Cross-Layer Scheduling in Cloud Systems

Hilfi Alkaff, Indranil Gupta and Luke M. Leslie
Department of Computer Science,
University of Illinois at Urbana-Champaign
Email: {alkaff2, indy, lmlesli2}@illinois.edu

***Abstract*—Today, cloud computing engines such as stream-processing Storm and batch-processing Hadoop are being increasingly run atop software-defined networks (SDNs). In such cloud stacks, the scheduler of the application engine (which allocates tasks to servers) remains decoupled from the SDN scheduler (which allocates network routes). We propose [1] 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 tandem with the selection of network routes that arise from these tasks. We present results from both cluster deployment and simulation, and using two representative network topologies: Fat-tree and Jellyfish. Our results show that cross-layer scheduling can improve throughput of Hadoop and Storm by between 26% to 34% in a 30-host cluster, and it scales well.**

***Keywords*—Cloud computing, Hadoop, Storm, SDN, Cross-layer**

I. Introduction

Real-time data analytics is a quickly-growing segment of industry. This includes, for instance, processing data from advertisement pipelines at Yahoo!, Twitter's real-time search, Facebook's trends, real-time traffic information, etc. In all 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).

Today, industry uses two categories of data analytics engines for real-time data analytics. The first category consists of cloud batch-processing systems such as Hadoop [14], [6], Hive [49], Pig [38], DryadLinq [54], etc. The second generation includes new stream-processing engines such as Storm [3].

Inside datacenters, these data analytics engines are run atop a variety of networks. Some datacenters use hierarchical network topologies like the Fat-tree [31], while others use newer networks like Jellyfish [46], Scafida [24], VL2 [19], DCell [22], etc. Yet, the scheduler of the application engine (which allocates tasks to servers) remains decoupled from the SDN scheduler (which allocates network routes). Concretely, while the application scheduler performs task placement in a manner that is aware of the data placement at servers, or it might adapt network flows in an end-to-end manner, e.g., [12], it does not usually coordinate with the network layer underneath.

In this paper, we explore a cross-layer approach to scheduling in such cloud stacks. 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 that, as far as real-time processing is concerned, both batch and streaming cloud engines can be represented as dataflow graphs. Figure 1 shows a sample Storm application graph, where each vertex is a computation node, called a *bolt* in Storm (source bolts are called *spouts*). Edges represent the flow of data, and data takes the form of tuples. Each bolt processes the tuples by, e.g., filtering, mapping, joining, etc. 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 each vertex of the dataflow graph a *Computation Node*. 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, e.g., 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 [34]. 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 [9].

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 approach on two topologies that lie at opposite ends of the diversity spectrum. Our first topology, the Fat-tree, is structured, and allows diversity for only a limited number of routes, i.e., routes traversing closer to the root. Our second topology, Jellyfish [46], uses a random graph for the network, thus offering an extremely high diversity of route choices for all host pairs. Any other network topology would lie in-between Fat-tree and Jellyfish on this diversity spectrum, and its behavior with our cross-layer approach can be interpreted from our results for Fat-tree and Jellyfish.

The goal of our cross-layer approach is a placement of tasks and a selection of routes that together achieves high throughput for the application. Our approach involves the application-level scheduler using available end-to-end bandwidth information to schedule tasks at servers. The application scheduler calls the

[1]This work was supported in part by AFOSR/AFRL grant FA8750-11-2-0084, NSF grant CCF 0964471, NSF grant CNS 1319527, and NSF grant CNS 1409416.

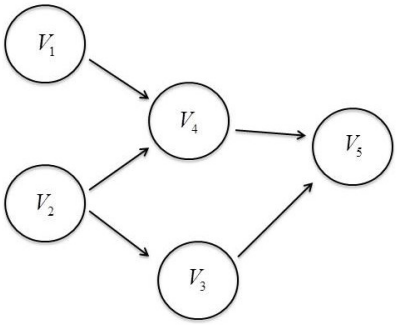

Fig. 1: Stream-processing engines such as Storm can be represented using a dataflow graph. Each vertex is a bolt processing data, and edges show the flow of data.

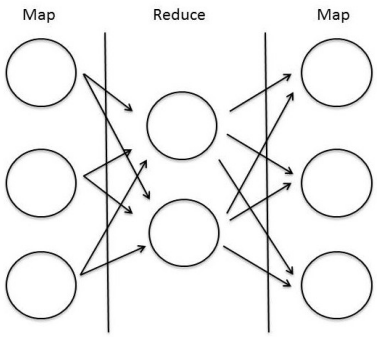

Fig. 2: Batch-processing engines such as Hadoop, Hive, and Pig can be represented as a dataflow graph.

SDN controller which finds the best routes for those end-to-end paths. This is done iteratively. While making their decisions, each of the application scheduler and the SDN scheduler need to deal with a very large state space. There are combinatorially many ways of assigning tasks to servers, as well as routes to each end-to-end flow.

To address the state space challenge, we use Simulated Annealing. Inspired by metallurgy, this approach [50] 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 and in the SDN level.

