots, N$, where we keep the following parameters for each bin: the minimum and maximum flow size ($s^{(1)}_{l,n}$, $s^{(2)}_{l,n}$), the total flow size (in number of bytes) $b_{l,n}$, and the number of flows $c_{l,n}$. In essence, we compress a set of flow sizes into a histogram of flow sizes with flexible bin boundaries ($s^{(1)}_{l,n}$, $s^{(2)}_{l,n}$). Using the compressed flow state, we can approximate (4) by:

$$\mathrm{FCT}^{\mathrm{FAIR}}(f_0, l) \approx \frac{1}{B_l}\left(s_{f_0} + \sum_{n=1}^{p} b_{l,n} + s_{f_0}\sum_{n=p+1}^{N} c_{l,n}\right), \tag{18}$$

where $p = m_l(s_{f_0})$ and $m_l(s) \in \{1, \ldots, N\}$ is the index of the bin containing flow size s (i.e., $s^{(1)}_{l,m_l(s)} \le s < s^{(2)}_{l,m_l(s)}$). Other predictions in § 4.1 can be approximated similarly, although we only need to approximate (4) if the invariance condition in Proposition 4.1 is satisfied. Note that storing $b_{l,n}$ is optional as it can be approximated by $c_{l,n}s$ for some $s \in [s^{(1)}_{l,n}, s^{(2)}_{l,n})$.

For CCT prediction, the compressed flow state in each bin has two additional attributes: $d_{l,n}$, denoting the total load on link l (i.e., sum of $s_{c,l}$ over coflows in this bin), and $e_{l,n}$, denoting the total normalized load on link l (i.e., sum of $s_{c,l}/s_c$ over coflows in the bin). Assuming $q = m_l(s_{c_0})$, for Fair/LAS scheduling, we can approximate (11) by

$$\mathrm{CCT}^{\mathrm{FAIR}}(c_0, l) \approx \frac{1}{B_l}\left(s_{c_0,l} + \sum_{n=1}^{q} d_{l,n} + s_{c_0}\sum_{n=q+1}^{N} e_{l,n}\right), \tag{19}$$

and (13) by

$$\sum_{c\in F_l} \Delta\mathrm{CCT}^{\mathrm{FAIR}}(c, l) \approx \frac{s_{c_0,l}}{B_l s_{c_0}}\left(\sum_{n=1}^{q} b_{l,n} + s_{c_0}\sum_{n=q+1}^{N} c_{l,n}\right). \tag{20}$$

[1]Note that NEAT defines "task" differently from the MapReduce framework.

Under TCF scheduling, we can approximate the righthand side of (17) by

$$\frac{1}{B_l}\left(s_{c_0,l}+\sum_{n=1}^{q} d_{l,n}+s_{c_0,l}\sum_{n=q+1}^{N} c_{l,n}\right). \quad (21)$$

Here storing $e_{l,n}$ is also optional as it can be approximated by $d_{l,n}/s$ for some $s \in [s_{l,n}^{(1)},\ s_{l,n}^{(2)})$.

Remark: The compressed flow state has a size that is linear in the number of bins (which is a design parameter), regardless of the number of flows in the network. The cost of this compression is the loss of accuracy in the FCT/CCT prediction, caused by uncertainty in the sizes of flows (coflows) that are in the same bin as the newly arrived flow (coflow). The bin sizes can be set based on the datacenter traffic distribution. For example, for heavy tailed traffic [24], one can use exponentially growing bin sizes to have smaller bin sizes for short flows and large bin sizes for long flows.

Reduced Communication Overhead: NEAT uses distributed components (network daemons) spread across the network to maintain the flow states (see Figure 2). To minimize the communication overhead between the task placement daemon and the network daemons, the network daemons do not always report their updated flow states; instead, the task placement daemon pings the network daemons to get the predicted task performance when it needs to place a new task.

