Interference and Locality-Aware Task Scheduling for MapReduce Applications in Virtual Clusters

Xiangping Bu
Department of Electrical & Computer Engineering
Wayne State University
Detroit, Michigan 48202
xpbu@wayne.edu

Jia Rao
Department of Computer Science
University of Colorado at Colorado Springs, Colorado
jrao@uccs.edu

Cheng-Zhong Xu
Department of Electrical & Computer Engineering
Wayne State University
Detroit, Michigan 48202
czxu@wayne.edu

ABSTRACT

MapReduce emerges as an important distributed parallel programming paradigm for large-scale applications. Running MapReduce applications in clouds presents an attractive platform-as-a-service usage model for enterprises. In a virtual MapReduce cluster, the interference between virtual machines (VMs) causes performance degradation of map and reduce tasks and renders existing data locality-aware task scheduling policy, like delay scheduling, no longer effective. On the other hand, virtualization offers an extra opportunity of data locality for co-hosted VMs. In this paper, we present a task scheduling strategy to mitigate interference and meanwhile preserving task data locality for MapReduce applications. The strategy includes an interference-aware scheduling policy, based on a task performance prediction model, and an adaptive delay scheduling algorithm for data locality improvement. We implement the interference and locality-aware (ILA) scheduling strategy in a virtual MapReduce framework. We evaluated its effectiveness and efficiency on a 72-node Xen-based virtual cluster. Experimental results with 10 representative CPU and IO-intensive applications show that ILA is able to achieve a speedup of 1.5 to 6.5 times for individual jobs and yield an improvement of up to 1.9 times in system throughput in comparison with four other MapReduce schedulers. It improves data locality of map tasks by up to 65%.

1. INTRODUCTION

MapReduce has become an important distributed parallel programming paradigm for applications with various computational characteristics in large-scale clusters [11]. It forms the core of technologies powering big IT businesses like Google, IBM, Yahoo and Facebook. Providing MapReduce frameworks as a service in clouds becomes an attractive usage model for enterprises [2]. A MapReduce cloud service allows users to cost-effectively access a large amount of computing resources without creating MapReduce frameworks of their own. Users are able to flexibly adjust the scale of MapReduce clusters in response to the change of the resource demand of their applications.

MapReduce services in clouds typically run in virtual clusters. This usage model raises two new challenges. First, interferences between co-hosted virtual machines (VMs) can significantly affect the performance of MapReduce applications. Although virtualization provides performance isolation to a certain extent, there is still significant interference between VMs running on a shared hardware infrastructure. A MapReduce cloud service has to deal with the interference coming from contentions in various hardware components, including CPU, memory, I/O bandwidth, and their joint effects.

In a virtual MapReduce cluster, the interference may cause variation in VM capacity and uncertainty in task performance [27], and ultimately impair the correctness and effectiveness of the MapReduce key components, such as the task scheduler, fault tolerance mechanism, and configuration strategy. Our experimental results show that interference could slow down a job by 1.5 to 7 times. Performance degradation of MapReduce jobs due to VM interference was also observed in Amazon EC2 [1, 27]. There were studies on mitigating VM interference in virtual clusters through dynamic resource allocation or interference-aware task scheduling [20, 9, 17]. However, the MapReduce framework further complicates the interference problem on virtual clusters. MapReduce cloud service requires mitigating VM interference while maintaining the framework's features, such as job fairness and task data locality.

The second challenge for MapReduce cloud services is preserving good data locality for tasks of each job. In a MapReduce framework, the task scheduler assigns each task to an available node "closest" to its input data to leverage the data locality. To achieve good data locality while preserving job fairness in shared MapReduce clusters, Zaharia *et al.* proposed a delay scheduling algorithm to postpone a scheduled job for a few seconds if it can not launch a local task [26]. Such locality-aware task schedulers largely assume that the tasks are short lived. Unexpected task slowdown caused by interference in virtual clusters may render them no longer effective.

The MapReduce framework running on a distributed file system offers several levels of data locality. A task and its input data can locate in the same server node (*node locality*), the same rack (*rack locality*) or in different racks (*off-rack*). Server virtualization adds one more layer of locality: tasks and their data being placed on the co-hosted VMs of the same physical server. We refer to this as *server locality*. Data exchange between co-hosted VMs is often as efficient as local data access because inter-VM communication within one physical server is optimized by Hypervisor and does not consume any network bandwidth. *Server locality is much easier to achieve than node locality, although they are expected to deliver similar level of performance.* When applied to virtual MapReduce clusters, existing task schedulers designed for physical clusters are not able to leverage this extra layer of data locality and lose the opportunity for achieving better performance.

In this paper, we present an interference and locality-aware (ILA) task scheduler to address the challenges in the provisioning of fair share MapReduce cloud services. There were recent studies on improving the performance of MapReduce applications in the cloud through resource allocation [22] or data and VM placement [18, 21]; see [18] for a concise survey. In contrast, ILA focuses on task scheduling optimization in a virtual MapReduce cluster. ILA relies on an application-level task scheduling strategy to adapt to changes of data and VM deployment and cloud resource allocation. It requires no modification for the underlying resource management. ILA can efficiently speed up the jobs by mitigating VM interference and preserving data locality. We summarize the contributions of this paper as follows:

1. We develop an exponential interference prediction model to estimate task slowdown caused by interference in the virtual MapReduce cluster. We also introduce a Dynamic Threshold policy to schedule tasks based on the prediction model.

2. We develop an Adaptive Delay Scheduling algorithm, which improves the Delay Scheduling algorithm [26] by adjusting delay intervals of ready-to-run jobs in proportion to their input size. The algorithm also takes into account data locality in all layers including the *server locality.*

3. We develop a meta scheduling strategy to integrate the interference-aware scheduling and locality-aware scheduling algorithms and implement the algorithm in an interference and locality-aware (ILA) scheduling system. We evaluated the efficiency of the system on a 72-node Xen-based virtual MapReduce cluster. Experimental results with representative CPU and I/O-intensive applications demonstrate that the ILA scheduler can achieve a speedup of 1.5-6.5 times for individual jobs and yield an improvement of up to 1.9 times in system throughput compared with four recently proposed task schedulers. It improves data locality among map tasks by up to 65%.

The rest of this paper is organized as follows. Section 2 introduces the background and motivation of ILA scheduling. Section 3 presents the system architecture. Section 4 shows the design of our ILA Scheduler. Evaluation setting and results are given in Section 5. Related work is discussed in Section 6. Section 7 concludes the paper with remarks on limitation and possible future work.

2. BACKGROUND AND MOTIVATION

2.1 Hadoop in Virtualized Environments

In this paper, we use Hadoop implementation of MapReduce framework as an example to illustrate the concepts of VM interference and data locality [3]. Hadoop partitions each job into a number of map and reduce tasks. Each map task runs map functions on one data block (64MB by default) of an input file. A reduce task receives intermediate results from data dependent map tasks and generates final results. A MapReduce framework consists of a *master* and multiple *slaves*. The *master* is responsible for management of the framework, including user interaction, job queue organization and task scheduling. Each *slave* has a fixed number of map and reduce slots to perform tasks. The job scheduler located in the *master* assigns tasks according to the number of free task slots reported by each *slave* through a heartbeat protocol.

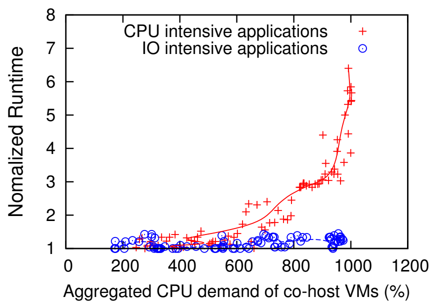
Hadoop running on a distributed file system HDFS assumes that the data storage is co-located within the compute cluster. MapReduce framework can exploit task locality without incurring extra management overhead. In the ILA framework, each *slave* is deployed on one VM with attached local disk image. It acts as both compute and data node. The HDFS was built across all the VMs. There exist other storage architectures in cloud environments. Amazon used distinct infrastructures for storage and compute. It is not suitable for MapReduce applications due to the requirement of loading data from a storage server to HDFS before the job execution and keeping large duplicated datasets during the execution. Recent works proposed several storage infrastructures to enable local data access for compute cluster and enhance the performance of MapReduce applications [18, 21]. The ILA framework helps deal with the interference and locality challenges in different storage architectures.

