CAM: A Topology Aware Minimum Cost Flow Based Resource Manager for MapReduce Applications in the Cloud

Min Li†, Dinesh Subhraveti‡, Ali R. Butt†, Aleksandr Khasymski†, Prasenjit Sarkar‡
† Dept. of Computer Science, Virginia Tech; ‡ IBM Almaden Research Center
{limin,butta,khasymskia}@cs.vt.edu, dineshs@us.ibm.com,
psarkar@almaden.ibm.com

ABSTRACT

MapReduce has emerged as a prevailing distributed computation paradigm for enterprise and large-scale data-intensive computing. The model is also increasingly used in the massively-parallel cloud environment, where MapReduce jobs are run on a set of virtual machines (VMs) on pay-as-needed basis. However, MapReduce jobs suffer from performance degradation when running in the cloud due to inefficient resource allocation. In particular, the MapReduce model is designed for and leverages information from the native clusters to operate efficiently, whereas the cloud presents a virtual cluster topology overlying or hiding actual network information. This results in two *placement anomalies*: loss of *data locality* and loss of *job locality*, where jobs are placed physically away from their data or other associated jobs, adversely affecting their performance.

In this paper we propose, CAM, a cloud platform that provides an innovative resource scheduler particularly designed for hosting MapReduce applications in the cloud. CAM reconciles both data and VM resource allocation with a variety of competing constraints, such as storage utilization, changing CPU load and network link capacities. CAM uses a flow-network-based algorithm that is able to optimize MapReduce performance under the specified constraints — not only by initial placement, but by readjusting through VM and data migration as well. Additionally, our platform exposes, otherwise hidden, lower-level topology information to the MapReduce job scheduler so that it makes optimal task assignments. Evaluation of CAM using both micro-benchmarks and simulations on a 23 VM cluster shows that compared to a state-of-the-art resource allocator, our system reduces network traffic and average MapReduce job execution time by a factor of 3 and 8.6, respectively.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems — *distributed applications*; G.2.2 [**Discrete Mathematics**]: Graph Theory — *graph algorithms, network problems*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
HPDC'12, June 18–22, 2012, Delft, The Netherlands.
Copyright 2012 ACM 978-1-4503-0805/12/06 ...$10.00.

General Terms

Design, Algorithms, Performance, Evaluation.

Keywords

MapReduce, Cloud computing, Min-cost flow network, VM placement, VM resource management

1. INTRODUCTION

MapReduce is an established framework for processing large-scale data-intensive applications. The MapReduce model helps businesses to process massive quantities of data in reasonable time and extract valuable insights hidden within by distributing the job across a large number of cost-effective cluster nodes. In particular, for many applications, such as converting archived media into a streaming format for Internet delivery, the processing is needed only once, and hence the resources required for processing are also needed only for a specific duration.

Combining the MapReduce framework with the cloud provides a number of unique advantages. It is particularly appealing for organizations that need to analyze large amounts of data without having to acquire and manage large cluster resources. The user does not need to own the cluster resources required to run the job, which removes the entry barrier [6], enabling even small businesses to perform detailed analysis on their data. An organization can focus on its core business rather than being occupied by lower-level cluster maintenance. The cloud provides the flexibility of dedicating as many or as few virtual machines (VMs) and storage resources as needed based on the required turnaround time [1, 2, 8]. The user only pays for the resources for the duration of time they are used.

While cloud offers great promise, the storage infrastructure of existing cloud environments is poorly suited for MapReduce computation. Clouds are typically built on commodity clusters with node-local disks for their cost-effectiveness and scalability. Several issues impact the turnaround time of MapReduce jobs running in these cloud environments.

First, *running MapReduce jobs in the cloud has an expensive ingestion phase*, where the dataset needs to be copied from a central persistent store into the compute cluster for processing. For large datasets, ingestion represents a significant portion of the turnaround time. Moreover, clouds feature a stateless model and any data and VMs copied to the physical/hypervisor cluster are discarded once the job is completed. Subsequent jobs require transferring the data again. Alternatively, it may be possible to have the MapReduce tasks access the data directly from the remote store via suitable remote data access protocols such as NFS, iSCSI [14], or FibreChannel [19]. Such remote access has several disadvantages. For one,

all data would have to be accessed over the network. MapReduce model achieves its efficiency by ensuring that tasks can access their data locally. Thus, fetching data over the network severely affects job performance. Furthermore, the bisectional bandwidth between the compute cluster and the central store can easily become a bottleneck. A large number of tasks, all accessing their data from the central store, can quickly saturate the link and render the system inefficient.

A further alternative is to co-locate the data with the compute cluster. However, spreading out the data across the local disks of cluster nodes constrains the scheduling choices available for placing VMs. VMs accessing the data located on a particular node must all be placed on that node, but other constraints such as amount of memory or licenses may not permit this. Providing reliability of persistent data located on the hypervisor cluster is also a challenge. Data stored on a centralized storage device is typically protected from disk failures through internal replication. Providing a similar replication facility across disjoint local file systems storing the data is difficult. A cluster file system may be used to combine the local storage attached to individual nodes in the cluster, but most existing cluster file systems are designed for a central storage model in a storage area network (SAN), and perform poorly on local storage [4]. For example, cluster file systems typically stripe files across all available disks in the cluster to maximize throughput, whereas such a strategy limits the performance in commodity clusters where network is the bottleneck.

Similar issues also apply to the VM images that compose a MapReduce job. The virtual image files need to be copied to the hypervisor nodes before starting a job, which introduces a very high startup latency. As before, running directly from the remote storage is not a scalable solution and co-locating the images with the cluster limits scheduling choices.

Second, *the cloud masks the physical topology of the underlying infrastructure*, which can potentially inhibit optimal scheduling of MapReduce tasks. The MapReduce model is designed for and leverages information from the native clusters to operate efficiently, whereas the cloud presents a virtual cluster topology. For instance, the VMs associated with a job may be placed across multiple racks. However, this information is not typically visible to the application. Furthermore, the cloud may also change the initial assignment by migrating the VMs to different nodes in the cluster based on runtime load and other constraints. While these functions add flexibility, they also make application-level scheduling challenging. This results in two *placement anomalies*: (1) Loss of data locality, where a task may be placed away from the physical location of its data; and (2) Loss of job locality, where a task may be placed away from the physical locations of other tasks that it communicates with. Map-intensive jobs are adversely affected by loss of data locality and reduce-intensive jobs are impacted by loss of job locality.

Third, *the multi-tenant cloud environment may result in interference between MapReduce applications* and other applications sharing the environment. Scheduling decisions made at the beginning of the job may become invalid during the course of the job, when VMs are migrated around or due to changing workloads. An optimal allocation of resources might become suboptimal leading to poor performance.

In order to efficiently address aforementioned issues, we present CAM, a platform that is designed to host MapReduce applications in the cloud. CAM provides a cluster file system that supports a uniform file system name-space across the cluster by integrating the discrete local storage of the individual nodes. The shared file system enables a VM to be placed on any cluster node or subsequently migrated as necessary. We leverage GPFS [10] in CAM to query and specify the physical locations of an image and its replicas, which can then be used for CAM-directed placement of VMs and data.

CAM avoids the placement anomalies with an innovative resource scheduler for the cloud, especially designed for improving the performance of MapReduce jobs. Specifically, this paper makes the following contributions:

- CAM adopts a three level approach to maximize locality. (1) **Data placement:** Data is placed within the cluster based on offline profiling of the jobs that most commonly run on the data. Rather than accommodating an arbitrary data placement, strategically placing the data can significantly improve locality. (2) **VM/job placement:** For a given job, CAM selects the best possible physical nodes to place the set of VMs that represent the job. (3) **Task placement:** In order to further minimize the possibility of a placement anomaly, CAM exposes, otherwise hidden, *compute, storage, and network* topologies to the MapReduce job scheduler such that it makes optimal task assignments. This is crucial as, for example, what appears to be a directly attached local disk within a VM could in fact be physically located on a different node.

