ThroughputScheduler: Learning to Schedule on Heterogeneous Hadoop Clusters

Shekhar Gupta, Christian Fritz, Bob Price, Roger Hoover, and Johan de Kleer
Palo Alto Research Center, Palo Alto, CA, USA
{sgupta, cfritz, bprice, rhoover, dekleer}@parc.com

Cees Witteveen
Delft University of Technology, The Netherlands
c.witteveen@tudelft.nl

Abstract

Hadoop is the de-facto standard for big data analytics applications. Presently available schedulers for Hadoop clusters assign tasks to nodes without regard to the capability of the nodes. We propose *ThroughputScheduler*, which reduces the overall job completion time on a clusters of heterogeneous nodes by actively scheduling tasks on nodes based on optimally matching job requirements to node capabilities. Node capabilities are learned by running probe jobs on the cluster. ThroughputScheduler uses a Bayesian, active learning scheme to learn the resource requirements of jobs on-the-fly. An empirical evaluation on a set of sample problems demonstrates that ThroughputScheduler can reduce total job completion time by almost 20% compared to the Hadoop FairScheduler and 40% compared to FIFOScheduler. ThroughputScheduler also reduces average mapping time by 33% compared to either of these schedulers.

1 Introduction

Map-Reduce frameworks, such as Hadoop, are the technology of choice for implementing mayn big data applications. However, Hadoop and other frameworks typically assume a homogeneous cluster of server nodes and assign tasks to nodes regardless of their capabilities, while in practice, data centers may contain a heterogeneous mix of servers. When the jobs executing on the cluster also have heterogeneous resource requirements, which is typical, then it is possible to significantly increase processing throughput by actively matching jobs to server capabilities [2, 4, 6]. In this paper, we present the *ThroughputScheduler*, which actively exploits the heterogeneity of a cluster to reduce the overall execution time of a collection of concurrently executing jobs with distinct resource requirements. This is accomplished without *any* additional input from the user or the cluster administrator.

Optimal task allocation requires knowledge about both the resource requirements of jobs and the resource capabilities of servers, e.g., their relative CPU and disk I/O speeds. The ThroughputScheduler derives server capabilities by running "probe" jobs on the cluster nodes. These capabilities drift very slowly in practice and can be evaluated at infrequent intervals, e.g., at cluster set-up. In contrast, each new job has a-priori unknown resource requirements. We therefore present a learning scheme to learn job resource requirements on-the-fly.

The practicality of our solution relies on the structure of jobs in Hadoop. These jobs are subdivided into *tasks*, often numbering in the thousands, which are executed in parallel on different nodes. Mapping tasks belonging to different jobs can have very different resource requirements, while mapping tasks belonging to the same job are very similar. This is true for the large majority of practical mapping tasks, as Hadoop divides the data to be processed into evenly sized blocks. For a given job, we can therefore use online learning to learn a model of its resource requirements from a small number of mapping tasks in an *explore* phase, and then *exploit* this model to optimize the allocation of the remaining tasks. As we will show, this can result in a significant increase in throughput and never reduces throughput compared to Hadoop's baseline schedulers (FIFO and FairScheduler). We focus on minimizing the overall time to completion of mapping tasks, which is typically the primary driver of overall job completion time.

The next section reviews scheduling in Hadoop, followed by a discussion of related work. We then define a model of task completion time based on server capabilities and task requirements. We derive a Bayesian experimental design for learning the parameters of this model online, and present a real-time heuristic algorithm to optimally schedule tasks onto available cluster nodes using this model. Finally, we show empirically that ThroughputScheduler can reduce overall job execution time by up to 40% on a heterogeneous Hadoop cluster.

2 Hadoop Scheduler

In this section we briefly review the scheduler of Hadoop YARN [1]. YARN has a central entity called the resource manager. The resource manager has two primary modules: *Scheduler* and *ApplicationManager*. For every incoming job the ApplicationManager starts an *ApplicationMaster* on one of the slave nodes. The ApplicationMaster makes resource requests to the resource manager and is also responsible for monitoring the status of the job. Jobs are divided into *tasks* and for every task the scheduler assigns a *container* upon the request from the corresponding ApplicationMaster. A container specifies the node to run the task on and a fixed amount of resources (memory and CPU cores). YARN supports allocating containers based on the available resources (as of now just based on memory) on the nodes, but it has no mechanism to determine the actual resource requirements of a job.

