Purlieus: Locality-aware Resource Allocation for MapReduce in a Cloud

Balaji Palanisamy
College of Computing
Georgia Tech
balaji@cc.gatech.edu

Aameek Singh
IBM Research - Almaden
aameek@us.ibm.com

Ling Liu
College of Computing
Georgia Tech
lingliu@cc.gatech.edu

Bhushan Jain
IBM India Software Lab
bhujain1@in.ibm.com

ABSTRACT

We present Purlieus, a MapReduce resource allocation system aimed at enhancing the performance of MapReduce jobs in the cloud. Purlieus provisions virtual MapReduce clusters in a locality-aware manner enabling MapReduce virtual machines (VMs) access to input data and importantly, intermediate data from local or close-by physical machines. We demonstrate how this locality-awareness during both map and reduce phases of the job not only improves runtime performance of individual jobs but also has an additional advantage of reducing network traffic generated in the cloud data center. This is accomplished using a novel coupling of, otherwise independent, data and VM placement steps. We conduct a detailed evaluation of Purlieus and demonstrate significant savings in network traffic and almost 50% reduction in job execution times for a variety of workloads.

1. INTRODUCTION

In most modern enterprises today, *big data* [5] and *big data analytics* play a key role in delivering value to the business. Whether it is using click stream analysis to identify customer buying behavior [7] or detecting fraud from millions of transactions [9], analyzing large amounts of data efficiently and quickly makes businesses more profitable. One of the technologies that made big data analytics popular and accessible to enterprises of all sizes is MapReduce [3] (and its open-source Hadoop [23] implementation). With the ability to automatically parallelize the application on a cluster of commodity hardware, MapReduce allows enterprises to analyze terabytes and petabytes of data more conveniently than ever. Today MapReduce forms the core of technologies powering enterprises like Google, Yahoo and Facebook.

Further, MapReduce offered as a service in the cloud provides an attractive usage model for enterprises. A recent Gartner survey shows increasing cloud computing spending with 39% of enterprises having allotted IT budgets for it [1]. A MapReduce cloud service will allow enterprises to cost-effectively analyze large amounts of data without creating large infrastructures of their own. Using virtual machines (VMs) and storage hosted by the cloud, enterprises can simply create virtual MapReduce clusters to analyze their data.

An important challenge for the cloud provider is to manage multiple virtual MapReduce clusters executing concurrently, a diverse set of jobs on shared physical machines. Concretely, each MapReduce job generates different loads on the shared physical infrastructure – (a) computation load: number and size of each VM (CPU, memory), (b) storage load: amount of input, output and intermediate data, and (c) network load: traffic generated during the map, shuffle and reduce phases. The network load is of special concern with MapReduce as large amounts of traffic can be generated in the shuffle phase when the output of map tasks is transferred to reduce tasks. As each reduce task needs to read the output of *all* map tasks [3], a sudden explosion of network traffic can significantly deteriorate cloud performance. This is especially true when data has to traverse greater number of network hops while going across *racks* of servers in the data center [4]. Further, the problem sometimes is exacerbated by TCP *incast* [37] with a recent study finding goodput of the network reduced by an order of magnitude for a MapReduce workload [13].

To reduce network traffic for MapReduce workloads, we argue for improved **data locality** for both Map and Reduce phases of the job. The goal is to reduce the network distance between storage and compute nodes for both map and reduce processing – for map phase, the VM executing the map task should be *close* to the node that stores the input data (preferably local to that node) and for reduce phase, the VMs executing reduce tasks should be close to the map-task VMs which generate the intermediate data used as reduce input. Improved data locality in this manner is beneficial in two ways – (1) it reduces job execution times as network transfer times are big components of total execution time and (2) it reduces cumulative data center network traffic. While map locality is well understood and implemented in MapReduce systems, reduce locality has surprisingly received little attention in spite of its significant potential impact. As an example, Figure 1 shows the impact of improved reduce locality for a Sort workload. It shows the Hadoop task execution timelines for a 10 GB dataset in a 2-rack 20-node physical cluster[1], where 20 Hadoop VMs were placed without and with reduce locality (top and bottom figures respectively). As seen from the graph, reduce locality resulted in a significantly shorter shuffle phase helping reduce total job runtime by 4x.

In this paper, we present Purlieus – an intelligent MapReduce cloud resource allocation system. Purlieus improves data locality during both map and reduce phases of the MapReduce job by carefully coupling data and computation (VM) placement in the cloud. Purlieus categorizes MapReduce jobs based on how much data they access during the map and reduce phases and analyzes the network flows between sets of machines that store the input/intermediate data and those that process the data. It places data on those machines that can either be used to process the data themself or are close to the machines that can do the processing. This is in contrast

SC '11, November 12-18,2011, Seattle, Washington, USA
Copyright 2011 ACM 978-1-4503-0771-0/11/11 ...$10.00.

[1] Complete experimental setup is described in Section 5.

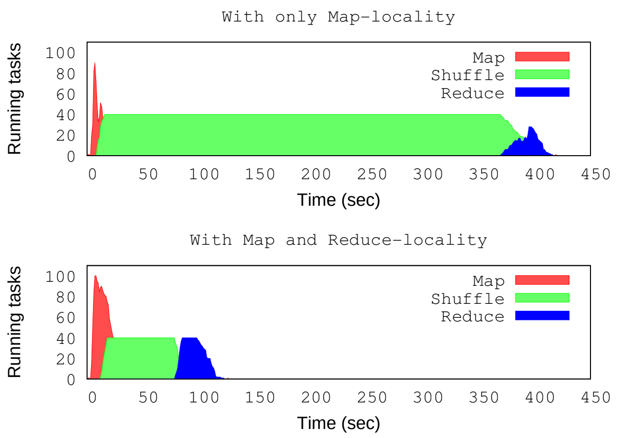


Figure 1: Impact of Reduce-locality. Timeline plotted using Hadoop's *job_history_summary*. *Merge* and *Waste* series are omitted since they were negligible

to conventional MapReduce systems which place data independent of map and reduce computational placement – data is placed on any node in the cluster which has sufficient storage capacity [3, 23] and only map tasks are attempted to be scheduled local to the node storing the data block.

Additionally, Purlieus is different from conventional MapReduce clouds (e.g., Amazon Elastic MapReduce [14]) that use a separate compute cloud for performing MapReduce computation and a separate storage cloud for storing the data persistently. Such an architecture delays job execution and duplicates data in the cloud. In contrast, Purlieus stores the data in a dedicated MapReduce cloud and jobs execute on the same machines that store the data without waiting to load data from a remote storage cloud.

To the best of our knowledge, Purlieus is the first effort that attempts to improve data locality for MapReduce in a cloud. Secondly, Purlieus tackles the locality problem in a fundamental manner by coupling data placement with VM placement to provide both map and reduce locality. This leads to significant savings and can reduce job execution times by close to 50% while reducing up to 70% of cross-rack network traffic in some scenarios.

2. SYSTEM MODEL

In our system model, customers using the MapReduce cloud service load their input datasets and MapReduce jobs into the service. This step is similar to any typical cloud service which requires setting up of the application stack and data. There is one key distinction, however. Typically cloud service providers use two distinct infrastructures for storage and compute (e.g. Amazon S3 [16] for storage and Amazon EC2 [15] for compute). Executing a MapReduce job in such infrastructures (e.g. using Amazon Elastic MapReduce [14]) requires an additional *loading* step, in which data is loaded from the storage cloud into the distributed filesystem (e.g. Hadoop's HDFS) of the MapReduce VMs running in the compute cloud before even the job begins execution. Such additional loading has two drawbacks –(1) depending upon the amount of data required to be loaded and connectivity between the compute and storage infrastructures, this step adversely impacts performance, and (2) while the job is running (often for long durations) the dataset is duplicated in the cloud – along with the storage cloud original, there is a copy in the compute cloud for MapReduce processing, leading to higher costs for the provider.