In a large cluster, the task placement daemon has to contact a large number of network daemons to place a task. To further reduce the overhead, the task placement daemon uses local information to reduce the number of network daemons it needs to contact. Specifically, it only contacts a network daemon if (i) the node it resides on is sufficiently close to the input data (e.g., in the same rack or a rack one-hop away from the input data), and (ii) if the node state (i.e., the smallest residual size of flows currently scheduled on this node) is no smaller than the size of the current task. Note that without contacting the network daemon, the task placement daemon does not know the current node state. In our design, the task placement daemon caches the node states previously reported by the network daemons and uses the cached values as estimates.

6. EVALUATION

We evaluate NEAT using trace-driven simulations based on ns2 [37] as well as experiments on a small-scale testbed.

6.1 Simulation Settings

Datacenter Topology: We use a 160 node, 3-tier multi-rooted topology for our evaluation comprising layers of ToR (Top-of-Rack) switches, aggregation switches and core switches, similar to [38, 39, 40]. Each host-ToR link has a capacity of 1 Gbps whereas all other links are of 10 Gbps. The end-to-end round-trip propagation delay (in the absence of queuing) between hosts via the core switch is 300μs.

Traffic Workloads: We consider traffic workloads that are derived from patterns observed in production datacenters.

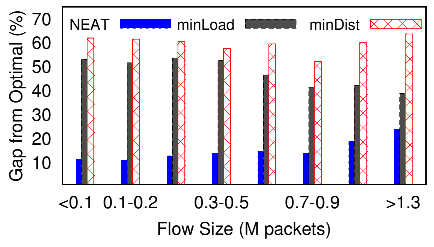
Scheme	Parameters
DCTCP	qSize = 250 pkts
	markingThresh = 65
L^{2}DCT	minRTO = 10 msec
PASE	qSize = 250 pkts
	minRTO (flows in top queue) = 10 msec
	minRTO (flows in other queues) = 200 msec
	numQue = 8

Table 1: Default parameter settings used in simulation.

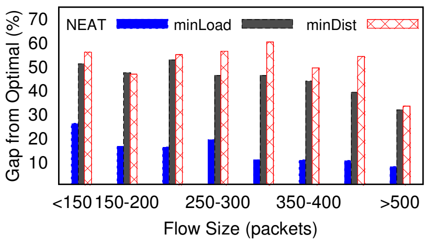
We evaluate NEAT using the benchmark Hadoop (map-reduce)[1] and web-search [16] workloads. These workloads contain a diverse mix of short and long flows with a heavy-tailed flow size distribution. In the web-search workload, more than 75% of all bytes are from 50% of the flows with sizes in the range [1,20MB]. The hadoop workload is less skewed: ~50% of the flows are less than 100MB in size and 4% flows are larger than 80GB. Note that we always use these settings unless specified otherwise.

Protocols Compared: We compare NEAT with two task scheduling strategies: load aware placement (minLoad) and locality aware placement (minDist). We consider several state-of-art datacenter network scheduling strategies including DCTCP (that implements Fair) [24], L^{2}DCT (that implements LAS) [15], and PASE (that implements SRPT) [17] for flow scheduling and Varys [21] for coflow scheduling. minLoad places each task on the node with the minimum network load, defined as the utilization ratio of its link to ToR. minDist places each task as close as possible to its input data, as proposed by earlier schemes [12]. We also test coflow performance using smallest coflow first (SCF) heuristic. We implemented DCTCP, L^{2}DCT, PASE, and Varys using the source code provided by the authors to evaluate their scheme. The parameters of these protocols are set according to the recommendations provided by the authors, or reflect the best settings, which we determined experimentally (see Table 1). For FCT prediction, we use Fair sharing based prediction model unless stated otherwise and for CCT prediction, we use the prediction models corresponding to each evaluated coflow scheduling scheme.

Performance Metrics: We consider the average flow completion time (AFCT) for the flow-based scheduling and coflow completion time (CCT) for coflow-based scheduling schemes. We use *gap from optimal* as the performance metric. For example, for FCT based schemes, the gap from optimal is calculated as $(FCT - FCT^{opt})/FCT^{opt}$, where FCT^{opt} is the optimal FCT defined as the time it takes to complete the data transfer when that is the only flow (coflow) in the network. Note that the gap from optimal equals slowdown (a.k.a. stretch) minus one. The gap from optimal tells us the margin for performance improvement for different kind of network scheduling policies. For example, we observe in our evaluation that the room for improvement is more for suboptimal scheduling policies, in terms of FCT, (such as FAIR, LAS) and less for near-optimal network scheduling policies (such as SRPT).