The technical contributions of this paper are as follows:

- We design a novel cross-layer scheduling framework enabling cloud applications to coordinate with the underlying SDN network scheduler.
- We show how to quickly locate near-optimal cross-layer scheduling decisions via Simulated Annealing.
- We show generality by implementing our approach for Hadoop and Storm at the application level, and in Fat-tree and Jellyfish network topologies.
- We perform real deployment as well as simulated cluster experiments. In a 30-host cluster, our approach improves throughput for Hadoop by 26-31% and for Storm by 30-34% (ranges depend on network topology).

This paper is structured as follows. Section II presents our Simulated Annealing-based approach to scheduling. Section III discusses implementation details for both Storm and Hadoop. Section IV presents our experiments. We present related work in Section V, and conclude in Section VI.

II. Design

In this section, we present our cross-layer scheduling approach inspired by Simulated Annealing (SA).

Our cross-layer scheduling framework is run whenever a new job arrives. Our approach computes good placement and routing paths for the new job. For already-running jobs, their task placement and routing paths are left unchanged – this is due to three motivating reasons: i) it reduces the effect of the new job on already-running jobs; ii) migrating already-running tasks may be prohibitive, especially since real-time analytics jobs are typically 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, our SA approach uses a utility function that considers the throughput for all jobs in the cluster.

Notation: For a network topology graph, G, its vertex set, V, comprises both the physical servers (i.e., servers) and the routers. Its edge set, E, consists of the links between servers. Each job J_i that is running on this topology contains T_i tasks. We denote a server as M_i.

Pre-Computation of the Topology Map: When our data processing framework starts, we run a modified version of the Floyd-Warshall Algorithm [16] 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 the *Topology Map*. The keys of the Topology Map consist of a pair of servers, (M_i, M_j), where $M_i, M_j \in V$ and $i < j$. The values associated with each key are the k shortest paths, which we call $K_1, K_2, \cdots, K_k$. Each path is stored along with the available bandwidth on each of its constituent links. The available bandwidth is updated periodically by the SDN layer.

We argue that the Topology Map hash table is small in size. 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 $= 1000^2/2$ nodes $\times$ 10 hops/node pair $\times$ 10 different paths = 50 Million. With a few bytes for each entry, this is still $O(GB)$, which can be maintained in-memory in a typical commercial off-the-shelf (COTS) server.

This hash table can be further compacted because the k shortest paths for (M_i, M_j), are likely to overlap in their intermediate routers. Thus, instead of storing $K_1, K_2, \cdots, K_k$ individually, we store the subgraph $G'_{i,j}$ that is a union of these k paths. For the above example with 1000 servers, this compacted hash table was only 6 MB.

Finally, creating this hash table is fast – for our running example above, creation takes around 3 minutes.

States: The SA 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 $P_1, P_2, \cdots$ in G whose end-points are a pair of servers. Thus, each P_i is obtained from the Topology Map. We define the *neighbors of a state* $\mathcal{S}$ as those states that differ from $\mathcal{S}$ in exactly one path P_i.

At the application level, a state $\mathcal{S}$ consists of the current placements of the worker tasks across servers $M_i \in V$. 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) [50] is an iterative approach to reach an optimum in a state space. Given the current state $\mathcal{S}$, the core SA heuristic considers some of its neighboring states $\mathcal{S}'$, and probabilistically decides between either: i) moving the system to a neighbor state $\mathcal{S}'$, or ii) staying in state $\mathcal{S}$. This is executed iteratively until a convergence criterion is met.

When a user submits a job (Hadoop job or Storm application topology), our system runs in two phases. In the first phase, our application-level scheduler consults the Topology Map to determine the servers on which placement will yield the best throughput, along with the network paths that are the best matches for the end-to-end flows in that placement. As mentioned earlier, we neither change the task placement for already-running jobs, nor their allocated paths. In the second phase, the SDN controller sets up the requested paths.

We discuss the first phase in more detail. During this phase, Algorithm 1 is run at both the application and routing levels. The former level calls the functions INITSTATE and GENSTATE from Algorithm 2, while the latter calls them from 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 (line 16 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 selects one prospective next-state from among these. Then, the utility of this prospective state is computed by the COMPUTEUTIL function.

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

Our de-allocation heuristic works as follows. From Section I, recall that a real-time data analytics application is a graph of Computation Nodes. A Computation Node which is adjacent to either a source Computation Node (called a *spout* in Storm) or a sink Computation Node is more likely (than other Computation Nodes) to directly affect the overall throughput of the new job. Hence, for each Computation Node, C_i, we first calculate its priority, $\mathcal{P}_{C_i}$ = number of source Computation Nodes adjacent to C_i + number of sink Computation Nodes adjacent to C_i. Then, we probabilistically select a Computation Node with probability = $\frac{\mathcal{P}_{C_i}}{\sum_j \mathcal{P}_{C_j}}$.