In contrast, we propose a dedicated MapReduce service, in which data is directly stored on the same physical machines that run MapReduce VMs. This prevents the need for a wasteful data loading step before executing a MapReduce job. Since MapReduce input data is often predominantly used for MapReduce analysis, storing it into a dedicated cloud service provides the greatest opportunity for optimization. The challenge for this design is the ability to transition data stored on physical machines to the MapReduce VMs in a seamless manner – i.e. without requiring an explicit data-loading step. This is accomplished in the following manner.

In our proposed service, when customers upload their data into the service, the data is broken up into chunks corresponding to MapReduce blocks and stored on a distributed filesystem of the *physical* machines. The placement of data – deciding which machines to use for each dataset – is done intelligently based on techniques described later. When the job begins executing (i.e. MapReduce VMs are initialized) the data on physical machines is seamlessly made available to VMs using two specific techniques – (1) *loopback mounts*: For a job, when its data is loaded into the cloud, the chunks being placed on each machine are stored via a loopback mount [2] into a single data file (we refer to it as a vdisk-file), this provides access similar to any local filesystem, even though all data is being stored in a single file on the *physical* filesystem. and (2) *VM disk-attach*: The vdisk-file is then attached to the VM as a block device using server virtualization tools (e.g. KVM's `virsh attach-device` command[2]). The VM can then mount the vdisk file like it would any typical filesystem. The mount point of this vdisk-file inside the VM serves as the MapReduce DFS directory (e.g. Hadoop's `data.dir` configuration variable).

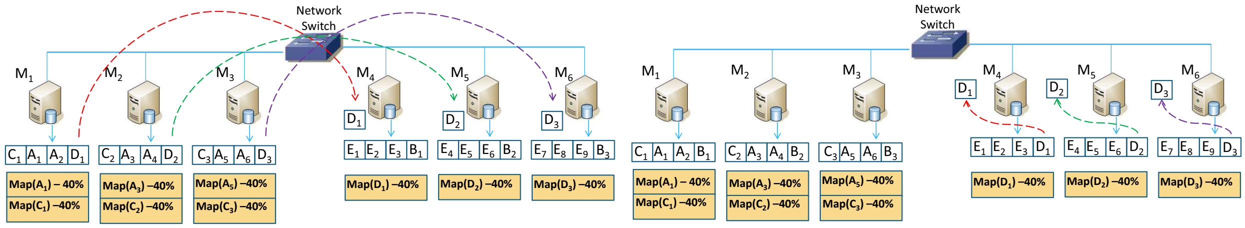
We implemented this architecture on our cluster of CentOS 5.5 physical machines with KVM as the hypervisor. Figure 2 shows the sequence of steps used to store data persistently on a physical machine and seamlessly transfer it to one of its VMs without requiring additional loading.

1. Create a vdisk file on the hypervisor (for instance, 5 GB)
```
dd of=vdisk-file bs=1M count=0 seek=5192
```
2. Format as ext2:
```
mkfs.ext2 -F vdisk-file
```
3. Loopback mount the vdisk file:
```
mount -t ext2 -o loop vdisk-file vdisk-mount
```
4. Store input data into `vdisk-mount` in a MapReduce chunk format e.g. as a simplification, by creating a MapReduce cluster on the physical machines.
5. Unmount vdisk-mount. `vdisk-file` represents persistent data for each VM.
6. Upon VM initialization, the vdisk file is attached to the VM as a block device
```
virsh attach-device vm vdisk-file-cfg.xml
```
7. VM can mount the block device like a new disk
```
mount -t ext2 /dev/sdb /data-dir
```
/data-dir contains dataset blocks and used as Hadoop *dfs.data.dir*
8. Virtual MapReduce cluster is initialized between the VMs by starting the MapReduce cluster - each VM reports the data blocks to the MapReduce *NameNode* to initialize the filesystem. Then job execution can begin.
9. After job execution, VMs can be destroyed. On subsequent initializations, only steps 6 onwards need to be performed.

Figure 2: Dataflow from physical to virtual machines

These series of steps ensure that data is loaded onto the same physical machines that host the VMs for MapReduce computation and even while the VMs can be non-persistent (e.g. customer may destroy VMs between different job executions to minimize cost), the data is persistently stored on the physical machines. Secondly

[2] Similar commands exist for Xen and VMware



(a) Load-unaware data placement

(b) Load-aware data placement

Figure 3: Load Awareness in Data placement

by using the VM disk-attach step, we are able to seamlessly transition this data into the VM cluster without requiring explicit loading. In contrast, a separate compute and storage cloud infrastructure would require paying the data loading overhead each time the VMs are initialized[3]. In our architecture, note that if a MapReduce VM is required to be placed on a physical machine other than the one containing that job's data chunks, the vdisk file can be copied over to the appropriate physical machine and then attached to the VM. This step is similar to traditional MapReduce's *remote-read* operation.

3. PURLIEUS: PRINCIPLES AND PROBLEM ANALYSIS

In our proposed system model, the cloud provider faces two key questions – (1) *Data Placement*: Which physical machines should be used for each dataset? and (2) *VM Placement*: Where should the VMs be provisioned to process these data blocks? Poor placement of data or VMs may result in poor performance. Purlieus tackles this challenge with a unique coupled placement strategy, where data placement is aware of likely VM placement and attempts to improve locality. In this section, we describe the principles of our design and provide a formal analysis of the problem.

3.1 Principles

We argue that unlike traditional MapReduce, where data is placed independently of the type of job processing it or the loads on the servers, in a multi-tenant virtualized cloud these attributes need to be accounted during data placement.

1. Job Specific Locality-awareness: *Placing data in the MapReduce cloud service should incorporate job characteristics* - specifically the amount of data accessed in the map and reduce phases. For example, a job that processes a lot of reduce data (referred to as a *reduce-input heavy* job) is best served by provisioning the VMs of MapReduce cluster close to each other on the cluster network, as each reducer reads the outputs of all mappers. If the VMs are far from each other, each reducer will read map outputs over longer network paths increasing job execution time and also increasing cross-rack traffic in the data center. On the other hand, *map-input heavy* jobs that generate little intermediate data do not benefit by placing its data blocks close to each other on the cluster. An efficient data placement scheme could distribute data blocks for such a *map-input heavy* job across the network to preserve resources for placing *reduce-input heavy* jobs on closely connected machines.

Specifically, we use three distinct classes of jobs – (1) Map-input heavy (e.g. a large $grep$ workload that generates small intermediate data simply indicating if a word occurs in input data), (2) Map-and-Reduce-input heavy (e.g. a $sort$ workload: intermediate data is equal to input data) and (3) Reduce-input-heavy (e.g. a *permutation generator* workload which generates permutations of input strings). Purlieus uses different data placement strategies for different job types with the goal of improving data locality[4].

2. Load Awareness: *Placing data in a MapReduce cloud should also account for computational load (CPU, memory) on the physical machines*. A good technique should place data only on machines that are likely to have available capacity to execute that job, else remote-reads will be required to pull data from busy machines to be processed at less-utilized machines.

For example, in Figure 3(a), consider datasets A, B, C, D and E placed on six physical machines, M_1 to M_6. A load unaware placement may colocate the blocks of datasets A, C and D together as shown in Figure 3(a), even if jobs execute on A, C, D more frequently and generate higher load than B and E. Here, when the job on the dataset D arrives and requests for a virtual cluster of 3 VMs, say each with 40% CPU resources of the physical machine, even though it would be best to place the VMs on the physical machines, M_1, M_2 and M_3 as they contain the data blocks of the dataset D, the system may be forced to place the VMs on M_4, M_5 and M_6, resulting in remote reads for the job executing on dataset D. In contrast, the load-aware data placement shown in Figure 3(b) is able to achieve local execution for all the map tasks as it is able to host the VMs on the physical machines containing the input blocks.