To coordinate the allocation of resources for concurrent jobs, Hadoop provides three different schedulers: FIFO-, Fair- and CapacityScheduler. FairScheduler is the most popular scheduler among all because it enables fairness among concurrently executing jobs by giving them equal resources. All of Hadoop's schedulers are unaware of the actual resource profiles of jobs and the capabilities of node in the cluster and therefore often allocate resources sub-optimally.

3 Related Work

Recently, researchers have realized that the assumption of a homogeneous cluster is no longer true in many scenarios and have started to develop approaches that improve Hadoop's performance on heterogeneous clusters.

Speculative execution, a feature of Hadoop where a task that takes longer to finish than expected gets reexecuted preemptively on a second node assuming the first may fail, can lead to degraded performance on heterogeneous clusters. This is because the scheduler's model of how long a task should take does not take the heterogeneous resources into account, leading to many instances of unnecessary speculative executions for tasks executing on slower nodes. The *LATE Scheduler* [9] improves speculative executing for heterogeneous clusters, to only speculatively execute tasks that will indeed finish late using the concept of *straggler* tasks [3]. However, the approach assumes that the hardware capabilities and the task resource profiles are already known rather than being discovered automatically.

The *Context Aware Scheduler for Hadoop* (CASH) [5] assigns tasks to the nodes that are most capable to satisfy the tasks' resource requirements. Similar to our approach CASH learns resource capabilities and resource requirements to enable efficient scheduling. However, unlike our online learning, CASH learns capabilities and requirements in offline mode. The performance of CASH is evaluated on a Hadoop simulator rather than a real cluster. Tian et al. propose a dynamic scheduler which learns job resource profile on the fly [8]. Their scheduler only considers the heterogeneity in the workload and assumes a homogeneous cluster to assign tasks to nodes. An architecture of a resource-aware cloud-driver for heterogeneous Hadoop clusters was proposed to improve the performance and increase fairness [7]. The cloud-driver tries to improve the performance by providing more efficient fairness among jobs in terms of resource allocation. Unlike our approach, the cloud-driver assumes that cluster capabilities are already known and it has abstract knowledge of job resource requirements.

4 Approach

In this section we describe the design for a scheduler that optimizes the assignment of tasks to servers. To do this, we need the *task requirements* and *server capabilities*. Unfortunately, these requirements and capabilities are not directly observable as there is no simple way of translating server hardware specifications and task program code into resource parameters. We take a learning based approach which starts with an *explore phase* where parameters are learned followed by an *exploit phase* in which the parameters are used to allocate tasks to servers. To learn these parameters by observation, we propose a task execution model that links observed execution times of map tasks to the unobservable parameters. We assume that map tasks belonging to the same job have very similar resource requirements. In the remainder of this section, we introduce the task model and then describe the explore and exploit phases.

4.1 Task Model

The task performance model predicts the execution time of a task on a server given the task resource requirements and the capabilities of the server node. We model a task as a set of resource specific operation types such as reading data from HDFS, performing computation, or transferring data over the network. The task resource requirements are represented by a vector $\theta = [\theta_1, \theta_2, \dots, \theta_N]$ where each component represents the total requirement for an operation type (e.g., number of instructions to process, bytes of I/O to read). The capabilities of the server are described by a corresponding vector $\kappa = [\kappa_1, \kappa_2, \dots, \kappa_N]$ which represent rates for processing the respective operation type (e.g., FLOPS or I/O per second).