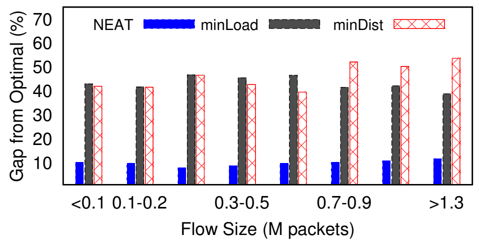


(a) Hadoop

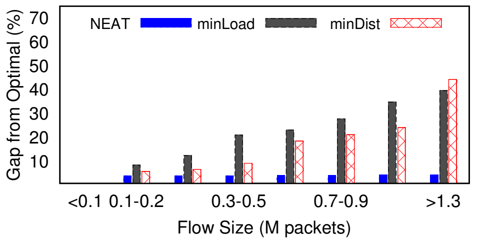


(b) Websearch

Figure 5: Flow placement under DCTCP

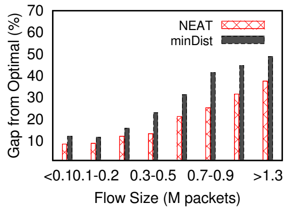


(a) LAS (L^2DCT)

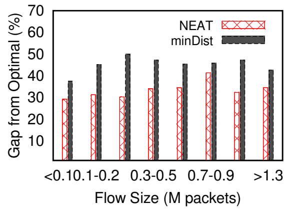


(b) SRPT (PASE)

Figure 6: Flow placement under Hadoop workload



(a) Varys



(b) SCF

Figure 7: CoFlow placement for Hadoop workload

6.2 Macrobenchmarks

NEAT improves application performance upto 3.7x, compared to the existing task placement algorithms that only consider data and compute, by incorporating network state and network scheduling policies into task placement decisions. Below, we first discuss NEAT performance in placing flows and then its performance in placing coflows.

Flow placement performance under Fair sharing: Figure 5 shows that NEAT outperforms minLoad and minDist by up to 3.6x for web-search workload and up to 3.7x for Hadoop workloads when the network employs Fair sharing scheduling policy. This is because NEAT chooses the node with minFCT to finish flows faster and uses node state NEAT to avoid placing short flows on same links with long flows. Note that many of the existing policies deployed in the datacenter employ Fair sharing discipline. This shows that even if datacenter does not implement any optimized network scheduling mechanism, NEAT can improve application performance significantly by being aware of network state.

Flow placement performance under different network scheduling policies: Figure 6 shows NEAT performance for Hadoop workload when the network uses L^2DCT (LAS) and PASE (SRPT) as network scheduling policies. NEAT improves the performance by 2.7x compared to minLoad and by upto 3.2x compared to minDist when used with L^2DCT. NEAT also improves the performance by 30% compared to minLoad and by upto 20% compared to minDist when used with PASE. Note that PASE is a near-optimal algorithm and there is very little performance improvement margin for NEAT. Also, observe that minDist and minLoad perform differently under different scheduling policies, whereas, NEAT consistently performs better than both for all the evaluated network scheduling policies. We observe similar performance trends with websearch workloads.

