Cross-Layer Scheduling in Cloud Computing Systems [*] (Research Paper)

Hilfi Alkaff
Department of Computer Science
University of Illinois at Urbana-Champaign
alkaff2@illinois.edu

Indranil Gupta
Department of Computer Science
University of Illinois at Urbana-Champaign
indy@illinois.edu

Abstract
Today, cloud computing engines such as stream-processing Storm and batch-processing Hadoop are routinely run atop SDN networks (software-defined networks). Yet, in these cloud stacks, the schedulers of these application engines are decoupled from the network. We propose a new approach that performs cross-layer scheduling between the application layer and the networking layer. This coordinated scheduling orchestrates the placement of application tasks (e.g., Hadoop maps and reduces, or Storm bolts) in coordination with the selection of network paths that arise from these tasks. We present results from both cluster deployment and simulation, and using two different network topologies: Fat-tree and Jellyfish. Our results show that cross-layer scheduling can improve throughput of Storm and Hadoop by between 22% to 34% in a 30-host cluster.

Categories and Subject Descriptors
H.3.4 [**Systems and Software**]: Distributed systems; C.2.4 [**Distributed Systems**]: Network operating systems

General Terms
Algorithms, Design, Experimentation, Measurement, Performance

Keywords
Cloud computing, Hadoop, Storm, Scheduling, Cross-layer

1. INTRODUCTION
The recent years have seen an explosion in sources of big data. Today, data analytics engines need to process this data in real-time – examples include processing data from advertisement pipelines at Yahoo!, Twitter's real-time search, Facebook's trends, real-time traffic information, etc. In all of these applications, it is critical to optimize throughput as this metric is correlated with revenues (e.g., in ad pipelines), with user satisfaction and retention (e.g., for real-time searches), and with safety (e.g., for traffic).

Two categories of systems are widely used in industry today to meet this need for real-time data analytics. The first category consists of cloud batch-processing systems such as Hadoop [12, 3], Hive [48], Pig [36], DryadLinq [53], etc. The second generation includes new stream-processing engines that have begun to emerge and are already being used widely in industry – these include Storm [2].

Today, these data analytics engines are run atop a variety of networks. The most common network topology in deployment is hierarchical, and it contains multiple racks (each with a top of the rack switch) connected by a core switch. The most popular variant of this is the Fat-tree topology [29], which assigns more links (thus more bandwidth) to switches closer to the root of the tree. At the same time, novel topologies have begun to emerge such as Jellyfish [45], Scafida [22], VL2 [17], DCell [20], etc.

Yet, neither stream-processing engines nor batch-processing frameworks are attuned to the structure of the network underneath. These systems perform task placement in a manner that is aware of the data placement at servers, or adapt network flows in an end-to-end manner, e.g., [10]. However, true coordination between the scheduler at the application level and the network remains largely unexplored. In other words, the cloud engine's scheduler takes into account data-locality, CPU and memory consumption, but not the structure of the network underneath.

In this paper, we explore a cross-layer approach to scheduling in such cloud stacks. Our proposed framework operates at two levels that complement each other. The upper application-level scheduler decides which physical server each worker task should be allocated to. The lower network-level scheduler lies at the routing level in which the physical routers and links are interconnected. It decides how paths are selected for each end-to-end pair of communicating hosts.

To achieve generality across cloud stacks, we need to be general both: i) across cloud engines, as well as ii) across multiple types of networks. To address the first challenge, we observe as far as real-time processing is concerned, both batch and streaming cloud engines can be represented as

[*]This work was supported in part by AFOSR/AFRL grant FA8750-11-2-0084, and in part by NSF grant CCF 0964471 and NSF grant 1319527.

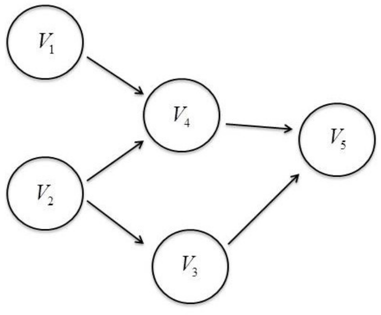

Figure 1: Stream-processing engines such as Storm can be represented using a dataflow graph. Each vertex is a bolt processing data, and edges show flow of data.

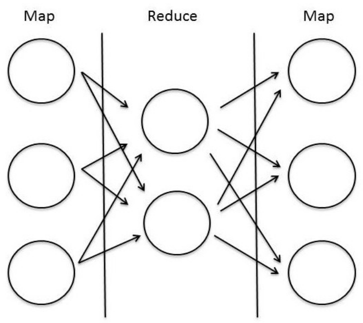

Figure 2: Batch-processing engines such as Hadoop, Hive, and Pig can be represented as a dataflow graph.

dataflow graphs. Figure 1 shows a sample Storm application graph, where each vertex is a computation node, called a "bolt" in Storm. Edges represent flow of data, which is in the form of tuples. Each bolt processes the tuples, e.g., it may filter it, or perform a map, join, etc. The source bolts are called "spouts." The overall Storm graph for an application, called a "Storm topology", is allowed have to cycles. At the same time, Figure 2 shows that Hadoop, either single stage or in multiple stages as in Hive or Pig, can also be represented as a dataflow graph from map tasks to reduce tasks (and if necessary, to another level of map tasks, then reduce tasks, and so forth).

We call the vertices of such dataflow graphs as "Computation-Nodes ". A Computation-Node is a bolt in Storm, while in Hadoop it is a map task or reduce task. We observe that dataflow graphs arising from batch-processing frameworks like Hadoop are more structured than, and are thus a subset of, those from stream-processing engines like Storm, i.e., the former are constrained to be acyclic and staged.

To address the second challenge (generality across networks), we leverage the abstraction of Software-Defined Networking (SDN), manifested in popular systems like OpenFlow [32]. The SDN approach to managing networks splits the control plane from the data plane, and allows the former to live in a centralized server called the SDN Controller. The controller installs forwarding rules inside the routers, monitors flow status, and can change these rapidly. As measured in [7], SDN controllers can install flows within a few milliseconds,

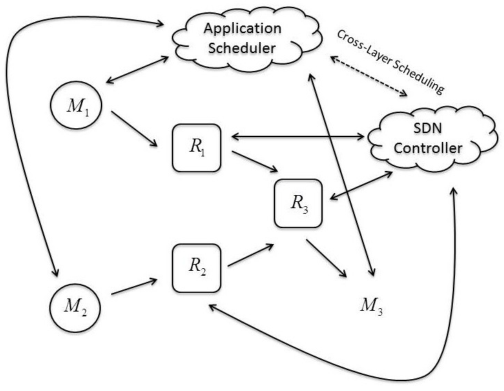

Figure 3: Application master is responsible for scheduling and monitoring worker servers, M_i, while SDN controller for the routers, R_j.

which is comparable to a traditional non-SDN network.

Essentially, the SDN controller allows us to generally handle any topology in the underlying network. Our only requirement is that there be multiple paths between any given pair of end hosts. To be representative, we evaluate our cross-layer scheduling on one popular network topology and one recently-proposed network topology. In particular, we evaluate our approach with a hierarchical topology (i.e., Fat-tree), and a random topology (i.e., Jellyfish). This choice is motivated by the fact that these two topologies are at opposite ends of the spectrum of route diversity. On the one hand, the Fat-tree topology is more structured, allowing diversity for only a limited number of routes, i.e., those that traverse closer to the root of the topology. On the other hand, Jellyfish uses a random graph for the network, thus offering an extremely high diversity of route choices for all host pairs.