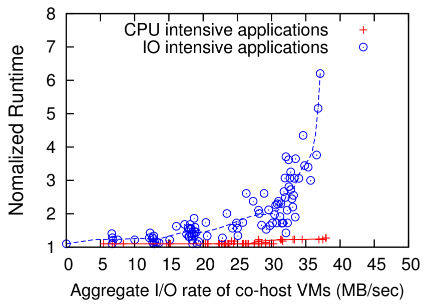
2.2 Virtual Machine Interference

Virtual cluster is the most common platform for cloud computing services. When multiple VMs are sharing hardware resources, the performance of their hosted applications may degrade due to imperfect VM isolation. We illustrate this problem using 5 Xen VMs deployed on a physical server for the execution of a benchmark of 10 representative CPU- and I/O-intensive applications; see Table 3 in Section 5 for their computation and I/O characteristics. Each *slave* VM was configured with 3 VCPUs, 2GB memory and with 3 map slots and 1 reduce slot. The VMs competed for 10 physical cores and one shared disk. One of the VMs executed the benchmark applications one by one and profiled the execution time for each application task. The other co-hosted VMs ran randomly selected applications from the benchmark as background jobs. Figure 1 and Figure 2 show the task completion time of each job, which is normalized to that of the task running alone on a dedicated VM. In Figure 1, the total CPU demand of co-hosted VMs is represented in the percentage of one physical core. We can see that, for CPU-intensive applications, there is no significant slowdown until the background demand reaches the capacity of 800 (8 cores). It is expected that I/O-intensive applications were insensitive to the total amount of CPU demands. Similarly, Figure 2 shows an exponential increase of the completion time of I/O intensive applications with the aggregated I/O traffic from co-hosted VMs. Previous works mitigated VM interference through dynamic resource allocation or interference-aware scheduling [9, 17, 20]. But applying their approaches in virtual MapReduce clusters may degrade system performance greatly due to the unawareness of MapReduce's features, like job fairness and data locality.

2.3 Data Locality

Recall that a virtual MapReduce cluster defines data locality in four layers: *node locality*, *server locality*, *rack locality*, and *off rack*. We show their respective effect on performance using 24 VMs deployed on 12 physical machines with 2 VMs on each. Each VM was allocated sufficient resource to eliminate interference effect. The physical machines were installed on 2 racks, connected by a 100 Mbps ethernet switch (for the purpose of creating network contention scenarios).





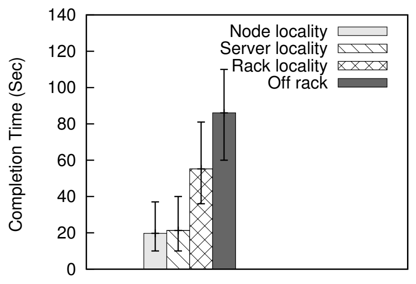


Figure 1: Effect of CPU interference. Figure 2: Effect of I/O interference. Figure 3: Effect of data locality.

We ran TeraSort application in the benchmark with 120 map tasks over 12GB input data. From Figure 3 we can see that tasks with *server locality* would finishe in approximately the same time as those with *node locality*. However, tasks with *rack local* and *off rack* data access could take as long as 3x and 4x time to complete, respectively. The non-local data access dramatically degraded the performance.

There were many task scheduling algorithms designed to preserve task data locality for MapReduce applications in physical clusters. When they are applied to virtual clusters, the data locality can not be maintained effectively due to the presence of VM interference. Most of the existing approaches assume that the tasks are largely short lived and the task slots are not occupied for too long by any job. Thus, for a given task to be run, even the target nodes with local data are not available, they are assumed to be free up soon. The scheduler can always launch local tasks for each job with a few seconds delay. In a virtual MapReduce cluster, VM interference could prolong short-lived tasks and render the data locality policy ineffective. To demonstrate this issue, we built a virtual cluster with 24 VMs on 6 physical servers, each with 4 VMs deployed. For comparison, we also built a physical cluster with 24 physical machines. The clusters were run with a Hadoop framework, which deployed the delay scheduling algorithm [26] to enhance data locality. Table 1 shows that for the physical cluster, the approach can achieve 98.4% *node locality* for a total of 120 tasks. In contrast, in the virtual cluster, the most beneficial *node* and *server locality* are reduced to 72.3% and 2%, respectively. As a result, we observed 100% slowdown in job completion time. Thus it requires a specifically designed task scheduler for MapReduce virtual clusters.

Table 1: Degrees of data locality in different clusters

Locality	Node	Server	Rack	Off Rack	Time
Physical	98.4%	N/A	1.6%	0.0%	170 sec
Virtual	72.3%	2%	18.3%	7.2%	342 sec

3. SYSTEM ARCHITECTURE

The ILA scheduler works in a Hadoop virtual cluster. Figure 4 illustrates the architecture of the target system. The cluster consists of a number of physical servers, each of which has the same virtualized environment. Multiple VMs are allocated onto each physical server hosting running applications, supported by Hadoop HDFS. Hadoop framework is deployed on top of the virtual cluster with a single *master* and multiple *slaves*. Each *slave* is configured to run within one VM and the *master* is deployed on a dedicated physical machine with secondary backup.

The core of ILA-based task management is located in the *master*, consisting of four major components: 1) the *Interference-Aware Scheduling Module* (IASM) to mitigate the interference between tasks running on co-hosted VMs with the help of an interference prediction model; 2) the *Locality-Aware Scheduling Module* (LASM) maintains good data locality for map tasks by using Adaptive Delay Scheduling algorithm; 3) the Task Profiler estimates the task's demand of each job and feeds task information to IASM and LASM modules; 4) the ILA scheduler instructs IASM and LASM modules to conduct interference-free high-locality task management. To collect the running states of the servers, we deployed a VM resource monitor in each VM and a physical resource monitor in each physical server. They send resource consumption status to ILA scheduler periodically.

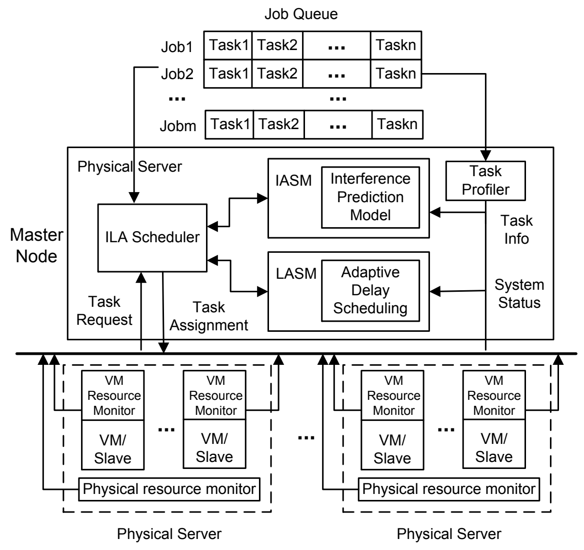


Figure 4: System Architecture.

4. ILA SCHEDULER DESIGN

4.1 Interference Prediction Model

In this section, we present a model to characterize the impact of the interference. We focus on the CPU and I/O bandwidth resources. The CPU-bound and I/O-bound workloads are the most common workloads for MapReduce clusters.

Nonlinear prediction model. On the application level, the interference can be perceived as the performance variation, including job runtime [28] and I/O throughput [9]. For generality, instead of using the absolute completion time, we considered the task slowdown rate (S) as the prediction target, which is defined on the task's real completion

Table 2: System metrics

	Parameters
System CPU	c_u: Local CPU usage in DomU c_a: Aggregated CPU usage of co-hosted VMs c_0: CPU usage in Dom0
System I/O	r_u: Local read rate in DomU w_u: Local write rate in DomU r_a: Aggregated read rate of co-hosted VMs w_a: Aggregated write rate of co-hosted VMs io_u: I/O utilization of physical server
Task	t_c: Average CPU demand t_r: Average read rate t_w: Average write rate

time (T_{real}) over the run time without interference (T), $S = T_{real}/T$. Such normalization helps the model deal with applications with different magnitudes in completion time.

The interference comes from two main sources : co-existed tasks in the same scheduled VM and the co-hosted VMs on the same physical server. Their impact on performance also varies with the application's demand. The prediction result should depend on the characteristics of the scheduled tasks as well as the resource consumption status of the target VM and co-hosted VMs on the same physical server. In an Xen-based environment, the privileged domain (*Dom0*) has direct access to hardwares. It is in charge of resource management of all guest domains (*DomU*). All of the effects should be included in the model. We selected CPU usage, disk Read/Write rate and I/O utilization to represent task demand and system status. These parameters are selected through covariance ranking and statistical hypothesis testing in order to keep the model simple and accurate. These eleven performance-critical parameters are listed in Table 2.

We first constructed separate interference prediction models for pure CPU-bound and I/O-bound applications. As shown in Figure 1, the characteristic of the data points is fit for an exponential curve. We constructed an exponential interference model for pure CPU-intensive application as follows:

$$\hat{S}_{cpu} = \alpha_{cpu} \exp(\gamma_t t_c + \sum_{i=1}^{3} \gamma_i CPU_i + C_{cpu}) + C_1, \quad (1)$$

where the task performance depends on its own CPU demand as well as all the CPU relative metrics listed in Table 2, represented as CPU_i, including the CPU usage of the scheduled VM, co-hosted VMs and *Dom0*. γ and C represent the coefficient and constant in the model.

