Network Latency and Application Performance Aware Cluster Scheduling in Data Centers

Diana Andreea Popescu and Andrew W. Moore

Abstract

Data-center-based cloud computing has revolutionized the way businesses use computing infrastructure. Instead of building their own data centers, companies rent computing resources and deploy their applications on cloud hardware. Providing customers with well-defined application performance guarantees is of paramount importance. A user's application performance is subject to the constraints of the resources it has been allocated and to the impact of the network conditions in the data center. Given the network latency variability observed in data centers, applications' performance is also determined by their placement within the data center. We present NoMora, a cluster scheduling architecture whose core is represented by a latency-driven, application-performance-aware cluster scheduling policy. The policy places the tasks of an application taking into account the expected performance based on the measured network latency between pairs of hosts in the data center. If a tenant's application experiences increased network latency, and thus lower application performance, their application may be migrated to a better placement. Experiments on a testbed and in simulations show that our architecture improves the overall average application performance by up to 32.5 and 42 percent, respectively, demonstrating that application performance can be improved by exploiting the relationship between network latency and application performance.

Introduction

Cloud computing has revolutionized the way businesses use computing infrastructure. Instead of building their own data centers, companies rent computing resources from cloud providers (e.g., Amazon AWS, Google Cloud Platform, and Microsoft Azure), and deploy their applications on cloud provider hardware. Network latency variability is still common in multi-tenant data centers [1, 2], and even small amounts of delay, on the order of tens of microseconds, may lead to significant drops in application performance [3]. An important factor in achieving predictable application performance is understanding the networking requirements of the application in terms of bandwidth and latency. Once these requirements have been determined, they have to be incorporated into the data center management stack. This can be done in-network, through scheduling [4], or prioritizing the application's flows [5], and/or at the end host, through bandwidth allocation [6]. In these situations, the placement of the application's tasks is assumed to be known before incorporating its network resource demands. If the tasks' placements are not known a priori or if they can be changed, the network resource demands can be incorporated at a higher level in the data center management stack, namely in the cluster scheduler. Previous works [5–9] have looked at providing network bandwidth and tail latency guarantees, and, as a result, the application would meet its performance guarantees. However, none of the existing cluster schedulers have considered placing an application's tasks taking into account their expected application performance determined by the current network conditions. Instead of considering network-level metrics such as flow completion time to improve application performance, our work demonstrates how high-level application performance metrics such as job completion time can be linked to network-level measurements, leading to improvements in application-level performance.

If we know how the application reacts to latency and the current network conditions, we can place the tenant's application in the data center ensuring the best performance achievable under the current network conditions. By measuring dynamically the network latency in the data center and having a model of the application performance dependent upon network latency, the relationship between network latency and application performance can help cloud customers to determine the performance their application can achieve under certain network conditions and can guide cloud operators in selecting the network latency ranges that best suit the needs of their customers.

In this article, we present a cluster scheduling architecture, NoMora,[1] whose core is a cluster scheduling policy that places the tasks of a job (application) taking into account the expected application performance based on the measured network latency between pairs of hosts in the data center. If a tenant's application experiences increased network latency, and thus lower application performance, the application may be migrated to different hosts.

NoMora paves the way for *application-performance-aware cluster schedulers*. Instead of meeting the job's resource requirements, in our work we consider meeting the job's performance requirements.

Given the current network conditions, the optimal performance may not be achievable. In this case, if the tenant whose job has to be scheduled is content with their job running with less than optimal performance, the job is admitted into the system

[1] Mora means delay in Latin, so the name refers to applications being scheduled to not have network delay affecting their performance.

Digital Object Identifier: 10.1109/MNET.001.2100414

The authors are with the University of Cambridge.

0890-8044/22/$25.00 © 2022 IEEE

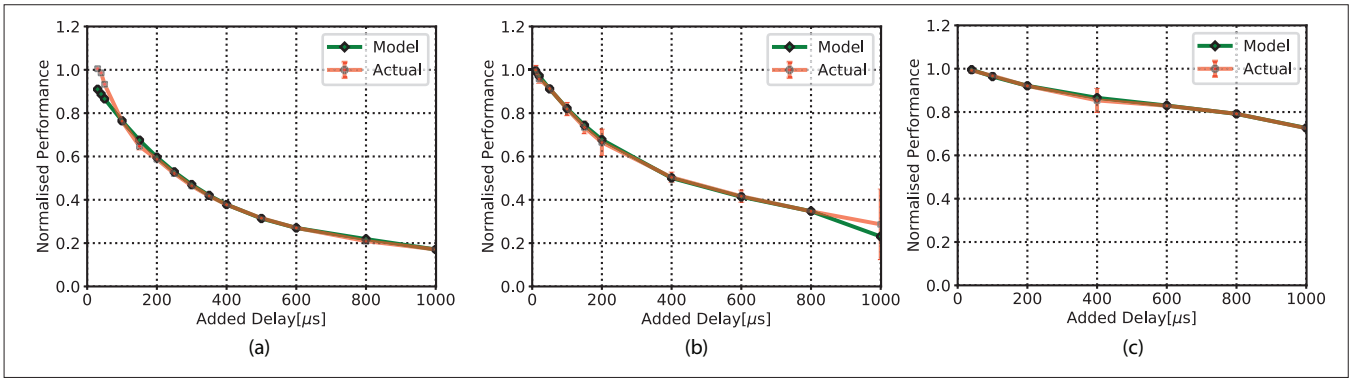

FIGURE 1. Applications' experimental results (actual) and model on the results (model): a) Memcached; b) STRADS Lasso Regression; c) Tensorflow MNIST.

with the best achievable performance given the limits of the current network conditions. Otherwise, admission control is performed, the job being scheduled only if it can run with optimal performance.