- CAM reconciles resource allocation with a variety of other competing constraints such as storage utilization, changing CPU load and network link capacities using a flow-network-based algorithm that is able to simultaneously satisfy the specified constraints. Each placement decision not only considers the existing data and VM assignments in the cluster, but also evaluates the cost of readjusting existing assignments in response to data movement and VM migration to derive the best net configuration possible.

- We evaluate CAM using both micro-benchmarks and simulations on a 23 VM cluster. We show that compared to a state-of-the-art resource allocator, our system reduces network traffic by up to 3 times, and achieves $8.6x$ speedup of MapReduce jobs on average.

The rest of the paper is organized as follows. Sections 2 and 3 provide an overview of CAM's architecture and usage model. Section 4 details the design of data and VM placement techniques used in CAM. Section 5 presents our experimental results. Finally, we summarize related work in Section 6 and conclude in Section 7.

2. ARCHITECTURE

CAM is designed as an extension to IBM ISAAC product [12]. ISAAC implements key cloud functions such as creating and deleting VMs and their persistent volumes, placing the VMs based on load and capacity, maintaining availability of cloud services through clustering and fail-over mechanisms. The architectural components of CAM are implemented as extensions to related counterparts in ISAAC. In particular, we have integrated ISAAC with the GPFS-SNC [10] file system to provide a suitable cluster file system needed by CAM, and have extended ISAAC to support data and VM placement based on techniques we describe in Section 4. Figure 1 illustrates the components of CAM and their interactions when deployed in a cloud environment. The physical resources supporting the cloud consists of a cluster of hypervisor (physical) nodes with local storage directly attached to the individual nodes.

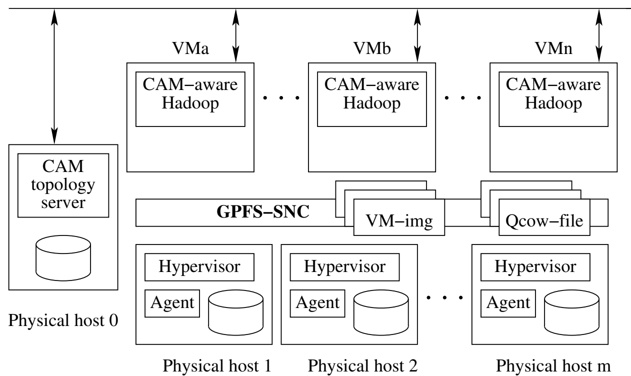

Figure 1: CAM architecture components.

2.1 GPFS-SNC Storage Layer

CAM uses GPFS-SNC [10] to provide its storage layer. GPFS-SNC is designed as a cloud storage platform, which supports timely and resource-efficient deployment of VMs. GPFS-SNC manages the local disks directly attached to a cluster of commodity physical machines. More specifically, it has a number of unique features that make it a cloud-friendly storage system. First, GPFS-SNC supports co-locating all blocks of a file at one location, rather than stripping the file across machines. This enables a VM I/O request to be serviced locally from the stored location instead of remotely from physical hosts across the network. CAM leverages this feature to ensure that co-located VM images are stored at one location and can be accessed efficiently. Second, GPFS-SNC supports an efficient block-level pipelined replication scheme, which guarantees fast distributed recovery and high I/O throughput through fast parallel reads. This feature is useful for CAM for achieving efficient failure recovery. Finally, GPFS-SNC specifies a user-level API that can be used to query the physical location of files. CAM uses this API to determine actual block location, and uses this information to infer storage closeness for data and VM placement.

2.2 Topology Awareness

MapReduce task scheduler uses the topology information of the cluster nodes to decide task assignments. The information is supplied by the user as a part of the job configuration file when the job is submitted. However, in an attempt to abstract hardware level details and present a simple interface to the user, existing cloud implementations do not expose the information about the topology of the cluster or the actual placement of VMs to the MapReduce scheduler [2]. Furthermore, the initial configuration provided by the user may become stale when the VMs are moved later.

CAM addresses these issues with three main components that together provide topology awareness as shown in Figure 1. First, the *CAM topology server* provides the additional topology information required to enable the MapReduce scheduler to place the tasks optimally. The topology server is an integral component of the ISAAC cloud service infrastructure and provides a REST interface, which the scheduler invokes. The information exposed by the topology server consists of network and storage topologies, and other dynamic node-level information such as CPU load. Second, a set of agents running on the physical nodes of the cluster periodically collect and convey to the topology server, a variety of pieces of data about the respective node, such as utilization of outbound/inbound network bandwidth, IO utilization and CPU/memory/storage load. The topology server consolidates the dynamic information it receives from the agents and serves it along with topology information about each job running in the cluster. The topology information is derived from existing VM placement configuration. Third, a new MapReduce task scheduler interfaces with the topology server to obtain accurate and current topology information. The scheduler readjusts task placement accordingly whenever a change in the configuration is observed. Note that CAM needs to provide a different scheduler because the standard MapReduce scheduler is designed to make the placement decisions only based on a static configuration file and only at the beginning of the job [9]. While the MapReduce task scheduler is modified to leverage the topology and physical host resource utilization information in CAM, the MapReduce applications can run without any modification.

The **network topology** information is represented by a distance matrix that encodes the distance between each pair of VMs as cross-rack, cross-node, or cross-VM. Current MapReduce task schedulers consider rack and node localities but lack the notion of VM locality. When two VMs are placed on the same node, they are connected through a virtual network connection implemented as a part of the hypervisor. By virtue of the fact that the VMs share the same node hardware, the virtual network provides a high-speed medium that is significantly faster than the inter-node or inter-rack links. The network traffic between the VMs on the same node does not have to pass through the external hardware link. The network virtual device simply forwards the traffic in-memory through highly optimized ring buffers. CAM extends the MapReduce scheduler to consider this fine-grain locality information to make optimal placement choices for the tasks.

The **storage topology** information is provided as a mapping between each virtual device containing the dataset and the VM to which the device is local. In the native hardware context, a SATA disk attached to a node can be directly accessed through the PCI bus. In the cloud, however, the physical blocks belonging to a VM image attached to a VM could be located on a different node. Even though a virtual device might appear to be directly connected to the VM, the image file backing the device could be across the network, and potentially closer to another VM in the cluster than the one it is directly attached to in the virtual setup. The topology server queries the physical image location through the GPFS API and presents the information to the MapReduce scheduler.

The specific APIs provided by the topology server is described in Table 1. $get_VM_distance$, provides MapReduce task scheduler with hints of the network distance between two VMs. The distance is estimated based on observed data transfer rates between the VMs, and is expressed in units of bandwidth. $get_block_location$, enables MapReduce to get the actual block location instead of the location of a VM, thus guaranteeing data locality. The rest of the calls are used to facilitate the MapReduce task scheduler to query the I/O and CPU contention information related to network and disk utilization. The MapReduce task scheduler can leverage this additional information to make smarter decisions, such as placing I/O intensive tasks on physical hosts that have idle I/O resources.

3. CAM USAGE MODEL

CAM is a cloud platform with specific interfaces and support for running MapReduce jobs. The dataset to be processed is initially placed on GPFS. This is in contrast to most cloud models, which segregate storage and compute resources, and require the dataset to be moved from the storage cloud to the compute cloud for processing. Co-locating storage and compute clusters avoids the expensive ingestion phase for each job run. Data placed in this manner can be used by each subsequent job in CAM.