Similarly, as shown in Figure 2, the slowdown rate of I/O-bound applications demonstrate an exponential relationship with background I/O rate. We constructed a non-linear exponential model for pure I/O-bound applications as follows:

$$\hat{S}_{io} = \beta_{io} \exp(\tau_{tr} t_r + \tau_{tw} t_w + \sum_{i=1}^{5} \tau_i IO_i + \tau_0 c_0 + C_{io}) + C_2, \quad (2)$$

where task performance is estimated based on its read and write I/O demands as well as all the I/O relative metics listed in Table 2, represented as IO_i, including Read/Write I/O throughout of the scheduled VM, the co-hosted VMs and the physical I/O utilization. τ and C represent the coefficient and constant in the model. Notice that the CPU usage of *Dom0* was also introduced into the model. This is crucial because all the requests from guest VMs are routed through *Dom0*. Handling a large number of I/O requests on behalf of guest domain will consume substantial CPU resources in *Dom0*.

For general applications, both of the CPU and I/O resource can affect their performance. We introduced the final general interference prediction model based on the two special models above. We constructed a linear model to quantify the joint impact of CPU and I/O resource on performance, as follows:

$$\hat{S} = \alpha \hat{S}_{cpu} + \beta \hat{S}_{io} + C_3, \quad (3)$$

The experiments in Section 5.4 show that the general model can achieve as high as 90% prediction accuracy. It brings 10%-15% improvement over the linear and quadratic models used in previous works [9].

Model Training. The model was initially constructed through offline training. For generality, we selected 10 applications shown in Table 3 for interference profile. Each application was running on one VM with various workloads on the same VM and co-hosted VMs. There were 6 VMs deployed on the same physical server. We developed two kinds of workload generators to generate CPU and I/O-bound workloads. The CPU workload generator conducts a set of arithmetic operations in a loop with variable time intervals. The I/O workload generator repeatedly reads from or writes to a file, which is much larger than the allocated memory to avoid OS caching effect. Both generators are able to issue workloads with any level of intensities by adjusting the length of sleep interval between each iteration. We also created 200 workload combinations by randomly selecting real applications in Table 3 as the background applications in profiling the interference. All the required metrics were collected during the experiments and used as the input data for modeling process.

In the nonlinear modeling process, we used the *Gauss-Newton* algorithm [10] to generate the coefficients that minimize sum of squared errors (SSE). The *Gauss-Newton* method is an interactive process that gradually updates the parameters to obtain the optimal solution. We also employed a stepwise algorithm [12] to simplify the model as much as possible. This stepwise process repeatedly adds or removes possible variables from the equation and evaluates the new re-fitting models.

Online Model Adaptation. Although the proposed prediction model is general for all kinds of applications, it keeps updating in order to achieve more accuracy for current applications. The time for modeling process is less than 2 seconds on a 3.0 GHz Inter Xeon processor. It can be dynamically re-calibrated when the accuracy is not acceptable with negligible overhead. The *Guass-Newton* method will be triggered whenever there are k new observations, k is set to 100 in this work. Thus the model can be easily adapted to a new cloud platform with different applications, virtual machines, operation systems and cloud infrastructures.

4.2 Data Locality Improvement

The impact of data locality on performance is difficult to predict because it involves the status of multiple levels of network nodes, including VMs, physical servers and switches. Instead of using explicit models, we propose an heuristic approach, namely Adaptive Delay Scheduling, to improve data locality. Compared with Delay Scheduling, the new approach is much more efficient and suitable for the virtualized environment.

Fair and Delay Scheduling. In practice, sharing a cluster between multiple users is more common and highly beneficial than dedicated clusters due to low building cost and data consolidation. We build our new scheduler on top of the existing fair scheduler. Briefly, at each scheduling interval, the fair scheduler sorts all the jobs according to their running tasks. It always assign available compute node to the job that is farthest below its fair share [26]. However, such strict scheduling order may conflict the data locality. The scheduled job may not find a free node to launch a local task.

Delay scheduling is a simple but effective approach to improve locality by temporarily relaxing fairness [26]. The key idea is when the scheduled job can not launch a local task on the available node, the scheduler will delay this job and skip to process the next one, until the delayed job find a free node to run local tasks or the accumulated delay time exceeds predefined intervals. There are multiple wait time thresholds for different levels of data locality. For example, in the latest version of Hadoop, the default maximum wait time for a *node local* task (T_{node}) is 5 seconds, after which the scheduler will try to launch a *rack local* task. The default extra wait time for a *rack local* task (T_{rack}) is also 5 seconds. When the wait time goes beyond $T_{rack}+T_{node}$, the scheduler will launch any task of the delayed job without considering the data locality.

Adaptive Delay Interval. The Delay Scheduling approach delays all the jobs for the same amount of time as long as they do not have local data access. However, the impact of data locality varies with the task's input file size. For jobs with small input file, their performance are insensitive to data locality, as shown in Section 5.3. But the scheduler forces them to take unnecessary delay to achieve high data locality. In practice, CPU-intensive applications, such as machine learning applications, usually have small input file. Even for data intensive applications, their tasks' input file size could also be changed through specific job configuration. We propose an adaptive delay scheduling algorithm with the delay interval proportional to the task input file size, defined as follows. Note that the HDFS block size is the largest unit for each map task, which is treated as the upper bound for task input file size.

$$\hat{T}_{ij} = \begin{cases} 0 & If \;\; F_j/F_b \leq 0.01; \\ F_j * T_i/F_b & Otherwise, \end{cases} \tag{4}$$

where T_i is the maximum wait time for locality level i and $\hat{T}_{ij}$ is the actual wait time for job j. F_j represents the input file size of the tasks in job j and F_b represents the HDFS block size. When the input size less than 1% of the block size, the scheduler will launch the task without any delay.

Server Locality-Aware Scheduling. The virtualized environment adds one more layer in network topology: the co-hosted VMs. We defined it as "server local" if the data is not on the compute VM but on the same physical server. The inter-VM communication within one physical machine is more efficient than the cross-machine communication. It will not be affected by the outside network traffic because the communication is optimized by Hypervisor. As shown in Section 2.3, the *server local* task performs closely to the *node local* task. Thus we set a small delay interval for *server locality*, 0.5 second in this work. After failing to find a *node local* task, the scheduler will quickly search for a *server local* task instead of searching for a *rack local* one.

The *server locality* information is usually unavailable in virtual cluster. We designed several methods to detect the VM's physical location. Users can input the VM deployment information through XML configuration file. The framework can also automatically generate the information by using *traceroute* from each VM to locate their physical hosts. The first hop is always the *Dom0* or Hypervisor process for the physical server. The information may also be provided by specially desinged management system, as used in [18].

Improvement Analysis. We explore how much the performance improves if we try to achieve both *node* and *server locality* instead of only the former one. It is assumed that the cluster consists of N physical servers, with M VMs per server. Let P_j denote the number of nodes on which job j has local data, and let Q_j denote the number of physical servers on which the P_j are deployed. The probability $\mathbb{P}(Q,P)$ of that all of the P VMs deployed on Q physical servers is as follows:

$$\mathbb{P}(Q,P)=\begin{cases} \binom{N}{Q}\binom{MQ}{P}/\binom{MN}{P} & If \;\; Q=\lceil\frac{P}{M}\rceil; \\ \frac{\binom{N}{Q}\binom{MQ}{P}}{\binom{MN}{P}}-\binom{N}{Q}\sum_{i=\lceil\frac{P}{M}\rceil}^{Q-1}\frac{\binom{Q}{i}}{\binom{N}{i}}\mathbb{P}(i,P) & Otherwise, \end{cases}$$
$$where \qquad M,N,P,Q\in\mathbb{Z}, P\in[1,NM], Q\in[\lceil\frac{P}{M}\rceil,N]. \tag{5}$$

Thus, for job j, the expected value of occupied physical server $\bar{Q}_j$ is:

$$\bar{Q}_j = E[Q_j] = \sum_{i=\lceil P_j/M\rceil}^{N} \mathbb{P}(i,P_j) * i, \tag{6}$$

where $\lceil P_j/M\rceil$ serves as a lower bound of the possible number of physical machines and N is the upper bound; see Appendix A for the proof.

Let $p_j = P_j/NM$ be the fractions of nodes that can launch *node local* tasks for job j. Thus, $q_j = \bar{Q}_j M/NM = \bar{Q}_j/N$ will be the fractions of nodes that can launch *server local* tasks for job j. We denote K as the number of skipped schedule intervals for job j in order to achieve good data locality. If job j has been waiting for K_n skips, the probability that it does *not* find a *node local* task is $(1-p_j)^{K_n}$. Similarly, after K_s skips, the probability that it does *not* find a *server local* task is $(1-q_j)^{K_s}$. If we want to achieve the same level of data locality, we have $\frac{K_n}{K_s}=\frac{\ln(1-q_j)}{\ln(1-p_j)}$. For our tested cluster with 12 physical servers and 6 VMs per sever, if the expected data locality is 99% and p_j is 10%, K_n/K_s will be 5.99. If the expected data locality is 95% and p_j is 40%, K_n/K_s will be 5.86. Since *node local* and *server local* tasks have close performance, our *server locality* aware scheduling approach will greatly reduce the delay time without compromising performance.