NoMora

The NoMora cluster scheduling architecture combines three key elements:

1. Functions of application performance dependent upon network latency
2. Network latency measurement system
3. The latency-driven application performance-aware cluster scheduling policy

For 2, systems such as PTPmesh [10], Pingmesh [11], and NetNORAD [12], can provide the most recently measured network latency between hosts in a data center. The data collected by these systems is fed in real time to our cluster scheduler to aid in making task placement or migration decisions.

Application Performance Functions

We study three client-server applications and distributed machine learning frameworks, Memcached (https://memcached.org/), STRADS (https://github.com/sailing-pmls/stradsl) and Tensorflow (https://www.tensorflow.org/), with different corresponding workloads.

Memcached: a widely used, in-memory, key-value store for arbitrary data. We use the Mutilate (https://github.com/leverich/mutilate) load generator, with the Facebook "ETC" workload, representative of general-purpose key-value stores, and as application performance metric the number of queries/requests per second (QPS).

STRADS: a distributed framework for machine learning algorithms targeted to moderate cluster sizes. We use the Lasso Regression application. Workers communicate only with the master server. We do not allow the use of stale parameters during iterations (no pipelining), and the injected network latency does not change the scheduling of the parameters. While pipelining can reduce the impact of network latency by overlapping network communication with computation, due to the usage of stale parameters and the potential for dependencies between iterations, it may lead to a slower convergence rate. The application performance is the *training time*, which is the same as the job completion time in our case.

Tensorflow: a widely used machine learning framework. We use the MNIST dataset (http://yann.lecun.com/exdb/mnist/) for the handwriting recognition task as input data and Softmax Regression for the training of the model. Tensorflow follows a master-worker model. The parameter updates from workers are aggregated before being applied in order to avoid stale gradients, this being similar to not using the pipeling in STRADS. The application performance metric used is the training time, which in our case is the same as the job runtime.

To determine experimentally the relationship between network latency and application performance, we used a methodology previously described in [3], where we injected increasing amounts of constant latency in a networked system, delaying the packets by a fixed amount of time, and we measured how the application performance changes depending on the amount of inserted network latency. We construct *functions that predict application performance dependent upon network latency* for different applications. To model the relationship between network latency and application performance, we use SciPy's (https://www.scipy.org/) `curve_fit` function. We first normalize the values of the application performance with respect to the baseline performance, which was obtained without any additional latency. We then model the relationship between network latency and normalized application performance by fitting a polynomial function to the results, where the independent variable is the static latency, and the dependent variable is the normalized performance: *normalised_performance* = *p*(*constant_latency*). The experimental data along with the resulting models are presented in Fig. 1. The models have two functions: a constant function, whose value is the baseline performance, and this function gives the performance up to a threshold beyond which additional latency leads to a drop in application performance (40 μs for Memcached, 20 μs for STRADS, 40 μs for Tensorflow), and a polynomial function of order three fit on the experimental data beyond this threshold. The coefficient of determination, R^2, to assess goodness of fit is 0.976 for the Memcached model, 0.994 for the STRADS model, and 0.996 for the Tensorflow model.

Latency-sensitive distributed applications are usually synchronous, meaning that the application blocks on waiting to receive data from a different host over the network before proceeding with the next step. This is the case for the machine learning frameworks we studied, STRADS and Tensorflow, which follow the parameter server

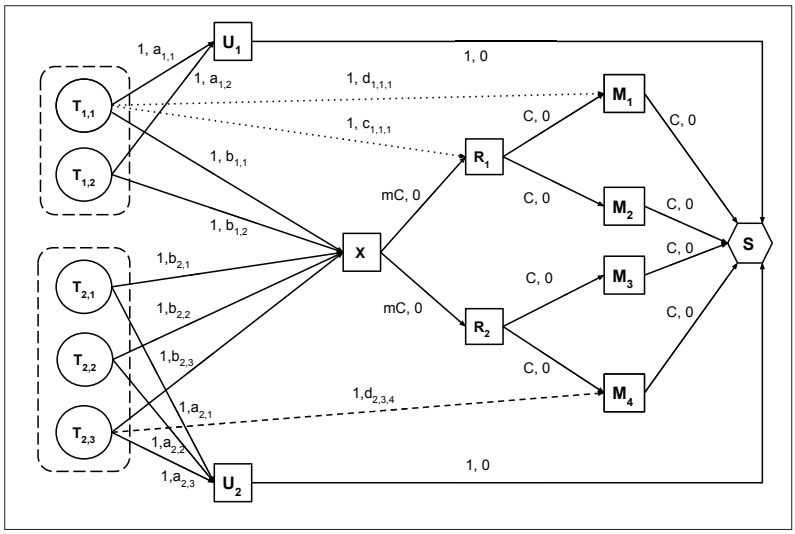

FIGURE 2. A general flow network with annotated capacities and costs on arcs. Job J_1 has tasks $T_{1,1}$ and $T_{1,2}$. Job J_2 has tasks $T_{2,1}$, $T_{2,2}$ and $T_{2,3}$. The unscheduled aggregators are U_1 and U_2. The machines in the cluster are M_1, M_2, M_3, and M_4. Rack aggregators are R_1 and R_2. The cluster aggregator is X. The sink vertex is S.