Figure 3 summarizes our overall approach. The figure shows worker servers M_i and routers R_j. At the application level, the scheduler decides where to place each Computation-Node of the application's dataflow graph (tasks in a Hadoop cluster, bolts in a Storm cluster). In our approach, the application scheduler consults the SDN scheduler to find out the bandwidth available end-to-end across servers. The SDN controller in turn has the flexibility to adjust the routes taken by end-to-end paths.

What we desire is a placement of tasks as well as selection of routes, that achieves optimality (in the formal sense) of throughput achieved by the applications. However, the state spaces that the application scheduler and the SDN scheduler need to explore are both enormous. There are combinatorially many ways in which tasks can be scheduled at servers, leading to a large state space for the application scheduler. There are also combinatorially many routes to select for each end-to-end flow in the underlying SDN, leading to a large state space at the network level. Exploring these state spaces exhaustively is prohibitively expensive.

In order to explore these state spaces efficiently and achieve good performance, we leverage the philosophy of Simulated Annealing. Inspired by metallurgy, this approach [49] probabilistically explores the state space and converges to an optimum that is close to the global optimum. It avoids getting stuck in local optima by utilizing a small probability of jumping away from it – for convergence, this probability decreases over time. We realize the simulated annealing approach in both the application-level (for task placement) and in the SDN level (for route selection).

The technical contributions of this paper are as follows.

- We design a novel cross-layer scheduling framework enabling cloud applications to make better scheduling decisions in a network topology-aware manner.
- We show how to use a simulated annealing algorithm to make fast and good cross-layer scheduling decisions.
- We implement our approach for Hadoop and Storm at the application level, and in Fat-tree and Jellyfish topologies at the network level.
- We perform simulation and real cluster experiments. Our cluster experiments show that our approach improves throughput for Hadoop by 22-31% and for Storm by 30-34% (ranges depend on network topology).

This paper is structured as follows. Section 2 presents our simulated annealing-based approach to scheduling. Section 3 discusses implementation details for both Storm and Hadoop. Section 4 presents our experiments. We present related work in Section 5, and conclude in Section 6.

2. DESIGN

In this chapter, we present our simulated annealing (SA)-based approach to cross-layer scheduling.

Our cross-layer scheduling framework is run whenever a new job arrives. Our approach computes a good placement and routing paths for the new job. For already-running jobs, their task placement and routing paths are left unchanged. We adopt this approach for three reasons: i) to reduce the effect of the new job on already-running jobs; ii) migrating already-running tasks may be prohibitive, especially since real-time analytics tasks may be short; and iii) the state space is much larger when considering all running jobs, instead of just the newly arrived job. At the same time, the utility function used by our SA approach considers the throughput for all jobs in the cluster.

Considering the new job, both the application-level and the routing-level run the same simulated annealing algorithm shown as pseudocode in Algorithm 1. Since the state abstractions are different for these two levels, two of the functions in Algorithm 1 are different for the two layers – these are shown in Algorithm 2 and 3. We elaborate on these algorithms in detail below.

Let us name the network topology graph as $\mathcal{G}$. Its vertex set $\mathcal{V}$ represents the physical servers (i.e., servers) as well as routers. Its edge set $\mathcal{E}$ consists of the links between the servers. Each job $\mathcal{J}_i$ that is running on this topology contains $\mathcal{T}_i$ tasks.

Algorithm 1 Simulated Annealing Algorithm (for both Application Level and Routing Level)

```
function MAIN($graph$)
    $bestUtil \leftarrow 0$
    $bestState \leftarrow Null$

    for $i \leftarrow 1$ to 5 do
        $t \leftarrow initialTemperature$
        $currState \leftarrow initState(graph)$
        $currUtil \leftarrow 0$

        for $j \leftarrow 1$ to $maxStep$ do
            $newState \leftarrow genState(graph, currState)$
            $newUtil \leftarrow computeUtil(graph, newState)$

            if $transition(currUtil, newUtil, t)$ then
                $currState \leftarrow newState$
                $currUtil \leftarrow newUtil$
            end if

            if $currUtil \geq bestUtil$ then
                $bestUtil \leftarrow currUtil$
                $bestState \leftarrow currState$
            end if

            $t \leftarrow t^{0.95}$
        end for
    end for

    return $bestState$
end function

function TRANSITION($oldUtil, newUtil, temperature$)
    if $newUtil > oldUtil$ then
        return 1
    else
        return $e^{\frac{newUtil-oldUtil}{temperature}}$
    end if
end function

function COMPUTEUTIL($graph, state$)
    $util \leftarrow 0$
    $jobs \leftarrow$ All existing jobs + new job

    for Each sink COMPUTATION-NODE $s$ in $jobs$ do
        $sinkUtil \leftarrow$ inf

        for Each root COMPUTATION-NODE of $s$ do
            $path \leftarrow$ TOPOLOGY-MAP.$get(sink, root)$
            $sinkUtil \leftarrow min(sinkUtil, min\ bandwidth\ in\ path)$
        end for
    end for

    return $util$
end function
```

Algorithm 2 Simulated Annealing Functions at the Application Level (Task Placement)

```
 1: function initState($graph, servers$)
 2:     $currAllocation \leftarrow \{\}$
 3:
 4:     for Each task $t$ in graph.Computation-Node () do
 5:         $server \leftarrow$ random($servers$)
 6:         $currAllocation[server].add(t)$
 7:     end for
 8: end function
 9:
10: function genState($graph, servers, currAllocation$)
11:     $task \leftarrow$ de-alloc heuristic($graph$.Computation-Node())
12:     $server \leftarrow currAllocation.find(task)$
13:     Deallocate $task$ from the server
14:     $newMachine \leftarrow$ random($servers$)
15:     $currAllocation[newMachine] \leftarrow task$
16:
17:     Call the routing-level SA to calculate a good set of
18:     routes for all end to end host pairs
19:     return $currAllocation$
20: end function
```

Algorithm 3 Simulated Annealing Functions at the Routing Level (SDN Routes)

```
 1: function initState($graph, communicatingMachines$)
 2:     $currRoutes \leftarrow \{\}$
 3:
 4:     for Each pair in $communicatingMachines$ do
 5:         $routes \leftarrow$ Topology-Map.$get(pair)$
 6:         $currRoutes[pair] = random(routes)$
 7:     end for
 8: end function
 9:
10: function genState($graph, servers, currRoutes$)
11:     $communicatingPair \leftarrow$ de-path heuristic($currRoutes$)
12:     $routes \leftarrow$ Topology-Map.$get(communicatingPair)$
13:     $newRoute \leftarrow$ path heuristic($routes$)
14:     $currRoutes[communicatingPair] \leftarrow newRoute$
15:
16:     return $currRoutes$
17: end function
```

Pre-computation of Topology-Map: When our data processing framework first starts, we run a modified version of the Floyd-Warshall Algorithm [14] in order to compute, for each pair of hosts, the k shortest paths between that pair. The results of this computation are then cached in a hash table, which we call "Topology-Map ". This Topology-Map is consulted for future scheduling decisions. In this hash table, each item stands for a pair of servers, with a key of $(\mathcal{M}_i, \mathcal{M}_j)$ where $\mathcal{M}_i, \mathcal{M}_j \in \mathcal{V}$, $i < j$. The values associated with the key are the k shortest paths which we name $\mathcal{K}_1, \mathcal{K}_2, \cdots \mathcal{K}_k$, stored along with the available bandwidth on their constituent links. Thus, $\mathcal{K}_i$ is a sub-graph of $\mathcal{G}$.