4.3 System Monitoring and Task Profiling

The scheduling decision is based on system resource consumption status and task resource demands. We deployed one VM resource monitor in each VM and one physical resource monitor in each physical server. We used the standard Xen tool *Xentop* to monitor the CPU usage of *Dom0* and each guest domain. Physical server I/O utilization and Read/Write I/O rate of each VM were measured via Linux *iostat* tool. We modified the *TaskTracker* in each *slave* (VM) to collect the resource consumption status. The information was sent to the *JobTracker* located in *master* node through periodical heartbeat operations. The resource information of physical servers was sent to *JobTracker* via TCP connections.

There is no standard tool to directly estimate the resource demand of an incoming task. We estimated the information through task profiling. For a MapReduce job, it usually consists of many small tasks. The tasks mostly have the same resource demand because they are often run in a data partition model for large problems. We can estimate the task demand of job j $\hat{D}_j$ based on the measured demand of the finished ones D_j, as follows:

$$\hat{D}_{tj} = \begin{cases} init & If \quad t = 0; \\ D_{0j} & If \quad t = 1; \\ \alpha * D_{(t-1)j} + (1-\alpha) * \hat{D}_{(t-1)j} & Otherwise. \end{cases} \tag{7}$$

The value $\hat{D}_j$ is used to estimate the demand before the task running and the actual demand D_j measured after the completion is used to update the estimation for subsequent tasks. The footnote t represents the update time interval. Thus $\hat{D}_{tj}$ represents the estimated demand of all the tasks in job j scheduled during the interval t. Initially, when there is no finished task, i.e. $t = 0$, we set the *init* demand to a high value to avoid interference. We used a decayed model to estimate the task demand, which makes the new observations more relevant in prediction than old ones. We set α to 0.8 in this work. In this paper, we modified *JobTracker* to collect the information. The consumed CPU time and the read/written file size of each task were obtained using MapReduce *CPU* and *FileSystem* Counter, respectively.

4.4 ILA Scheduling

ILA scheduler performs the interference and locality-aware scheduling operations on top of the fair scheduling. At each interval, it selects a job from a wait queue sorted according to the job's fairness. However, occasionally, the goal of mitigating interference and maintaining data locality may conflict with each other. ILA scheduler always considers the interference mitigation first due to the following two reasons: 1) VM interference causes much more performance degradation than remote data access. Scheduling a non-local task only affect the individual task. In contrast, VM interference may affect not only the scheduled task but also all the tasks running on the same VM or physical server. 2) No interference is a precondition for achieving good data locality. Any unexpected task slowdown would make the data locality policy ineffective.

Interference-Aware Scheduling. Whenever ILA scheduler receives a task request from one VM, it collects the VM's resource status as well as the information of its co-hosted VMs and its physical host. Then it searches down the sorted job list and gets the task's profile of the first job. Taking those as inputs, the prediction model returns a quantitative value to evaluate the interference. Previous works employed a Min-Min heuristics [9] to schedule the tasks based on the interference prediction. The scheduler always assign the "least-interference" task to available VMs. However, such tasks may still lead to the severe contention if all the tasks are resource intensive.

In this paper, we propose a scheduling strategy based on slowdown rate thresholds, as shown in Algorithm 1. We set a static threshold H to 1 by default, which means no task slowdown due to interference. When a free node $Node_i$ requests a task, ILA scheduler collects the system information and evaluates the tasks $Tasks_j$ of each job on the sorted list. ILA only evaluates a job once using its current estimated task demand $\hat{D}_t$, instead of testing all individual tasks in the job, since all the tasks in a job have similar demand. If the predicted slowdown rate $\hat{S}_j$ is not higher than H, ILA scheduler accepts job j and stop searching. Otherwise it refuses the job and processes the next one. If eventually no job satisfies the condition, the scheduler rejects $Node_i$ and lets it wait for resource releasing. Such static threshold method could lead to many idle slots and degrade the performance. For example, if current H is 1 and the number of running slots Z_R on the sever is 2, we assume that the completion time slowdown rate of all the running tasks in the same server is no higher than current H. Then the server's throughput is $Z_R/H = 2$. If we increase H to 2, which makes Z_R to go up to 6, we have the throughput $6/2 = 3$. Although all the tasks are slowed down, the throughput is improved. There is a tradeoff between individual task performance and the job's degree of parallelism.

We introduced a dynamic threshold H_d to deal with the problem. ILA scheduler tries to increase the parallelism if the number of idle slots Z_I becomes more than one for k seconds (20 seconds in our experiments). Within one physical server, the scheduler compares the current throughput Z_R/H_d with the predicted throughput if adding one more task $(Z_R + 1)/\hat{S}_j$. If the latter is larger, ILA schedules the task in and updates H_d as $\hat{S}_j$. H_d should not be increased endlessly. It will be decreased gradually if $\hat{S}_j$ is no larger than H_d, which means H_d has become over set.

Algorithm 1 Interference-Aware Scheduling

```
1: When a hearbeat is received from a free node $n$:
2: Collect system information $N_{info}$;
3: Given a job $j$
4: Fetch task's profile $Tasks_j$;
5: Predict the slow down rate $\hat{S} = Model(N_{info}, Tasks_j)$;
6: Get the number of running slots $Z_R$ and idle slots $Z_I$
7: if $Z_I > 1$ for $k$ seconds then
8:     // use dynamic threshold
9:     if $(Z_R + 1)/\hat{S}_j > Z_R/H_d$ and $\hat{S}_j > H_d$ then
10:        update $H_d = \hat{S}$ and return the accepted job $j$;
11:    else
12:        if $\hat{S}_j <= H_d$ then
13:            update $H_d = min(H_d - 1, \hat{S}_j)$ and return the accepted job $j$;
14:        else
15:            reject job $j$
16:        end if
17:    end if
18: else
19:    $H_d = H$; //use the predefined threshold
20:    if $\hat{S} <= H$ then
21:        return the accepted job $j$
22:    else
23:        reject job $j$;
24:    end if
25: end if
```

Locality-Aware Scheduling. The job that has passed through interference check is sent to LASM. The module searches all the tasks in the job and selects one whose input data is deployed closest to the requesting VM. We define the level of data locality according to the corresponding network hierarchy: L_0, L_1, L_2 and L_3 represent *node local*, *server local*, *rack local* and *off rack*, respectively. L_j denotes the maximum allowed locality level for job j. L_{jmin} denotes the minimal achievable locality level among all the tasks in job j given the requesting VM. If L_{jmin} is no higher than L_j, ILA scheduler accepts the task. Otherwise, ILA skips the job's scheduling unless its accumulated wait time W_j becomes lager than delay thresholds. $\hat{T}_{ij}$ denotes the wait time for locality level i for job j, $i \in [0, 2]$. No delay is needed in L_3. The locality-aware scheduling algorithm is shown in

Algorithm 2.

Algorithm 2 Locality-Aware Scheduling

```
1: System maintains four variables for each the job $j$:
2: maximum allowed level $L_j$; accumulated wait time $W_j$;
3: task input file size $F_j$;
4: the delay interval $\hat{T}_{ij}$ of each level $i$;
5: Get job $j$ from Interference-aware scheduling module;
6: The free VM is $vm_n$;
7: if $F_j/F_b <= 0.01$ then
8:     return any unlaunched task $t$ in job $j$ ;
9: else
10:    In job $j$, find the task $t$ with the minimal locality level for
       $vm_n$; // the task whose input file located “closest” to $vm_n$
11:    if $L_{jmin} <= L_j$ or $W_j >= \sum_{l=L_j}^{(L_{jmin}-1)} \hat{T}_{lj}$ then
12:        set $W_j = 0$
13:        $L_j = L_{jmin}$; // reset $L_j$ as the recently accepted level
14:        return the accepted task $t$ in job $j$;
15:    else
16:        reject job $j$ and update $W_j$
17:    end if
18: end if
```

In this algorithm, each job's maximum allowed locality level L_j is initialized to 0, i.e. the *node locality.* At each scheduling interval, L_j is reset to the locality level of the last accepted task in job j. If the scheduler can not find a sufficiently “close” task, the job only needs to wait for the cumulative delay interval, which is calculated from L_j to the minimal achievable level L_{jmin}, instead of from level 0. For example, if $L_j = 1$ and $L_{jmin} = 2$, the job only needs to wait for the time of T_{1j}, instead of $T_{0j} + T_{1j}$. This strategy tends to reduce the unnecessary delay for the jobs, for which the low locality levels are really difficult to achieve. We set the *level 0* delay to a very small value, 0.5 second. The delay intervals of other levels were all set to 5 seconds.