design for the applications we used, with workers that exchange messages with the parameter server over the network. These frameworks use, in the current iteration, the parameters computed during the previous iteration. This pattern makes them highly dependent on the network, and especially on network latency. Similarly, for key-value stores, like Memcached, that act as an intermediate caching layer between the client and the storage system, the request-response latency is very important, since the store needs to provide fast access to the data. The overall throughput of the Memcached server will decrease when additional latency is injected in the network.

Background on Cluster Scheduling

NoMora extends the Firmament cluster scheduler [13] that models the problem of assigning tasks to machines as a minimum-cost flow problem, which is a centralized scheduler that considers the entire workload across the whole cluster. A scheduling policy defines a flow network representing the cluster, where the nodes define tasks and resources. A job (application) can have several tasks, and a cluster has machines (servers). Events such as task arrival, task completion, machine addition to the cluster, and machine removal from the cluster change the flow network. When cluster events change the flow network, Firmament's min-cost max-flow solver computes the optimal flow on the updated flow network. After the solver finishes running, Firmament extracts the task placements from the optimal flow and applies these changes in the cluster. We next give an overview of how the cluster scheduling problem is mapped to the minimum-cost maximum-flow optimization problem [13, 14].

Flow Network: First, we provide a high-level overview of the structure of the flow network, which can be seen in Fig. 2. By *flow network* we refer to a directed graph where each arc has a capacity and a cost to send flow across that arc. Each submitted task $T_{i,j}$, representing task j of job J_i, is represented by a vertex in the graph, and it generates one unit of flow. The sink S drains the flow generated by the submitted tasks. A task vertex needs to send a unit of flow along a path composed of directed arcs in the graph to the sink S. The path can pass through a vertex that corresponds to a machine (host) M_m, meaning the task is scheduled to run on that machine, or it can pass through a special vertex for the unscheduled tasks of that job, U_i, meaning the task is not scheduled. In this way, even if the task is not scheduled to run, the flow generated by this task is routed through the unscheduled aggregator to the sink.

The graph can have an arc between every task and every machine, but this would make the computation of an optimal scheduling solution in a short time prohibitive, as the graph would scale linearly with the number of machines in the cluster. To reduce the number of arcs in the graph, a cluster aggregator X and rack aggregators R_r have been introduced, inspired by the topology of a typical data center.

Capacity Assignment: Each arc in the flow network has a capacity c for flow, bounded by c_{min} and c_{max}, with c_{min} usually 0, while c_{max} depends on the type of vertices connected by the arc and on the cost model. The capacity of an arc between a task and any other vertex is 1. If a machine has C CPU cores and a rack has m machines, the capacity of an arc between a rack aggregator and a machine is C, and the capacity of an arc between the cluster aggregator and a rack is $C \times m = Cm$. The capacity of an arc between a machine and the sink is C. The capacity between an unscheduled aggregator U_i and the sink S can be used to ensure a fair allocation of runnable tasks between jobs [14]. For the NoMora policy, we set this capacity to 1.

Cost Assignment: The cost on an arc represents how much it costs to schedule any task that can send flow on this arc on any machine that is reachable via this arc. Table 1 provides on overview of the capacities and costs of different arcs.

Task to Machine Arc: The cost on the arc between a task vertex $T_{i,j}$ and a machine vertex M_m is denoted by $d_{i,j,m}$, and is computed according to information regarding the task and machine. In most cases, this cost is being decreased by how much the task has already run, $\beta_{i,j}$.

Task to Resource Aggregator Arc: The cost on the arc between a task vertex $T_{i,j}$ and a rack aggregator vertex R_r, denoted $c_{i,j,r}$, represents the cost to schedule the task on any machine within the rack, and is set to the worst case cost among all costs across that rack. The cost on the arc between a task vertex $T_{i,j}$ and the cluster aggregator X, denoted by $b_{i,j}$, represents the cost to schedule the task on any machine within the cluster, and is set to the worst case cost among all costs across the cluster:

$$b_{i,j} = \max_r c_{i,j,r}.$$

Task to Unscheduled Aggregator Arc: The cost on the arc between a task vertex $T_{i,j}$ and the unscheduled aggregator U_i, denoted by $a_{i,j}$, is usually larger than any other costs in the flow network. The cost on this arc increases as a function of the task's wait time $\alpha_{i,j}$, in order to force the task to be scheduled, and it is scaled by a constant *wait time factor* ω, which increases the cost of tasks being unscheduled. Thus, the cost to the

unscheduled aggregator U_i is $a_{i,j} = \omega \times \alpha_{i,j} + \gamma$, to which a constant cost factor γ, which is larger than any other possible arc costs, is added.