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

```
1:  function MAIN(graph, hosts)
2:      bestUtil ← 0
3:      bestState ← Null
4:      for i ← 1 to 5 do
5:          t ← initialTemperature
6:          currState ←INITSTATE(graph, hosts)
7:          currUtil ← 0
8:          for j ← 1 to maxStep do
9:              newState←GENSTATE(graph, hosts, currState)
10:             newUtil←COMPUTEUTIL(graph, newState)
11:             r ← random()
12:             if TRANSITION(currUtil, newUtil, t)>r then
13:                 currState ← newState
14:                 currUtil ← newUtil
15:             end if
16:             if currUtil ≥ bestUtil then
17:                 bestState ← currState
18:                 bestUtil ← currUtil
19:             end if
20:             t ← t^0.95
21:         end for
22:     end for
23:     return bestState
24: end function
25:
26: function TRANSITION(oldUtil, newUtil, temperature)
27:     if newUtil > oldUtil then
28:         return 1
29:     else
30:         return e^((newUtil−oldUtil)/temperature)
31:     end if
32: end function
33:
34: function COMPUTEUTIL(graph, state)
35:     util ← 0
36:     jobs ← existing jobs + new job
37:     for Each sink Computation Node s in jobs do
38:         sUtil ← inf
39:         for Each root Computation Node of s do
40:             path ← TopologyMap.get(sink, root)
41:             sUtil ← min(sUtil, min bandwidth in path)
42:         end for
43:         util ← util + sUtil
44:     end for
45:     return util
46: end function
```

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 in Algorithm 3.

We first describe our path heuristic. We prefer to allocate paths that have the lowest number of hops. This has two advantages: i) it minimizes the latency and maximizes bandwidth

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

```
1: function INITSTATE(graph, servers)
2:     currAllocation ← {}
3:     for Each task t in graph.ComputationNode() do
4:         server ← random(servers)
5:         currAllocation[server].add(t)
6:     end for
7:     return currAllocation
8: end function
9:
10: function GENSTATE(graph, servers, currAllocation)
11:     task ← de-alloc heuristic(graph.ComputationNode())
12:     server ← currAllocation.find(task)
13:     DEALLOCATE(task, server)
14:     newMachine ← random(servers)
15:     currAllocation[newMachine] ← task
16:     routes ← MAIN(graph, servers) // routing level SA
17:     Set the routes for currAllocation
18:     return currAllocation
19: end function
```

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

```
1: function INITSTATE(graph, commMachines)
2:     currRoutes ← {}
3:     for Each pair in commMachines do
4:         routes ← TopologyMap.get(pair)
5:         currRoutes[pair] = random(routes)
6:     end for
7:     return currRoutes
8: end function
9:
10: function GENSTATE(graph, commMachines, currRoutes)
11:     commPair ← de-path heuristic(currRoutes)
12:     routes ← TopologyMap.get(commPair)
13:     newRoute ← path heuristic(routes)
14:     currRoutes[commPair] ← newRoute
15:     return currRoutes
16: end function
```

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 these, we select that 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 the route which has highest number of hops. We break ties by selecting 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, s, it finds all its parent Computation Nodes, t, and the network bandwidth (in the currently chosen network paths) from t to s. It then calculates the *minimum* bandwidth among all the parents of s. This is the utility of s. We use minimum bandwidth because an application's overall throughput is bottlenecked by the minimum bandwidth between source and sink nodes. 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 along with sink Computation Nodes of the new job.

Next, 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, $newUtil$, is greater than the current total utility, $currUtil$, we always transition to the new state. If the new utility is lower, however, we transition with a probability that is related to both the difference in utilities and 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 more robust, we repeat the entire SA exploration 5 additional times: this helps ensure that we are not stuck in a local optima.

While our current utility function accounts only for bandwidth, COMPUTEUTIL can be extended by considering other metrics, such as CPU utilization, I/O utilization, etc., for the constituent tasks. As a result, our approach can be combined with existing approaches such as Mesos [25], Spark [56], etc.

Fault-tolerance and Dynamism: Upon a failure or change in the network, the SA algorithm can be re-run quickly. 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 explored state space. Upon a server failure, the application-level SA scheduler runs only for the affected tasks (and it calls the routing-level SA).

To support faster indexing for link failures, our approach also maintains a parallel hash table to the Topology Map, wherein the key is an edge E_i from the network topology graph, and the associated value is the set of end host pairs who are using E_i in at least one of their k shortest paths. This facilitates fast edge updates upon network changes.

III. IMPLEMENTATION

In this section, we discuss implementation details of how we integrated our cross-layer scheduling approach into the Storm scheduler and the Hadoop YARN scheduler.

A. Storm

Background: Storm processes streams, defined as unbounded sequences of tuples. As shown in Figure 1, there are five main abstractions in Storm: bolts (Computation Node in the dataflow graph), spouts (source Computation Nodes), sink bolts, topologies (the entire dataflow graph for one job), and

stream groupings (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, V_1-V_5 are bolts, and V_5 is a sink.

Storm allows the user 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, stream grouping decides which data goes to which tasks, e.g., hash-based, range-sharded by key, etc..)

When the Storm cluster first starts, each of the Storm master nodes runs a daemon called Nimbus [3], which is responsible for distributing code and monitoring the status of the worker servers. Each server (worker) runs a daemon called the Supervisor, which listens for commands from Nimbus. In order to remain stateless and fail-fast, Nimbus and Supervisor servers do not communicate directly. Instead, they utilize Zookeeper [26], a centralized service that provides distributed synchronization and 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 upon 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 SA Algorithm from Section II is called first, which then places the bolts (including sinks) and spouts accordingly on the servers. The algorithm then asks the SDN controller to allocate the chosen paths for each host pair.

B. Hadoop

Background: The latest versions of Hadoop (2.x) use the new YARN scheduler [51]. YARN decouples cluster scheduling from job-related functions: the former lives 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 server's NM, which starts the 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 upon changes in the system (as with Storm). 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 SA Algorithm from Section II.