ILA Scheduling. The ILA scheduling algorithm is shown in Algorithm 3. At each scheduling interval, the scheduler sorts all the jobs according to their fair shares. The job that is farthest below its fair share obtains the free node first. Whenever a task request comes, ILA scheduler searches the sorted list and select the first job whose tasks do not cause interference. Then it searches the job's task list and pick a task that has the “closest” data access. If failing to find a satisfactory task, the scheduler rejects the node and lets it wait for the next scheduling interval.

Algorithm 3 ILA Scheduling

```
1: System maintains the job queue $Q$;
2: When a hearbeat is received from a free node $n$:
3: Collect system information $N_{info}$;
4: Sort jobs in $Q$ according to the fairness policy;
5: for each job $j$ in $Q$ do
6:     $job = IASM(job_j, N_{info})$;
7:     if $job$ == null then
8:         skip current job $j$, process the next one;
9:     else
10:        $task = LASM(job, n)$;
11:        if $task$ == null then
12:            skip current job $j$, process the next one;
13:        else
14:            assign $task$ to the node $n$;
15:            break the loop;
16:        end if
17:    end if
18: end for
19: if $task$ == null then
20:     reject node $n$;
21: end if
```

Table 3: A summary of MapReduce benchmarks

Name	Type	Introduction
TeraSort	I/O	Sort the input data into a total order
TeraGen	I/O	Generate and write data into system
Grep	I/O	Extract matching regular expression
RWrite	I/O	Random write words into log file
WCount	I/O	Count words in the input file
PiEst	CPU	Estimate Pi using Monte Carlo method
Bayes	CPU	Contruct Bayes Classifier on input data
Kmean	CPU	Cluster analysis using K-mean method
Canopy	CPU	Cluster analysis using Canopy method
Matrix	CPU	Matrix add and multiplication

5. EVALUATION

5.1 Experimental Setup

We evaluated the ILA scheduling framework in a 72-node Xen-based private virtual cluster, which consists of 12 physical servers, each configured with 12 CPU cores, 32GB memory and one 500GB disk. Each server hosted 6 VMs and each VM was configured with 2 VCPUs and 2GB memory. The 6 VMs were configured to compete for 10 cores and one shared disk. The virtual cluster spanned 2 racks and was connected by a 1Gbps Ethernet.

We installed a modified version of Hadoop 0.20.205 equipped with ILA scheduler, system resource monitors and task profilers. Based on hardware capacity, we configured each *slave* with 2 map slots and 1 reduce slot, for a total of 144 map slots and 72 reduce slots in the cluster. The HDFS block size was set to 128 MB to improved performance according to a Facebook's report [26]. All other parameters were set to their default configurations.

We evaluated ILA scheduler using 10 MapReduce applications, most of which were widely used in evaluations of MapReduce framework by previous works [21, 15, 7, 27, 26, 8]. Table 3 shows their main characteristics. The machine learning applications were from mahout project [4]. In the experiments, we compared the performance of ILA scheduler with 4 other main competitors in practical use: 1) *PureFair scheduler* conducts fair scheduling using greedy method to maintain data locality, i.e. always selecting the “closest” task from the scheduled job without any delay [11, 26]; 2) *Delay scheduler* uses delay scheduling algorithm to achieve good data locality by slightly compromising fairness restriction [26]; 3) *Longest Approximate Time to End (LATE) scheduler* improves MapReduce applications' performance in heterogenous environment, like virtualized environment, through accurate speculative execution [27]; 4) *Capacity scheduler*, introduced by Yahoo, supports multiple queues for shared users and guarantees each queue a fraction of the capacity of the cluster [6]; We also compared two variants of ILA: 1)*Interference-Aware Only (IAO) scheduler* only conducts interference-free scheduling with the help of IASM, but uses greedy method to maintain data locality; 2) *Locality-Aware Only (LAO) scheduler* only uses LASM to conduct Adaptive Delay Scheduling to improve data locality without considering VM interference.

5.2 Performance of ILA Scheduler

We evaluated ILA scheduler through a set of macro benchmarks based on the workload trace from Facebook reported in [26], according to which, the job size, in terms of number of map tasks, presents the distribution as shown in the first

Table 4: Job type and job size distribution

JobSize	%	#	# of I/O(size)	# of CPU(size)
1-2	54%	14	6 TeraSort(2)	8 Kmean(1)
3-20	14%	4	2 TeraSort(10)	2 Bayes(20)
21-150	14%	3	1 RWrite(40) 2 Grep(120)	
151-300	6%	1	1 WCount(250)	
301-500	4%	1		1 PiEst(480)
> 500	8%	2	1 TeraGen(600)	1 PiEst(1000)

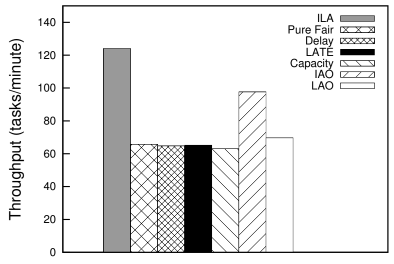


Figure 6: Throughput due to different schedulers.

two columns of Table 4. We adjusted the total number of jobs based on our cluster's scale and generated a job submission schedule for 25 jobs. According to the trace, the distribution of inter-arrival times between jobs was roughly exponential with a mean of 14 seconds. It makes our submission schedule 253 seconds long. The tested applications are listed in Table 4 with the columns representing the job size, the percentage of the total jobs, the actual number of running jobs, the number of I/O-bound jobs and the number CPU-bound jobs, respectively.

Figure 5 shows the average completion time of each type of jobs due to different task schedulers. The results are normalized with respect to the performance due to ILA scheduler. Compared with interference-oblivious schedulers, including PureFair, Delay, LATE, Capacity and LAO schedulers, ILA scheduler could speed up individual jobs by 1.5-6.5 times. It led to an improvement of 1.1-2.0 times in job completion time in comparison with IAO scheduler. Figure 6 shows the system throughput due to different schedulers. ILA scheduler yielded an improvement of up to 1.9 time in throughput over interference-oblivious schedulers and led to an improvement of 1.3 times over IAO scheduler. The interference-oblivious schedulers had similar performance. The delay-based scheduling algorithm in Delay and LAO schedulers could not speed up the jobs because the tasks were significantly slowed down due to resource contention no matter how they access the data. Capacity scheduler also lost its effectiveness in such virtualized environments because it was unable to guarantee each job queue's resource portion in the presence of VM interference. LATE scheduler could not maintain its efficiency due to severe resource contention. IAO scheduler only considers the effect of interference. The greedy data locality policy is attributed to its performance degradation.

Figure 7 shows the percentage of tasks in the jobs with local data access. Since *node local* tasks and *server local* tasks have similar performance, the calculated local data access includes both *node local* access and *server local* access. Jobs are demonstrated in two groups: I/O-bound jobs and CPU-bound jobs. Recall that the Adaptive Delay Scheduling algorithm manages to achieve good data locality for tasks with

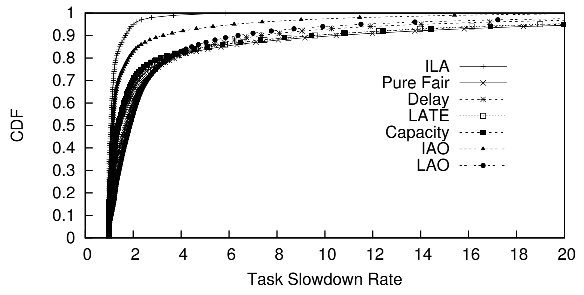


Figure 8: CDF of task completion time.

sufficiently large input file. Thus, for CPU-bound jobs with small input files, ILA and LAO scheduler used default greedy locality policy. (*TeraGen* and *RWrite* are not shown because they have no input files). From Figure 7, we can see that for I/O-bound jobs, ILA brought the average local data access up to 90%. It gained 65% improvement over PureFair, LATE, Capacity and IAO schedulers, and 20% improvement over Delay schedulers. As expected, schedulers using greedy locality policy, such as PureFair, LATE, Capacity and IAO schedulers, achieved the lowest local data access. For Delay scheduler, three major factors are attributed to its worse performance compared with ILA scheduler: first, it introduces much longer delay after failing to find a *node local* task. Second, launching premature *rack local* tasks instead of *server local* ones due to the unawareness of *server locality.* Third, many unexpected long tasks caused by interference make the delay scheduling less efficient. For CPU-bound jobs, their performance are insensitive to data locality. ILA scheduler generated 60% local task without any delay. Although Delay scheduler achieved the highest percentage, it may cause unnecessary long time delay and harm the performance.