CoFlow placement performance under different network scheduling policies: NEAT improves the CCT performance for Hadoop workloads by upto 25% for two coflow scheduling policies, 1) Varys (Figure 7(a)) and 2) Smallest Coflow First (SCF) (Figure 7(b)). Coflow placement requires a group of flows to be placed in a batch. For coflow placement, NEAT works as described in Section 5.1, while minLoad and minDist are modified as follows - For minLoad, we place each flow of the coflow separately, in the decreasing order of flow sizes, on a node that has the minimum network load. For minDist, our goal is to place flows of the same coflow within the same rack and close to input data and we start by placing the largest flow of a coflow first. NEAT performs better under both coflow scheduling policies, however its gains are limited by the performance of underlying coflow scheduling policy. For example, minDist with Varys scheduling performs better than NEAT with SCF scheduling. The reason for this is that Varys, which adjusts the flow rates based on the smallest bottleneck heuristic, is a much better heuristic than SCF. However, we observe that the gap from optimal is substantial (30% to 50%), for these strategies, for large flow sizes with Varys and nearly for all flow sizes with SCF. The reason for this gap is that the optimal completion time is not always achievable, even under the optimal scheduling, as our definition of optimal ignores resource competition from other flows/coflows. Also, many of these protocols implement certain approximations, instead of the actual optimal scheduling algorithm, to be more deployment friendly.

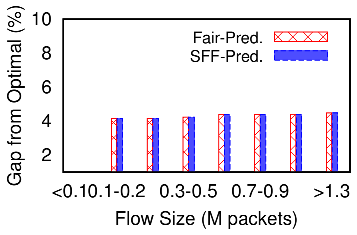


Figure 8: Fair prediction vs SRPT prediction

6.3 Microbenchmarks

In this section, we discuss various aspects of NEAT design, with the help of several micro-benchmarks. We analyze NEAT dynamics using Hadoop workload under SRPT (PASE) network scheduling policy, unless stated otherwise. Our evaluation shows that NEAT optimizations (such as prediction assuming FAIR policy only, preferred hosts) improve the application performance and it can predict task completion times quite accurately.

Benefits of using Fair predictor: Figure 8 shows the NEAT performance when task completion time is predicted using the Fair-sharing model (Eq. 4) or the SRPT-sharing model (Eq. 7) when the underlying network scheduling mechanism is SRPT. Both predictors achieve similar performance, therefore, even though the network performs SRPT flow scheduling, NEAT can perform FCT prediction by assuming the fair-sharing principle. This validates our proposition, in §4.1.4, which states that for flow based network scheduling mechanisms, task placement using minimum predicted task completion time, assuming FAIR predictor, can provide better performance for any flow based network scheduling policy.

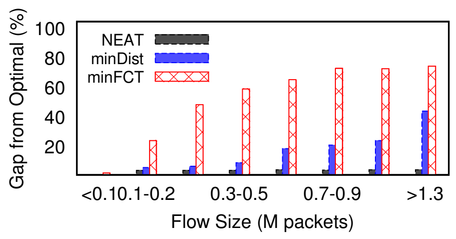


Figure 9: Benefits of using preferred hosts placement

Benefits of using preferred hosts aware placement: Figure 9 shows the benefits of using preferred hosts for making task placement decisions. To evaluate the impact on performance, we compare NEAT performance to minFCT. minFCT strategy makes task placement decision based on the predicted task completion times alone and ignores the node states i.e., it places each task on to a node with the smallest predicted FCT and does not consider the priority of the tasks already scheduled on that node. Figure 9 shows that minFCT placement degrades application performance by up to 50%, and performs even worse than minDist placement. NEAT on the other hand achieves better performance by being aware of node states. Intuitively, minFCT hurts performance in two ways: i) it groups short flows together, thus increasing fair sharing among short flows. ii) for long flows it selects nodes with many short flows, thus long flows experience more preemption by the currently scheduled and newly arriving short flows. We note that the comparison between minFCT and minDist is workload-dependent, e.g., for the workloads, where short and long flows have proportional number of bytes, minFCT performs similar to or better than minDist placement. Thus, using preferred hosts based task placement not only minimizes communication overhead, but it also improves application performance.