In theory, some of these operations could take place simultaneously. For instance, some computation can occur while waiting for disk I/O. In practice this does not have a large impact on Hadoop tasks we studied. We therefore assume that the requirements for each operation type are processed independently. The time required to process a resource requirement is the total magnitude for the requirement divided by the processing rate. The total time T^j to process all resource requirements on server j is the sum of the times for each operation type

$$T^j = \sum_k \frac{\theta_k}{\kappa_k^j} + \Omega^j \tag{1}$$

where Ω^j is the overhead to start the task on the server. We assume that every job imposes the same amount of overhead on a given machine. In this paper, we consider a two dimensional model in which $\kappa = [\kappa_c, \kappa_d]$ represents computation and disk I/O server capabilities and $\theta = [\theta_c, \theta_d]$ represents the corresponding task requirements. Hence, the task duration model reduces to:

$$T^j = \frac{\theta_c}{\kappa_c^j} + \frac{\theta_d}{\kappa_d^j} + \Omega^j. \tag{2}$$

The parameters κ_c and κ_d abstractly capture many complex low-level hardware dependencies. For example, κ_c internally accounts for the kind of operations needed to be performed (flops or integer ops or memory ops). Similarly, κ_d is dependent on disk speed, seek time, etc. In practice, it is very difficult to build a task model as a function of these low level parameters. To keep the model simple and easier to understand we use such abstract parameters.

4.2 Explore

We learn server resource capabilities and task resource requirements separately. First we learn server capabilities offline. Then using these capabilities we actively learn the resource requirements for jobs online.

4.2.1 Learning Node Capabilities

We assume server capabilities κ^j's and overhead Ω^j do not change frequently and can be estimated offline. The server parameters are estimated by executing probe jobs. Since the time we measure is the only dimension with fixed units, the value of the parameters is underdetermined. We resolve the unidentifiability of the system by choosing a 'unit' map task to define a baseline. The unit map task has an empty map function and it does not read or write from/to HDFS.

The computation (θ_c) and disk task requirements (θ_d) are both zero, therefore Equation 2 allows us to estimate Ω. Multiple executions are averaged to create an accurate point estimate. Note that Ω includes some computation and disk I/O that occur during start up.

One could imagine attempting to isolate the remaining parameters in the same fashion, however, it is difficult to construct a job with zero computation or zero disk I/O. Instead we construct jobs with two different levels of resource usage defined by a fixed ratio η.

Let's assume we aim to determine κ_c. First we run a job $J_c^1 = \langle \theta_c, \varepsilon_d \rangle$ with fixed disk requirement ε_d (J_c^1 might be a job which simply reads an input file and processes the text in the file). We compute the average execution time of this job on each server node. According to our task model the average mapping time for every machine i can be given as

$$T_1^i = \frac{\theta_c}{\kappa_c^i} + \frac{\varepsilon_d}{\kappa_d^i} + \Omega^i \tag{3}$$

Next we run a job J_c^η which reads the same input but the processing is multiplied by η compared to J_c^1. Therefore, the resource requirements of J_c^η can be given as $J_c^\eta = \langle \eta\theta_c, \varepsilon_d \rangle$. The average mapping time for every node can be given as

$$T_n^i = \frac{\eta\theta_c}{\kappa_c^i} + \frac{\varepsilon_d}{\kappa_d^i} + \Omega^i \tag{4}$$

We solve for $\frac{\varepsilon_d}{\kappa_d}$ in equations 3 and 4, set them equal and solve for κ_c^i to get:

$$\kappa_c^i = \frac{\theta_c(\eta - 1)}{T_n^i - T_1^i} \tag{5}$$

This equation gives us κ_c^i in terms of a ratio. To make it absolute, we arbitrarily choose one node as the reference node. We set $\kappa_c^1 = 1$ and $\kappa_d^1 = 1$ and then solve equation 5 for θ_c. Once we have the task requirements θ_c in terms of the base units for server one, we can use this job requirement to solve for the server capabilities on all the other nodes. Similarly we estimate κ_d.

Normally in Hadoop, the output of map tasks goes to multiple reducers and may be replicated on several servers. This would have the effect of introducing network communication costs into the system. To avoid that while learning node capabilities, we set the number of reducers to zero and set the replication factor to one.

Table 1 gives an example of computed server capability parameters for a five node cluster of heterogenous machines. The algorithm correctly discovers that there are two classes of machines.

4.2.2 Learning Job Resource Profile

In this phase the resource requirements for tasks are learned in an online manner without interrupting production use of the cluster. To enable online learning we collect task completion time samples from actual production

Node	κ_c	κ_d	Ω
Node1	1	1	45
Node2	1	1	45
Node3	7.5	2.5	5.3
Node4	7.5	2.5	5.3
Node5	7.8	2.6	4.8

Table 1: Recorded Node Capabilities and Overhead

jobs. With every new time sample we update our belief about the resource profile $[\theta_c, \theta_d]$ of the job.