ILA scheduler improved jobs' performance by accelerating each individual task. Figure 8 plots the cumulative distribution of completion time of individual tasks under different schedulers. The completion time is normalized with respect to that of the task running with *node local* data access and without any interference. We can see that under ILA scheduling, 70% of the tasks (2440 tasks in total) proceed without any slowdown, 90% run with less than 1.5x slowdown and 99% run with less than 2.5x slowdown. IAO performed slightly worse with 83% of tasks runnig less than 2x slower and 99% running less than 10x slower. The task slowdown rate significantly rised under the interference-oblivious scheduling. For PureFair, Delay, LATE, Capacity and LAO schedulers, only 15% of the tasks could run without any interference and up to 60% run with more than 2x slowdown. The completion times of 10% tasks were increased by more than 10 times.

From the results, we can make several observations. First, small jobs tend to be improved more by ILA scheduler than large jobs, as shown in Figure 5. One of the reasons is that small jobs are easily affected by interference. If one task of a small job is slowed down greatly, the whole job's completion time is very likely increased due to waiting for the slow task. However, for a large job, the task slowdown can be amortized by other concurrent normal running tasks as long as the delayed tasks are not in the last batch. Moreover, for a large job, the performance degradation in individual tasks may be compromised by the increased parallel degree. But for small jobs, the degree of parallelism is limited by

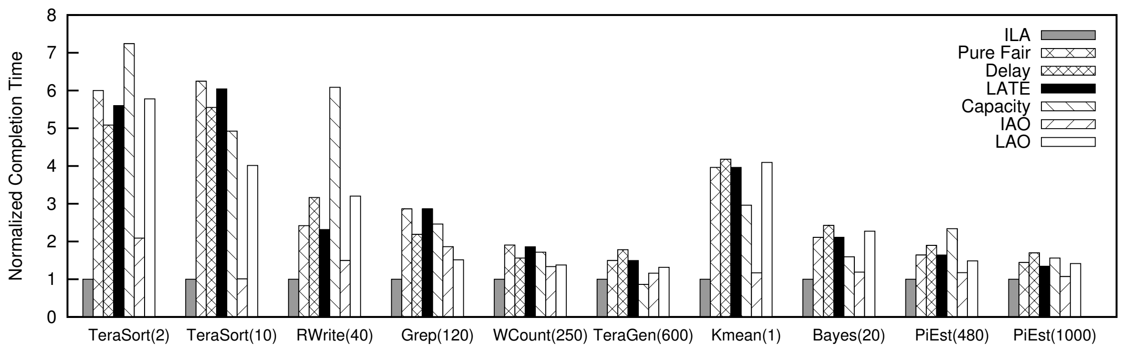


Figure 5: Job completion time due to different schedulers.

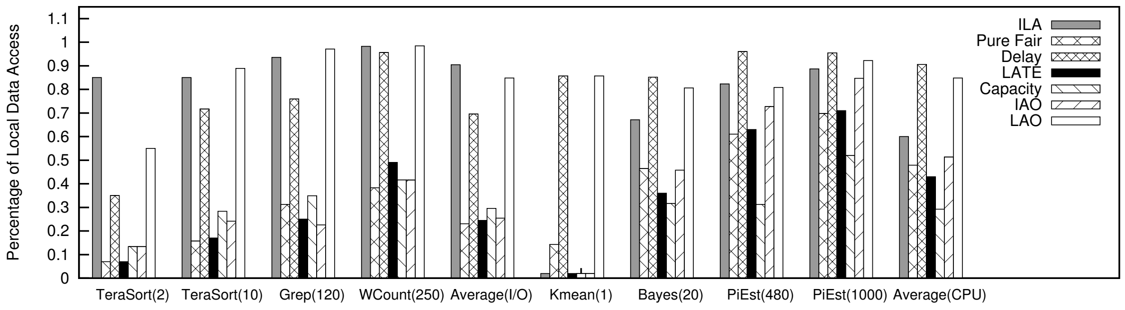


Figure 7: Data locality due to different schedulers.

the number of tasks. Another reason for small jobs gaining more improvement is that launching *node local* tasks is much more difficult for them. Compared with large jobs, small jobs have much fewer input file blocks, which makes them having much less nodes with local data. Therefore, achieving the *server locality* could significantly increase the number of local nodes without degrading the performance. As shown in Figure 7, for the small job $TeraSort(2)$, the schedulers which are unaware of the *server locality* could only achieve less than 35% local access. ILA was able to improve it to 85%.

The second observation is that interference could cause much severer performance degradation than remote data access. Remote data access could only affect individual tasks but interference may impose impact on the scheduled task and all of the co-hosted tasks. As shown in Figure 5 and Figure 8, LAO scheduler led to more job and task slowdown than IAO scheduler. This observation explains why ILA scheduler always mitigate interference before improving data locality if there is a conflict between these two aspects.

5.3 Benefits of Interference and Locality Aware Scheduling

In this section, we demonstrate effectiveness of IASM and LASM modules with the specifically designed experiments.

Benefit from IASM. IASM is the component in ILA conducting the interference-aware scheduling. To isolate the effect from data locality, we selected the $Matrix$ and $TeraGen$ applications in evaluation because neither of them requires input data. Each $Matrix$ job was comprised of 150 map tasks and each $TeraGen$ job contained 400 map tasks. We submitted 3 $Matrix$ jobs and 3 $TeraGen$ jobs to the cluster alternatively with 15 seconds time interval.

Figure 9 shows the normalized completion time of each type of jobs due to different schedulers. Since there is no data locality effect, ILA and IAO were reduced to the same scheduler, which speeded up the jobs by 2.0-3.0 times compared with the interference-oblivious schedulers. Delay and LAO schedulers became equivalent to PureFair scheduler without the data locality effect. There was also no obvious improvement achieved by either LATE or Capacity scheduler. Their performance were mainly limited by the side effect of interference. From Figure 9, we can also see that job's completion time only varied within 18% due to ILA scheduling. In contrast, under the interference-oblivious scheduling, job's performance fluctuated in a much wider range from 10% to 70%. The reason is that interference seldom evenly affects all the tasks in one job. Task's performance heavily depends on the current system status of the host VM, the physical server and its own characteristics. Thus only the interference aware scheduling could provide a stable and predictable system.

Benefit from LASM. LASM module is in charge of the locality aware scheduling. To demonstrate its effectiveness, we eliminated the influence from VM interference by designing a micro-benchmark with a set of elaborately modified applications. We carefully adjusted the resource demand of $PiEst$ and $Grep$ applications by injecting idle loops into the programs. These modified applications, noted as $MPiEst$ and $MGrep$, still consume sufficient resources to maintain their characteristics but will not cause resource contention even running on co-hosted VMs. Each $MPiEst$ job contained 100 map tasks and each $MGrep$ contained 350 tasks. We submitted 3 $MPiEst$ jobs and 3 $MGrep$ jobs and also run some independent applications to mimic background network traffic without interfering with the scheduling.

Figure 10 shows the normalized completion time and the percentage of local tasks due to different schedulers. For the

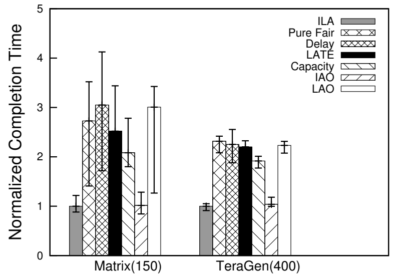
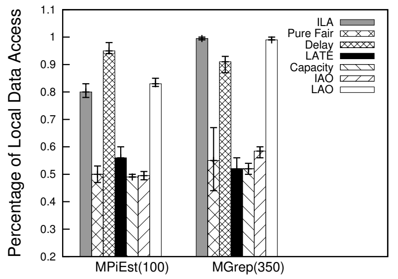
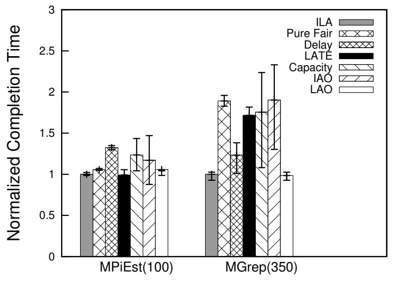


Figure 9: Benefits of interference-aware scheduling.



(a) Job data locality



(b) Job completion time

Figure 10: Benefits of locality-aware scheduling.

I/O-bound job *MGrep*, the schedulers with LASM modules (ILA and LAO) brought the percentage of local tasks to nearly 100%. In contrast, the schedulers using the default greedy locality policy (PureFair, LATE, Capacity and IAO) only achieved less than 58% locality. As a result, these schedulers slowed down the jobs by nearly 1.8 times. The Delay Scheduling demonstrated its advantage over the greedy locality policy. It yielded 90% local tasks. However, due to the awareness of the *server locality*, ILA scheduler achieved a 10% improvement in data locality and a 23% improvement in completion time over Delay scheduler.

For CPU-bound job $MPiEst$, without any delay, LASM-based schedulers launched 80% local tasks and the schedulers with greedy locality policy launched 50%. Only Delay scheduler postponed tasks in order to improve data locality. Since $MPiEst$ is insensitive to data access, although having achieved 95% locality, Delay scheduler was still the worst one due to unnecessary delay for each task. It slowed down $MPiEst$ job by 1.3 times compared with ILA. This result shows that LASM could speed up jobs through dynamically setting delay intervals according to their input file sizes.