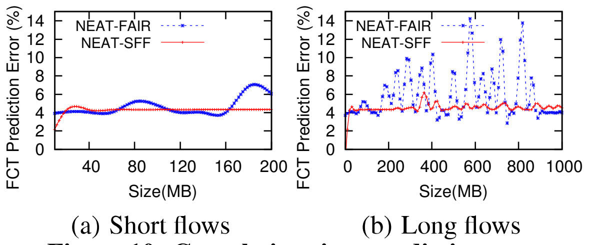


Figure 10: Completion time prediction error

Flow completion time prediction accuracy: NEAT can predict flow completion times with reasonable accuracy as shown in Figure 10. NEAT performance depends on accurately predicting completion times. It is important to keep the prediction error small because NEAT only considers currently active tasks while predicting FCTs and future task arrivals may make current placement decisions sub-optimal. Figure 10 reports the prediction accuracy, which is calculated as $(FCT^{flow} - FCT^{pred})/FCT^{pred}$, where FCT^{flow} is the actual FCT and FCT^{pred} is the FCT predicted by NEAT. Figure 10 shows the prediction accuracy for short flows (Figure 10(a)), and long flows (Figure 10(b)) and we observe that the prediction error increases with the flow size. This is because, with the increase in flow size, the flow spends more time in the network and is affected more by the tasks arriving later in time. In many practical datacenter applications, most tasks generate short flows, for which NEAT can predict the FCT within 5% error. This experiment shows that NEAT is not very sensitive to mis-prediction in task completion time estimate and it is able to achieve good performance even in the presence of small inaccuracies in task completion time estimates.

6.4 Testbed Evaluation

We discuss testbed implementation and evaluation results.

Implementation: NEAT follows a master/slave architecture similar to Hadoop-Yarn [41], and we implement NEAT in Scala using support from Akka for messaging between task placement daemon (TD) and network daemons (ND). We currently implement NEAT as an application layer program by extending Varys implementation [21] and leave the integration with Hadoop-Yarn as a future work. Similar to Yarn, the TD acts as the resource manager (RM) and makes task placement decisions, and NDs act as node managers, which maintain flow state of the active jobs and share the network state upon receiving requests from the task placement daemon. The TD accepts a user job similar to RM and invokes network daemon APIs at the nodes (endhosts). Cluster framework (e.g., Spark or Hadoop) drivers can submit a job using the `register()` API to interact with NEAT. Upon receiving a request from the TD, NDs update their local states and reply with predicted task performance and node state. The TD then runs our placement algorithm

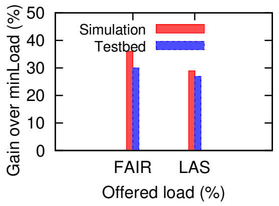


Figure 11: Testbed: minLoad vs NEAT

to schedule flows on selected nodes. To support multiple network scheduling policies, we reuse Varys' `put()` and `get()` method to send and receive data between different nodes. Cluster frameworks can use NEAT `InputStream` and `OutputStream` to leverage various network scheduling policies. The desired network scheduling policy can be specified during network configuration phase.

Evaluation Results: We evaluate NEAT using a 10-node single-rack cluster consisting of DELL servers (with 1G NICs) and PICA-8 Gigabit switch with ECN support enabled. We consider scenario with all-to-all traffic based on Hadoop workload [1] and generate traffic such that the average network load is 50%. We first generate the traffic using ns2 and replay the same traffic in the testbed. The hadoop workload generates many long flows as ∼50% of the flows are less than 100MB in size and 4% flows are larger than 80GB. We compare NEAT to minLoad task placement under Fair (implemented by DCTCP) and LAS (implemented by L^2DCT) network scheduling policies. Because of the small scale of the testbed, we do not compare with minDist (since all node pairs have the same distance), and only evaluate using flow-based network scheduling policies.

Our evaluation shows that, compared to minLoad, NEAT can improve application performance by upto 30% under Fair scheduling and upto 27% under LAS scheduling policy, see Figure 11. The gain in testbed scenario is much less than the large scale simulations, and the reason for small gain is the small scale of the testbed. In our evaluation, we observe that a lot of long flows are generated that run simultaneously in the network and share links with short flows. Both the placement algorithms spread large flows across all the nodes in the network, over the time, and therefore increase the completion time of short flows because they share the network links and switch buffers, most of which are occupied by long flows. This effect is not observed in large scale networks where a flow can mostly have many available nodes to chose from. Although, the size of the testbed limits the amount of improvement, NEAT is able to improve the performance under both the network scheduling policies by making better placement decisions. We also compare testbed results to ns2 simulations using same settings as testbed setup and observe similar performance gains.