The placement of data is driven by the nature of MapReduce jobs that are typically run on the data. For instance, if the dataset is

API	Description
`int get_VM_distance(string vm1, string vm2)`	Returns the distance between two VMs.
`struct block_location get_block_location(string src, long offset, long length)`	Returns the actual location of blocks.
`int get_vm_networkinfo(string VM, struct networkinfo)`	Returns the network utilization information of physical host on which the VM is running.
`int get_vm_diskinfo(string vm, string device, struct diskinfo)`	Returns disk utilization information.
`int get_VM_cpuinfo(string vm, struct cpuinfo)`	Returns CPU utilization information of the physcial host on which the VM is running.

Table 1: The key APIs provided by CAM to the MapReduce scheduler.

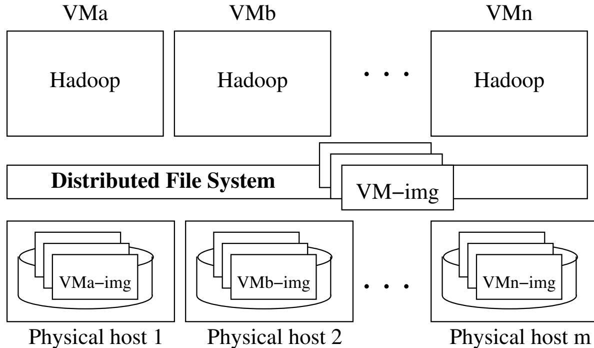

Figure 2: Setup of CAM for supporting MapReduce in the cloud.

primarily used as input for various pattern search jobs, data locality is likely to be more important than task locality. The user can specify the nature of expected workloads, or the workload characteristics can be automatically derived based on previously observed I/O patterns.

The user submits a MapReduce job by providing the application, e.g., relevant java class files, indicating a previously uploaded dataset corresponding to the job, and the number and type of VMs to be used for the job. Each VM typically supports several MapReduce task slots depending on the number of virtual CPUs and virtual RAM allocated to the VM. The more the number of VMs assigned to a job, the quicker the job finishes.

CAM determines an optimal placement for the set of new VMs requested by the user by considering a variety of factors such as current workload distribution among the cluster nodes, distribution of the input dataset required by the job, and the physical locations of the required master VM images. The images required to boot the VMs on the selected nodes are created from the respective master images using a copy-on-write mechanism provided by GPFS, which allows fast provisioning of a VM image instance without requiring a data copy of the master image. The job class files are copied into the cloned VM image by mounting the image as a loopback file system. These changes are private to the cloned image. Next, the data images are attached to the VMs and the respective device files are mounted within the VM for the MapReduce tasks to access the data contained within them.

Figure 2 illustrates the setup. Each machine is equipped with local disks. There is a distributed file system installed on top of these physical machines. The VM image files are stored in the distributed file system. Moreover, there is a cloud manager that allocates the resources for MapReduce jobs, and manages the data placement and VM placement.

4. MIN-COST FLOW BASED PLACEMENT

In this Section, we present how CAM manages Data and VM placement using a min-cost flow based approach. In our model, we assume that it is possible for the cloud provider to profile a job and estimate its characteristics such as job type (Map-Reduce intensive, Map intensive, or Reduce intensive), and input, output and intermediate data sizes. For our current implementation, we rely on user-provided or predetermined job descriptions to identify a job's type. However, the system can be easily extended to determine the amount of time an application spends in different phases (Map, Reduce), and use this information to determine a job's type. For example, a job that spends more than 30% of the time in Map can be considered as Map-intensive.

4.1 Data Placement

We express the problem of optimally placing data in a given cloud cluster architecture as an instance of the well-known min-cost flow problem [13]. To achieve this, we break down the placement problem into three sub-problems, namely guaranteeing VM closeness, avoiding hotspots, and balancing physical storage utilization according to different job types. We capture the three constraints via similarly named factors in our model. *VM closeness* expresses how close data should be placed to VMs so that the network traffic between the corresponding VMs is minimized. *Hotspot factor* expresses the expected load on a machine, and identifies machines that do not have enough computational resource to support the VM(s) assigned to them. To avoid a hotspot, data needs to be placed on the least-loaded machine. This can be determined by measuring the current computational resource load of the machine and adding it to the expected computational requirements of the VMs that will work with the data to be placed on the machine. *Storage utilization* expresses the percentage of total physical machine storage space that is in use.

Job Type	VM closeness	Hotspot factor	Storage utilization
MR-intensive	Yes	Yes	Yes
M-intensive	No	Yes	Yes
R-intensive	No	No	Yes

Table 2: Significance of considered cost factors for different job types.

Table 2 shows the significance of the three factors on the performance of different MapReduce workloads. For workloads that are both Map and Reduce intensive, related data should be placed close together and on the least loaded machine. For Map intensive workloads, the data should be placed on the least loaded machine,

but does not necessarily need to be placed close together due to the light shuffle traffic in such workloads. For Reduce intensive workloads, the only concern is the storage utilization of the machine on which the VM is to be placed. For all types of workloads, it is desirable to place data evenly across racks to minimize the need to rearrange data over time for supporting migrating VMs.

We use these factors in constructing a min-cost flow graph that encodes the factors. Then we employ an extended solver to minimize the global cost of the graph, thus solving the original problem of determining how data should be placed in the virtualized cloud.

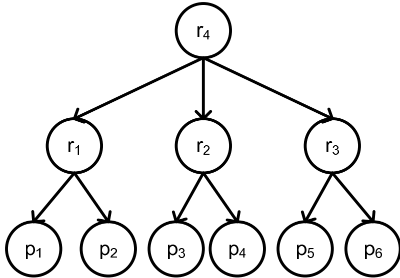

Figure 3: Sample network topology for data placement.

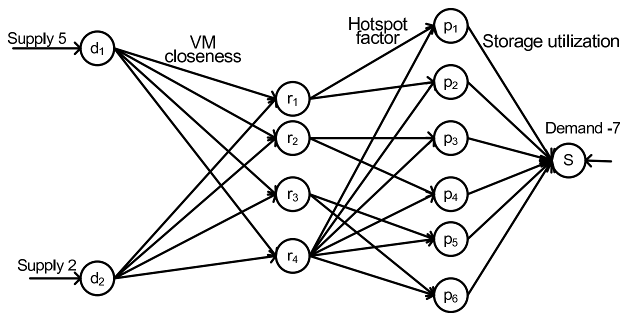

Figure 4: Flow graph for sample data placement.

Figure 3 shows a sample network topology, which consists of six physical nodes ($p_1, \ldots, p_6$) organized into three racks (r_1, r_2, r_3) with one master rack/switch (r_4) connecting the racks. Note that our model can support any topology where the network traffic can be estimated. There are several challenges when min-cost flow is used in our problem space. First, the three factors described above have to be encoded into the graph. Second, the correlation between different VMs images placement has to be encoded (which is shown to be non-trivial [13]). The flow-network model is aimed at minimizing the flow cost, however, we employ the model to also consider VM closeness as an objective, which requires it to solve correlated constraints, i.e., a set of VMs would have to be placed together, but it does not matter where. Third, the three factors capture different costs that are not directly comparable to each other. For instance, *VM closeness* of 1 may signify the cost of copying 1 GB of data within a local rack, where as *Hotspot factor* of 1 may signify the cost of using a physical machine that has 1% more load than the least-loaded machine in the cluster. The two costs are clearly not the same. Thus, we need a way to formulate the three factors in the same units for encoding them into a min-flow graph.

In contrast to the extant data placement techniques that work at the granularity of the data blocks, our unit of data placement is a VM image. Such coarse placement is justified in CAM as the goal is to ensure that an entire image is available at one location. Moreover, our underlying storage layer of GPFS-SNC avoids striping the data across different physical machines, thus making block-level placement unnecessary.