In Purlieus, while placing data blocks, it is ensured that the expected load on the servers does not exceed a configurable threshold. This incorporates the frequency and load generated by jobs executing on datasets stored on these servers. It is important to note that information about expected loads is available to a cloud provider by monitoring the cloud environment. Typically with MapReduce, a set of jobs are repeatedly executed on a similar input data set – e.g. periodic execution of indexing on web crawled data. This allows the cloud provider to understand the load characteristics of such jobs and use this knowledge to optimize its environment. Additionally, there are many proposals that *profile* MapReduce jobs via trial executions on a small subset of data [10, 31, 6, 8, 18]. These show that understanding MapReduce job characteristics can be quick and reasonably accurate. For the scope of this work, we assume that the expected load on each dataset is known. Also that the cloud provider has enough data to estimate job arrival rate and the mean execution time. However, we do not completely rely on the accuracy of these estimates, rather use them as an additional guiding measure. In Section 5, we will demonstrate that our proposed techniques perform well even when such estimates are partly erroneous.

3. Job-specific Data Replication: Traditionally, data blocks in MapReduce are replicated within the cluster for resiliency. While the job is executing, any replica of the block can be used for pro-

[3] In Amazon Elastic MapReduce [14], by default VMs are destroyed after job completion, thus requiring data loading for each run of the job. Alternatively, using a –alive option VMs can be made persistent across job runs, but users have to pay for them for the entire duration.

[4] For completeness a Map-and-Reduce-input light class can also be considered, however locality has little impact on its performance

cessing. Purlieus handles replication in a different manner. Depending upon the type and frequency of jobs, we place each replica of the entire dataset based on a particular strategy. For example, if an input dataset is used by three sets of MapReduce jobs, two of which are reduce-input heavy and one map-input heavy, we place two replicas of data blocks in a reduce-input heavy fashion and the third one using map-input heavy strategy. This allows maintaining greater data locality, especially during the reduce phase, since otherwise by processing data block replicas *far* from other input data blocks during the map phase, the reducers may be forced to read more data over the network.

3.2 The Data Placement Problem

Next, we formally analyze the data placement problem. We start with notations for representing datasets, physical cloud infrastructure and their relationship.

Datasets and Jobs: Let $\mathcal{D} = \{D_i : 1 \leq i \leq |\mathcal{D}|\}$ be the set of datasets that need to be stored in the MapReduce cloud. For the sake of presentation simplicity, assume that each dataset is associated with only one MapReduce job-type and that the replication factor is 1[5]. Each dataset D_i is divided into uniform sized blocks $B_{i,j} : 1 \leq i \leq |\mathcal{D}|, 1 \leq j \leq Q_i\}$ where $\mathcal{Q} = \{Q_i : 1 \leq i \leq |\mathcal{Q}|\}$ represent the number of blocks for D_i.

We assume that the job arrivals on the datasets follow a Poisson process and let $\lambda = \{\lambda_i : 1 \leq i \leq |\mathcal{D}|\}$ denote the arrival rate of the jobs on the datasets. After a job starts, it first executes map tasks. We denote the mean size of the expected map output of each block of dataset, D_i by $mapoutput(D_i)$.

Cloud Infrastructure: Let $\mathcal{M} = \{M_k : 1 \leq k \leq |\mathcal{M}|\}$ denote the set of physical machines. Each physical machine, M_k has some compute resources with capacity $Pcap(M_k)$[6] and some storage resources (disk) with capacity denoted by $Scap(M_k)$. In the data center, the physical machines are connected to each other by a local area network. Let $dist(M_l, M_m)$ denote the *distance* between the physical machines M_l and M_m – we use number of network hops as the $dist$ measure.

Relationship Notation: Let $P_i \in \mathcal{M}$ be the set of servers used to store the dataset D_i and X_i^k be a Boolean variable indicating if the physical machine M_k is used to store the dataset Di. Therefore, $M_k \in P_i$ if X_i^k =1. Let $\mathcal{N} = \{N_i : 1 \leq i \leq |\mathcal{N}|\}$ denote the number of machines used to store D_i. Thus,

$$\sum_k X_i^k = N_i, \forall i$$

Within P_i, let $Y_{i,j}^k$ be the Boolean variable indicating if the specific block $B_{i,j}$ is present in the physical machine $M_k \in \mathcal{M}$. Thus, in order to ensure that the blocks are evenly distributed among the nodes in P_i, we have

$$\forall i, k \sum_{1 \leq j \leq Q_i} Y_{i,j}^k = \frac{Q_i}{N_i}$$

Locality based Cost: To capture locality, we define a cost function that measures the amount of data transfer during job execution. Consider a job, A on the dataset, D_i. Let $V(A)$ be the set of virtual machines used by job A and let $Pnode(v)$ represent the physical machine hosting the VM, $v \in V(A)$. The total cost of a MapReduce application is the sum of map and reduce costs that represent the overhead involved in the data transfers during the map and reduce phases.

$$Cost(A, D_i) = Mcost(A, D_i) + Rcost(A, D_i)$$

Here, $Rcost$ incorporates the *shuffle time* of the job. If $Snode(B_{i,j})$ $\in P_i$ is the physical machine storing the data block, $B_{i,j}$, and its map task gets scheduled on the physical machine, $Cnode(B_{i,j})$ that hosts some VM, $v \in V(A)$, we consider

$$Mcost(A, D_i) = \sum_{1 \leq j \leq Q_i} size(B_{i,j}) \times dist(Snode(B_{i,j}), Cnode(B_{i,j}))$$

This cost definition captures the amount of data and the distance it travels over the network. Similarly, the reduce cost can be computed as the overhead involved in transferring the map outputs to the servers where the reducers are executed. Let $L(A)$ be the set of reduce tasks for job A. As each reduce task, $rtask_l, l \in L(A)$ needs to see the output of all the map tasks, the map outputs need to be transferred to the corresponding reduce tasks. Therefore the reduce cost is given by:

$$Rcost(A, D_i) = \sum_{1 \leq j \leq Q_i, 1 \leq l \leq L(A)} dist(Cnode(B_{i,j}), Cnode(rtask_l) \times m_{out}(A, B_{i,j}, rtask_l)$$

where $m_{out}(A, B_{i,j}, rtask_l)$ is the amount of output data generated by the map task on the data block, $B_{i,j}$ that gets transferred to reduce task, $rtask_l$ that is run on $Cnode(rtask_l)$. To improve locality, the goal is to minimize $Mcost$ and $Rcost$, subject to not violating the storage capacity constraint on physical machines

$$\forall k \sum_{i,j} Y_{i,j}^k \leq Scap(M_k)$$

Minimizing Map Cost: To minimize map cost, the computations should get placed on the same physical machines storing the map-input blocks ($dist$ is zero). The *data placement* technique, in turn, should try to maximize the probability of such co-location. This is achieved by upper-bounding the expected resource load on the servers for hosting the VMs at any given time. By placing data blocks such that every server has a low expected utilization, there is higher probability that the server will be available to host a VM when a request for a job on the datasets arrives. Concretely, we model each physical machine, M_k as a $M/M/1$ single server queue. Let a dataset, D_i have a service time distribution with mean, μ_i, where μ_i is the mean time to process the blocks by each VM and $\rho_i = \frac{\lambda_i}{\mu_i}$. Therefore the expected number of jobs on the dataset D_i running on the physical machine M_k is given by

$$W_i^k = \frac{\rho_i}{\rho_i - \mu_i}.X_i^k$$

Now, the expected load on physical machine M_k is given by

$$E^k = \sum_i W_i^k \times CRes(D_i)$$

where $CRes(D_i)$ denotes the computational resource required by each VM of the job on D_i, given by the type of VM chosen by the user (e.g. Amazon EC2's *small* VM instance that uses 1.7 GB memory and 1 vCPU). We upper-bound the expected load on any physical machine based on the load parameter, α.

$$\forall k, E^k \leq \alpha \times Pcap(M_k)$$

[5] If the dataset is associated with multiple jobs of different job types, as mentioned earlier different replicas are used to support each type. Our model can be directly extended to incorporate different replicas of the same dataset

[6] Though we present a scalar capacity value, compute resources may have multiple dimensions like CPU and memory. To handle this, our model can be extended to include a vector of resources or compute dimensions can be captured in a scalar value, e.g. the volume metric presented in [12].