We assume that the observed execution time T^j is normally distributed around the value predicted by the task duration model given by Eq. 2. Given a distribution over resource parameters $[\theta_c, \theta_d]$, the remaining uncertainty due to changing conditions on the server (i.e., the observation noise) is given by a standard deviation σ_j.

$$T^j \sim \mathcal{N}\left(\frac{\theta_c}{\kappa_c^j} + \frac{\theta_d}{\kappa_d^j} + \Omega^j,\ \sigma^j\right) \tag{6}$$

Starting with prior beliefs about task requirements $p(\theta_c, \theta_d)$ and the execution model based likelihood function $p(T^j \mid \theta_c, \theta_d, \kappa_c^j, \kappa_d^j, \sigma^j)$, Bayes' rule allows us to compute a joint posterior belief over $[\theta_c, \theta_d]$:

$$p(\theta_c, \theta_d \mid T^j, \kappa_c^j, \kappa_d^j, \sigma^j) = \alpha p(T^j \mid \theta_c, \theta_d, \kappa^j, \sigma^j) p(\theta_c, \theta_d)$$

For our two-dimensional CPU and disk usage example, the likelihood has the form (Empirically we observed an observed variance of approximately +/- 3 indicates a standard deviation of 1, therefore, $\sigma^j = 1$):

$$p(T^j \mid \theta_c, \theta_d; \kappa_c, \kappa_d) = \frac{1}{\sqrt{2\pi}} \exp \frac{\left(T^j - \frac{\theta_c}{\kappa_c^j} - \frac{\theta_d}{\kappa_d^j} - \Omega^j\right)^2}{2}$$

Note that the execution time is normally distributed around a line defined by the server capabilities $[\kappa_c, \kappa_d]$. The joint distribution of the likelihood is not a bivariate normal, but a univariate Gaussian tube around a line. This makes sense, as a given execution time could be due to a slow CPU and fast disk or a fast CPU and slow disk.

When a job is first submitted we assume that the resource requirements for its tasks are completely unknown. Assuming an uninformative prior, the posterior distribution after the first observation is just proportional to the likelihood.

$$p(\theta_c, \theta_d \mid T^j) = \frac{1}{\sqrt{2\pi}\sigma^j} \exp \frac{\left(T^j - \frac{\theta_c}{\kappa_c} - \frac{\theta_d}{\kappa_d} - \Omega^j\right)^2}{2}$$

For the second and subsequent updates we have a definite prior distribution and likelihood function. These two are multiplied to obtain the density of the second posterior update. Let the first experiment be on machine j with capability κ^j and let the observed time be T^j. Let the second experiment be on machine k with capability κ^k and let the observed time be T^k. The resulting posterior distribution is

$$p(\theta_c, \theta_d \mid T^j, T^k) = \frac{1}{\sqrt{2\pi}} \exp\left[\frac{\left(T^j - \frac{\theta_c}{\kappa_c^j} - \frac{\theta_d}{\kappa_d^j} - \Omega^j\right)^2}{2} + \frac{\left(T^k - \frac{\theta_c}{\kappa_c^k} - \frac{\theta_d}{\kappa_d^k} - \Omega^j\right)^2}{2}\right] \tag{7}$$

We omit the derivation for space, but we do give the update rules here. With every time sample we can recover the mean μ_{θ_c,θ_d} and covariance matrix $\Sigma_{\theta_c,\theta_d}$ by using the property of the bivariate Gaussian distribution. Expanding the exponent of Equation 7 and collecting the θ_c and θ_d term gives us a conic section in standard form:

$$a_{20}{\theta_c}^2 + a_{10}\theta_c + a_{11}\theta_c\theta_d + a_{01}\theta_d + a_{02}{\theta_d}^2 + a_{00} = 0 \tag{8}$$