The size of the hash table is small. For instance, in a 1000 server-cluster, we found that topologies such as Jellyfish and Fat-tree (with an appropriate number of routers) have at most 10-hop acyclic paths between any two nodes. Setting $k = 10$ implies that the number of entries that need to be stored is at most $= 1000^2$ nodes $\times$ 10 hops/node pair $\times$ 10 different paths = 100 Million. With a few bytes for each entry, this is still O(GB), which can be maintained in-memory in a typical COTS (off the shelf) server today.

The above hash table can be compacted further. The set of k shortest paths between a pair of servers $\mathcal{M}_i, \mathcal{M}_j$ are likely to overlap in their intermediate routers. As a result, instead of storing the values as $\mathcal{K}_1, \mathcal{K}_2, \cdots \mathcal{K}_k$, we store the subgraph $\mathcal{G}'_{i,j}$ consisting of the paths $\mathcal{K}_1, \ldots \mathcal{K}_k$ for server pair $(\mathcal{M}_i, \mathcal{M}_j)$. In the above scenario with 1000 servers, this reduces the total hash table memory utilization to 6 MB.

Finally, the pre-computation step to create this hash table is fast – for our running example above, this takes around 3 minutes.

States: The simulated annealing approach requires us to define the notions of *states* and *state neighborhood.* Since our framework works at two levels, these mean different things at each level. In the routing level, each state $\mathcal{S}$ consists of all the k-shortest paths $\mathcal{P}_1, \mathcal{P}_2, \ldots$ in $\mathcal{G}$ whose end-points are a pair of servers. Thus each $\mathcal{P}_i$ is obtained from the hash table Topology-Map. We define the *neighbors of a state* $\mathcal{S}$ as those states that differ from $\mathcal{S}$ in exactly one path $\mathcal{P}_i$.

At the application level, a state $\mathcal{S}$ consists of the current placements of the worker tasks across servers $\mathcal{M}_i$. The neighbors of a state $\mathcal{S}$ are defined as those states that differ from $\mathcal{S}$ in the placement of exactly one worker task.

State Space Exploration: Simulated annealing (SA) [49] is an iterative probabilistic technique for locating a good approximation of the best solution. The main idea of SA is to use a heuristic which considers some neighboring states $\mathcal{S}'$ of the current state $\mathcal{S}$, and probabilistically decides, in each iteration, between either: i) moving the system to a neighbor state $\mathcal{S}'$, or ii) staying in state $\mathcal{S}$.

The main flow of our framework is as follows. Firstly, a user submits a job. For Hadoop, this may be a new Hadoop job. For the Storm scenario, this may be a new dataflow graph with its own Storm topology.

Then our system runs in two phases. In the first phase, our scheduler that runs on the application level consults the Topology-Map to evaluate which are the servers on which placement of the new tasks will yield the best throughput, along with which network paths are the best matches for the end-to-end flows for that placement. As mentioned earlier, we neither change the task placement for already-running jobs, nor their allocated paths. Thereafter, in the second phase, the application requests the SDN controller to set up the requested paths. The first phase is shown in the pseudocode for Algorithm 1.

In the first phase, the same SA Algorithm 1 is run at both the application and routing levels. The former calls Algorithm 2 and the latter calls Algorithm 3. The primary SA works at the application level, and drives the network-level SA. This means that every iteration of the SA at the application level calls the routing-level SA as a black box (penultimate statement in *genState* of Algorithm 2), which in turn runs SA on the network paths for the end hosts in the current state, converges to a good set of paths, and then returns. When the application-level SA converges, we have a new state and the new job can be scheduled accordingly.

Algorithm 1 begins by initializing the state in the function *InitState*. Thereafter, in each iteration (step) it first runs *genState* to generate the neighboring states of the current state and select one prospective next-state from among these. Then, the utility of this prospective state is computed by *computeUtil*.

To calculate a potential next-state, the application-level SA (*genState*) uses a *de-allocation heuristic* to select one Computation-Node, de-allocate it from its current server, and then allocate it to new server chosen at random.

Our de-allocation heuristic works as follows. From Section 1, recall that Computation-Node is defined as a vertex in the dataflow graph takes input data from incoming edges and produces output data on its outgoing edges. In a dataflow graph, Computation-Nodes that are adjacent to either a source Computation-Node (called a "spout" in Storm) or a sink Computation-Node are more likely (than other Computation-Nodes) to directly affect the overall throughput of the new job. For each Computation-Node C_i, we calculate its priority, P_{C_i}, equal to the sum of the number of source and sink Computation-Nodes that are directly adjacent to it in the dataflow graph. We then probabilistically select a Computation-Node with probability $= \frac{P_{C_i}}{\sum_j P_{C_j}}$.

At the routing-level, our system uses a *de-path heuristic* that selects one host pair and removes its route, and then a *path heuristic* to assign this host pair a new route. These appear in *genState* of Algorithm 3.

We first describe our path heuristic. First, we prefer paths that have the lowest number of hops. This has two advantages: i) it minimizes the latency and maximizes bandwidth between the end hosts, and ii) it intersects and interferes with the least number of other network routes. However, there might be multiple such lowest-hop paths. Among them we then select path which has the highest available bandwidth (thereafter, ties are broken randomly).

The de-path heuristic is the inverse of the path heuristic. That is, we prefer to de-allocate that route which has highest number of hops – if there are ties, we select the path with the lowest available bandwidth.

Utility Function and Transitions: The utility function for a given state (in the SA algorithm) is calculated by the *computeUtil* function. The utility function estimates the effective throughput across *all* jobs in the cluster, both already-running and the new job. It does so by iterating across all the sink Computation-Nodes of these jobs. For each sink it finds all its root Computation-Nodes r, and the network bandwidth (in the currently chosen network paths) from r to s. It then calculates the minimum bandwidth among all the roots of s. This is the utility of s. The total utility of a state is then computed as the sum of utilities of all sink Computation-Nodes across all the existing jobs in the cluster as well as the sink Computation-Nodes of the new job.

Then, the SA approach decides in the *transition*() function whether to transition to the new state or stay in the old state. If the new total utility (overall throughput) of the new proposed state *newUtil* > the current total utilization *currUtil*, we always transition to the new state. If the new utility is lower however, we transition with a probability that is related to the difference in utilities and also to how long the SA algorithm has been running. First, if the utility difference is large, then the probability of transitioning is small. However if the utility difference is small, meaning the prospective new state is worse in utility but not too far below the current status, then there is a good probability of transitioning. Second, the SA algorithm has a higher probability of making such jumps early on – later on, the *temperature* of the SA would have grown small and the probability of transitioning would thus be small. This ensures eventual convergence of the SA algorithm. Finally, to make our SA exploration robust, we repeat the entire SA exploration 5 additional times: this ensures that that we are not stuck in a local optima, and have indeed converged.

Fault-tolerance and Dynamism: The SA algorithm can be re-run quickly whenever there is a failure or change in the network. The SDN controller monitors the link status in the background. Upon an update to any link's bandwidth, or addition or removal of links, the SDN controller calls the application scheduler, which then updates its Topology-Map. Thereafter, we run the SA algorithm but only at the routing level, and only for flows that use the updated link. This reduces the state space to be explored. Similarly, upon a server failure, the application-level SA scheduler runs only for the affected tasks (it also calls the routing-level SA).