5.4 Interference Prediction Model Analysis

In this section, we evaluate the accuracy of the exponential interference prediction model. For comparison purpose, we also designed and implemented two other models: linear model and quadratic model, which are used in previous work [9]. These two models require the same performance critical parameters for interference prediction, as shown in Table 2. They estimate that the job's performance and the system resource consumption presents a linear and a quadratic relationship, receptively,

To assess the prediction accuracy of the interference model, we selected the coefficient of determination (R^2) as one of the measurement. R^2 is widely used in the context of statistical models whose purpose is the prediction of future outcomes on the basis of related information. It is defined as $R^2 = 1 - (\sum_{i=0}^{K}(y_i - \hat{y}_i)^2 / \sum_{i=0}^{K}(y_i - \bar{y}_i)^2)$, where K is the total number of samples. y, $\hat{y}$ and $\bar{y}$ denote the actual value, predicted value and the mean of actual values, receptively. Table 5 shows the R^2 value for each type of jobs due to different models. The evaluated job set includes all the 10 applications listed in Table 3. We can see that, compared with the other two models, the exponential model led to higher R^2 values for both types of jobs, which means higher accuracy. It brought overall R^2 value to 0.887. In contrast, linear model and quadratic model only achieve 0.657 and 0.714 for overall R^2 value, receptively.

We also evaluated the exponential model using a more

Table 5: R^2 due to different models.

Model	I/O-bound	CPU-bound	Overall
Linear	0.676	0.611	0.657
Quadratic	0.722	0.672	0.714
Exponential	0.895	0.879	0.887

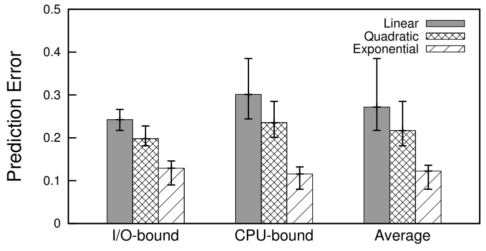


Figure 11: Prediction error due to different models.

direct metric, the *prediction error*, which is defined as $|y - \hat{y}|/y$. Figure 11 shows the yielded *prediction errors* for each type of jobs due to different models. The box heights represent the average prediction errors and the error bars represent the prediction accuracy deviations among all of the evaluated jobs. We can see that the exponential model could reduce the average prediction error by 15% compared with the linear model and by 10% compared with the quadratic model. The exponential model led to an average of 12% error rate. It was able to keep the prediction error below 14% for all kinds of jobs. According to the experimental results in Section 5.2 , such accuracy of the interference prediction model is acceptable for ILA scheduling. It could efficiently mitigate the interference and make most of the tasks running without being slowed down.

5.5 Effectiveness of Dynamic Threshold

As discussed in Section 4.4, we introduced a dynamic threshold policy to deal with the tradeoff between the individual task performance and the degree of the job's parallelism during interference-aware scheduling. In this section, we evaluate the effectiveness of the dynamic threshold policy by comparising with the static ones.

We generated three types of workloads including mixed, pure CPU and pure I/O applications. The compared polices included static policies with the threshold H set as 1, 2, 4, 6 and 8, and the policy with no threshold. Figure 12 shows the system throughputs due to different polices. We observed that dynamic threshold policy always achieved the highest throughput for all kinds of workloads. In contrast,

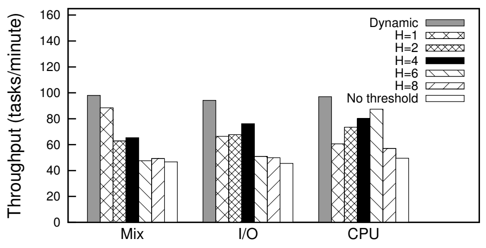


Figure 12: Performance improvement due to dynamic threshold policy.

the no-threshold policy, i.e. the interference-oblivious policy, always performed worst. The static policies could not yield consistent performance. For example, when H is 1, the scheduler achieved nearly highest throughput under Mix workload, but caused 40% performance degradation under CPU-bound workload. That is because under the mixed workload, the scheduler is easy to find a task without any interference due to the diversity in task demands. However, under a pure CPU or I/O-bound workload, increasing the tasks' parallelism with a little compromising individual task performance may be more beneficial to the whole job. Thus, there is no single optimal static threshold for all kinds of workloads. We can see that dynamic threshold policy was able to improve the throughput by up to 40%-100% over static policies.

5.6 ILA Overhead and Scalability

For the 72-node Xen-based experiment setup, ILA scheduler consumed less than 1% CPU resource on a 3.0 GHz Inter Xeon processor. To evaluate the scalability of ILA, we developed a workload simulator to mimic a cluster with 2400 VMs and 400 physical machines. Each physical machine contained 6 VMs. Each VM was configured with 2 map slots and 1 reduce slot. The simulator ran 100 jobs with different sizes. There were a total of 15000 map tasks and 5000 reduce tasks. We assumed the average task length to be 10 seconds. Thus there were 720 tasks finishing per second in average. The simulation result shows that ILA was able to correctly schedule 720 tasks per second on a 3.0 GHz Inter Xeon processor.

6. RELATED WORK

There have been many studies devoted to improving system throughout, job completion time and fairness of large scale cluster applications, especially MapReduce applications, in physical or virtualized environments. Existing works have addressed these issues via task scheduling optimization, adaptive resource management or data locality improvement.

Task scheduling for cluster applications. Following on the MapReduce seminar work [11], many researches focused on improving task scheduling algorithms for this framework. Yahoo's Capacity scheduler supports multiple queues for shared users and guarantees each queue a fair share of the capacity of the cluster [6]. Facebook's fairness scheduler uses delay scheduling algorithm to achieve good data locality by slightly compromising fairness restriction [26]. In [27], Zaharia, *et al.* proposed Longest Approximate Time to End (LATE) scheduling algorithm to improve MapReduce applications' performance in heterogenous environment through accurate speculative execution. However, none of them could maintain its effectiveness in virtualized cloud environments due to VM interference.

A large body of work has studied the task scheduling algorithm in other distributed frameworks. For example, in [14], Isard *et al.* introduced a task scheduler *Quincy* for Microsoft's *Dryad* computing environment [5] to achieve good data locality while maintaining fairness through an optimization process; In [24], the *Condor* scheduler was extended to grid to improve performance within locality constraints. However, these approaches are designed for physical clusters. Their performance may be compromised by the virtualization overhead and interference when moving to the cloud.

Resource management and interference mitigation. As MapReduce cloud service becomes an attractive usage model, virtual resource management for MapReduce applications has draw more and more attentions. Sandholm *et al.* proposed a system with proportional shared mechanism that dynamically adjusts resource allocations to MapReduce jobs [23]. Park *et al.* introduced a locality-aware dynamic VM reconfiguration technique to improve MapReduce's performance [22]. In [21], a resource management framework *Purlieus* was proposed to enhance the performance of MapReduce jobs by coupling data and VM placement. In [18], Li *et al.* proposed, CAM, a topology aware resource manager for MapReduce applications in clouds using a minimum cost flow method. CAM focused on optimizing data and VM placement with considerations of task data locality as well as resource utilization, including both computational and storage resources. In contrast, ILA addresses the management issue from a different perspective. It improves the performance of MapReduce applications via application-level task scheduling optimization. ILA is able to adapt to any data/VM deployment and resource allocation policy. It requires no modification for the underlying resource management. Moreover, ILA can not only avoid server overload due to inappropriate VM deployment or resource allocations, but also mitigate interference between co-hosted VMs.

Overcoming the interference between co-hosted VMs is one of the essential challenges in cloud management. Hardware or operating system solutions have been extensively studied in previous works, including dynamical cache partition [25], intelligent memory management [19] and improved operating system scheduling [29]. Our approach deals with the interference problem without any modification of existing hardware platforms or operating systems. In [20], *Q-Clouds* suggested to mitigate the interference by dynamically tuning resource allocation to VMs using an online feedback control method. Most recently, TRACON, an interference-aware task scheduler was proposed for data-intensive applications in virtual environment [9]. It is able to effectively mitigate I/O interference with the help of a prediction model. As shown in Section 5.4, their model is less accurate than ILA's exponential prediction model. Also TRACON focused on mitigating interference with no consideration of data locality.

Locality improvement for cluster applications. A lot of works are devoted to improving data locality for data-intensive cluster applications. In [7], a system that replicated blocks based on their popularity was presented to alleviate data hotspots and speed up jobs. In [15], Jin *et al.* proposed an availability-aware MapReduce data placement policy for non-dedicated distributed computing environment.