Preemption: If preemption is enabled, the scheduler can preempt a task that it is running on a machine, which means the flow pertaining to that task is routed via the unscheduled aggregator, or migrate the task to a different machine, meaning that the flow is routed via that new machine's vertex. If preemption is not enabled, a scheduled task will have in the flow network only the arc to the machine on which it is currently running, with all the other arcs being removed once the task is scheduled. Preempting a task presents a trade-off between migrating the task to a better placement and the amount of time the task has already run (on the current machine or on a different one). If preemption is enabled, the amount of time the task has already run, $\beta_{i,j}$, can be subtracted from the $cost(T_{i,j}, M_m)$; then $d_{i,j,r} = cost(T_{i,j}, M_m) - \beta_{i,j}$. This leads to fewer task migrations happening, because it becomes less advantageous to preempt a task and restart it on another machine after migration the more time the task is running, essentially wasting the work that has already been done.

Latency-Driven, Application-Performance-Aware Policy

We propose a new *latency-driven, application-performance-aware policy* whose goal is to place distributed applications in a data center in a manner that gives them improved application performance. This generally leads to grouping tasks as close as possible, in a rack or on the same machine, for the applications for which latency matters, such as Memcached and machine learning frameworks (STRADS, Tensorflow). For tasks that do not fit within the same rack or on the same machine, the policy finds the machine that offers the best application performance among the available placements. On the other hand, applications such as Spark, for which additional latency of up to 1 ms does not matter [3], will have more freedom when being placed within the data center. If the network conditions change, a task whose performance degrades can be migrated to a better placement.

Since the applications we studied earlier are client-server applications or worker-master applications, we consider that the server/master has a special role, because it has to be running before the clients/workers. We call the server (for client-server applications)/master (for master-workers applications) the *root task*. Thus, the policy needs to schedule the root task first. The root task is scheduled immediately in any place available in the cluster. The other tasks of the job (clients/workers) are not scheduled until the root task is scheduled. While this adds delay in scheduling for these tasks, the delay is minimal, since they will be scheduled in the next scheduling round based on the placement of the root task. Next, a task's placement is determined based on the application performance function and current network latencies to the job's root task placement.

Flow Network: The flow network is similar to the one in Fig. 2. When a job is submitted, the root task $T_{i,0}$ is assigned a single arc to the cluster aggregator, with a cost of 0, which means that the root task will be scheduled immediately on any available machine. After it is scheduled, the root task will have an arc from the root task to the machine on which it is running. The other tasks of the job will wait for the root to be scheduled first, and they do not have any arcs initially. After the root task is scheduled, each task $T_{i,j}$ will have preference arcs to the cluster aggregator X, to rack aggregators R_r and machines M_m based on the cost to schedule the task on those resources and on the parameters of the policy.

Arc	Capacity (max)	Value
$T_{ij} \rightarrow U_i$	1	a_{ij}
$T_{ij} \rightarrow X$	1	b_{ij}
$T_{ij} \rightarrow R_r$	1	$c_{ij,r}$
$T_{ij} \rightarrow M_m$	1	$d_{ij,m}$
$X \rightarrow R_r$	mC	0
$R_r \rightarrow M_m$	C	0
$M_m \rightarrow S$	C	0
$U_i \rightarrow S$	1	0

TABLE 1. Arcs in the flow network.

Cost Assignment: The cost assignment is also called *cost model* [13]. In NoMora we consider the cost to the root task computed based on application performance function dependent on network latency for a job (as built earlier), combined with measured network latency between the root task machine and the machine under consideration. We also factor the task wait time (the time a task waits before being scheduled) when computing the cost of the arc to the unscheduled aggregator, and, if preemption is enabled, how much time a task has run before being preempted (*preemption cost*). Finding a good placement can mean waiting more time until a suitable machine is free (the task wait time increases), or preempting a task that is already running (if this task is restarted from the beginning on another machine, the time the task has already run is lost).

Assuming the root task is running on machine M_{root} and a task j, $T_{i,j}$, of job J_i can be scheduled on machine M_m, the cost of the arc from $T_{i,j}$ to M_m is

$$d_{i,j,m} = cost(T_{i,j}, M_m) = \frac{1}{p\left(max\left(latency(M_{root}, M_m)\right)\right)} \quad (1)$$

where $p(max(latency(M_{root}, M_m)))$ is the expected application performance for the measured network latency between machine M_{root} and machine M_m, as determined earlier. We invert the performance because when the performance is smaller, the cost assigned to the arc is higher, making the machine to which the arc points less desirable for running the task on. Since in data centers typically there are multiple paths between two machines, in order to be conservative, we use the maximum latency value measured between the two machines because due to equal-cost multi-path routing (ECMP), we cannot know which of the available paths the application's flows will take.