Here, a low value of α would indicate a conservative data placement where the expected load on the physical machines is less and therefore there is a high probability for a job on a data chunk on a physical machine to get executed locally.

Minimizing Reduce Cost: With the above method for minimizing map cost, now the key optimization is to improve reduce locality. At the time of data placement, the node used to host the VM that processes the data, $Cnode(B_{i,j})$ is not fixed. Hence $Rcost$ can not be obtained precisely during data placement. Instead, we compute an estimated reduce cost during data placement – we assume that at the time of job execution, the VMs get placed on the physical machines storing the data block, which based on the previous map cost optimization should be likely. Assuming every VM, $v \in V(A)$ runs equal number of reducers (i.e each VM runs $\frac{|L(A)|}{|V(A)|}$ reducers) and every map output being uniformly distributed among the reducers, now the optimization is

$$min \sum_i Rcost_{est}(A, D_i)$$

$$Rcost_{est}(A, D_i) =$$

$$\sum_{1 \leq j \leq Q_i, v \in V(A)} dist(Snode(B_{i,j}), Pnode(v)) \times \frac{mapoutput(D_i)}{\frac{L(A)}{|V(A)|}}$$

where $Pnode(v)$ is the physical machine hosting the VM, v and $mapoutput(D_i)$ is the mean size of the expected map output of each block of dataset, D_i. While being an estimate, this definition serves as a useful guideline for placement decisions, which as our evaluations show provides significant benefits.

It is easy to see that an optimal solution for this problem is NP-Hard – both data and VM placement involve bin-packing, which is known to be NP-Hard [28]. Therefore, we use a heuristics based approach, which is described next.

4. PURLIEUS: PLACEMENT TECHNIQUES

Next, we describe Purlieus's data and VM placement techniques for various classes of MapReduce jobs. The goal of these placements is to minimize the total $Cost$ by reducing the $dist$ function for map (when input data, Q_i is large) and/or reduce (when intermediate data, m_{out} is large).

4.1 Map-input heavy jobs

Map-input heavy jobs read large amounts of input data for map but generate only small map-outputs that is input to the reducers. For placement, mappers of these jobs should be placed close to input data blocks so that they can read data locally, while reducers can be scheduled farther since amount of map-output data is small.

4.1.1 Placing Map-input heavy data

As map-input heavy jobs do not require reducers to be executed close to each other, the VMs of the MapReduce cluster can be placed anywhere in the data center. Thus, physical machines to place the data are chosen only based on the storage utilization and the expected load, E_k on the machines. As discussed in the cost model, E^k denotes the expected load on machine, M_k.

$$E^k = \sum_i W_i^k \times CRes(D_i)$$

To store map-input heavy data chunks, Purlieus chooses machines that have the least expected load. This ensures that when MapReduce VMs are placed, there is likely to be capacity available on machines storing the input data.

4.1.2 VM placement for Map-input heavy jobs

The VM placement algorithm attempts to place VMs on the physical machines that contain the input data chunks for the map phase. This results in lower $MCost$ – the dominant component for map-input heavy jobs. Since data placement had placed blocks on machines that have lower expected computational load, it is less likely, though possible that at the time of job execution, some machine containing the data chunks does not have the available capacity. For such a case, the VM may be placed close to the node that stores the actual data chunk. Specifically, the VM placement algorithm iteratively searches for a physical machine having enough resources in increasing order of network distance from the physical machine storing the input data chunk. Among the physical machines at a given network distance, the one having the least load is chosen.

4.2 Map-and-Reduce-input heavy jobs

Map-and-reduce-input heavy jobs process large amounts of input data and also generate large intermediate data. Optimizing cost for such jobs requires reducing the $dist$ function during both their map and reduce phases.

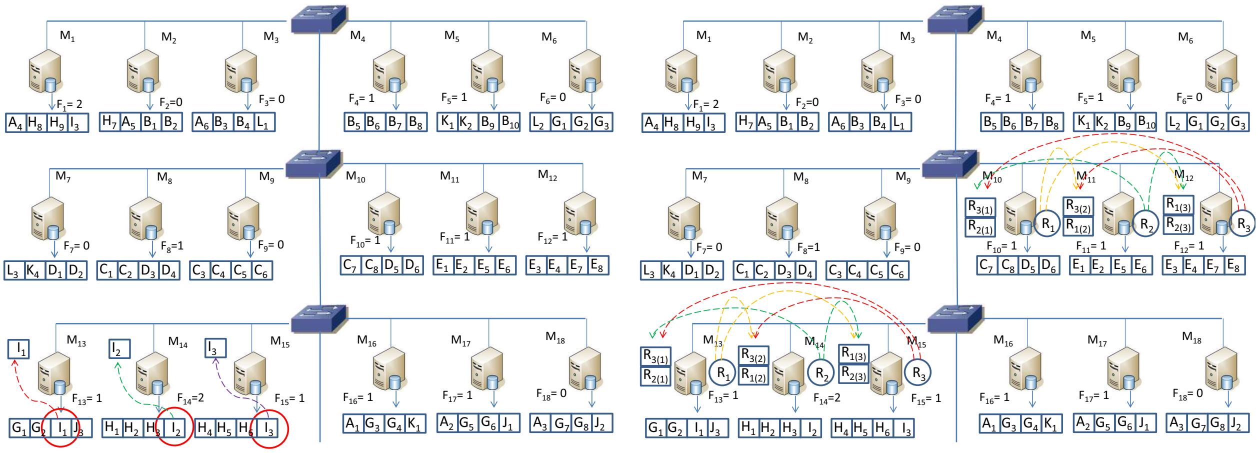
4.2.1 Placing Map-and-Reduce-input heavy data

To achieve high map-locality, data should be placed on physical machines that can host VMs locally. Additionally, this data placement should support reduce-locality – for which the VMs should be hosted on machines close to each other (preferably within the rack) so that reduce traffic does not significantly load the data center network. Ideally, a subgraph structure that is densely connected, similar to a *clique*, where every node is connected to every other node in 1-hop would be a good candidate for placing the VMs. However, it may not always be possible to find cliques of a given size as the physical network may not have a *clique* or even if it does, some of the machines may not have enough resources to hold the data or their expected computational load may be high to not allow VM placement later. An alternate approach would be to find subgraph structures similar to cliques. A number of clique relaxations have been proposed, one of which is *k-club* [27]. A *k-club* of a graph G is defined as a maximal subgraph of G of diameter k. While finding *k-club* is NP-Complete for a general graph, data center networks are typically hierarchical (e.g. *fat-tree* topologies) and this allows finding a *k-club* in polynomial time. In a data center tree topology, the leaf nodes represent the physical machines and the non-leaf nodes represent the network switches. To find a *k-club* containing n leaf nodes, the algorithm simply finds the sub-tree of height $\frac{k}{2}$ containing n or more leaf nodes.

For map-and-reduce-input heavy jobs, data blocks get placed in a set of closely connected physical machines that form a *k-club* of least possible k (least possible height of the subtree) given the available storage resources in them. If several subtrees exists with the same height, then the one having the maximum available resource is chosen. As an illustration, in Figure 4(a), the input data blocks, I_1, I_2, and I_3 are stored in a closely connected set of nodes M_{13}, M_{14} and M_{15} that form a *k-club* of least possible k in the cluster.