We address these challenges as follows. Consider the corresponding min-cost flow graph for Figure 3 as shown in Figure 4. Here, two data items d_1 and d_2 with requests for 5 and 2 VM images, respectively, are submitted to the cloud. The number of VM images requested by a data item is denoted as the data item's *supply* for our flow graph. Conversely, we add a sink node S to the graph, that can "support" the VMs. The number of VMs that a sink node can handle is assigned as a *demand* value. In our example, S has a demand -7 and is the only place that can receive all the flows. Each flow graph edge has two parameters attached to it, the capacity of the edge and the cost for a flow to go through the edge. The data nodes, represented by d_1 and d_2 in the graph, have outgoing links to each rack with *VM closeness* as costs. The *Hotspot factor* is encoded in the links from the racks to each physical node p within its range. Note that even though r_4 serves as a switch between the racks, it is shown in the graph as directly connected to all the physical nodes. This is to ensure that the least-loaded machine can be chosen for Map-intensive jobs without being constrained by the network topology. All the physical nodes, $p_1, \ldots, p_6$, are linked to the sink node with *Storage utilization* as link costs. Note that there is no direct link from data item node d_j to the associated physical host p_i. This is to support scaling up the system, as otherwise the number of links in the graph will increase with increasing number of data items and physical nodes (much faster than the number of racks). Consequently, making it inefficient to solve for min-flow on the graph.

	Data set d_j	Rack r_k	Physical host p_i	Sink S
Supply	$\sum(N_{d_j})$	0	0	-$\sum(N_{d_j})$
Incoming link from	N/A	N_{d_j}	Rack	Physical host
Outgoing link to (cap., cost)	Rack (N_{d_j}, α_{jk})	Physical host (Cap_i, β_i)	Sink (Cap_i, γ_i)	N/A

Table 3: Values assigned to the flow graph for data placement used in CAM.

Table 3 provides the details of how we encode the various factors and system information in our min-cost flow graph. N_{d_j} is the number of VM images requested by dataset d_j. α_{jk} captures *VM closeness*. The cost, α_{jk}, of outgoing link from the dataset d_j to physical host p_i on which the data is placed on rack r_k is estimated conservatively by the traffic in the shuffle phase as follows:

$$\alpha_{jk} = size_{intermediate} * \frac{num_{Reducer} - 1}{num_{Reducer}} * distance_{max}, \quad (1)$$

where $distance_{max}$ is the maximum network distance between any two nodes in the rack r_k, and $size_{intermediate}$ and $num_{Reducer}$ are the total size of data output by the Map phase and the number of reducers, respectively, of the MapReduce job running on data set d_j . Note that given its higher $distance_{max}$ a higher level rack-/switch, e.g., r_4 in our example, would have a higher α than the lower racks, e.g., r_1, r_2 and r_3, based on this formula. The *Hotspot factor* is captured using β_i for physical node p_i, and is estimated by the current and expected load as follows:

$$\beta_i = a * (load_{exp} + load_{curr} - load_{min}), \quad (2)$$

where $load_{curr}$ and $load_{min}$ represent the current load and minimum current load, respectively. a is a parameter that acts as a knob to tune the weight of the *Hotspot factor* with respect to other costs. Moreover, based on guidelines from [21] the expected load is determined as $load_{exp} = \sum_j (\rho_j/(1-\rho_j) * CRes(d_j)$, where $\rho_j = \lambda_j/\mu_j$. Here, λ_j represents the number of d_j's associated jobs that arrive within a give time interval, μ_j represents the mean time for each VM to process a block, and $CRes(d_j)$ represents the compute resources required by jobs running on data set d_j. *Storage utilization* of a physical node p_i is captured by γ_i, which is determined by the current storage utilization compared with minimum storage utilization of all p_is.

$$\gamma_i = b * (storageUtil_{p_i} - storageUtil_{min}) \quad (3)$$

Here, b is another parameter used to fine tune the weight of *Storage utilization* with respect to the other two factors. Finally, $Cap_i = freespace_{p_i}/size_{VMImg}$, is a conservative estimation of the capacity of each physical host calculated as the ratio of the available storage capacity of p_i and the size of the VM image. We assume that all VM images have the same size (10 GB in our experiments) when initially uploaded to the cloud.

To enable the graph to capture the correlation between VM image placement for one data request, we extend the solver to take into account an additional parameter for each edge, *split factor*, which specifies whether flows from a node are allowed to be split across different links, and is either *true* or *false*. In our example, *split factor* for all the links from d_1 and d_2 are set to *false*. This implies that all the flows from data nodes will wholly go through one of the $r_1, \ldots, r_4$, but will not be split between the racks.

Once a new data upload request comes in, the cloud server updates the graph and computes a global optimal solution. The graph is updated as follows. First, the graph is cleaned of data and state from the previous iteration. This is done by deleting the data nodes that correspond to the datasets that have finished uploading, and their outgoing links from the graph. Next, the cost of the edges corresponding to the *Hotspot factor* and *Storage utilization* of the physical nodes where the data was stored need to be updated using equations 2 and 3. Then, a new data node d_j is created for the new data upload request with edges to each rack node r_k with costs calculated based on the above equation 1. Once the graph is updated, a new min-flow value is calculated, which is then used by the cloud scheduler. This process ensures that the cloud scheduler is timely provided with updated information to accommodate varying loads.

4.2 VM Placement

The goal of VM placement is to maximize the global data locality and job throughput. Our model considers both VM migration and delayed scheduling of a job as part of the optimal solution. Delaying a job is used to explore better data locality opportunities that can arise in the near future, while minimizing time wasted during the waiting. Migrating a VM belonging to a job enables our scheduler to make room for other suitable jobs or to explore better location opportunity. There are two assumptions that we make about how VMs are migrated. First, we assume that once the VMs for a job are allocated, the job will not be suspended or killed. There is no preemption, which guarantees that the job will have some quota of resources at all times during its life span. Second, even if some of the VMs belonging to a job get migrated, their total number remains the same. We model the VM placement as minimum cost flow problem, which has similar characteristics to the min-cost flow based data placement.

An example graph for VM placement is shown in Figure 5. Each job v_j is submitted to the system at the source node with the number of requested VMs, N_{v_j}, as the value of supply. The goal of the VM allocator is to either keep the job unscheduled (allocate 0) or allocate N_{v_j} VMs for each request. There is a single sink node, S, in the system with demand equal to minus the sum of the supply. The request from each job acts as a flow that goes either through the rack nodes, r_k, or through the unscheduled nodes, u_j, and finally to the sink. If a job is unscheduled, none of its VMs are allocated. Otherwise, the flow goes through the physical nodes, p_i. Each job v_j has a "preferred" node pr_j that has outgoing links to a set of physical hosts that would be preferable for v_j to be scheduled on. Based on the min-cost solution, a allocation scheme with min-cost can be easily derived. If the VMs are allocated to the highest level rack, it implies that the VMs can be allocated arbitrarily to any set of nodes in the VMs under the rack. Once v_j is scheduled, a "running" node (ru_j) is added to the graph to keep track of information about the execution of v_j, which is then used by the solver to direct migration decisions.

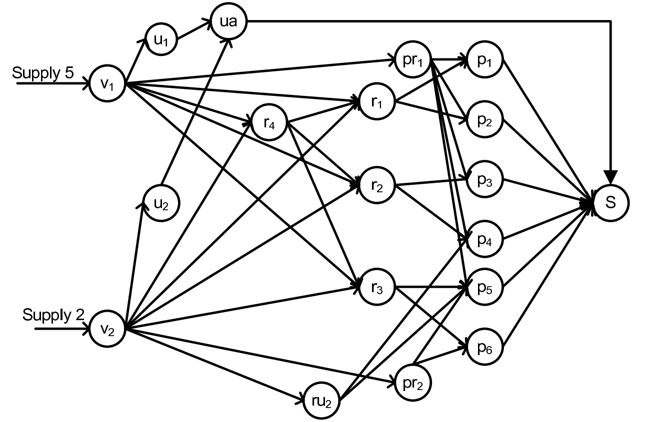

Figure 5: Flow graph for VM placement.