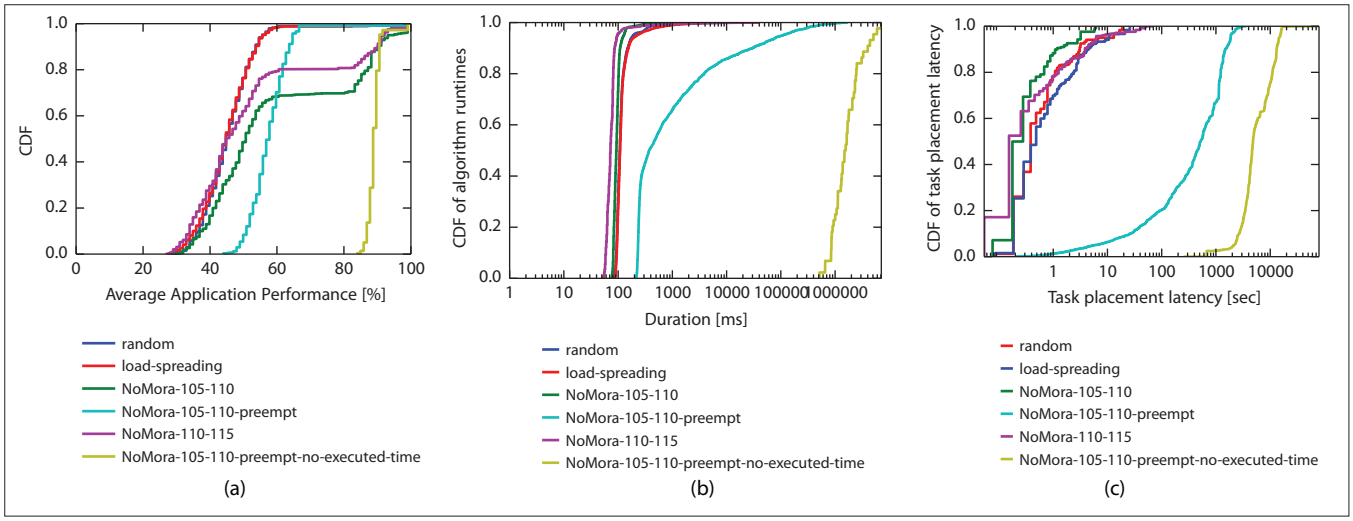

FIGURE 3. Evaluation metrics results for different policies on the Google workload: a) average application performance; b) algorithm runtime; c) task placement latency.

Similarly, the cost of the arc from $T_{i,j}$ to rack R_r is the cost to the worst cost machine in rack r:

$$c_{i,j,r} = cost(T_{i,j}, R_r) = \max_{m \in r} \frac{1}{p\left(max\left(latency(M_{root}, M_m)\right)\right)} \quad (2)$$

where $p(max(latency(M_{root}, M_m)))$ is the expected application performance for the measured network latency between machine M_{root} and a machine M_m in rack r. Similarly, to be conservative due to ECMP, we take the maximum value of the latencies between M_{root} and M_m.

The costs on the arcs are rounded to two significant digits and then multiplied by a factor of 100, since the costs must be integer numbers for the solver to understand. For a performance of 0.1, the cost is (1/0.1) × 100 = 1000.

Since the network latency is not constant in a data center, the costs associated with the arcs are updated based on the latest measured network latency values, and, as a result, the preference arcs for the tasks are updated. If preemption is enabled, the cost of the arcs for a running task will also be updated.

Cost Model Parameters: The cost model has two main parameters: p_m, threshold for the cost on an arc to a machine in order for that machine to be on the preferred list of machines on which the task can run, and p_r, threshold for the cost on an arc to a rack in order for that rack to be on the preferred list of racks in which the task can run. The first preference list comprises the machines on which the application may run to achieve the desired performance. This list should be kept small for the scheduling latency to take a reasonable amount of time. But having a small preference list means the application's placement options are limited. To mitigate this, the second preference list, which comprises the racks on which the application may run, was introduced. The second list is smaller than the first one, since the number of racks is smaller than the number of machines. This allows a bigger threshold to be set for the second parameter of the model, offering more placement options for the application's tasks, while keeping the first preference list small.

NoMora Evaluation

Simulation Experiments

Simulation Setup:

Cluster Workloads: As no public cluster workloads include information regarding sensitivity to network latency for applications, we assigned randomly the functions from Fig. 1 to the jobs in the Google workload [15]. For the experiments presented in this section, 50 percent of the jobs use the Memcached function, 25 percent use the STRADS function and 25 percent the Tensorflow function. We did not include the single task jobs, as they do not communicate with any other task. We used 24 hours of the trace. We set $\gamma = 1001$ for the simulation, since the performance does not drop below 0.1 for the functions used.

Topology: The cluster workload has 12,500 machines, which we grouped into racks of 48 machines and pods of 16 racks. These two numbers were chosen to reproduce a small cluster.

Network Latency Measurements: The simulator leverages the network latency measurements dataset from [2]. We assign them to machine pairs considering the physical distance between servers as a criterion, assuming a typical fat-tree topology for a data center. Latency values are provided every second in the simulation. For the latency values between cores on the same server we use a small constant.