These approaches were mainly designed for physical environment. However, in cloud environments, the predictions on popularity [7] or availability [15] may lose their effectiveness due to the unawareness of the presence of two levels of topology: physical server and VM level. Hotspots may still exist on the physical server level when the data placement is optimized for the VM level. Moreover, in cloud, data locality optimization does not necessarily lead to performance improvement due to the resource contention. ILA scheduler considers both interference and task data locality.

Performance optimization for MapReduce. The growing popularity of MapReduce has spurred many works on improving the MapReduce performance from system to application level. Kang *et al.* improved the performance of MapReduce virtual cluster by modifying the context-switching mechanism of the Xen credit scheduler [16]. Herodotou *et al.* proposed StarFish to improve MapReduce performance by automatically configuring Hadoop parameters [13]. In [8], an "outlier-control" framework *Mantri* was presented, which could detect the abnormal tasks and proactively take corrective action. These works are orthogonal to our ILA work.

7. CONCLUSION

This paper presents an interference and locality-aware scheduler for virtual MapReduce clusters. It relies on two scheduling modules: IASM and LASM. The former performs the interference-free scheduling with the assistance of a performance prediction model and the latter improves task data locality by using Adaptive Delay Scheduling algorithm. Experimental results show that ILA scheduler could achieve a speedup of 1.5-6.5 times for individual jobs and yield an improvement of up to 1.9 times in system throughput compared with 4 other schedulers. It improves data locality of map tasks by up to 65%. Although ILA scheduling algorithm is designed for MapReduce framework, it could be applicable to other virtual cluster schedulers.

In MapReduce clusters, besides locally running tasks, HDFS may also issue I/O requests for a remote data access. ILA considers this I/O flow as the background traffic and manages to mitigate the interference through task scheduling. In the future, we would like to extend ILA scheduler to mitigate the interference from HDFS through intelligent data placement and data node selection. In a MapReduce framework, the data transfer between map and reduce tasks consumes substantial network resource. ILA scheduler assumes the mappers evenly distributed across the cluster and the locality of reducers would not greatly affect the performance. Another direction of the future work is minimizing the communication overhead between map and reduce tasks by improving reducer's locality when the distribution of map tasks is skewed.

8. REFERENCES

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

[2] Amazon mapreduce. `http://aws.amazon.com/elasticmapreduce/`.

[3] Apache hadoop. `http://hadoop.apache.org`.

[4] The apache mahout project. `http://mahout.apache.org/`.

[5] Microsoft dryad project. `http://research.microsoft.com/en-us/projects/dryad/`.

[6] Yahoo! inc. capacity scheduler. `http://developer.yahoo.com/blogs/hadoop/posts/2011/02/capacity-scheduler/`.

[7] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. G. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: coping with skewed content popularity in mapreduce clusters. In *EuroSys*, pages 287–300, 2011.

[8] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, 2010.

[9] R. C.-L. Chiang and H. H. Huang. Tracon: interference-aware scheduling for data-intensive applications in virtualized environments. In *SC*, page 47, 2011.

[10] E. K. P. Chong and S. H. Zak. *An Introduction to Optimization, 3rd Edition*. Wiley Press, 2008.

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

[12] N. R. Draper and H. Smith. *Applied Regression Analysis*. John Wiley and Sons, 1981.

[13] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, pages 261–272, 2011.

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

[15] H. Jin, X. Yang, X.-H. Sun, and I. Raicu. Adapt: Availability-aware mapreduce data placement for non-dedicated distributed computing. In *ICDCS*, pages 516–525, 2012.

[16] H. Kang, Y. Chen, J. L. Wong, R. Sion, and J. Wu. Enhancement of xen's scheduler for mapreduce workloads. In *HPDC*, pages 251–262, 2011.

[17] P. Lama and X. Zhou. Ninepin: Non-invasive and energy efficient performance isolation in virtualized servers. In *DSN*, pages 1–12, 2012.

[18] M. Li, D. Subhraveti, A. R. Butt, A. Khasymski, and P. Sarkar. Cam: a topology aware minimum cost flow based resource manager for mapreduce applications in the cloud. In *HPDC*, pages 211–222, 2012.

[19] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *16th USENIX Security Symposium*, 2007.

[20] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *EuroSys*, pages 237–250, 2010.

[21] B. Palanisamy, A. Singh, L. Liu, and B. Jain. Purlieus: locality-aware resource allocation for mapreduce in a cloud. In *SC*, page 58, 2011.

[22] J. Park, D. Lee, B. Kim, J. Huh, and S. Maeng. Locality-aware dynamic vm reconfiguration on mapreduce clouds. In *HPDC*, pages 27–36, 2012.

[23] T. Sandholm and K. Lai. Mapreduce optimization using regulated dynamic prioritization. In *SIGMETRICS/Performance*, pages 299–310, 2009.

[24] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[25] Y. Xie and G. H. Loh. Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches. In *ISCA*, 2009.

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

[27] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, 2008.

[28] Y. Zhang, W. Sun, and Y. Inoguchi. Predicting running time of grid tasks based on cpu load predictions. In *GRID*, 2006.

[29] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS*, pages 129–142, 2010.

APPENDIX

A Appendix: Proof of the Expected Value $\bar{Q}$

We assume that the cluster consists of N physical servers, with M VMs per server, for a total of MN nodes. For a specific job, let P denote the number of nodes on which the job has local data. We first calculate the probability $\mathbb{P}(Q, P)$ of that all of the P VMs deployed on exactly Q physical servers.

We first consider a related problem: it is assumed that there are only Q physical servers in the cluster and the P VMs cover all the physical servers. We denote the number of ways of deployments as $\mathbb{N}(Q, P)$. For P VMs, there are in total $\binom{MQ}{P}$ different deployment situations without any constraint on the number of occupied physical servers, which may cover Q or $Q-1$, ... , or $\lceil \frac{P}{M} \rceil$ physical servers, if $Q > \lceil \frac{P}{M} \rceil$:

$$\begin{aligned}\binom{MQ}{P} &= \binom{Q}{Q}\mathbb{N}(Q,P) + \binom{Q}{Q\text{–}1}\mathbb{N}(Q-1,P) + ... + \binom{Q}{\lceil\frac{P}{M}\rceil}\mathbb{N}(\lceil\frac{P}{M}\rceil, P) \\ &= \mathbb{N}(Q,P) + \sum_{i=\lceil\frac{P}{M}\rceil}^{Q-1}\mathbb{N}(i,P)\end{aligned}$$

where

$$M, P, Q \in \mathbb{Z}, P \in [1, QM], Q \in (\lceil\frac{P}{M}\rceil, N] \tag{.8}$$

If $Q = \lceil \frac{P}{M} \rceil$, there is only one way to deploy. From equation A.12, we have:

$$\mathbb{N}(Q,P) = \begin{cases} 1 & Q = \lceil\frac{P}{M}\rceil \\ \binom{MQ}{P} - \sum_{i=\lceil\frac{P}{M}\rceil}^{Q-1}\mathbb{N}(i,P) & \lceil\frac{P}{M}\rceil < Q \leq N \end{cases} \tag{.9}$$

where

$$M, N, P, Q \in \mathbb{Z}, P \in [1, QM], Q \in [\lceil\frac{P}{M}\rceil, N]$$

Recall that, in the original problem, there are N physical servers instead of Q ones. Thus, the total number of deployment $\mathbb{T}(Q, P)$ that the P VMs cover Q physical servers will be as follows:

$$\mathbb{T}(Q,P) = \binom{N}{Q}\mathbb{N}(Q,P). \tag{.10}$$

Without any constraint, the total number of deployment situations for the P VMs will be $\binom{MN}{P}$. Thus the probability $\mathbb{P}(Q, P)$ of that all of the P VMs deployed on exactly Q physical servers will be $\mathbb{T}(Q, P)/\binom{MN}{P}$. Combine A.13 and A.14, we have:

$$\mathbb{P}(Q,P) = \begin{cases} \binom{N}{Q}\binom{MQ}{P}/\binom{MN}{P} & Q = \lceil\frac{P}{M}\rceil \\ \frac{\binom{N}{Q}\binom{MQ}{P}}{\binom{MN}{P}} - \binom{N}{Q}\sum_{i=\lceil\frac{P}{M}\rceil}^{Q-1}\frac{\binom{Q}{i}}{\binom{N}{i}}\mathbb{P}(i,P) & \lceil\frac{P}{M}\rceil < Q \leq N \end{cases}$$

where

$$M, N, P, Q \in \mathbb{Z}, P \in [1, NM], Q \in [\lceil\frac{P}{M}\rceil, N] \tag{.11}$$

According to formula A.15, if a job have P VMs holding its local data, the expected number of physical servers $\bar{Q}$ that store the local data will be:

$$\bar{Q} = E[Q] = \sum_{i=\lceil P/M\rceil}^{N}\mathbb{P}(i,P) * i \tag{.12}$$

where $\lceil P/M \rceil$ serves as a lower bound of the possible number of physical servers and N is the upper bound. Thus the expected number of the physical-local VMs is $\bar{Q}M$, which will be much larger than P.