To support faster indexing for link failures, our approach also maintains a parallel hash table to Topology-Map, wherein the key is an edge $\mathcal{E}_i$ in from the network topology graph, while its value is the set of end host pairs who are using $\mathcal{E}_i$ in at least one of their k shortest paths. This makes it easier to update an edge quickly on a network change.

3. IMPLEMENTATION

We discuss implementation details of how we integrated cross-layer scheduling into the Storm scheduler, and into the Hadoop YARN scheduler.

3.1 Storm

Background: Storm processes streams, defined as unbounded sequences of tuples. As shown in Figure 1, there are four main abstractions in Storm – bolts (Computation-Node in the dataflow graph), spouts (source Computation-Nodes), topologies (the entire dataflow graph for one job), and stream groupings (which specify how tuples are grouped before being sent to a bolt). Thus, in the case of Figure 1, by the above definitions, V_1 and V_2 are spouts, while V_3, V_4 and V_5 are bolts, and V_5 is a sink.

In reality, Storm allows a job to specify a parallelism level for each bolt: the bolt is then parallelized into that many tasks. Our SA approach treats these tasks as our schedulable tasks. Within a bolt, the stream grouping is used to send the appropriate data to the appropriate tasks. Two examples of stream grouping functions are hash-based, and range-sharded by key.

When the Storm cluster first starts, each of the master node(s) runs a daemon called Nimbus [2] that is responsible for code distribution and monitoring the status of the worker servers. Each server runs a daemon called the Supervisor which listens waits for commands from Nimbus. In order to remain stateless and fail-fast, Nimbus and Supervisor servers do not communicate directly. Instead they utilize Zookeeper [24], which is a centralized service that provides distributed synchronization and providing group services.

Cross-Layer Scheduling Implementation: We modified Nimbus to contact the centralized SDN controller and obtain information about the underlying topology. This is done at start time as well as when something changes in the system, e.g., upon a new job arrival, or when a link is added or removed. When a new job is submitted, our Simulated Annealing Algorithm from Section 2 is called first, which then places the bolts and spouts accordingly on the servers. Then it sends a request to the SDN controller to allocate the paths for each pair of hosts.

3.2 Hadoop

Background: The latest versions of Hadoop (2.x) use the new YARN scheduler [50]. YARN decouples cluster scheduling from job-related functions: the former lives are in a Resource Manager (RM), while the latter is in a per-job Application Master (AM). The AM negotiates with the RM for each resource request (e.g., a container for a task). Each server also runs a daemon called the Node Manager (NM) for monitoring and local resource allocation. When an AM has been granted resources by the RM on a server, the AM forwards this information to the NM on that server, which in turn starts up the appropriate task.

Cross-Layer Scheduling Implementation: We modified the RM to contact the SDN controller and obtain information about the underlying topology. This is done at start time as well as when something changes in the system, e.g., upon a new job arrival, or when a link is added or removed. When a new job is submitted, the RM receives from the AM the number of map tasks and reduce tasks in that job. With this information, our cross-layer scheduling framework places these tasks by using our Simulated Annealing Algorithm from Section 2. In Mapreduce/Hadoop, the predominant network traffic is the "shuffle" traffic which carries data from the map tasks to the appropriate reduce tasks. In Hive and Pig, we also include the traffic from reduce tasks to the underlying file system (HDFS), and from HDFS to map tasks. In either case, once the task placement has been done using our SA algorithm, the RM asks the SDN controller to allocate the paths for each pair of communicating tasks (map to reduce in Mapreduce, and map to reduce and reduce to map in Hive/Pig).

4. EVALUATION

In this section, we evaluate the following questions about our cross-layer scheduling approach implemented in Hadoop and Storm.

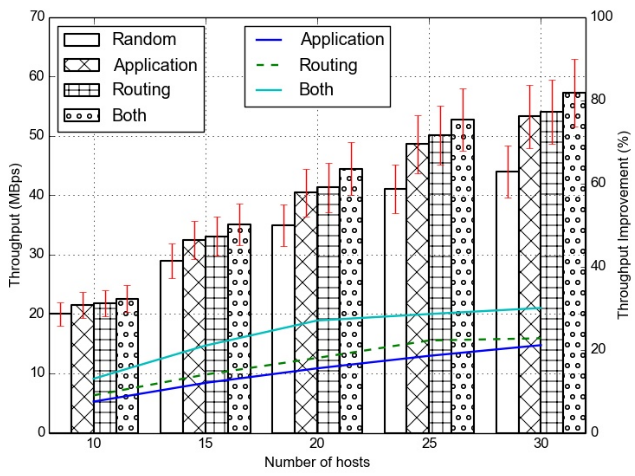

Figure 4: Cross-layer Approach's Throughput and Improvement in Fat-tree topology for Storm. Our SA approach improves throughput of vanilla Storm up to 30.0%.

1. What is the effect on throughput due to the adaptive SA algorithm running at the routing level, at the application level, and the cross-layer scheduling?
2. How does the system scale with number of servers?
3. How fast does our system make scheduling decisions?
4. How much are job completion times affected?

We use two network topologies to perform our experiments: Fat-tree [29] and Jellyfish [45]. The Fat-tree topology is a hierarchical topology that assigns more links to routers closer to the root. It is representative of structured topologies that are widely used in industry data centers today. Jellyfish on the other hand is a random graph topology that provides high throughput due its higher diversity of routes for any host pair. It is representative of unstructured topologies that are emerging out of networking research.

While there are many research testbeds for experimenting distributed systems such as Emulab [52] and AWS [1], there are unfortunately no widely-available and flexible testbeds for SDN research. As such we implemented our own software router and deployed it on an Emulab cluster. We use this setup for our deployment experiments (Sections 4.1 and 4.2). To measure the scalability of our system to hundreds of servers, we also supplement our results with a trace-driven simulation (Section 4.3).

We create the network topologies as follows, for a given number of servers. To create a Fat-tree topology, we create a large enough network topology that can accommodate the number of servers. Then, any non-core routers of the Fat-tree topology that are not connected to any server are pruned. In general, the Fat-tree topology can support $\frac{m^3}{4}$ nodes, where m is the number of links per routers. We use $m = 5$ links per router. Thereafter, we set up the Jellyfish topology using the same number of per-router links as the Fat-tree topology ($m = 5$).

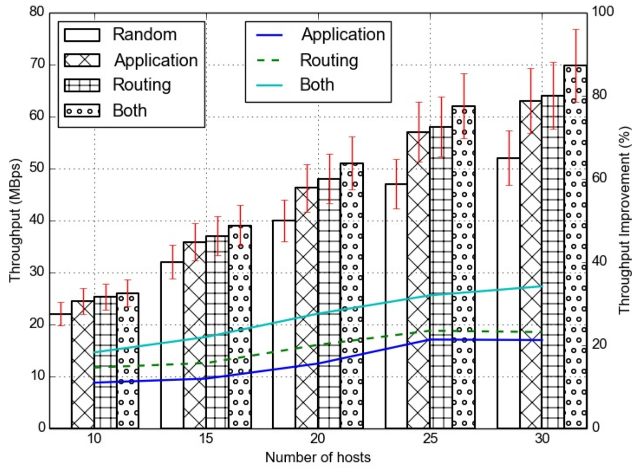

Figure 5: Cross-layer Approach's Throughput and Improvement in Jellyfish topology for Storm. Our SA approach improves throughput of vanilla Storm up to 34.4%.