Cost Model Parameters: We determine empirically through experimentation the values for the cost model parameters, p_m and p_r. We use p_m = 105 and p_r = 110 in most experiments, and also present results for p_m = 110 and p_r = 115. Choosing these parameters involves a trade-off between the improvement in application performance, task wait time, and algorithm runtime. The algorithm runtime depends on the number of arcs from each task to the resources, but also on the cluster size. As the number of arcs or the cluster size increases, so does the algorithm runtime. The two parameters of the cost model influence the number of arcs the graph has between task nodes and machine nodes or rack nodes, and hence the algorithm runtime, which depends on the flow network size and on the num-

ber of tasks considered per scheduling round. If the parameters' values are lower, the preference lists will be smaller. In this case, the applications' performance will be higher (only high-quality placements are considered), but they will have less placement options available for them to be scheduled, and thus the wait time may increase. The tasks will have to wait for the machines that offer the performance desired to have empty slots. However, setting a high parameter value means the preference lists will be larger, which could lead to an increase in the algorithm runtime. On the other hand, more placement options will be available for the tasks to be scheduled, reducing their wait time. In practice, with more placement options available, the tasks may be scheduled sooner, thus leading to fewer tasks being scheduled per round, resulting in a decrease in the algorithm runtime per scheduling round.

Evaluation Metrics: We compute the *average application performance*, which measures NoMora's task placement quality. It is computed as the application performance determined by the network latency in every measurement interval divided by the maximum application performance that could be achieved in every measurement interval, and it is computed for the job's total runtime.

The *algorithm runtime* is the time it takes for the min-cost max-flow algorithm to run to compute task placements. This also gives an indication of the time interval that is needed between latency measurements. If the algorithm runtime were on the order of minutes, running latency measurements every few seconds would be too often, since the measurements accumulated over the scheduler's runtime would not be used by the scheduler. If the algorithm runtime is on the order of milliseconds, it is better to run latency measurements every second or few seconds.

The *task placement latency* is the time between task submission and task placement, which includes the task wait time. The metric also captures how long the tasks are delayed when they are waiting for their root task to be placed first before them.

Simulation results

Placement Quality: We compare the NoMora policy, using different parameter values for the cost model, with a random policy that uses fixed costs (tasks always schedule if resources are idle), and a load-spreading policy that balances the tasks across machines. We enable preemption only for the NoMora policy, since the two other policies would not benefit from preemption due to their different scheduling goals.

The results for the average application performance for different policies can be seen in Fig. 3a. We compute the area marked by the y-axis, the cumulative distribution function (CDF), and the straight horizontal line with $y = 1$ for each policy. According to this computation, the maximum area corresponding to the maximum average application performance across applications is 100 percent, and it would be obtained for a vertical line at $x = 100$ percent. Next, we subtract from the NoMora policies areas the random and load-spreading areas to assess the placement improvement given by the NoMora policy.

The maximum overall improvement without preemption enabled is 13 percent over the random policy and 13.4 percent over the load-spreading policy, and is obtained for NoMora with parameters $p_m = 105$ and $p_r = 110$. If preemption is enabled and $\beta_{i,j} = 0$ (the time already executed by a task is not considered in the arc cost computation), the improvement is considerable, 42.4 percent over the random policy and 42.8 percent over the load-spreading one.

NoMora paves the way for application-performance-aware cluster schedulers. Instead of meeting the job's resource requirements, in our work we consider meeting the job's performance requirements.

The improvement in average application performance is not substantial when preemption is not enabled because of the root task's random placement. The tasks of the jobs are placed in the best available slots in relation to the root task's placement. In this way, we constrain the available placements, and the policy searches for placements in relation to a known location rather than trying to find a placement for all the tasks of a job simultaneously.

It can be seen that the CDF of NoMora with *preemption enabled* has a different shape than the other policies. This is due to task preemption, which can correct the initial placement if it is not good (because of the random placement of the root task), and it can also migrate tasks when their current placement is not good anymore. The improvement provided by the NoMora policies without preemption is evident from Fig. 3a, but it can also be seen that the CDFs start at approximately the same value (27–28 percent average application performance), and have an initially similar shape to the random and load-spreading CDFs. On the other hand, for NoMora with preemption enabled, the minimum average application performance is 44 and 84 percent, respectively, which means that the improvement in application performance happens across all jobs due to migration to better placements.

Algorithm Runtime: Figure 3b and Table 2 present results for the algorithm runtime for the load-spreading policy, random policy, and NoMora policy with and without preemption. For NoMora with parameters $p_m = 105$ and $p_r = 110$, we observe an improvement of 1.61× for the median runtime, and 2.66× and 3.92× at the 99th percentile, compared to the random policy and load-spreading policy, respectively.

NoMora with parameters $p_m = 105$ and $p_r = 110$ with preemption enabled takes a considerably longer amount of time, because of the higher number of arcs in the flow network compared to the case when preemption is not enabled (the arc preferences of the tasks that are running are not removed, unlike when preemption is not enabled), and the updates made to the flow graph (adding or changing running arcs to resources), further resulting in a larger number of tasks considered per scheduling round. This also translates into a larger task placement latency. This significant algorithm runtime means that preemption should be used with care. For example, only certain applications that explicitly demand to be migrated should be migrated, or migration can be triggered only if the application performance drops below a certain threshold.

Policy	Overall avg app. performance	Median	99th perc.	Maximum
Random	47.2%	108 ms	661 ms	18.89 s
Load-spreading	46.8%	109 ms	974 ms	25.88 s
NoMora $p_m = 105, p_r = 110$	60.2%	93 ms	248 ms	6.13 s
NoMora $p_m = 105, p_r = 110$ preempt	59%	373 ms	511 s	1719 s
NoMora $p_m = 110, p_r = 115$	51.85%	72 ms	486 ms	39.55 s
NoMora $p_m = 105, p_r = 110, \beta_{ij} = 0$ preempt	89.6%	1532s	6610s	7118 s

TABLE 2. Overall average application performance and algorithm runtimes for baseline and NoMora policies.