7. DISCUSSION

Input Data Placement: NEAT can leverage the knowledge of input data locations to improve scalability and make more accurate placement decisions. However, input data may not be optimally placed, which limits the performance of NEAT. In such cases, NEAT can be combined with strategies like corral [12] to make input data and task placement decisions based on the task execution history.

Flow Size Information: NEAT requires flow size information to predict task performance, however, flow size may not always be available. For example, the motivation behind using LAS-based network scheduling is to address this challenge [15]. In such scenarios, NEAT can use approximate flow sizes based on the task execution history for performance prediction, as we have shown that NEAT minimizes task completion time even with some inaccuracy in predicted task completion times (§6.3). We leave evaluation of such scenarios for future work.

Generalization of Network Topologies: Currently, NEAT assumes a single-switch topology and predicts task performance based on the edge links only, however, any link in the network can be bottleneck in other topologies. To make better placement decisions, one can use a distributed state maintenance mechanism similar to PASE [17] where a dedicated arbitrator maintains flow state for each link in the network. NEAT can choose the destination that gives the minimum completion time along the path from source to destination. Nevertheless, our evaluation shows that the single-switch abstraction already provides substantial performance improvement over existing solutions (section 6).

NEAT Scalability: NEAT introduces several optimizations to make system more scalable. For example, to mitigate the communication overhead among daemons, it uses node states while contacting nodes for performance prediction estimate. This reduces the amount of overhead messages sent over the network for communication among network and task placement daemons. Also, to reduce the overhead of task state maintenance by the network daemons, NEAT uses compressed flow state while predicting the task performance. As shown in §6.3, these optimizations help improve NEAT performance, however, we leave large scale scalability analysis of these optimizations as a future work.

8. CONCLUSION

This work proposes a task placement framework that takes into account the network scheduling policy and shows that significant performance gains can be obtained over task schedulers oblivious to network scheduling. NEAT task scheduler makes task placement decisions by using a pluggable task performance predictor that predicts the data transfer time of a given task under given network conditions and network scheduling policy. Despite several simplifying assumptions, NEAT achieves promising performance improvement under several commonly used network scheduling policies. Its pluggable architecture allows easy incorporation of sophisticated resource and application models.

Acknowledgement

This work is partially supported by the National Science Foundation under Grant Numbers CNS-1318563, CNS-1524698, and CNS-1421407, and the National Natural Science Foundation of China under Grant Numbers 61472184 and 61321491, and the Jiangsu High-level Innovation and Entrepreneurship (Shuangchuang) Program. Part of the work was done while Ali Munir was at IBM Research Watson. The authors would like to thank anonymous reviewers and shepherd Mosharaf Chowdhury for their useful feedback.

9. REFERENCES

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[2] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *ACM SIGMOD 2010*, pages 135–146.

[3] William Gropp, S Huss-Lederman, A Lumsdaine, E Lusk, B Nitzberg, W Saphir, and M Snir. Mpi. *The Complete Reference. The MPI-2 Extensions*, 2, 1998.

[4] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007.

[5] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278. ACM, 2010.

[6] Faraz Ahmad, Srimat T Chakradhar, Anand Raghunathan, and TN Vijaykumar. Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–12. USENIX Association, 2014.

[7] Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy Katz. The case for evaluating mapreduce performance using workload suites. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 390–399. IEEE, 2011.

[8] Paolo Costa, Austin Donnelly, Antony Rowstron, and Greg O'Shea. Camdoop: exploiting in-network aggregation for big data applications. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 3–3. USENIX Association, 2012.

[9] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276. ACM, 2009.

[10] Ganesh Ananthanarayanan, Srikanth Kandula, Albert G Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, volume 10, page 24, 2010.

[11] Jian Tan, Alicia Chin, Zane Zhenhua Hu, Yonggang Hu, Shicong Meng, Xiaoqiao Meng, and Li Zhang. Dynmr: Dynamic mapreduce with reduce task interleaving and maptask backfilling. In *ACM EuroSys*, page 2. ACM, 2014.

[12] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication*, pages 407–420. ACM, 2015.