4.2.2 VM placement for Map and Reduce-input heavy jobs

As data placement had done an optimized placement by placing data blocks in a set of closely connected nodes, VM placement algorithm only needs to ensure that VMs get placed on either the physical machines storing the input data or the close-by ones. This reduces the distance on the network that the reduce traffic needs to go over, speeding up job execution while simultaneously reducing cumulative data center network traffic. In the example shown



(a) Map-phase

(b) Reduce-phase

Figure 4: Data and VM placement. Bottom squares show data blocks placed on each machine. Squares next to a machine (e.g. I_1 near M_{13} for map-phase in figure 4(a)) indicates reading of the block for map processing. F measure denotes available computational capacity – for simplicity, number of VMs that can be placed on that machine. In reduce phase (figure 4(b)) , circled R_i indicates map outputs and square $R_i(j)$ indicates reading of intermediate data for reducer j from map task output, i.

in Figure 4, VMs for job on dataset I get placed on the physical machines storing input data. As a result, map tasks use local reads (Figure 4(a)) and reduce tasks also read within the same rack, thereby maximizing reduce locality (Figure 4(b)). In case node M_{15} did not have available resources to host the VM, then the next candidates to host the VM would be M_{16}, M_{17} and M_{18}, all of which can access the input data block I_3 by traversing one network switch and are close to the other reducers executing in M_{13} and M_{14}. If any of M_{16}, M_{17} and M_{18} did not have available resources to host a new VM, then the algorithm would iteratively proceed to the next rack (M_7, M_8, M_9, M_{10}, M_{11} and M_{12}) and look for a physical machine to host the VM. Thus the algorithm tries to maximize locality even if the physical machines containing input data blocks are unavailable to host the VMs.

4.3 Reduce-input heavy Applications

Jobs that are reduce-input heavy read small sized map-inputs and generate large map-outputs that serve as the input to the reduce phase. For these type of jobs, reduce locality is more important than map-locality.

4.3.1 *Placing Reduce-input heavy data*

As map-input to these jobs is light, the map-locality of the data is not as important. Therefore, the map-input data can be placed anywhere within the cluster as it can be easily transferred to the corresponding VMs during map execution. The data placement algorithm chooses the physical machine with maximum free storage. The example in Figure 4(a) shows the placement of input data blocks for dataset L consisting of L_1, L_2 and L_3 on M_3, M_6 and M_7 which are chosen only based on storage availability, even though they are not closely connected.

4.3.2 *VM placement for Reduce-input heavy jobs*

Network traffic for transferring intermediate data among MapReduce VMs is intense in reduce-input heavy jobs and hence the set of VMs for the job should be placed close to each other. For an example job using the dataset, L, containing L_1, L_2, and L_3 in Figure 4(a), the VMs can be hosted on any set of closely connected physical machines, for instance, M_{10}, M_{11} and M_{12}. These machines are within a single rack and form a *2-club* (diameter of 2 with a single network switch). Although the map phase requires remote reads from the nodes storing the input data, M_3, M_6 and M_7, it does not impact job performance much as the major chunk of data transfer happens only during the reduce phase. In the reduce phase, as VMs are placed in a set of densely connected nodes, the locality of the reads is maximized, leading to faster job execution.

4.4 Complexity of Techniques

There are two key operations used in our algorithms – (1) finding a *k-club* of a given size with available resources and (2) finding a node close to another node in the physical cluster. As noted before, with typical data center hierarchical topologies, both of these operations are very efficient to compute. As a result our techniques scale well with increasing sizes of datasets or the cloud data center.

5. EXPERIMENTAL EVALUATION

We divide the experimental evaluation of Purlieus into two – first, we provide detailed micro-benchmarking on effectiveness of our data and VM placement techniques for each MapReduce job class on a real cluster testbed of 20 physical machines. Then, we present an extensive macro analysis with mix of job types and evaluate scalability of our approach on a large cloud scale data center topology through a simulator which is validated based with experiments on the real cluster. We first start with our experimental setup.

5.1 Experimental setup

Metrics: We evaluate our techniques on two key metrics with the goal of measuring the impact of data locality on the MapReduce cloud service – (1) *job execution time*: techniques that allow jobs to read data locally result in faster execution; thus this metric measures the per-job benefit of data locality, and (2) *Cross-rack traffic*: techniques that read a lot of data across racks result in poorer throughput [4]; this metric captures such characteristics of the network traffic.

Data Placement Techniques: We compare two data placement schemes – our proposed *locality and load-aware data placement (LLADP)* accounts for MapReduce specific job characteristics and estimated loads on servers while placing data as described in Section 4. In contrast, the *random data placement* (RDP) scheme does not differentiate between job categories and places data blocks in a set of randomly chosen physical machines that have available storage capacity. It also has no knowledge of the server loads (analogous to conventional MapReduce data placement). Note that both the locality-aware and random data placement schemes are rack-aware [11]; no two replicas of a given data block are placed on the same cluster rack for reliability purposes.

VM Placement Techniques: We compare five techniques:

- *Locality-unaware VM Placement(LUAVP)*: LUAVP places VMs on the physical machines without taking into consideration the locations of the input data blocks for the job. The LUAVP scheme does try to pick a set of least loaded physical machines for placing the VMs.

- *Map-locality aware VM placement (MLVP)*: MLVP considers locality of only the input-data blocks for the map phase and considers the current load and resource utilization levels of the machines while placing the VMs (load-aware).

- *Reduce-locality aware VM placement (RLVP)*: RLVP does not consider map locality, but it tries to improve reduce locality by packing VMs in a set of closely connected machines. It is also load aware.

- *Map and Reduce-locality aware VM placement (MRLVP)*: MRLVP is aware of both map and reduce locality and is also load aware.

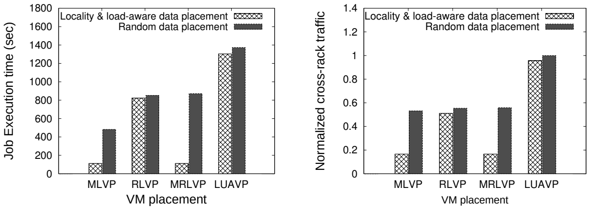
- *Hybrid locality-aware VM placement (HLVP)*: Our proposed HLVP technique adaptively picks the placement strategy based on type of the input job. It uses MLVP for map-input heavy, RLVP for reduce-input heavy jobs and MRLVP for map and reduce-input heavy jobs.

Key Comparison: The important comparison is between the combination of LLADP + HLVP (Purlieus proposal) with RDP + MLVP – analogous to traditional MapReduce. The other techniques help us understand the benefits of individual map or reduce locality as well as benefits gained from data vs. VM placement.

Cluster Setup: Our cluster consists of 20 CentOS 5.5 physical machines (KVM as the hypervisor) with 16 core 2.53GHz Intel processors. The machines are organized in two racks, each rack containing 10 physical machines. The network is 1 Gbps and the nodes within a rack are connected through a single switch. Each job uses a cluster of 20 VMs with each VM configured with 4 GB memory and 4 2GHz vCPUs. A description of the various job types and the dataset sizes is shown in Table 1. Each workload uses 320 map tasks. The *Grep* workload uses only one reducer since it requires little reduce computation while the *Sort* and *Permutation Generator* workloads use 80 reducers. The Hadoop parameter, *mapred.tasktracker.map.tasks.maximum* that controls the maximum number of map tasks run simultaneously by a task tracker is set as 5. Similarly, the *mapred.tasktracker.reduce.tasks.maximum* parameter is set as 5. Similar to typical data center topologies, the inter-rack link between the two switches becomes the most contentious resource as all the VMs hosted on a rack transfer data across this link to the VMs hosted on the other rack. For example, with 10 physical machines on each rack, and each physical machine hosting a nominal 8 VMs, 80 VMs (and thus, Hadoop nodes) on each rack will contend for the inter-rack link bandwidth of 1 Gbps. To simulate this contention in a more controlled environment that lets us accurately measure per-job improvements, we set the bandwidth of the inter-rack link to 100 Mbps while running one job at a time. The other alternative would be to run multiple jobs at the same time on the cluster, however, that would have made micro analysis on a per-job type basis tougher to evaluate.