In MapReduce/Hadoop, the predominant network traffic is the "shuffle" traffic, which carries data from the map tasks to the appropriate reduce tasks. HDFS nodes also become a server node in our approach, which is then considered as a parameter during our SA algorithm. 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).

IV. Evaluation

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

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 representative network topologies to perform our experiments: Fat-tree [31] and Jellyfish [46]. These two choices are conveniently located at opposite ends of a spectrum – Fat-tree is a structured (and hierarchical) topology with limited route diversity, while Jellyfish is an unstructured (and random) topology with high route diversity. Most other existing networks would fall in between these two ends of the spectrum. Thus, our experimental results with these two

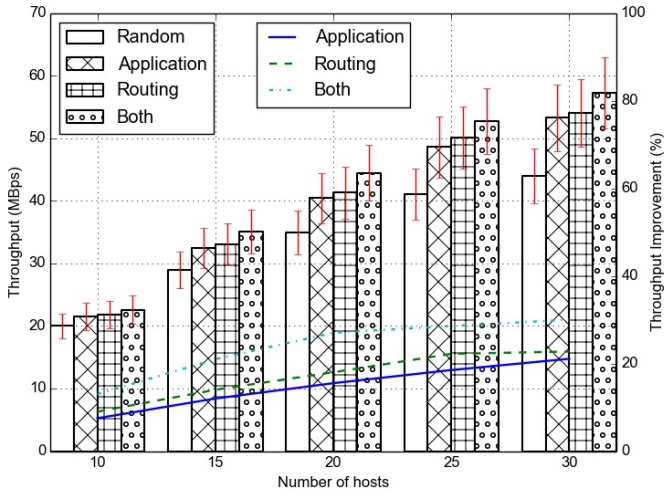

Fig. 3: Cross-layer Approach's Throughput and Improvement in Fat-tree topology for Storm. Our SA approach improves throughput of vanilla Storm up to 30.0%.

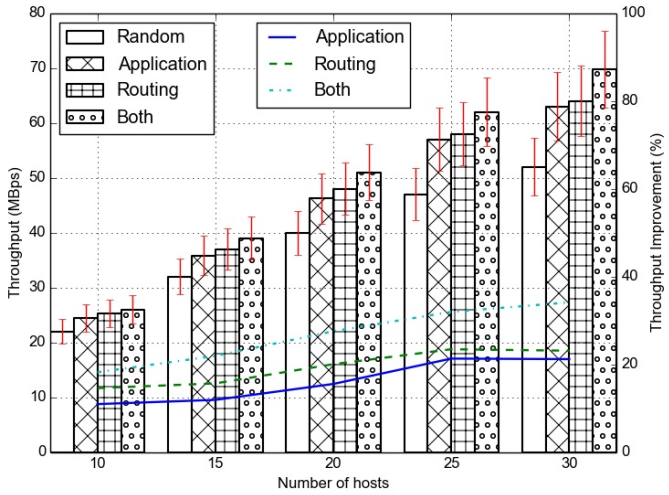

Fig. 4: Cross-layer Approach's Throughput and Improvement in Jellyfish topology for Storm. Our SA approach improves throughput of vanilla Storm up to 34.1%.

topologies should not be seen as merely samples in the space of topologies, but rather, given any other topology, its performance under our approach would lie somewhere in between that of Fat-tree and that of Jellyfish.

We present both deployment experiments on Emulab [53], and simulation results. For the former, due to the paucity of SDN testbeds, we wrote our own software router using ZeroMQ [4] and Thrift [2], and deployed it in Emulab.

For a given number of servers, we create the network topologies as follows. 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 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 ($m = 5$).

A. 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. Network links are 100 MBps.

We generate Storm application 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 random number of children sample from a Gaussian distribution with $\mu = 2 = \sigma$. Each spout generates between 1 MB to 100 MB of data – these tuples are of size 100 B. In order to measure raw network 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.

Storm jobs (topologies) are submitted to the cluster at the rate of 10 jobs/minute, and each trial runs for 10 minutes. Each spout attempts to push tuples as fast as possible, thus saturating the system throughput.

Throughput Results: Figures 3 and 4 show how the throughput scales with increasing Emulab cluster size in two topologies. Four systems are shown: "Random" uses random placement and routing [2], "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 Fat-tree (Figure 3), at 10 hosts, without any of our scheduling, the average job throughput is 20 MBps. When we turn on the application-level scheduler, the average throughput of the jobs rises by only 7.5%. Using only 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 better than application-only or routing-only approaches.

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

[2]We compare against Random since: i) existing schedulers are not network-aware; and ii) our techniques can be orthogonally combined with CPU- and I/O-aware scheduling (Section II).