The job type information is modeled as the cost of the edge from each job to the rack nodes in our flow based graph. The higher level rack has higher cost than the lower lever rack in terms of reduce traffic. We use conservative approximation to compute bounds on data transfer costs. The cost to the highest level rack is estimated by worst case VM arrangement with regards to the map and reduce traffic. Similar rules apply to the lower level rack. The cost of the edges to the unscheduled nodes are set to be increased over time so that delayed jobs get allocated sooner than recently submitted jobs. This cost also controls when a job stops waiting for better locality, and thus offers a knob to tune the trade-off between data locality and latency. The aggregated unscheduled nodes control how many VMs can remain unscheduled, which is another system parameter to control the system resource utilization and data locality trade-off. The cost of the edges to running nodes set is increased over time and is job-progress aware. For example, a reduce intensive job run during the reduce phase might not be suitable for migration to make room for a contending job request.

Similarly as for data placement, we provide means for expressing the cost of reading data across different level of racks, migrating VMs, and delay scheduling in the same units. For example, we can choose that the copying of 1 GB data across rack local switch costs the same as copying 0.5 GB data across one level higher rack,

	Job node set	Preferred node set	Running node set	Unscheduled node set	Unscheduled aggregator node	Rack node set	Physical host node set	Sink
supply	$\sum(N_{v_j})$	0	0	0	0	0	0	$-\sum(N_{v_j})$
Incoming link from	N/a	job	job	job	all unschedule nodes	job; higher rack	rack; preferred nodes set; running nodes set	physical host; unscheduled aggregator
Outgoing link to (cap., cost)	Rack(N_{v_j}, ρ_j) Prefer(N_{v_j}, θ_j) Run(N_{v_j}, ϕ_j) U(N_{v_j}, ϵ_j)	Physical host $(d_i, 0)$	Physical host $(r_i, 0)$	Unscheduled aggregator $(N_{v_j}, 0)$	Sink $(N_{unsched}, 0)$	Physical host $(N_{r_k}, 0)$	Sink $(N_{vm}, 0)$	N/A
flow	N_{v_j}	$0/N_{v_j}$	$0/N_{v_j}$	$0/N_{v_j}$	$0/N_{unsched}$	$0, N_{r_k}$	$0, 1$	$\sum(N_{v_j})$

Table 4: Values assigned to the flow graph for VM placement used in CAM.

or the same as setting up of and starting one VM, or the same as delaying a VM execution by say 10 seconds.

We categorize the various nodes in the graph into different types as shown in Table 4.

- Preferred node set (pr_j): These graph nodes point to a set of physical nodes p_i that have a job v_j's associated dataset stored on them. An edge from a preferred node to p_i has the cost of 0 and the capacity of the number of VM disk images stored on p_i.
- Running node set (ru_j): These are dynamically added nodes that point to p_is that are currently hosting the v_j's VMs. An edge from ru_j to p_i has a cost of 0 and the capacity of the number of VMs running on p_i.
- Unscheduled node set (u_j): These nodes provide information about currently unscheduled jobs. u_j has an outgoing edge with capacity of N_{v_j} and cost 0 to a unscheduled aggregator.
- Unscheduled aggregator node (ua): The graph contains a single unscheduled aggregator. u has an outgoing edge with cost 0 to the sink with capacity of $N_{unsched} = \sum(N_{v_j}) - M + M_{idle}$, where M is the total number of VMs that the cluster can support and M_{idle} denotes the number of idle VM slots allowed in the cluster.
- Rack node set (r_k): The rack node r_k represents a rack in the topology of the cluster. It has outgoing links with cost 0 to its subracks or, if it is at the lowest level, to physical nodes. The links have capacity N_{r_k} that is the total number of VM slots that can be serviced by its underlying nodes.
- Physical host node set (p_i): Each physical host p_i has an outgoing link to the sink with capacity the number of VMs that can be accommodated on the physical host N_{vm} and cost 0.
- Sink S: The single sink node with demand $-\sum(N_{v_j})$.
- Job node set (v_j): This set represents each job node v_j with supply N_{V_j}. It has multiple outgoing edges corresponding to the potential VM allocation decisions for v_j. These edges are discussed in the following:
  - Rack node set r_k: An edge to r_k indicates that r_k can accommodate v_j. The cost of the edge is ρ_j that is calculated by the map and reduce traffic cost. If the capacity of the edge is greater than N_{v_j}, it implies that the VMs of v_j will be allocated on some p_is on the rack.
  - Preferred node set (pr_j): An edge from job v_j to the job wide preferred nodes set pr_j has capacity N_{v_j} and cost θ_j. The cost is estimated by only the reduce phase traffic, because in this case map traffic is assumed to be 0.
  - Running node set (ru_j): A link from job v_j has capacity of N_{v_j} and cost $\phi_j = c * T$, where T is the time the job has been executing on the set of machines and c is a constant used to adjust the cost relative to other costs.
  - Unscheduled node set (u_j): An edge to the job-wide unscheduled node u_j has capacity N_{v_j} and cost ϵ_j, which corresponds to the penalty of leaving job v_j unscheduled. $\epsilon_j = d * T$, where T is the time that job v_j is left unscheduled and d is a constant used to adjust the cost relative to other costs. The *split factor* for this link is marked as `true`, which means the allocation of all the VMs are either satisfied or be delayed until the next round.

When a VM allocation request is submitted, the flow graph is updated to calculate a new global optimal solution for the VM scheduler. Similar to the update process for data placement, the graph is cleaned by removing unnecessary nodes and edges. For example, for each finished job v_j, the associated nodes including the unscheduled node u_j, the preferred node pr_j and the running node ru_j are deleted from the graph since the job has released its VM resources. Then, the costs of edges related to the jobs that are still running are updated according to Table 4 to reflect the jobs' current state. Next, a set of new nodes and edges are added into the graph for the current VM allocation request, namely, a job node, a related unscheduled node, and a preferred node. Moreover, the corresponding edge costs are again calculated as described in Table 4.

Once the solver outputs a min-cost flow solution, the VM allocation assignment can be obtained from the graph by locating where the associated flow leads to for each VM request v_j. Flow to an unscheduled node indicates that the VM request is skipped for the current round. If the flow leads to a preferred nodes set, the VM request is scheduled on that set of nodes. Finally, if the flow goes to a rack node, it implies that the VMs from the job are assigned to arbitrary hosts in that rack.

The number of flows sent to a physical host through rack nodes or preferred nodes set is not higher than the number of available VMs of each physical hosts. This is guaranteed by the specified

link capacity from physical host to sink. Thus, all VM requests that are allocated will be matched to a corresponding physical host.

5. EVALUATION

In this section we show the effectiveness of our approach through a set of Hive [22] based, I/O-bound micro-benchmarks running on a real cluster. We evaluate CAM's network and storage topology awareness against vanilla Hadoop, as to our best knowledge, CAM is the first technique that reintroduces the concept of data locality by exposing topology information in a cloud setting. We also compare CAM's data and VM placement against a state-of-the-art technique using a mix of workloads on a large simulated cluster.

Number of jobs	21	9	7	4	3	3	3
Map tasks / job	1	2	10	50	100	200	400

Table 5: Distribution of job sizes in terms of number of map tasks used for micro-benchmark tests.