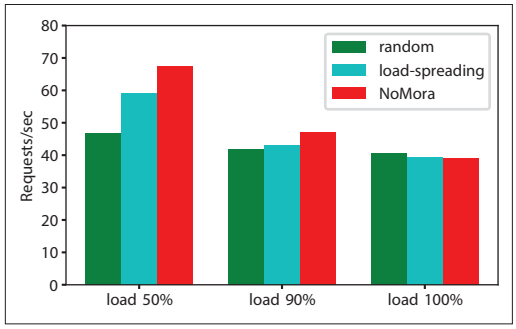

FIGURE 4. Overall average application performance (requests per second) for the Memcached workload using three policies: random, load-spreading, and NoMora.

The percentage of migrated tasks in the case when NoMora with preemption enabled and already executed time for a task considered in the arc computation is on average 0.022 percent per scheduling round, with a 99th percentile of 0.5 percent. If $\beta_{i,j} = 0$ (already executed time is not considered in the arc computation), a considerable number of task migrations take place: an average of 7.1 percent per scheduling round, with a 99th percentile of 10.07 percent, leading to a substantial algorithm runtime and task placement latency. This happens because the time a task has already run is ignored in the arc cost computation, meaning that the cost is based solely on the expected application performance under the given network conditions.

Task Placement Latency: Figure 3c presents the task placement latency. The NoMora policy with parameters p_m = 105 and p_r = 110 improves the median task placement latency by 1.56× compared to the random policy and by 1.79× compared to the load-spreading policy. The NoMora policy with parameters p_m = 110 and p_r = 115 improves the median task placement latency by 2.35× compared to the random policy and by 2.69× compared to the load-spreading policy. Policies with migration enabled lead to increased task placement latency.

Discussion: There are a number of factors that can influence the improvement in application performance that our cluster scheduling policy can achieve. If the applications are not latency-sensitive, but instead are throughput-intensive, our policy's benefits are limited. The simulation results will also be influenced by the number of hosts per rack and by the number of racks per pod. This is due to the fact that we assumed lower latencies between hosts within a rack compared to the latencies between pods under normal conditions, and we assigned the cloud latency traces accordingly. If there are more hosts per rack, and more racks per pod than the scenario we considered in our simulation, there will be a greater chance of fitting all the tasks of a job in the same rack or in the same pod. This means that the job will have better overall application performance due to the lower latencies between hosts in the same rack. Thus, the size of a job in terms of number of tasks should be taken into consideration when designing novel data center topologies. Another factor that can influence the results is the network utilization in the data centers. The performance of latency-sensitive applications will suffer if the network is congested.

Testbed Experiments

We evaluate NoMora on a testbed comprising 18 hosts. We use a tree topology, with one core switch and two leaf switches. Each leaf switch is connected to nine hosts. Each host has an Intel Xeon E5-2430L v2 Ivy Bridge CPU with six cores, running at 2.4 GHz with 64 GB RAM, and is equipped with an Intel X520 NIC with two SFP+ ports, being connected at 10 Gb/s through an Arista 7050Q switch. The two leaf switches are connected through the core switch, an Arista 7050Q using 40 Gb/s connections. The hosts run Ubuntu Server 18.04, kernel version 4.15.0-51-generic. A job is represented by a Memcached workload generated by five memaslap (from the libmemcached library) clients, and one Memcached server responding to the client requests. We use different loads for the cluster: 50 percent (10 jobs), 90 percent (18 jobs) and 100 percent (20 jobs). We use as background traffic iperf using a TCP stream between two servers in one of the leaves in order to have increased network latency between servers in that respective leaf. We run PTPmesh [10] to measure the network latency periodically between every two servers in our cluster. The latency measurements serve as input to NoMora when deciding where to place the jobs using the application performance function determined earlier for the Memcached workload. The results in Fig. 4 show that NoMora improves the overall application performance (measured in requests per second) compared to a random placement policy by 32.5 percent, and compared to a load-spreading policy by 20.46 percent when the load in the cluster is 50 percent. As the load in the cluster increases, the network becomes more utilized, and thus the improvement in the overall application performance decreases. When the load in the cluster is 90 percent, the NoMora policy performs better by 10.81 percent compared to a random policy, and by 8.47 percent compared to a load-spreading policy. When the cluster is fully utilized, all policies perform the same, because the network is congested under high utilization. As data center operators are not running their networks at high utilization, we show that our NoMora policy improves overall application performance.

Conclusion and Future Work

We introduce latency-driven, application-performance-aware cluster scheduling, and NoMora, a cluster scheduling framework that implements this

type of policy. It exploits functions that predict application performance based on network latency and dynamic network latency measurements between hosts to place tasks in a data center, providing them with improved application performance. We show that overall application performance is improved by up to 32.5 percent in testbed experiments and up to 42 percent in simulation experiments.

Our work paves the way for application-performance-aware cluster schedulers, which use network measurements to improve overall application performance.