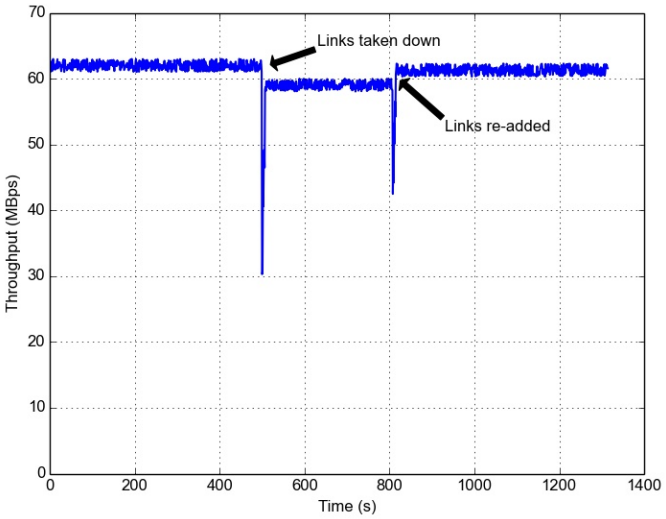

Fig. 5: Fault-tolerance of SA in Storm: Average task throughput intervals during one run of 5 topologies running in a Jellyfish cluster of 30 machines. 10% links failed at 500 s. Links brought back up at 800 s.

Figure 4 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. Essentially, more hosts and routers implies more route possibilities for the network scheduler to choose among.

Fault-tolerance: We run 5 topologies in a Jellyfish cluster of 30 servers and 30 routers. Figure 5 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. Right after the failure, the throughput plummets but recovers quickly (within 0.4 s) to within 6.8% of the pre-failure value. After link recovery, the scheduler recovers quickly to within 2% of the pre-failure throughput.

B. Hadoop YARN Deployment Experiments

Setup: Our Hadoop YARN deployment experiments use the same Emulab clusters and network topologies as Section IV-A.

We inject a job workload from the Facebook workload provided by the SWIM benchmark [5], with mappers forwarding data to reducers. Jobs are injected at the rate of 1 job/s, while map to reduce shuffle traffic ranges from 100 B to 10 GB.

Throughput Results: Figures 6 and 7 show the throughput and improvement due to application, routing, and cross-layer

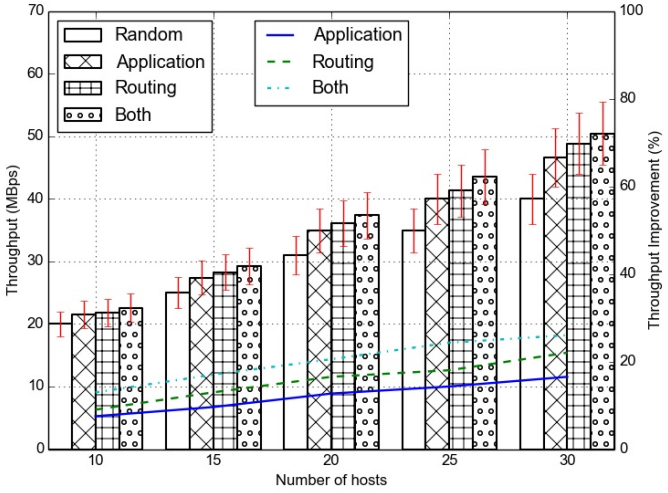

Fig. 6: Cross-layer Approach's Throughput and Improvement in Fat-tree topology for Hadoop. Our SA approach improves throughput of vanilla Hadoop by up to 26.0%.

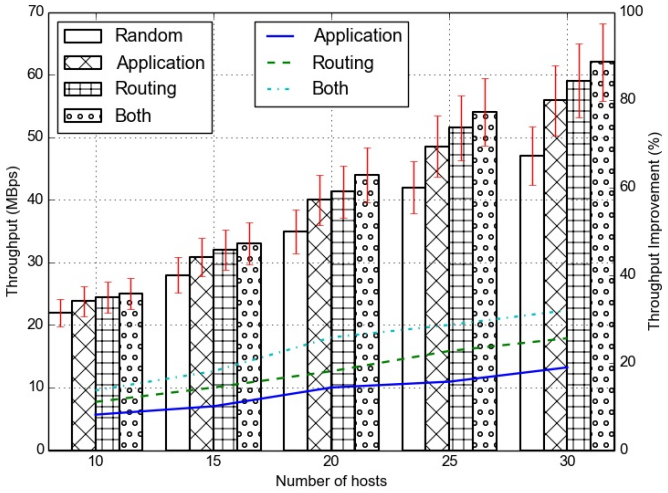

Fig. 7: Cross-layer Approach's Throughput and Improvement in Jellyfish topology for Hadoop. Our SA approach improves throughput of vanilla Hadoop by up to 31.9%.

("Both" bars and lines) under the Fat-tree and Jellyfish topologies respectively (same meanings as Section IV-A).

In Fat-tree (Figure 6), at 10 hosts, without any of our scheduling, the average job throughput is 20 MBps. When we turn on the routing and application level SA, the throughput increases by 7.5% and 5% respectively. With cross-layer scheduling, the throughput improvement is 9%. At 30 hosts, the cross-layer's throughput improvement grows to 26%.

Similar performance benefits are also seen in jobs running under the Jellyfish topology (Figure 7) 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 the scheduler running at both levels, the throughput of the jobs rises by 31.9%.