Workload Type	**Job**	**Input data**	**Output data**
Map-input heavy	Grep: word Search	20 GB	2.43 MB
Reduce-input heavy	Permutation Generator	2 GB	20 GB
Map and Reduce-input heavy	Sort	10 GB	10 GB

Table 1: Workload types



(a) Job Execution Time

(b) Network Utilization

Figure 5: Map and Reduce-input heavy workload

5.2 Micro-benchmarking Results

We first present evaluation of our proposed techniques for various MapReduce job types.

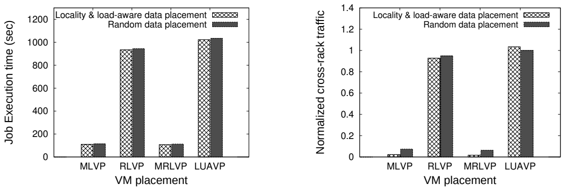
5.2.1 Map and Reduce-input heavy workload

In Figure 5, we study the performance for jobs that are both Map and Reduce-input heavy using the *Sort* workload on a dataset generated using Hadoop's RandomWriter. The job execution time in Figure 5(a) for map-and-reduce VM placement with locality and load-aware data placement (LLADP + MRLVP) shows the least value among all schemes with more than 76% reduction compared to RDP + MLVP. For data placement, MRLVP with RDP performs poorly indicating that *without a locality-aware data placement, it is hard to achieve high locality during VM placement* and therefore leads to higher job execution time. This justifies our coupled data placement and VM scheduling technique.

Also, RLVP does not perform well as it tries to consider only reduce locality. The LUAVP scheme places the VMs randomly without considering locality and therefore does not perform well either. An interesting trend here is that MLVP performs well with LLADP as the locality-awareness in data placement tried to place the data in a set of closely connected physical machines and hence, when the map-locality aware VM placement tries to place the VMs close to the input data, the reduce-locality is implicitly accounted for. These benefits can be explained by the trend in cross-rack traffic (normalized with respect to RDP + LUAVP) in Figure 5(b), showing 68% lesser cross rack reads when using LLADP + MRLVP compared to RDP + MLVP.

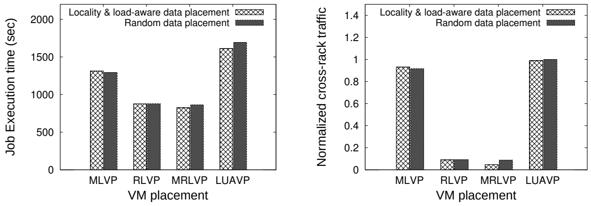
5.2.2 Map-input heavy workload

Next we evaluate data and VM placement for map-input heavy jobs using the *Grep* workload. Figure 6 compares our metrics with various schemes. In Figure 6(a), first notice that the job execution time for the locality-unaware VM placement (LUAVP) and reduce-locality aware VM placement (RLVP) schemes is much higher than that of map-locality aware (MLVP) and map-and-reduce locality aware (MRLVP) VM placements for both the random (RDP) and locality and load-aware data placement (LLADP) schemes. As map-input heavy jobs generate only small map-outputs and have little reduce traffic, the *techniques that optimize for map locality – MLVP and MRLVP perform much better than the reduce-locality only technique (RLVP) (up to 88% reduction in job execution time)*. The job execution time difference can be explained by cross-rack network traffic (Figure 6(b)), normalized with respect to



(a) Job Execution Time (b) Cross-rack traffic

Figure 6: Map-input heavy workload



(a) Job Execution Time (b) Cross-rack traffic

Figure 7: Reduce-input heavy workload

RDP + LUAVP, shows that map-locality awareness has a big impact. Lower cross-rack network traffic suggests that the data reads are more local to the rack, avoiding more than 95% of cross-rack traffic.

5.2.3 Reduce-input heavy workload

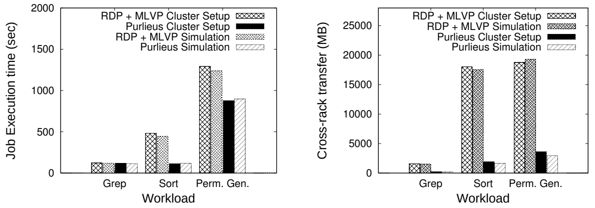
Figure 7 shows the performance for reduce-input heavy workload using a permutation generator job that generates and sorts the first 10 permutations of each record of a dataset generated by Hadoop's RandomWriter. We find in Figure 7(a) that RLVP and MRLVP have lower execution time for both the random (RDP) and locality-aware data placement (LLADP), having up to 32% faster execution time when compared to RDP + MLVP. Reduce-locality awareness in VM placement ensures that the reducers are packed close to each other and reduce traffic does not traverse a long distance on the network. Here, the underlying data placement scheme makes little impact as these jobs do not have large input data, so violating map locality does not cost much.

The LUAVP and MLVP schemes perform poorly since they do not capture reduce locality which is key for this reduce-intensive workload. A similar trend is seen for the ratio of cross-rack reads in Figure 7(b), where the (LLADP + MRVLP) technique has 10x+ higher number of reads within racks as compared to RDP + MLVP. **Summary**: *This micro-analysis demonstrates that data and VM placement techniques when applied judiciously to MapReduce jobs can have a significant impact on the job execution time as well as total datacenter traffic. To realize these benefits, the right technique needs to be applied for each MapReduce job type. Our Purlieus technique (LLADP + HLVP) identifies and uses the right strategy for each type of workload.*

5.3 Macro Analysis: Mix of workloads, Scalability and Efficiency

Following the per-job-type analysis, next we consider a mix of workloads and evaluate the scalability of the techniques with respect to the size of data center network and number of VMs in virtual MapReduce clusters using a mix of workload types.

For a thorough analysis at scales of 100s and 1000s of machines and with varying job, workload and physical cloud characteristics, we implemented a MapReduce simulator, called PurSim, similar to the existing NS-2 based MRPerf simulator [26]. However, unlike MRPerf, PurSim does not perform a packet-level simulation of the underlying network. Per-packet approach simulates every single packet over the network which makes it difficult to scale for even reasonably large workloads and cluster sizes. For instance, a per-packet simulator for a cluster size of 1000 hosts sending traffic at 1Gbps would generate 3×10^{10} packets for a 60 second simulation and simulating a million packets per second would take 71 hours to simulate just that one case [22]. Instead we use a *network flow* level simulation. Our discrete event simulator simulates the MapReduce execution semantics similar to the Hadoop implementation. The inter-node traffic is simulated in terms of network flows between the source-destination pairs. The simulation framework uses a data center of 1000 compute nodes with 1 Gbps network configured in the typical tree topology for the default setting. The performance metrics were averaged over the jobs executed during a 2 hour simulation period. By default, we use a mixed workload of jobs consisting of equal proportions of all MapReduce job types in Table 1. We use a 30 GB dataset for both the *Grep* and *Sort* workloads and a 2 GB dataset for *Permutation* workload. For the default setting, a total of 150 datasets were used, 50 for each of the job types and 3 replicas were created by default. The arrival rate of the jobs on the datasets is uniformly distributed from 200 to 2000 seconds.



(a) Job Execution Time (b) Cross-rack traffic

Figure 8: Simulator Validation

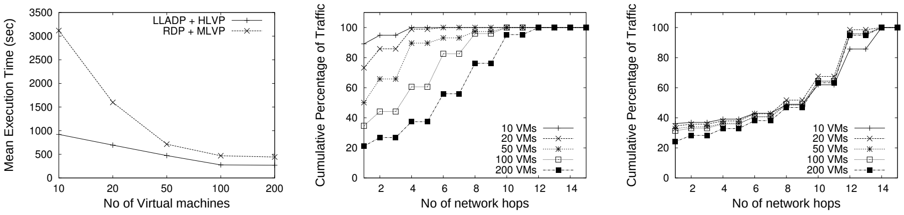
5.3.1 Simulator Validation

Before presenting our simulation experiments, we provide a validation of the simulator based on the experiments on our real 20 node cluster. To bootstrap the simulator, we used measurements obtained from the cluster experiments to configure simulator parameters, e.g. map and reduce compute times. We used the same settings from our cluster setup including the cluster network topology and workload characteristics in Table 1. As the key comparison is between the (RDP + MLVP) and Purlieus (LLADP + HLVP) schemes, we compare these two techniques for various job types.