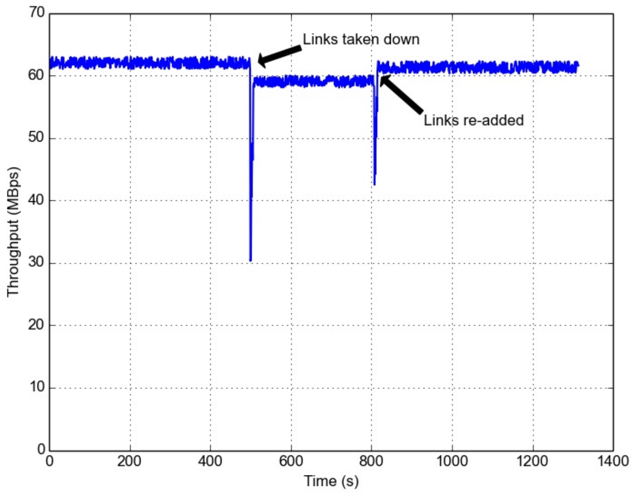

Figure 6: Fault-tolerance of SA in Storm: Average task throughput intervals during one run of 5 topologies running in a Jellyfish cluster of 30 machines. 10% links failed at 500 s. Links brought back up at 800 s.

4.1 Storm Deployment Experiments

Setup: Our Storm experiments are run on Emulab servers with a single 3 GHz processor, 2 GB of RAM and 200 GB disk, running Ubuntu 12.04. All network links have bandwidth of 100 MBps.

We generate Storm topologies (dataflow graphs) as follows. Each topology has two root nodes. There are a total of 10 bolts in the system. The spouts and bolts are allocated a number of children selected with a Gaussian distribution using $\mu = 2 = \sigma$. Each spout generates between 1 MB to 100 MB of data – these tuples are of size 100 B. In order to measure raw performance, bolts do not perform any processing but instead merely forward tuples they receive. This allows us to focus on the systems impact rather than be application-dependent.

In each of the experiments, the Storm topology is generated and submitted to the cluster at the rate of 10 jobs (topologies) per minute. We run the experiment for a total of 10 minutes. Each spout attempts to push tuples as fast as possible, and we measure the average achieved throughput.

Throughput Results: Figures 4 and 5 show how the throughput scales with increasing Emulab cluster size in two topologies. Four systems are shown: “Random” used random placement and routing, “Routing” uses SA at only the routing level, “Application” uses SA at only the application level, and “Both” uses the cross-layer scheduler. The bar graphs plot the raw throughput achieved by the system, while the lines plot the percentage improvement over the random routing.

In the Fat-tree topology (Figure 4), when the number of hosts is 10, we see that the average throughput of jobs running in the cluster is at 20 MBps when we do not run any of our scheduling. If we turn on the application-level scheduler, the average throughput of the jobs rises by only 7.5%. Using the routing-level scheduler, the improvement is 9.1%. When we have both of them turned on at the same time, the throughput improvement is 13.0%. This shows that the cross-layer approach is more preferable than solely using either the application-level or the routing-level approach.

As the number of hosts, routers, and links are increased, we observe that the percentage throughput improvement in steadily improves with scale. At 30 hosts, the throughput improvement of “Application”, “Routing” and “Both” rise to 21.1%, 22.7% and 30.0% respectively.

Figure 5 shows similar results for the Jellyfish network. When the number of hosts is 10, the percentage throughput improvement offered by “Application”, “Routing” and “Both” are 10.9%, 14.5% and 18.1% respectively. When we increase the number of hosts to 30, the throughput improvement of the three increases to 21.2%, 23.1% and 34.1% respectively.

From these two figures, we observe two trends. Firstly, since Jellyfish has a larger diversity of routes than Fat-tree, the improvement offered by each of “Application”, “Routing” and “Both” schedulers are higher for Jellyfish. Secondly, the throughput improvement scales with the number of hosts and routers that are present in the system. This is because a larger number of hosts and routers provides more route possibilities for the intelligent scheduler to choose from.

Fault-tolerance: We run 5 topologies, generated similar to the previous experiments, in a Jellyfish cluster of 30 servers and 30 routers. Figure 6 shows that at 500 s, we fail 10% of the links among the ones being used by at least one topology. At 800 s, these links are brought back up. The plot shows that the throughput drops immediately upon failure, but our scheduler reroutes the affected paths quickly within 0.4 s, and throughput returns to within 6.8% of the original throughput. After link recovery, the scheduler recovers quickly again to within 2% of the original throughput.

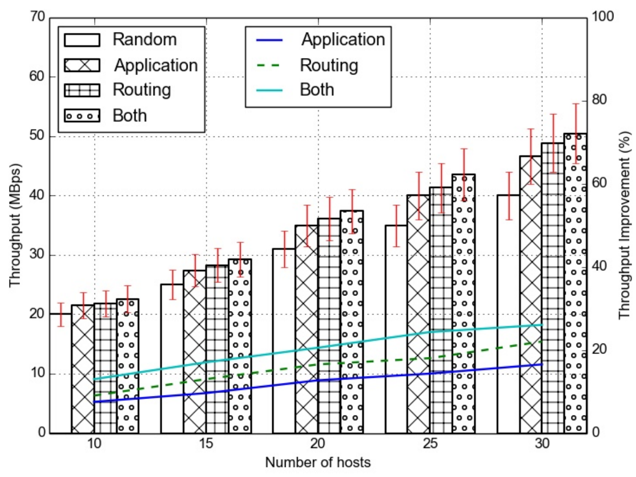

Figure 7: Cross-layer Approach's Throughput and Improvement in Fat-tree topology for Hadoop. Our SA approach improves throughput of vanilla Hadoop by up to 26.0%.

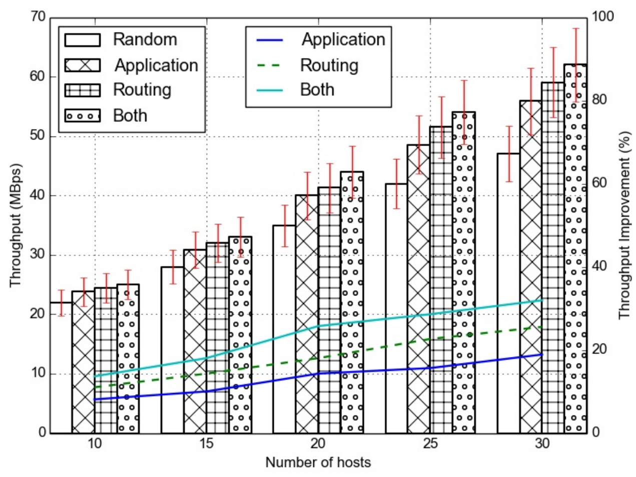

Figure 8: Cross-layer Approach's Throughput and Improvement in Jellyfish topology for Hadoop. Our SA approach improves throughput of vanilla Hadoop by up to 31.9%.

4.2 Hadoop YARN Deployment Experiments

Setup: Our Hadoop YARN deployment experiments use the same Emulab clusters and network topologies as described for the Storm experiments in Section 4.1.

Similar to the Storm experiments, the jobs that we run only forward data from mappers to reducers. We inject a job workload from the Facebook workload provided by the SWIM benchmark [4]. Jobs were injected in the rate of 1 job/s while the total amount of data that needs to be forwarded from the mappers to reducers range from 100 B to 10 GB.

Throughput Results: Figures 7 and 8 show the throughput and improvement due to application, routing, and cross-layer ("Both" bars and lines) under the Fat-tree and Jellyfish topologies respectively. The labels in these plots mean the same as in Section 4.1.