[13] Chi-Yao Hong, Matthew Caesar, and P Godfrey. Finishing flows quickly with preemptive scheduling. *ACM SIGCOMM Computer Communication Review*, 42(4):127–138, 2012.

[14] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. Detail: reducing the flow completion time tail in datacenter networks. *ACM SIGCOMM Computer Communication Review*, 42(4):139–150, 2012.

[15] Ali Munir, Ihsan Ayyub Qazi, Zartash Afzal Uzmi, Aisha Mushtaq, Saad N Ismail, M Safdar Iqbal, and Basma Khan. Minimizing flow completion times in data centers. In *INFOCOM, 2013 Proceedings IEEE*, pages 2157–2165. IEEE, 2013.

[16] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review*, 43(4):435–446, 2013.

[17] Ali Munir, Ghufran Baig, Syed M Irteza, Ihsan A Qazi, Alex X Liu, and Fahad R Dogar. Friends, not foes: synthesizing existing transport strategies for data center networks. In *Proceedings of the ACM conference on SIGCOMM*, pages 491–502. ACM, 2014.

[18] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, 2015.

[19] Mosharaf Chowdhury and Ion Stoica. Coflow: a networking abstraction for cluster applications. In *ACM Hotnets*, pages 31–36. ACM, 2012.

[20] Fahad R Dogar, Thomas Karagiannis, Hitesh Ballani, and Antony Rowstron. Decentralized task-aware scheduling for data center networks. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 431–442. ACM, 2014.

[21] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 443–454. ACM, 2014.

[22] Yangming Zhao, Kai Chen, Wei Bai, Chen Tian, Yanhui Geng, Yiming Zhang, Dan Li, and Sheng Wang. Rapier: Integrating routing and scheduling for coflow-aware data center networks. In *Proc. IEEE INFOCOM*, 2015.

[23] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. In *Proceedings*

of the ACM Conference on Special Interest Group on Data Communication, pages 393–406. ACM, 2015.

[24] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). *ACM SIGCOMM computer communication review*, 41(4):63–74, 2011.

[25] Dhruba Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11(2007):21, 2007.

[26] Mosharaf Chowdhury, Srikanth Kandula, and Ion Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 231–242. ACM, 2013.

[27] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, volume 11, pages 24–24. USENIX, 2011.

[28] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22. USENIX, 2011.

[29] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-resource fair queueing for packet processing. In *ACM SIGCOMM*, volume 42, pages 1–12. ACM, 2012.

[30] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *ACM EuroSys*, pages 365–378. ACM, 2013.

[31] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *ACM SoCC*, page 5. ACM, 2013.

[32] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, volume 8, page 7, 2008.

[33] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In *NSDI*, volume 13, pages 185–198. USENIX, 2013.

[34] Xiaoqi Ren, Ganesh Ananthanarayanan, Adam Wierman, and Minlan Yu. Hopper: Decentralized speculation-aware cluster scheduling at scale. In *ACM SIGCOMM*, pages 379–392. ACM, 2015.

[35] Hyunseok Chang, Murali Kodialam, Ramana Rao Kompella, TV Lakshman, Myungjin Lee, and Sarit Mukherjee. Scheduling in mapreduce-like systems for fast completion time. In *INFOCOM, 2011 Proceedings IEEE*, pages 3074–3082. IEEE, 2011.

[36] Fangfei Chen, Murali Kodialam, and TV Lakshman. Joint scheduling of processing and shuffle phases in mapreduce systems. In *INFOCOM, 2012 Proceedings IEEE*, pages 1143–1151. IEEE, 2012.

[37] The network simulator - ns-2. http://www.isi.edu/nsnam/ns/.

[38] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74. ACM, 2008.

[39] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. Deadline-aware datacenter tcp (d2tcp). *ACM SIGCOMM Computer Communication Review*, pages 115–126, 2012.

[40] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: Meeting deadlines in datacenter networks. In *ACM SIGCOMM Computer Communication Review*, pages 50–61. ACM, 2011.

[41] Apache hadoop yarn. https://goo.gl/qEdpo5.