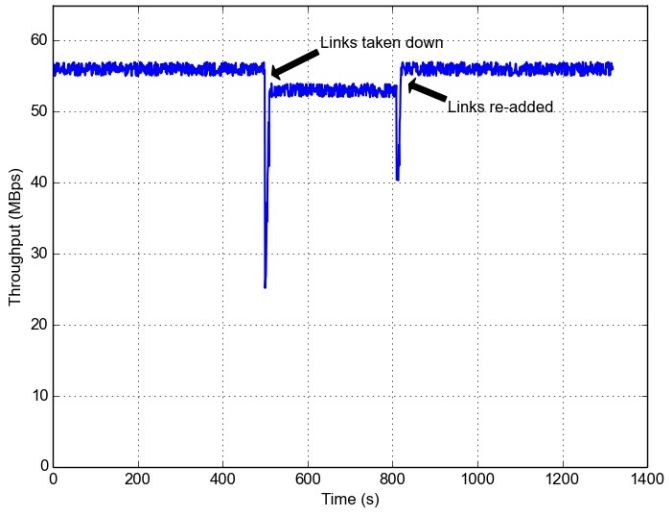

Fig. 8: Fault-tolerance of SA in Hadoop: Average task throughput intervals during one run of 5 WordCount programs running in a Jellyfish cluster. 10% of links are failed at 500 s. Links brought back up at 810 s.

Similar to the Storm results of Section IV-A, 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. Further, the throughput improvement also scales with the number of hosts and routers. 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 of Section II play a smaller part. If MapReduce jobs are chained (e.g., in Hive [49] or Pig [38]), the throughput improvement would approach that with Storm.

Fault-tolerance: We run 5 WordCount MapReduce jobs, each with 4 mappers and 4 reducers, in a Jellyfish cluster of 30 servers and 30 routers. In Figure 8, at 500 s we fail 10% of the utilized links. Upon failure, our scheduler reroutes affected paths within 0.35 s, returning throughput to within 4.7% of the pre-failure value. After the links recover at 810 s, the throughput is within 1.5% of the pre-failure throughput.

C. Simulations

In order to evaluate scalability, we perform simulation experiments run on a single server with a 4-core 2.50 GHz processor, with each link's bandwidth at 100 MBps. Each router has 15 links.

Scheduling Overhead: It is imperative for real-time jobs to be scheduled quickly. Figures 9 and 10 show that, for both Fat-tree and Jellyfish networks, the time to schedule a new job is small (under a second for 1000 servers), and it scales linearly with cluster size. The linear scaling is expected because of the linear increase in state space at the routing level.

Figure 9 shows for Storm-like dataflow graphs, scheduling decisions take 0.28 s in a 200-host Jellyfish topology. At 1000 hosts, scheduling decisions take 0.74 s on average. In Fat-tree,

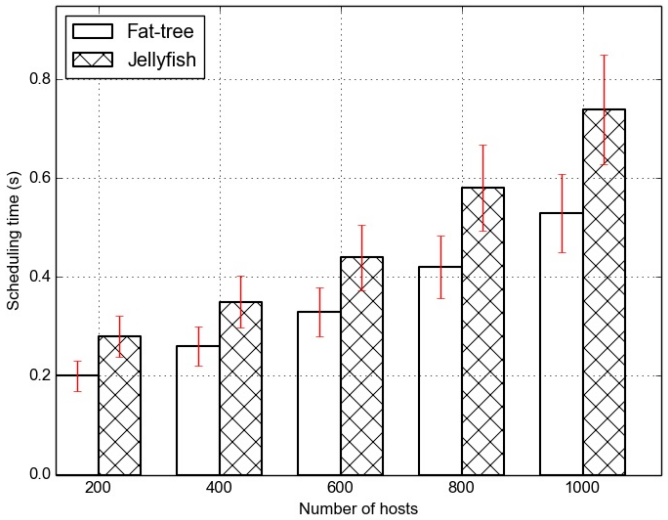

Fig. 9: Scheduling time for a new job for stream-processing dataflow graphs. At 1000 nodes, it takes only 0.53 s and 0.74 s per scheduling decision in Fat-tree and Jellyfish, respectively.

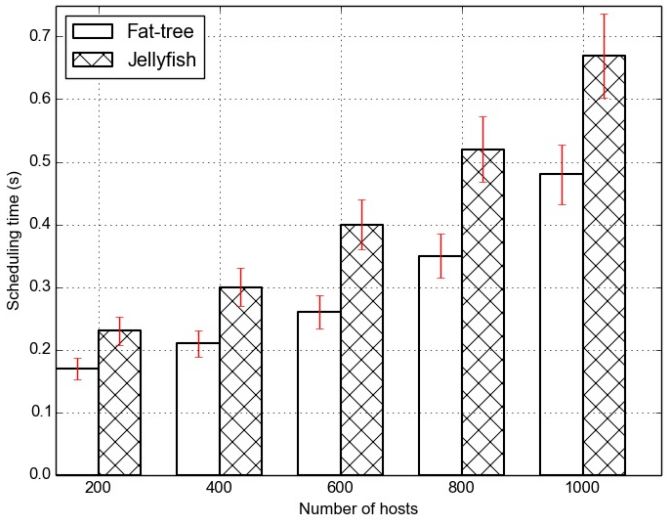

Fig. 10: Scheduling time for a new job for MapReduce/Hadoop dataflow graphs. At 1000 nodes, it takes only 0.48 s and 0.67 s per scheduling decision in Fat-tree and Jellyfish respectively.

scheduling decisions take 0.20 s on average. At 1000 hosts, the scheduling time only grows to 0.53 s. For Hadoop/Mapreduce-like dataflow graphs, Figure 10 shows that at 1000 hosts, scheduling can be completed in 0.48 s.