In the Fat-tree topology (Figure 7), when the number of hosts is 10, the average throughput of jobs running in the cluster is at 20 MBps when we do not run any of our scheduling. If we turn on the routing and application level SA, the throughput increases by 9.0% and 7.5% respectively. When we have the cross-layer scheduling enabled, the throughput improvement is 22.3%. This shows that the cross-layer approach is more beneficial than using either application-only or routing-only scheduling.

Similar performance benefits are also seen in jobs running under the Jellyfish topology (Figure 8) as we scale the number of hosts. With 30 hosts running in the cluster, throughput of the jobs rises by 25.5% and 18.8% as we turn on the routing-level and application-level scheduler respectively. When we have scheduler running at both levels, the throughput of the jobs rises by 31.9%.

Similar to the Storm results of Section 4.1, we observe that the throughput improvements for each of the three approaches (Application, Routing, Both) are higher in Jellyfish than in Fat-tree – once again, this is because of the higher route diversity in Jellyfish. Additionally, the throughput improvement also scales with the number of hosts and routers that are present in the system. However, unlike its Storm results, we observe that the throughput improvement offered by the application-level scheduler is slightly lower for Hadoop. This is because in our workload the Mapreduce jobs have only one stage of map and reduce tasks. Thus, the application-level heuristics discussed in Section 2 play a smaller part. With chained Mapreduce jobs (using Hive [48] or Pig [36]), the throughput improvement will approach that achieved by Storm.

Fault-tolerance: We run 5 WordCount Mapreduce jobs, with each having 4 mappers and 4 reducers, in a Jellyfish cluster of 30 servers and 30 routers. Figure 9 shows that after 500 s, we fail 10% of the links among the ones being used by at least one of the jobs. At 810 s, these links are brought back up. The plot shows that after failure, our scheduler reroutes affected paths within 0.35 s, returning throughput to within 4.7% of the original. After the links recover, the throughput is within 1.5% of the original throughput.

4.3 Simulation

We perform simulation experiments to measure time to schedule a job, the effect of scale, and job completion time. All experiments were run on a single server with a 4-core 2.50 GHz processor, and the bandwidth of each link in the simulation is set to 100 MBps. Each of the routers in the simulation has 15 links.

Scheduling Overhead: It is imperative for real-time jobs to be scheduled quickly. Figures 10 and 11 show the time for our cross-layer SA algorithm to run (for the new job), as a function of cluster size.

First we notice that for both Fat-tree and Jellyfish topologies, the time to schedule a new job scales linearly with

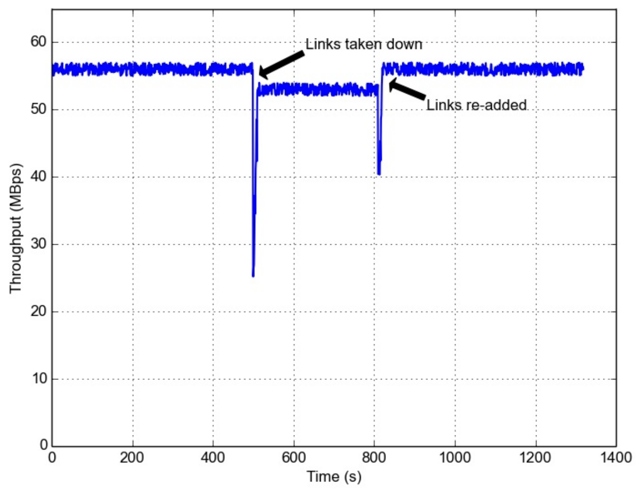

Figure 9: Fault-tolerance of SA in Hadoop: Average task throughput intervals during one run of 5 WordCount programs running in a Jellyfish cluster. 10% of links are failed at 500 s. Links brought back up at 810 s.

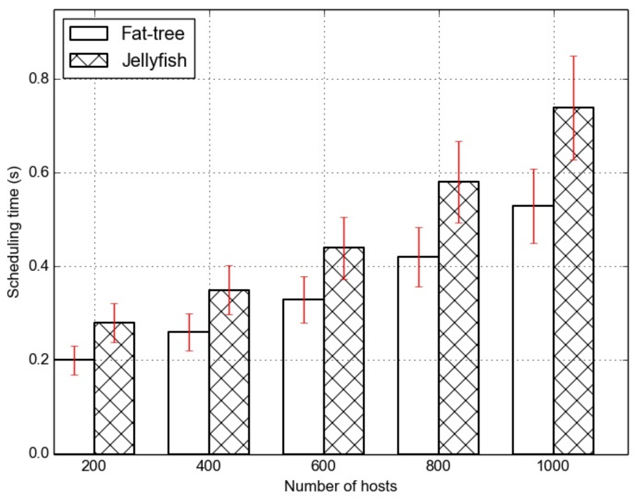

Figure 10: Scheduling time for a new job in the cross-layer scheduling approach for stream-processing dataflow graphs. At 1000 nodes, it only takes 0.53 s and 0.74 s per scheduling decision in Fat-tree and Jellyfish respectively.

cluster size, as expected – this is primarily because of the larger state space at the routing level. Second, we notice that this scheduling time is under a second for a cluster as large as 1000 servers.

Figure 10 shows for Storm-like dataflow graphs, scheduling decisions take 0.28 s in a 200-host Jellyfish topology. If we scale out to 1000 hosts, each scheduling decision takes 0.74 s on average. Similarly, in the Fat-tree topology, each scheduling decision takes 0.20 s in average to perform and if we scale it up to 1000 hosts, it takes 0.53 s on average.

For Hadoop/Mapreduce-like dataflow graphs, Figure 11 shows that a scheduling decision only takes 0.23 s on average in a 200-host Jellyfish topology. If we scale out to 1000 hosts, scheduling decisions take 0.67 s. Similarly, in the Fat-tree topology, scheduling decisions take 0.17 s on average and if we scale it up to 1000 hosts, it takes 0.48 s on average.

The scheduling time is slightly higher in the Jellyfish topology because of the higher diversity of routes. However, i) the scheduling time is small, and the ii) the overall throughput benefits are also higher for Jellyfish, as our results in the previous section indicated.

We only plotted scalability with cluster size as we found that the scheduling time does not depend much on the number of already-running jobs. This is because state spaces are independent of the number of jobs: both at the application-level and at the routing-level, our SA algorithm only considers available resources and the requirements of the new job.

We conclude that our cross-layer scheduling algorithms algorithm scales well with a large number of hosts in the cluster, and makes sub-second scheduling decisions with 1000 hosts.

Effect on Job Completion Times: Figures 12 and 13 show CDF plots of job completion time improvements in the Hadoop dataflow graph and Storm dataflow graph respectively, compared to a scheduler that picks random servers for the routing path selection. Figure 12 shows that our cross-layer scheduling framework improves Hadoop's job completion time by 34% (and 38%) at 50^{th} (and 75^{th}) percentile while Figure 13 shows that our cross-layer scheduling framework improves Storm's job completion time by 34% (and 41%) at 50^{th} (and 75^{th}) percentile.

We notice that only about 27% of Hadoop jobs and about 24% of Storm jobs suffer degradation in completion time in the Fat-tree topology. In the Jellyfish topology, around 18% Hadoop jobs and 16% Storm jobs degrade – these are lower than Fat-tree's because of Jellyfish's higher route diversity.