References

[1] D. A. Popescu, "Latency-Driven Performance in Data Centres," Univ. of Cambridge, Computer Lab, Tech. Rep. UCAM-CL-TR-937, June 2019; https://www.cl.cam.ac.uk/techreports/ UCAM-CL-TR-937.pdf.

[2] D. A. Popescu and A. W. Moore, "A First Look at Data Center Network Conditions Through the Eyes of PTPmesh," *Proc. 2018 IFIP/IEEE 2nd Network Traffic Measurement and Analysis Conf.*, ser. TMA '18, 2018.

[3] D. A. Popescu, N. Zilberman, and A. W. Moore, "Characterizing the Impact of Network Latency on Cloud-Based Applications' Performance," Univ. of Cambridge, Computer Lab, Tech. Rep. UCAMCL- TR-914, Nov. 2017; http://www.cl.cam.ac.uk/ techreports/UCAM-CL-TR-914.pdf.

[4] J. Perry *et al.*, "Fastpass: A Centralized 'Zero-Queue' Datacenter Network," *Proc. 2014 ACM Conf. SIGCOMM*, ser. SIGCOMM '14, 2014, pp. 307–18; http://doi.acm.org/10.1145/2619239.2626309.

[5] M. P. Grosvenor *et al.*, "Queues Don't Matter When You Can Jump Them!" *Proc. 12th USENIX Conf. Networked Systems Design and Implementation*, ser. NSDI'15, 2015, pp. 1–14; http://dl.acm.org/citation.cfm?id=2789770.2789771.

[6] L. Popa *et al.*, "Elasticswitch: Practical Work-Conserving Bandwidth Guarantees for Cloud Computing," *Proc. ACM SIGCOMM 2013*, ser. SIGCOMM '13, 2013, pp. 351–62; http://doi.acm.org/10.1145/2486001.2486027

[7] H. Ballani *et al.*, "Towards Predictable Datacenter Networks," *Proc. ACM SIGCOMM 2011*, ser. SIGCOMM '11, 2011, pp. 242–53; http: //doi.acm.org/10.1145/2018436.2018465.

[8] K. LaCurts *et al.*, "Choreo: Network-Aware Task Placement for Cloud Applications," *Proc. 2013 ACM Conf. Internet Measurement*, ser. IMC '13, 2013, pp. 191–204; http://doi.acm.org/10.1145/2504730.2504744.

[9] K. Jang *et al.*, "Silo: Predictable Message Latency in the Cloud," *Proc. 2015 ACM Conf. Special Interest Group on Data Communication*, ser. SIGCOMM '15, 2015, pp. 435–48; http://doi.acm.org/10.1145/2785956.2787479.

[10] D. A. Popescu and A. W. Moore, "PTPmesh: Data Center Network Latency Measurements Using PTP," *Proc. 2017 IEEE 25th Int'l. Symp. Modeling, Analysis, and Simulation of Computer and Telecommun. Systems*, Sept. 2017, pp. 73–79.

[11] C. Guo *et al.*, "Pingmesh: A large-Scale System for Data Center Network Latency Measurement and Analysis," *Proc. 2015 ACM Conf. Special Interest Group on Data Commun.*, ser. SIGCOMM '15, 2015, pp. 139–52; http://doi.acm.org/10.1145/2785956.2787496.

[12] A. Adams, P. Lapukhov, and J. H. Zeng, "NetNORAD: Troubleshooting Networks via End-to-End Probing"; https://code.facebook.com/posts/1534350660228025/netnorad-troubleshooting-networks-via-end-toend-probing/, 2016, accessed July 2021.

[13] I. Gog *et al.*, "Firmament: Fast, Centralized Cluster Scheduling at Scale," *Proc. 12th USENIX Conf. Operating Systems Design and Implementation*, ser. OSDI'16, 2016, pp. 99–115; http://dl.acm.org/citation.cfm?id=3026877.3026886.

[14] M. Isard *et al.*, "Quincy: Fair Scheduling for Distributed Computing Clusters," *Proc. ACM SIGOPS 22nd Symp. Operating Systems Principles*, ser. SOSP '09, 2009, pp. 261–76; http: //doi.acm.org/10.1145/1629575.1629601.

[15] C. Reiss *et al.*, "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis," *Proc. Third ACM Symp. Cloud Computing*, ser. SoCC '12, 2012, pp. 7:1–13; http://doi.acm.org/10.1145/2391229.2391236.

Biographies

Diana Andreea Popescu received her Ph.D. degree in computer science from the Computer Laboratory, Systems Research Group, University of Cambridge, United Kingdom, focusing on data center networking, network measurement, and cluster scheduling. During her Ph.D. work, she was a Marie Curie Early-Stage Researcher. As a research associate in the Systems Research Group, she worked on network management for IoT devices, applied machine learning to IoT network traffic, and federated learning. She is currently a visiting researcher in the Department of Computer Science and Technology, University of Cambridge.

Andrew W. Moore is a professor of networked systems with the Department of Computer Science and Technology, University of Cambridge, where he jointly leads the Systems Research Group working on issues of network and computer architecture with a particular interest in latency-reduction. His research contributions include enabling open network research and education using the NetFPGA platform; other research pursuits include low-power energy-aware networking, and novel network and systems data center architectures.