Overall, scheduling time is slightly higher in Jellyfish than in Fat-tree because it offers more routes for the SA exploration. We conclude that our cross-layer scheduling algorithm scales well with a large number of hosts in the cluster, and makes sub-second scheduling decisions with 1000 hosts. Furthermore, we note that jobs can be batch-scheduled if arrival rates become too high and that simulated annealing is agnostic to the number of jobs to be scheduled because it takes as input only the number of hosts and communication patterns between hosts.

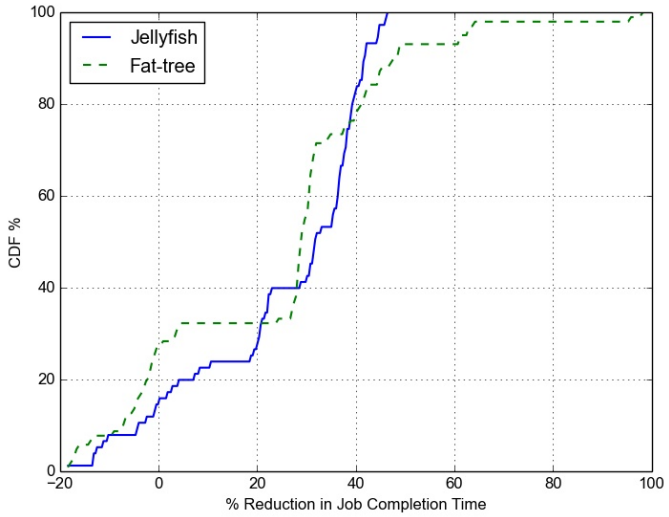

Fig. 11: CDF of job completion time improvements in Hadoop at 1000 hosts under cross-layer scheduling framework.

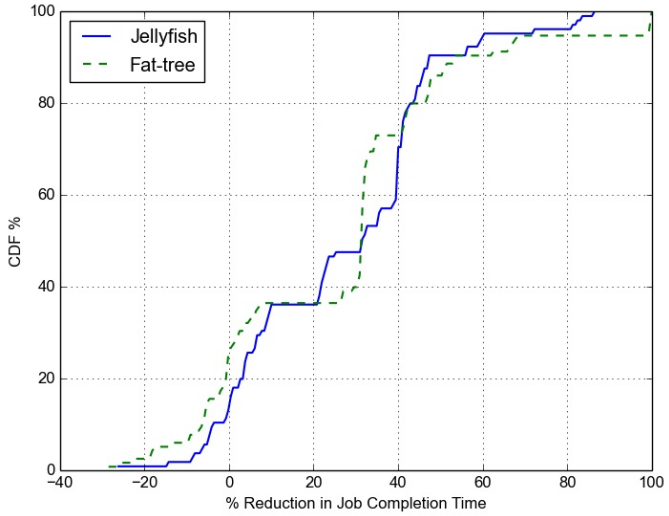

Fig. 12: CDF of job completion time improvements in Storm at 1000 hosts under cross-layer scheduling framework.

Effect on Job Completion Times: Figures 11 and 12 show CDF plots of job completion time improvements, compared to a scheduler that picks random routes. Cross-layer scheduling improves Hadoop by 34% (38%) at the 50^{th} (75^{th}) percentile, and Storm by 34% (41%) at the 50^{th} (75^{th}) percentile.

About 27% of Hadoop jobs and 24% of Storm jobs suffer degradation in completion time in Fat-tree (18% and 16% respectively in Jellyfish). However, our approaches do not cause starvation. The worst-case degradation is under 20% for Hadoop and 30% for Storm. By investigating our logs, we found that these are either larger jobs, or jobs submitted immediately after larger jobs. In essence, our heuristics schedule smaller jobs along network paths that have fewer hops. Thus, when a larger job arrives, its many flows are placed along longer routes. This slows it down, and affects future jobs until the large job is done. However, such large jobs would be rare

in real-time analytics workloads.

We conclude that our cross-layer scheduling algorithm 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).

Finally, we clarify that we do not show plots varying the number of jobs since scheduling time does not depend on the number of running jobs; state spaces at both layers depend only on the network and the new job. We also do not show job size variation because: i) for Hadoop real-time analytics, jobs are small, thus there is limited variability in job size; ii) for Storm, our experiments already saturate system throughput.

V. Related Work

Computation Scheduling: There has been significant work in improving scheduling of Mapreduce [14] and Hadoop [6]. Some systems improve job scheduling by preserving data locality [7], [27], [28], [55]. Others address fairness across multiple tenants for CPU and memory resources [17], [30], or heterogeneous tasks [44]. Systems like Jockey [15] compute job profiles using a simulator, and use the resulting dependency information to dynamically adjust resource allocation. Systems like Mantri [8] 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 [39] proposes decentralized scheduling that is within 12% of an ideal scheduler's performance, by using the power of two choices. We believe this idea can be applied to our work to design decentralized cross-layer SA scheduling.

Many streaming frameworks have been proposed such as Storm [3], Spark [56], Naiad [36], and TimeStream [41]. None of these schedulers expose or use information about the underlying network topology. Further, our cross-layer techniques are amenable with any of these systems.