In figure 8(a), we compare the job execution time of the two schemes for the three workloads. We find that for most cases, the execution time produced by the simulator is within 10% of the execution time obtained in our cluster experiments. The cross-rack transfer in Figure 8(b) shows that the simulator estimated cross-rack transfer matches closely with that of our cluster experiments, having less than 5% error in the cross-rack transfer estimated by the simulator. While not validated against large scale clusters, these low error rates when compared to our 20-node cluster experiments, provide good confidence in the quality of the simulator.

5.3.2 Mixed workload

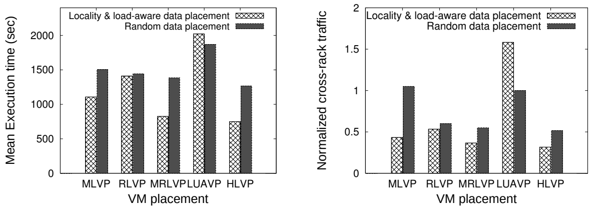
For our first macro analysis, we study the performance with a composite workload that consists of an equal mix of all MapReduce job categories with the default setting of 150 datasets and jobs using 20 VMs per job. Recall that Purlieus's HLVP decides on the placement policy based on the type of MapReduce job. For example, it uses RLVP for reduce-input heavy jobs and MRLVP for jobs that are both map and reduce heavy. The execution time in Figure 9(a) shows that HLVP works best for a mixed workload compared to all other VM placement policies. As discussed earlier,



(a) Job Execution time (b) CDF of Traffic with LLADP and HLVP (c) CDF of Traffic with RDP and MLVP

Figure 10: Varying number of Virtual Machines

a reduce-locality aware VM placement would lose map locality for map-input heavy jobs and a map-locality VM placement might lose reduce-locality while trying to achieve map-locality. While the map and reduce locality-aware VM placement could be a conservative policy for all types of jobs, it may not be needed in all cases and in fact may use valuable dense-collection of machines for jobs that do not need it. This explains the difference between HLVP and MR-LVP. – HLVP uses the right kind of resources for each job type. Overall, *HLVP with LLADP shows 2x faster execution time when compared to RDP + MLVP schemes* and a 9.1% improvement with most conservative policy of LLADP + MRVLP. Figure 9(b) shows the same trend with the normalized cross-rack traffic – LLADP + HLVP shows a lower cross-rack traffic (only 30.1%) compared to the RDP + MLVP. Overall, it is vivid that *with random data placement, it is hard to achieve a higher ratio of rack-local reads no matter what VM placement algorithm is used, thus validating our claim made in the Purlieus design.*



(a) Job Execution Time (b) Cross-rack traffic

Figure 9: Mixed workload

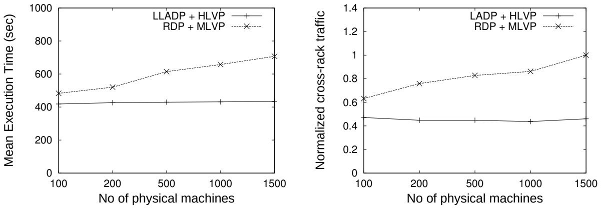
5.3.3 Impact of number of VMs

We study the impact of varying the number of VMs used for a given job in Figure 10 using the default PurSim setting. In figure 10(a), the number of VMs is varied from 10 to 200 and the average job execution time is compared. The job execution time decreases with increasing number of VMs but that decrease almost stops beyond a certain number of VMs (100 VMs in this case). The initial increase in number of VMs increases the computational parallelism and improves execution time. But as the number of VMs exceed a certain value, the reduce tasks get distributed across the network since not all of them can be placed on a set of closely connected machines (racks get exhausted). This reduction in data locality and increased network transmission time counters the improved parallelism. This also shrinks the advantage of Purlieus approach over RDP + MLVP. For instance, there is a performance gain of 2.3x in execution time while using 20 VMs and it drops down to a gain of 1.7x when 100 VMs are used. This is expected since when a large virtual cluster is provisioned, it is tough to provide both map and reduce locality. This impact can be further analyzed by visualizing the CDFs of number of network hops in both schemes with varying number of VMs in Figures 10(b) and 10(c). When number of VMs increase, there are more reads over longer network paths. However, we always find higher percentage of closer reads with (LLADP + HLVP) compared to (RDP + MLVP).

Overall, there are two key take-aways. First, Purlieus approach outperforms other approaches for varying sizes of virtual MapReduce clusters per job. Secondly, we notice that for a given job and cloud topology, there is a sweet-spot in the size of the virtual MapReduce cluster which gives the most bang for the buck. A tool that helps customers identify this would be very valuable.

5.3.4 Varying Network Size

In this experiment, we measure the job execution time and cross-rack traffic for various sizes of the cloud topology using 50 VMs for each job. The other parameters are based on PurSim's default setting. The job execution time in figure 11(a) is fairly constant for various network sizes with LLADP + HLVP. However, with RDP + MLVP, the data blocks gets distributed all over the network and with bigger clusters, the VMs are spread across the network and hence the reduce phase obtains poor locality leading to longer execution times. The normalized cross-rack traffic in Figure 11(b) is also indicative of the same trend. Thus, *Purlieus techniques work well with varying size of the cloud datacenter topology while conventional technique perform worse for larger network topologies.*

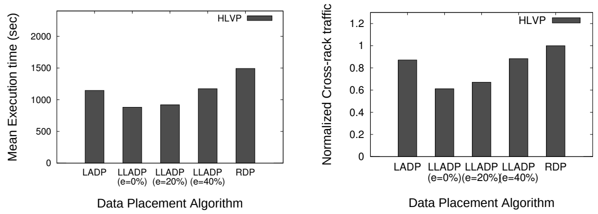


(a) Job Execution time (b) Cross-rack traffic

Figure 11: Varying Network Size

5.3.5 Impact of Load-aware Data Placement

Our next experiment evaluates the effectiveness of load-awareness in data placement. The experiments use the workload in the default PurSim setting using 20 VMs for each job. A good load-aware technique should make good decisions even with reasonably accurate estimates. We study the locality and load aware data placement (LLADP) with only locality-aware data placement (LADP) and random data placement (RDP). Figure 12 compares the LADP scheme with RDP and LLADP scheme for several load estimation error values, e. The estimation error, e directly corresponds to the percentage error in the estimation of the job arrival rates on the datasets. In figure 12(a), we find that without any load estimation error, LLADP $(e = 0\%)$ performs better than the load-unaware (LADP) and random placement (RDP) schemes. Also, we find that even with an estimation error of 20 % or 40%, the LLADP scheme performs better than the random and load-unaware (LADP) schemes. A similar trend is seen in Figure 12(b) for the cross-rack



(a) Job Execution time (b) Cross-rack traffic

Figure 12: Load Aware Data placement

traffic normalized with respect to (RDP + HLVP). It suggests that *even an approximate estimate of the arrival rate of the jobs on the datasets helps balance the expected load among physical machines and increases data locality*.