5.1 Micro-Benchmark Results

In this section, we use an I/O intensive workload based on a Hive benchmark to show the effectiveness of topology awareness and storage awareness for task placement. The reported numbers are averages across three runs of a test.

Our cluster consists of 4 RHEL 6.0 physical machines that use KVM as the hypervisor. Each machine has two quad-core 2.4 GHz Intel E5620 processors and 48 GB of main memory. The machines are organized in one rack and are connected to a dedicated Gigabit switch. We launched 23 VMs, 1 master and 22 slaves, on the four physical hosts. Each VM is configured with 1 GB of main memory and two 2.4 GHz vCPUs. Each VM has one map slot and one reduce slot, with a map block size of 64 MB. MapReduce fair scheduler is employed.

We generate a job submission schedule with 50 I/O-intensive Grep jobs using Poisson distribution with job inter-arrival time 10 seconds. To make the comparison consistent, we generate the submission schedule with a submission duration of 554 seconds, record it into a file and use it throughout the experiments. The size distribution of each Grep job for this experiment is shown in Table 5 and is based on the experiments performed by Zaharia et al. [24]. Thus, our schedule is representative of a typical workload of a production MapReduce cluster with a mix of many small jobs with a single map task per job, and a few large jobs with more than 100 map tasks per job. The input for the Grep jobs is generated using Teragen [5], with each map file consisting of 100 M records of size $0.1K$ for a total input size of 10 MB.

5.1.1 *Impact of Network Topology Awareness*

In our first experiment, we use our submission schedule to evaluate the impact of network topology awareness in a CAM-based implementation of Hadoop [5]. For this purpose, we measure the execution time for our schedule. We also measure the achieved locality expressed as the percentage of total Map tasks that are scheduled on the VMs (for VM locality) or physical nodes (for node locality) that have the associated data. As a base case for comparison we use vanilla Hadoop, which is unaware of the actual network topology and in this case cannot determine if two VMs are running on the same node or not.

The results are shown in Table 6. We observe that by exposing topology information to Hadoop, the node locality is improved by 6.4%, and the average job execution time reduces by 8%. Figure 6 shows a break-up of the node locality in terms of the number of map tasks for the two studied cases. Observe that network topology information effectively improves the node locality for jobs with 10 and 50 map tasks by 8% and 9%, respectively. Jobs with more than 50 map tasks see a decreasing improvement, because with the increased number of maps in the small cluster the chance of co-locating map tasks on the same node also increases. However, topology awareness is important even in such a small cluster as most MapReduce jobs have fewer than 50 map tasks [24]. Note that the relatively good performance of vanilla Hadoop is due to the fact that the test cluster consists of a small number of physical hosts located on the same rack.

System	VM locality	Node locality	Average execution time
Hadoop	29.1%	42.6%	48.3 s
CAM-based Hadoop	29.0%	49%	34.2 s

Table 6: Impact of network topology awareness on Hadoop performance.

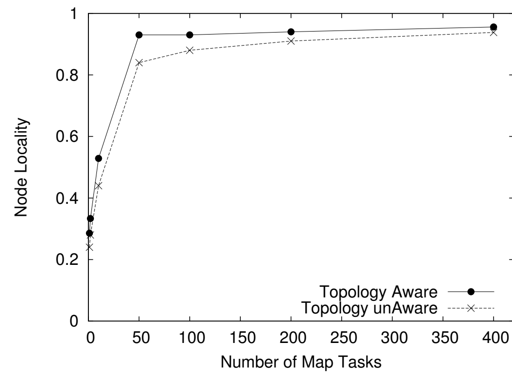

Figure 6: Breakdown of observed locality for jobs with different number of map tasks, with and without network topology awareness.

System	Average execution time
Hadoop	65.6 s
CAM-based Hadoop	48.3 s

Table 7: Impact of storage topology awareness on Hadoop performance.

5.1.2 *Impact of Storage Topology Awareness*

In our next experiment, we observe the impact of providing storage topology hints to Hadoop. For this test, we use the 22 VM slaves with local data, and then migrate 6 of the VM images from one physical host to another. This makes 27% ($= 6/22$) of the data to become remote. Once again we measure the average job execution time for our schedule. The results are sown in Table 7. We observe that storage awareness can help improve the MapReduce execution time for our job schedule by 26.5%, on average.

These results show that CAM-based Hadoop can provide better performance for Hadoop tasks by exposing network and storage topology information to the Hadoop scheduler.

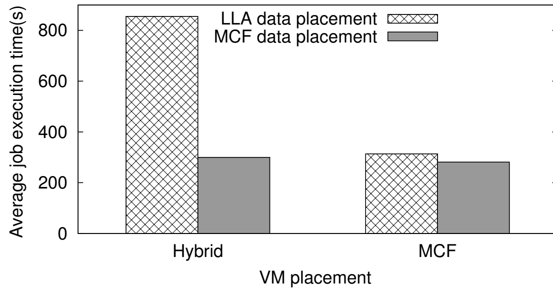

Figure 7: Execution time for Map-intensive workloads.

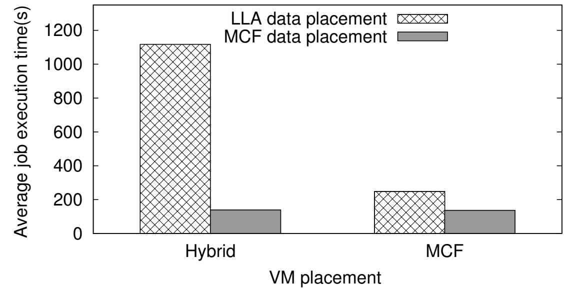

Figure 9: Execution time for MapReduce-intensive workloads.

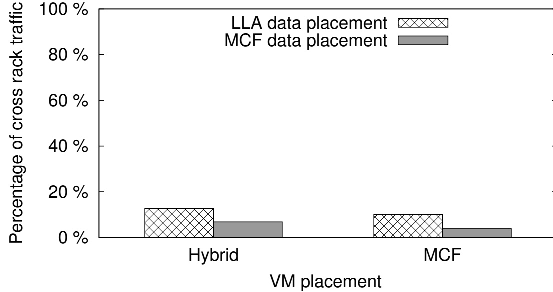

Figure 8: Fraction of data accessed remotely for Map-intensive workloads.

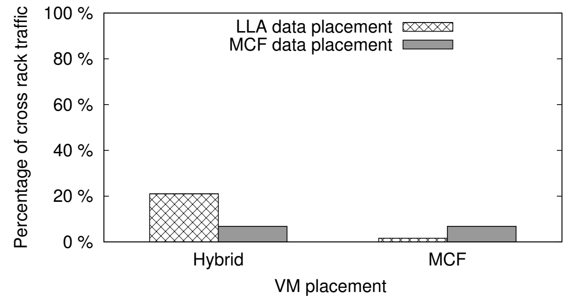

Figure 10: Fraction of data accessed remotely for MapReduce-intensive workloads.

5.2 Macro-Benchmark Results

In our next set of experiments, we show the effectiveness of our approach. For this purpose, we extend the simulator PurSim [21] to include the min-cost flow data placement, VM placement, network awareness, and storage awareness mechanisms described in Section 4. PurSim is a network flow level discrete event simulator that simulates the MapReduce execution semantics. Similarly as in previous tests, we generate a schedule with job size distribution based on Zaharia et al. [24] shown in Table 8. For these experiments the interarrival time is randomly generated between 60 and 90 seconds.

Number of jobs	38	16	14	8	6	6	4	8
Map tasks / job	1	2	10	50	100	200	400	800

Table 8: Distribution of job sizes in terms of number of map tasks used for macro-benchmark tests.

5.2.1 Data and VM Placement

In this section, we evaluate the effectiveness of min-cost flow (MCF) Data and VM placements used in CAM. We consider three types of MapReduce workloads, namely Map-intensive, MapReduce-intensive, and a workload with a mix of Map, MapReduce, and CPU-intensive jobs.