However, our approaches do not cause starvation. The worst-case degradation is under 20% for Hadoop and 30% for Storm. These degraded jobs are either longer jobs, or jobs submitted immediately after long jobs. In essence, our heuristics schedule short-running jobs along network paths that have a small number of hops. When a longer job arrives, our scheduler ends up scheduling the long-running jobs' many flows along longer paths. This in turn slows the longer job, and also disrupts the throughput of future submitted jobs until the longer job is done. As real-time analytics jobs in general do not run too long, our approach does not cause starvation.

We conclude that our cross-layer scheduling improves completion of a large majority of jobs in both Storm and Hadoop, and for network topologies that are both structured (Fat-tree) and unstructured (Jellyfish).

5. RELATED WORK

Computation Scheduling: There has been significant work in improving scheduling of Mapreduce [12] and Hadoop [3]. Some systems improve job scheduling by preserving data locality [5, 25, 26, 54]. Others fairness across multiple tenants for CPU and memory resources [15, 28]. Tasks requiring dif-

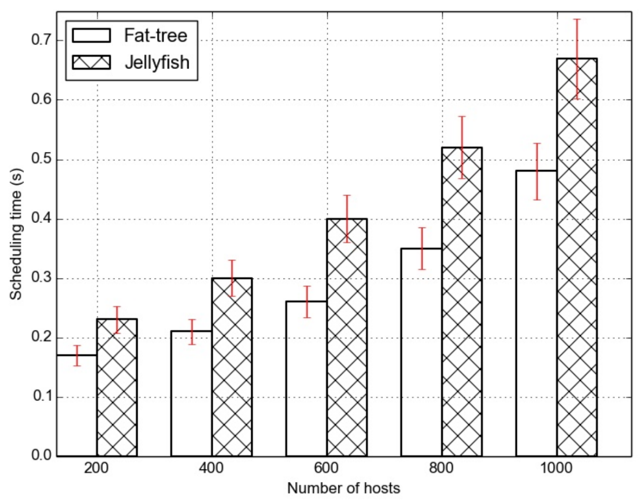

Figure 11: Scheduling time for a new job in cross-layer scheduling for Mapreduce/Hadoop dataflow graphs. At 1000 nodes, our cross-layer scheduling framework takes 0.48 s and 0.67 s per scheduling decision in Fat-tree and Jellyfish respectively.

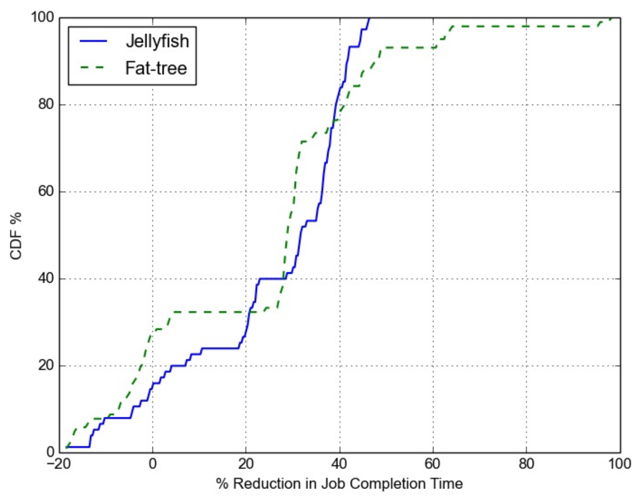

Figure 12: CDF of job completion time improvements in Hadoop at 1000 hosts when cross-layer scheduling framework is activated.

ferent resources can be allocated to appropriate servers [42]. Systems like Jockey [13] compute job profiles using a simulator, and use the resulting dependency information to dynamically adjust resource allocation. Systems like Mantri [6] look to mitigate stragglers in Mapreduce jobs. These systems are orthogonal to our work. Further they also largely target long batch jobs, while we target shorter real-time jobs.

Sparrow [37] proposes fully decentralized scheduling in milliseconds that is within 12% of an ideal scheduler's performance, by using the power of two choices. We believe our work opens up the door to similarly decentralized versions of our cross-layer SA scheduling algorithm.

Many streaming frameworks have been proposed such as Storm [2], Spark [55], Naiad [34], TimeStream [39]. However, none of these schedulers expose or use information about the underlying network topology.

Routing-Level Scheduling: Orchestra [10] and Seawall [43] propose weighted-fair approaches for network flows in order to make the shuffle phase more efficient in Mapreduce. Oktopus [8] and SecondNet [19] proposed static reservations throughout the network to implement bandwidth guarantees for the hose model and pipe model, respectively. Gatekeeper [41] proposes a per-VM hose model with work conservation, however its hypervisor-based mechanism works only for full bisection-bandwidth networks. FairCloud [38] introduces policies to achieve link-proportionality or congestion-proportionality. All of these systems are orthogonal to our work, and can be utilized alongside our system.

Software-Defined Networking (SDN): In the pre-SDN era, some systems provided custom routing protocols to applications [9], or even allowed applications to embed code in network devices to perform application processing [47]. In comparison, SDNs provide a more fine-grained control of the network. Much of the existing work on SDNs is targeted at either improving SDN scalability (e.g., enabling a hierarchy of policies to SDN controller [40]), enforcing correctness in the SDN (e.g., enabling consistent updates in SDN [40]) or enabling the SDN to work with broader range of the current network protocols (e.g., radio network [18], cellular network [30]).

Cross-Layer Scheduling: Some systems (e.g., [51]) considered a two-level optimization for data processing frameworks. They proposed batching intervals of job submissions for scheduling. However, that work did not consider concrete applications for their experiments. Flowcomb [11] improved Hadoop performance by sending application-level hints to the SDN centralized controller. In effect, that paper only explored routing-level optimization, while our work explores application-level placement as well as cross-layer scheduling.

Previously, cross-layer scheduling has been used in the HPC community [46, 56], and in operating systems [16, 31, 44]. Isolation is used widely to improve performance in virtualized clouds such as Amazon EC2 [1] and Eucalyptus [35] and in generic schedulers like YARN [50] and Mesos [23].

VM Placement: The cross-layer scheduling problem arises in virtualized settings as well. Some systems [27, 33] have used proposed migrating virtual servers by using Markov approximation that attempts to minimize the amount of VM migrations. [21] proposes a shadow routing based VM placement algorithm, also using combinatorial optimization. Such combinatorial optimization may work well for long-running virtual servers, but it is expensive for short jobs that process real-time data.

6. SUMMARY

We have proposed, implemented, and evaluated a cross-layer scheduling framework for cloud computation stacks running real-time analytics engines. We have integrated our simulated annealing-based algorithm into both a batch-

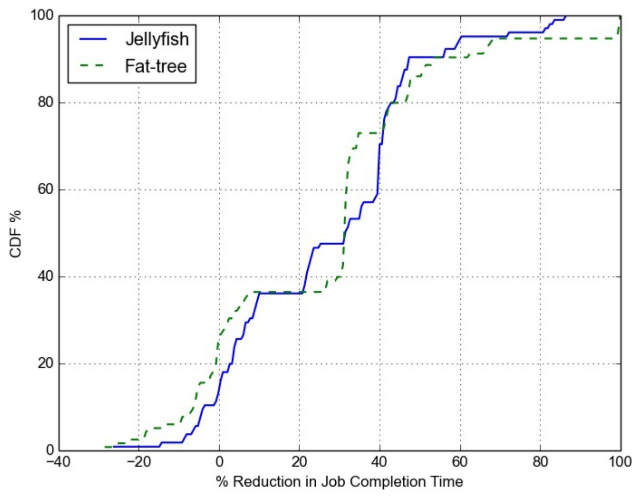

Figure 13: CDF of job completion time improvements in Storm at 1000 hosts when cross-layer scheduling framework is activated.

processing system (Hadoop YARN) as well as a stream-processing system (Storm). Our evaluations have used both structured network topologies (Fat-tree) and unstructured ones (Jellyfish). Our cross-layer approaches provide combined benefits from both the application-level and SDN-level schedulers. Our deployment experiments showed that our approaches improve throughput for Hadoop by 22-31% and for Storm by 30-34%. The throughput improvements are higher in network topologies with a higher route diversity.