Routing-Level Scheduling: Orchestra [12] and Seawall [45] create TCP sockets/connections in a weighted-fair manner in order to make Mapreduce's shuffle phase efficient. Oktopus [10] and SecondNet [21] proposed static reservations throughout the network to implement bandwidth guarantees for the hose and pipe models respectively. Gatekeeper [43] proposes a per-VM hose model with work conservation, however its hypervisor-based mechanism works only for full bisection-bandwidth networks. FairCloud [40] introduces policies to achieve link-proportionality or congestion-proportionality. All of these systems are orthogonal to our work, and can be combined with our techniques.

Software-Defined Networking (SDN): Before SDNs, some systems provided custom routing protocols to applications [11], or allowed applications to embed code in network devices [48]. SDNs provide more fine-grained control of the network. Much of the existing work on SDNs is targeted at improving SDN scalability (e.g., enabling a hierarchy of policies to the SDN controller [42]), or enforcing correctness in the SDN (e.g., enabling consistent updates in SDN [42]), or enabling the SDN to work with various protocols [20], [32].

Cross-Layer Scheduling: The two-level optimization approach has been explored in some data processing frameworks, e.g., [52] proposed batching intervals of job submissions for scheduling. However, they did not consider concrete applications in their experiments, and address neither streaming frameworks nor what should happen in the case of failures. Flowcomb [13] improved Hadoop performance by sending application-level hints to the SDN centralized controller. This only exploits routing-level optimizations, but not cross-layer techniques.

Previously, cross-layer scheduling has been used in the HPC community (e.g., [47]) and in OSs (e.g., [18], [33]). Isolation is used widely to improve performance in virtualized clouds such as AWS EC2 [1] and Eucalyptus [37] and in generic schedulers like YARN [51] and Mesos [25]. Our cross-layer approach can be integrated into systems like Mesos.

VM Placement: In the virtualized setting, some systems migrate virtual servers by using Markov approximation to minimize VM migrations [29], [35]. However, combinatorial optimization for VM placement [23] may be expensive for short jobs that process real-time data.

VI. 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-processing system (Hadoop YARN) and 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 26-31% and for Storm by 30-34%. The throughput improvements are higher in network topologies with more route diversity, and our cross-layer scheduling approach allows improvements to be maintained in the case of link failure.

References

[1] Amazon EC2, http://aws.amazon.com/ec2.

[2] Apache Thrift, https://thrift.apache.org/.

[3] Storm: Distributed and fault-tolerant realtime computation, http://storm-project.net/.

[4] Zeromq: Code Connected, http://storm-project.net/.

[5] Statistical workload injector for mapreduce (SWIM), Aug. 2013, https://github.com/SWIMProjectUCB/SWIM/wiki.

[6] Apache Hadoop, Oct. 2013, http://www.hadoop.apache.org/.

[7] 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. 6th ACM Eurosys*, pages 287–300, 2011.

[8] 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. 7th Usenix OSDI*, volume 10, page 24, 2010.

[9] M. Appelman and M. de Boer. Performance analysis of OpenFlow hardware, 2012.

[10] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *ACM SIGCOMM CCR*, volume 41, pages 242–253, 2011.

[11] 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.

[12] 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.

[13] 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*, 2013.

[14] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *CACM*, 51(1):107–113, 2008.

[15] 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.

[16] R. W. Floyd. Algorithm 97: shortest path. *CACM*, 5(6):345, 1962.

[17] 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.

[18] P. Goyal, X. Guo, and H. M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proc. 2nd Usenix OSDI*, volume 96, pages 107–121, 1996.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] Y. Guo, A. L. Stolyar, and A. Walid. Shadow-routing based dynamic algorithms for virtual machine placement in a network cloud. In *Proc. 32nd IEEE Infocom*, pages 620–628, 2013.

[24] L. Gyarmati and T. A. Trinh. Scafida: a scale-free network inspired data center architecture. *ACM SIGCOMM CCR*, 40(5):4–12, 2010.

[25] 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.

[26] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proc. Usenix ATC*, volume 8, page 11, 2010.

[27] 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.

[28] 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.

[29] J. W. Jiang, T. Lan, S. Ha, M. Chen, and M. Chiang. Joint VM placement and routing for data center traffic engineering. In *Proc. 31st IEEE Infocom*, pages 2876–2880, 2012.

[30] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang. Multi-resource allocation: Fairness-efficiency tradeoffs in a unifying framework. In *Proc. 31st IEEE Infocom*, pages 1206–1214, 2012.

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

[32] 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.

[33] 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.

[34] 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.

[35] X. Meng, V. Pappas, and L. Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *Proc. 29th IEEE Infocom*, pages 1–9, 2010.

[36] 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.

[37] 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.

[38] 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.

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] A. Shieh, S. Kandula, A. Greenberg, and C. Kim. Seawall: performance isolation for cloud datacenter networks. In *Proc. 2nd Usenix HotCloud*, page 1, 2010.

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

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

[48] 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.

[49] 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.

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

[51] 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.

[52] 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.

[53] 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.

[54] 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. 8th Usenix OSDI*, volume 8, pages 1–14, 2008.

[55] 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.

[56] 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.