We consider two data placement strategies, namely load and locality aware (LLA) data placement and MCF data placement. We also consider two VM placement strategies, namely Hybrid VM placement and MCF VM placement. The LLA and Hybrid strategies are defined in Purlieus, and are used as a baseline for comparison to CAM's MCF based approach.

Figures 7, 9, and 11 show the average execution time for the three workloads considered. Similarly, Figures 8, 10, and 12 plot the percentage of total data (for the tests) that is accessed remotely across the rack under each combination of VM and data placements for the three studied workloads. For the Map-intensive workload, the combination of MCF VM and data placement produces a $3x$ speedup over the baseline, which is due to a $3.3x$ decrease in relative cross rack traffic. On the other hand, the same combination for the MapReduce-intensive traffic produces an $8x$ speedup with a corresponding 3 fold decrease in network traffic. The MCF placements see the best speedup of $8.6x$ verses the baseline for the mixed workload, due to the fact that they all but eliminate cross network traffic.

For all workload types using either the MCF data placement combined with the baseline VM placement, or conversely using the MCF VM placement with the baseline data placement, produces a significant speedup. Hence, the MCF graphs constructed for both placement optimization problems successfully optimize the respective factors and produce an optimal solution. Note that combining both techniques does not yield further significant benefit.

5.2.2 Impact of Network Topology Awareness

In this experiment, we configure our VM cluster to run 100 jobs simultaneously on 192 VMs using Hadoop fair share scheduling mechanism. Table 9 shows that network topology awareness im-

Network topology awareness	VM locality	Average job execution time
Unaware	82%	24.6 s
Aware	99%	22.4 s

Table 9: Impact of network topology awareness on Hadoop performance.

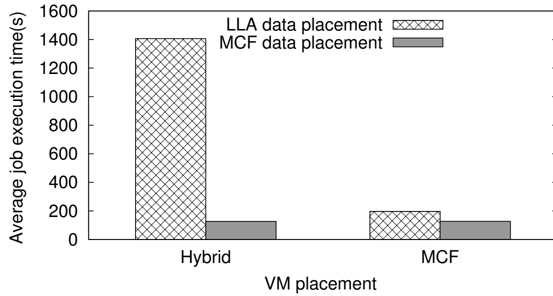

Figure 11: **Execution time for Mixed workloads.**

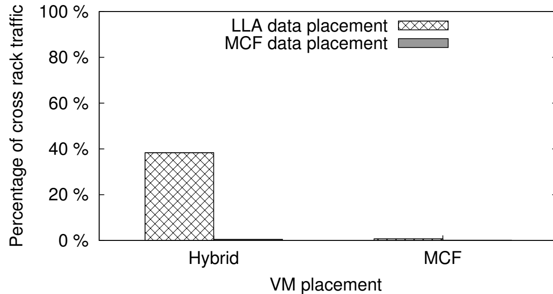

Figure 12: **Fraction of data accessed remotely for Mixed workloads.**

proves the map tasks locality on average by 7%, and reduces the average job execution time by 9%. Figure 13 shows the breaks up for the percentage of map VM locality with respect to the number of map tasks. We observe that network topology awareness is most effective for jobs that have less than 50 map tasks, improving locality by 24.2% on average.

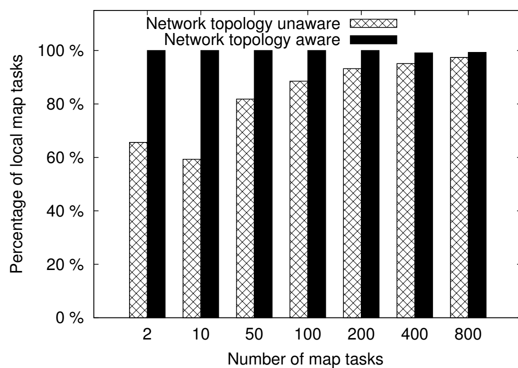

Figure 13: **Impact of network topology awareness on locality of map tasks broken down in terms of number of map tasks.**

5.2.3 *Impact of Storage Topology Awareness*

In our next experiment, we demonstrate the effectiveness of providing storage topology information hints to MapReduce. We vary the number of VM image files that are placed remotely with respect to the physical node where the VM is to be run. First, we measure how loss of VM image locality affects the average execution time. Figure 14 shows the results. We observe that as more VMs are placed remotely, the average job execution time increases. For example, with 80% remote VM images, the average Mapreduce job execution time worsens 36% compared to the all local images case (0%). As seen from the previous experiments, CAM-based Hadoop achieves all local images using the storage topology information, and thus offers an effective solution for VM placement.

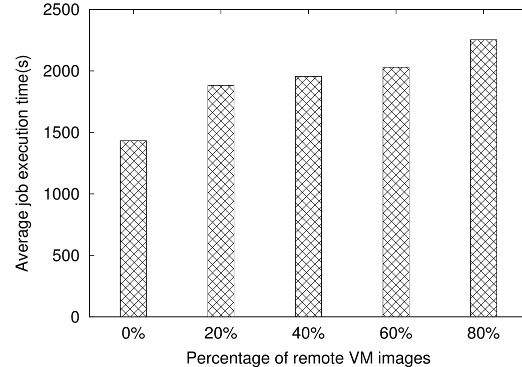

Figure 14: **Impact of storage topology awareness on MapReduce performance in terms of percentage of remote VM images.**

Next, we measure the locality of map tasks achieved with varying VM images placed remotely from their physical host. Figure 15 shows the percentage of the number of local map tasks with varying remote VM images. We see that without exposing storage topology information, the locality of map tasks is decreased. Conversely, the amount of data accessed remotely across the rack increased. Thus, by exposing storage locality, CAM can minimize the cross rack traffic due to remotely accessing VM images.

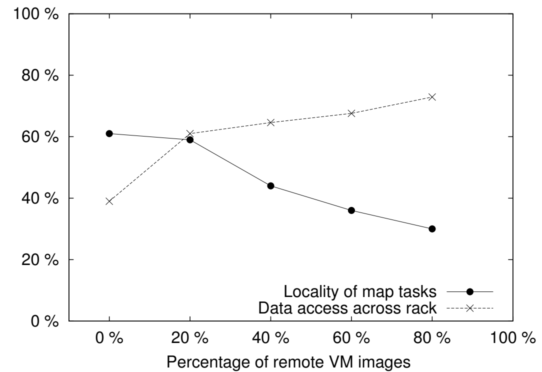

Figure 15: **Impact of storage topology awareness on MapReduce performance.**

5.2.4 *Scalability of CAM*

We now discuss the scalability of our min-cost flow model. In our macro-benchmark experiments we found the overhead to be negligible in a cluster of size 192 after repeated tests. As shown in Quincy [13], even for a large cluster size (thousands of nodes) a similarly-sized flow network can be solved in a few seconds, which is significantly smaller than the running time of typical MapReduce

jobs. That is because techniques such as successive approximation push-relabel can process large-scale graphs efficiently. Moreover, the overhead of the solver is incurred only once, when the job is submitted for scheduling.

In summary, CAM offers to simultaneously meet the different constrains to co-locate VM and data on physical hosts, and improves overall MapReduce in the cloud application performance.