While we have only considered placement of tasks for new jobs, our work opens the avenue to tackle the more complex problem of migration of tasks of already-executing jobs in the cluster. This needs to be done efficiently but also infrequently, to balance off the benefits from rescheduling with the overheads of migration.

7. REFERENCES

[1] Amazon EC2.
[2] Storm, distributed and fault-tolerant realtime computation.
[3] Apache Hadoop, Oct. 2013.
[4] Statistical workload injector for mapreduce (SWIM), Aug. 2013.
[5] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: coping with skewed content popularity in mapreduce clusters. In *Proc. ACM Eurosys*, pages 287–300, 2011.
[6] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in Map-Reduce clusters using Mantri. In *Proc. Usenix OSDI*, volume 10, page 24, 2010.
[7] M. Appelman and M. de Boer. Performance analysis of OpenFlow hardware, 2012.
[8] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *ACM SIGCOMM CCR*, volume 41, pages 242–253, 2011.
[9] P. Chandra, A. Fisher, C. Kosak, T. E. Ng, P. Steenkiste, E. Takahashi, and H. Zhang. Darwin: Customizable resource management for value-added network services. In *Proc. 6th IEEE ICNP*, pages 177–188, 1998.
[10] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with Orchestra. *ACM SIGCOMM CCR*, 41(4):98, 2011.
[11] A. Das, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and C. Yu. Transparent and flexible network management for big data processing in the cloud. In *Proc. 5th Usenix HotCloud*.
[12] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *CACM*, 51(1):107–113, 2008.
[13] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proc. 7th ACM Eurosys*, pages 99–112, 2012.
[14] R. W. Floyd. Algorithm 97: shortest path. *CACM*, 5(6):345, 1962.
[15] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *Proc. 11th Usenix NSDI*, 2011.
[16] P. Goyal, X. Guo, and H. M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proc. Usenix OSDI*, volume 96, pages 107–121, 1996.
[17] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *ACM SIGCOMM CCR*, volume 39, pages 51–62, 2009.
[18] A. Gudipati, D. Perry, L. E. Li, and S. Katti. Softran: Software defined radio access network. In *Proc. 2nd ACM HotSDN*, pages 25–30, 2013.
[19] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *Proc. 6th ACM Conf. on Emerging Networking Experiments and Technologies*, page 15, 2010.
[20] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. Dcell: a scalable and fault-tolerant network structure for data centers. In *ACM SIGCOMM CCR*, volume 38, pages 75–86, 2008.
[21] Y. Guo, A. L. Stolyar, and A. Walid. Shadow-routing based dynamic algorithms for virtual machine placement in a network cloud. In *Proc. IEEE Infocom*, pages 620–628, 2013.
[22] L. Gyarmati and T. A. Trinh. Scafida: a scale-free network inspired data center architecture. *ACM SIGCOMM CCR*, 40(5):4–12, 2010.
[23] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proc. 8th Usenix NSDI*, pages 22–22, 2011.
[24] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proc. Usenix ATC*, volume 8, pages 11–11,

2010.

[25] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS OSR*, 41(3):59–72, 2007.

[26] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proc. 22nd ACM SOSP*, pages 261–276, 2009.

[27] J. W. Jiang, T. Lan, S. Ha, M. Chen, and M. Chiang. Joint VM placement and routing for data center traffic engineering. In *Proc. IEEE Infocom*, pages 2876–2880, 2012.

[28] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang. Multi-resource allocation: Fairness-efficiency tradeoffs in a unifying framework. In *Proc. IEEE Infocom*, pages 1206–1214, 2012.

[29] C. E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Trans. Computers*, 34(10):892–901, Oct. 1985.

[30] L. E. Li, Z. M. Mao, and J. Rexford. Toward software-defined cellular networks. In *Proc. IEEE European Wshop. on SDN*, pages 7–12, 2012.

[31] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanovic, and J. Kubiatowicz. Tessellation: Space-time partitioning in a manycore client OS. *Proc. 1st Usenix HotPar*, 3:2009.

[32] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM CCR*, 38(2):69–74, 2008.

[33] X. Meng, V. Pappas, and L. Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *Proc. IEEE Infocom*, pages 1–9, 2010.

[34] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proc. 24th ACM SOSP*, pages 439–455, 2013.

[35] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus open-source cloud-computing system. In *Proc. 9th IEEE/ACM Intnl. Symp. Cluster Computing and the Grid*, pages 124–131, 2009.

[36] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proc. ACM SIGMOD*, pages 1099–1110, 2008.

[37] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proc. 24th ACM SOSP*, pages 69–84, 2013.

[38] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. Faircloud: sharing the network in cloud computing. In *Proc. ACM SIGCOMM*, pages 187–198, 2012.

[39] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. Timestream: Reliable stream computation in the cloud. In *Proc. 8th ACM Eurosys*, pages 1–14, 2013.

[40] M. Reitblatt, N. Foster, J. Rexford, and D. Walker. Consistent updates for software-defined networks: Change you can believe in! In *Proc. 10th ACM HotNets*, page 7, 2011.

[41] H. Rodrigues, J. R. Santos, Y. Turner, P. Soares, and D. Guedes. Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks. *Proc. 3rd Usenix Wshop. I/O Virtualization*, 2011.

[42] T. Sandholm and K. Lai. Mapreduce optimization using regulated dynamic prioritization. In *Proc. 11th ACM Intnl. Joint Conf. on Measurement and Modeling of Computer Systems*, pages 299–310, 2009.

[43] A. Shieh, S. Kandula, A. Greenberg, and C. Kim. Seawall: performance isolation for cloud datacenter networks. In *Proc. Usenix HotCloud*, pages 1–1, 2010.

[44] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating system concepts*, volume 8. Wiley, 2013.

[45] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: networking data centers randomly. In *Proc. 9th Usenix NSDI*, pages 17–17, 2012.

[46] G. Staples. Torque resource manager. In *Proc. 2006 ACM/IEEE Conference on Supercomputing*, page 8, 2006.

[47] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, 1997.

[48] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endowment*, 2(2):1626–1629, 2009.

[49] P. J. Van Laarhoven and E. H. Aarts. *Simulated annealing*. Springer, 1987.

[50] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proc. 4th ACM SoCC*, page 5, 2013.

[51] G. Wang, T. Ng, and A. Shaikh. Programming your network at run-time for big data applications. In *Proc. 1st ACM HotSDN*, pages 103–108, 2012.

[52] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. *ACM SIGOPS OSR*, 36(SI):255–270, 2002.

[53] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proc. Usenix OSDI*, volume 8, pages 1–14, 2008.

[54] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proc. 5th ACM Eurosys*, pages 265–278, 2010.

[55] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proc. 24th ACM SOSP*, pages 423–438, 2013.

[56] S. Zhou. Lsf: Load sharing in large heterogeneous distributed systems. In *Intnl. Wshop. on Cluster Computing*, 1992.