6. RELATED WORK

To the best of our knowledge, Purlieus, with its coupled data and VM placement, is unique in exploiting both map and reduce locality for MapReduce in a cloud. We briefly review some of the related work in this area. There have been several efforts that investigate efficient resource sharing while considering fairness constraints [34]. For example, Yahoo's capacity scheduler uses different job queues, so each job queue gets a fair share of the cluster resources. Facebook's fairness scheduler aims at improving the response times of small jobs in a shared Hadoop cluster. Sandholm et al [33] presented a resource allocation system using regulated and user-assigned priorities to offer different service levels to jobs over time. Zaharia et al. [24] developed a scheduling algorithm called LATE that attempts to improve the response time of short jobs by executing duplicates of some tasks in a heterogenous system. Herodotou et al. propose *Starfish* that improves MapReduce performance by automatically tuning Hadoop configuration parameters [31]. The techniques in Purlieus are complementary to these above mentioned optimizations. Recent work, *Mantri*, tries to minimize outliers by making network-aware task placement, task restarting and protecting the output of valuable tasks [4]. It also identifies that cross-rack traffic during the reduce phase is a crucial factor for MapReduce performance. However, without a locality-aware data placement scheme in *Mantri*, there are only limited opportunities for optimizations during task placement. Purlieus solves the fundamental problem of optimizing data placement so as to obtain a highly local execution of the jobs during scheduling, minimizing the cross-rack traffic during both map and reduce phases. As seen in evaluations, Purlieus benefits from its locality-aware data as well as computation placement.

A large body of work has explored the placement of applications in a virtualized data center to minimize energy consumption [36], perform load balancing [35] or perform server consolidation [32]. These approaches primarily focus on the *bin-packing* aspect and place applications (VMs) independent of the underlying data placement. Purlieus differs from these in terms of its consideration of both input and intermediate data locality for MapReduce. Recently, motivated by MapReduce, there has been work on resource allocation for data intensive application, especially in the cloud context [29, 17]. Gunarathne et al.[17] present a new MapReduce runtime for scientific applications built using Microsoft Azure cloud infrastructure services. Tashi [29] identifies the importance of location awareness but does not propose a complete solution. Tara [38] presents an architecture for optimized resource allocation using a genetic algorithm. Quincy [25] is a resource allocation system for scheduling concurrent jobs on clusters, but it considers only input data locality and does not optimize for locality of any intermediate data generated during job execution which is a key factor to scaling MapReduce in large data centers. Purlieus differentiates from these through its locality optimizations achieved for both input and intermediate data. Also, as discussed in Section 4, without an efficient underlying data placement, even a sophisticated locality-aware compute placement may not be able to achieve high data locality.

7. CONCLUSIONS

This paper presents Purlieus, a resource allocation system for MapReduce in a cloud. We present a system architecture for the MapReduce cloud service and describe how existing data and virtual machine placement techniques lead to longer job execution times and large amounts of network traffic in the data center. We identify *data locality* as the key principle which if exploited can alleviate these problems and develop a unique coupled data and VM placement technique that achieves high data locality. Uniquely, Purlieus's proposed placement techniques optimize for data locality during both map and reduce phases of the job by considering VM placement, MapReduce job characteristics and load on the physical cloud infrastructure at the time of data placement. Our detailed evaluation shows significant performance gains with some scenarios showing close to 50% reduction in execution time and upto 70% reduction in the cross-rack network traffic.

We plan to extend our work in two directions. First, for placement techniques we would like to capture relationships between datasets, e.g. if two datasets are accessed together (MapReduce job doing a *join* of two datasets), their data placement can be more intelligent while placing their blocks in relation to each other. Second, we plan to develop online techniques to handle dynamic scenarios like changing job characteristics on a dataset. While core principles developed in this work will continue to apply, such scenarios may use other virtualization technologies like live data and VM migration.

8. ACKNOWLEDGEMENTS

The first and third authors are partially supported by the National Science Foundation under NetSE program and CyberTrust program, as well as an IBM Faculty award.

9. REFERENCES

[1] B. Igou "User Survey Analysis: Cloud-Computing Budgets Are Growing and Shifting; Traditional IT Services Providers Must Prepare or Perish". Gartner Report, 2010

[2] *http://en.wikipedia.org/wiki/Loop_device*

[3] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[4] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha and E. Harris. Reining in the Outliers inMap-Reduce Clusters using Mantri. In *OSDI*, 2010.

[5] *http://en.wikipedia.org/wiki/Big-data*

[6] S. Babu. Towards Automatic Optimization of MapReduce Programs. In *SOCC*, 2010.

[7] *http://en.wikipedia.org/wiki/Clickstream*

[8] K. Kambatla, A. Pathak and H. Pucha. Towards Optimizing Hadoop Provisioning in the Cloud. In *HotCloud*, 2009.

[9] Cloudera. *http://www.cloudera.com/blog/2010/08/hadoop-for-fraud-detection-and-prevention/*

[10] K. Morton, A. Friesen, M. Balazinska, D. Grossman. Estimating the Progress of MapReduce Pipelines. In *ICDE*, 2010.

[11] Hadoop DFS User Guide. *http://hadoop.apache.org/*.

[12] T. Wood, P. Shenoy, A. Venkataramani and M. Yousif. Black-box and Gray-box Strategies for Virtual Machine Migration. In *NSDI*, 2007.

[13] Y. Chen, R. Griffith, J. Liu, R. H. Katz and A. D. Joseph. Understanding TCP Incast Throughput Collapse in Datacenter Networks. In *WREN*, 2009.

[14] Amazon Elastic MapReduce. *http://aws.amazon.com/elasticmapreduce/*

[15] Amazon Elastic Compute Cloud. *http://aws.amazon.com/ec2/*

[16] Amazon Simple Storage Service. *http://aws.amazon.com/s3/*

[17] T. Gunarathne, T. Wu, J. Qiu, G. Fox MapReduce in the Clouds for Science. In *CloudCom*, 2010.

[18] M. Cardosa, P. Narang, A. Chandra, H. Pucha and A. Singh. STEAMEngine: Optimizing MapReduce provisioning in the cloud. *Dept. of CSE, Univ. of Minnesota*, 2010.

[19] M. Al-Fares, A. Loukissas and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.

[20] C. Guo, H. Wu, K. Tan, L. Shiy, Y. Zhang, S. Luz. DCell: A Scalable and Fault-Tolerant Network Structure for Data Centers. In *SIGCOMM*, 2008.

[21] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, S. Sengupta. VL2: A Scalable and Flexible Data Center Network . In *SIGCOMM*, 2009.

[22] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.

[23] Hadoop. http://hadoop.apache.org.

[24] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI*, 2008.

[25] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[26] G. Wang, A. Butt, P. Pandey, K. Gupta. A Simulation Approach to Evaluating Design Decisions in MapReduce Setups. *MASCOTS*, 2009.

[27] R. J. Mokken. Cliques, clubs and clans. In *Quality and Quantity*, 1973.

[28] M. R. Garey, D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman. ISBN 0-7167-1045-5.

[29] M. A. Kozuch, M. P. Ryan, R. Gass et al. Tashi: Location-aware Cluster Management. In *ACDC*, 2009.

[30] K. Kambatla, A. Pathak, and H. Pucha. Towards optimizing hadoop provisioning in the cloud. In *HotCloud*, 2009.

[31] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, S. Babu Starfish: A Selftuning System for Big Data Analytics. In *CIDR*, 2011.

[32] G. Khanna, K. Beaty, G. Kar, and A. Kochut. Application performance management in virtualized server environments. In *NOMS*, 2006.

[33] T. Sandholm and K. Lai. Mapreduce optimization using dynamic regulated prioritization. In *ACM SIGMETRICS/Performance*, 2009.

[34] Scheduling in hadoop. *http://www.cloudera.com/blog/tag/scheduling/*.

[35] A. Singh, M. Korupolu, and D. Mohapatra. Server-storage virtualization: Integration and load balancing in data centers. In *IEEE/ACM Supercomputing*, 2008.

[36] A. Verma, P. Ahuja, and A. Neogi. pMapper: Power and Migration Cost Aware Placement of Applications in Virtualized Systems. In *ACM Middleware*, 2008.

[37] A. Phanishayee, H. Shah, E. Krevat, D. Andersen, G. Ganger, G. Gibson, B. Mueller, V. Vasudevan. Safe and Effective Fine-grained TCP Retransmissions for Datacenter Communication. In *SIGCOMM* 2009.

[38] G. Lee, N. Tolia, P. Ranganathan, R. Katz. Topology-Aware Resource Allocation for Data-intensive workloads. In *APSys*, 2010.