6. RELATED WORK

Resource allocation for MapReduce in the cloud has received a lot of attention in recent works [13,15,18,21,25]. The project closest to our own is a resource allocation system called Purlieus, developed by Palanisamy et al. [21]. Purlieus arrives at a job-local data and VM placement solution according to heuristics specifically developed for different job types, such as Map-input or Reduce-input heavy. The system defines both data and VM locality, as well as physical machine load, which are similar to the notion of *VM closeness* and *Hotspots* in CAM. Unlike Purlieus, CAM employs a min-cost flow based approach, which can consider both VM migration as well as delayed scheduling to arrive at a global optimal placement. Additionally, CAM optimizes for *Storage utilization*, which allows it to do data, as well as CPU utilization, load balancing, consequently improving the overall VM placement. Moreover, CAM uses both the actual location of data and network topology as its inputs, whereas Purlieus relies on the virtual topology which may be different from the physical topology.

LATE [25] improves MapReduce performance in the cloud by performing effective speculative execution to reduce the job running time, while ignoring speculative task locality. CAM is different from LATE in that it couples the data placement, VM placement, and task placement to systematically improve data locality for MapReduce in the cloud.

There are several efforts that focus on MapReduce task scheduling in terms of data locality and fairness. Mantri [3] manages outliers in a resource and cause aware manner on native cluster. Delay scheduling [24] target fairness scheduling while maximize the map tasks locality on native clusters by delaying a task multiple times. Although delay scheduling offers a simple technique to provide better locality, it does not consider global scheduling, thus it loses the opportunity for achieving better performance. Moreover, the effectiveness of delay scheduling relies on the assumption that most tasks comprise of either small or long jobs. Quincy [13] uses similar graph techniques, but it differs from CAM in terms of problem space and associated flow network construction. Quincy strikes a balance between fairness and data locality, while CAM focuses on optimizing data/VM placement of MapReduce applications in the cloud. As a result, the factors encoded in the flow graph (VM closeness, Hotspot, etc.) are fundamentally different from that of Quincy's.

A plethora of VM placement and migration techniques are proposed in the cloud to optimize for minimizing network traffic, energy, meeting SLA requirement, etc. [7,11,16,17,20,23]. The VM placement problem is essentially a bin-packing problem for which various heuristics are applied. Such work is different from CAM, because unlike CAM it does not consider data placement and task placement, which are critical for MapReduce applications in the cloud.

7. CONCLUSION

In this paper, we have presented the design of CAM, a platform with an innovative resource scheduler designed to address performance degradation of MapReduce jobs when running in the cloud. CAM adopts a three level approach to avoid placement anomalies due to inefficient resource allocation: placing data within the cluster that run jobs that most commonly operate on the data; selecting the most appropriate physical nodes to place the set of virtual machines assigned to a job; and exposing, otherwise hidden, compute, storage and network topologies to the MapReduce job scheduler. CAM uses a flow-network-based algorithm that is able to reconcile resource allocation with a variety of other competing constraints such as storage utilization, changing CPU load and network link capacities. Evaluation of our approach using both micro-benchmarking and simulation on a 23 VM cluster shows that compared to a state-of-the-art resource allocator, CAM reduces network traffic and average MapReduce job execution time by a factor of 3 and 8.6, respectively.

CAM leads to important follow-on work. While we observe promising results in our experiments, we would like to further tune our min-cost flow model for data and VM placement and validate its effectiveness in larger real VM clusters. The topology information exposed by CAM could be leveraged by other topology-sensitive applications. We would like to create a standardized API that could be adopted as a cloud standard such that applications are able to access the service regardless of the specific cloud implementation.

8. ACKNOWLEDGMENT

This paper is based upon work supported in part by the National Science Foundation under Grants CCF-0746832, CNS-1016793, and CNS-1016408. We are thankful to our shepherd Dr. Michela Taufer for her feedback in preparing the final draft of the paper, and to Guanying Wang for his initial ideas on using min-cost flow approach for VM and data placement for supporting virtualized MapReduce.

9. REFERENCES

[1] The Rackspace Cloud. `http://www.rackspacecloud.com/`.

[2] Amazon. Amazon Elastic Compute Cloud (Amazon EC2). `http://www.amazon.com/b?ie=UTF8&node=201590011`.

[3] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proc. 9th USENIX conference on Operating systems design and implementation (OSDI)*, 2010.

[4] R. Ananthanarayanan, K. Gupta, P. P, H. Pucha, P. Sarkar, M. Shah, and R. Tewari. Cloud analytics: Do we really need to reinvent the storage stack? In *Proc. of USENIX conference on Hot topics in cloud computing (HotCloud)*, 2009.

[5] Apache Software Foundation. Hadoop. `http://hadoop.apache.org/core/`.

[6] S. Baker. How the new york times uses clouds? 2007. `http://www.businessweek.com/the_thread/blogspotting/archives/2007/12/how_thenew_yor.html`.

[7] N. Bobroff, A. Kochut, and K. Beaty. Dynamic placement of virtual machines for managing sla violations. In *Proc. 10th IFIP/IEEE International Symposium on Integrated Network Management (IM'07)*, 2007.

[8] J. Cala and P. Watson. Automatic software deployment in the azure cloud. *Lecture Notes in Computer Science: Distributed Applications and Interoperable Systems*, 6115:155–168, 2010.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. 6th USENIX conference on Operating systems design and implementation (OSDI)*, 2004.

[10] K. Gupta, R. Jain, I. Koltsidas, H. Pucha, P. Sarkar, M. Seaman, and D. Subhraveti. GPFS-SNC: An enterprise storage framework for virtual-machine clouds. *IBM Journal of Research and Development*, 55(6):2:1 –2:10, 2011.

[11] F. Hermenier, X. Lorca, J.-M. Menaud, G. Muller, and J. Lawall. Entropy: a consolidation manager for clusters. In *Proc. ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2009.

[12] IBM. IBM service agility accelerator for cloud. `http://www.ibm.com/developerworks/wikis/display/tivolidoccentral/IBM+Service+Agility+Accelerator+for+Cloud`.

[13] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proc. ACM 22nd symposium on Operating systems principles (SOSP)*, 2009.

[14] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, E. Zeidner. Internet Small Computer Systems Interface (ISCSI). `http://tools.ietf.org/html/rfc3720`.

[15] K. Jaewon, Z. Yanyong, and B. Nath. Tara: Topology-aware resource adaptation to alleviate congestion in sensor networks. *IEEE Transactions on Parallel and Distributed Systems*, 18(7):919 –931, 2007.

[16] A. Kochut. On impact of dynamic virtual machine reallocation on data center efficiency. In *Proc. IEEE International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems (MASCOTS)*, 2008.

[17] A. Kochut and K. Beaty. On strategies for dynamic resource management in virtualized server environments. In *Proc. IEEE International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems (MASCOTS)*, 2007.

[18] M. A. Kozuch, M. P. Ryan, R. Gass, S. W. Schlosser, D. O'Hallaron, J. Cipar, E. Krevat, J. López, M. Stroucken, and G. R. Ganger. Tashi: location-aware cluster management. In *Proc. 1st workshop on Automated control for datacenters and clouds (ACDC)*, 2009.

[19] K. Malavalli and B. Stovhase. Distributed computing with fibre channel fabric. In *Proc. Thirty-Seventh IEEE Computer Society International Conference (Compcon)*, 1992.

[20] X. Meng, V. Pappas, and L. Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *Proc. 29th conference on Information communications (INFOCOM)*, 2010.

[21] B. Palanisamy, A. Singh, L. Liu, and B. Jain. Purlieus: Locality-aware resource allocation for mapreduce in a cloud. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[22] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *Proc. IEEE 26th International Conference on Data Engineering (ICDE)*, 2010.

[23] H. N. Van, F. D. Tran, and J.-M. Menaud. Autonomic virtual resource management for service hosting platforms. In *Proc. ICSE Workshop on Software Engineering Challenges of Cloud Computing*, 2009.

[24] 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 European conference on Computer systems (EuroSys)*, 2010.

[25] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proc. 8th USENIX conference on Operating systems design and implementation (OSDI)*, 2008.