There is a transformation to map between the coefficients of a conic in standard form and the parameters of a Gaussian distribution. The mean and covariance of the distribution with the same elliptical form is given by:

$$\begin{bmatrix} \mu_{\theta_c} \\ \mu_{\theta_d} \end{bmatrix} = \begin{bmatrix} (a_{11}a_{01} - 2a_{02}a_{10})/(4a_{20}a_{02} - a_{11}^2) \\ (a_{11}a_{10} - 2a_{20}a_{01})/(4a_{20}a_{02} - a_{11}^2) \end{bmatrix} \tag{9}$$

$$\Sigma_{\theta_c\theta_d}^{-1} = \begin{bmatrix} a_{20} & \frac{1}{2}a_{11} \\ \frac{1}{2}a_{11} & a_{02} \end{bmatrix} \tag{10}$$

For every new time sample we compute coefficients a_{nm} for equation 8. These coefficients determine the updated value of μ_{θ_c}, μ_{θ_d}, and $\Sigma_{\theta_c,\theta_c}$.

Because we recover both the mean and the covariance of task requirements, we can quantify our degree of uncertain about task requirements, and hence decide whether to keep exploring or starting to exploit this knowledge for optimized task scheduling. In this paper we sample tasks until we get a determinant for the covariance matrix $|\Sigma_{\theta_c,\theta_d}| < 0.007$. Table 2 summarizes resource requirements learned by the online inference mechanism for some of the Hadoop example jobs. When we compare the 'Pi' job, which calculates digits of Pi, to RandomWriter, which writes bulk data, we see that the algorithm correctly recovers the fact that Pi is compute intensive (large μ_{θ_c}) whereas RandomWrite is disk intensive (large μ_{θ_d}). Other Hadoop jobs show intermediate resource profiles as expected. The J_{IO} job will be described further in the experimental section. The '# of Tasks' column gives the number of tasks executed to reach the desired confidence.

4.3 Exploit

Once the resource profile of a job is learned to sufficient accuracy we switch from explore to exploit. The native Hadoop scheduler sorts task/machine pairs according to

Job	μ_{θ_c}	μ_{θ_d}	$\lvert\Sigma_{\theta_c\theta_d}\rvert$	# of Tasks
Pi	24.00	6.30	0.0038	109
Random Writer	27.26	234.62	0.0061	28
Grep	15.82	8.10	0.0038	90
WordCount (1.5 GB)	43.50	22.50	0.00614	31
WordCount (15 GB)	138.05	206.40	0.00615	32
J_{IO}	5.60	96.46	0.0063	30

Table 2: Job resource profile measurements with variance and number of tasks executed

whether they are local (data for the task is available on the machine), on the same rack, or remote. We introduce our routine based on our task requirements estimation called "SelectBestJob" to break ties within each of these tiers as shown in Algorithm 4.1: If we have two local jobs, we would run the one most compatible with the machine first.

Algorithm 4.1: THROUGHPUTSCHEDULER(Cluster, Request)

```
for each Node N ∈ Cluster
do
    JobsWithLocalTasks ← N.GETJOBSLOCAL(Request)
    JobsWithRackTasks ← N.GETJOBSRACK(Request)
    JobsWithOffSwitchTasks ← N.GETJOBSOFFSWITCH(Request)
    if LocalJobs ≠ NULL
      then
        J ← SELECTBESTJOB(LocalJobs, N)
        ASSIGNTASKFORJOB(N, J)
    else if RackJobs ≠ NULL
      then
        J ← SELECTBESTJOB(RackJobs, N)
        ASSIGNTASKFORJOB(N, J)
    else
        J ← SELECTBESTJOB(OffSwitchJobs, N)
        ASSIGNTASKFORJOB(N, J)
```

Algorithm 4.2: SELECTBESTJOB(*NodeN*, *ListofJobs*)

```
return $(\operatorname{argmin}_{J\in\text{ListOfJobs}} \frac{\text{norm}(\theta_c^J)}{\text{norm}(\kappa_c^N)} + \frac{\text{norm}(\theta_c^J)}{\text{norm}(\kappa_c^N)})$
```

SelectBestJob, shown in Algorithm 4.2, selects job J that minimizes a *score* for task completion on node N. However, rather than using absolute values of θ_c, θ_d, κ_c and κ_d, we use the normalized value of these parameters to define the score. While absolute values represent expected time of completion, which can be measured in seconds, job selection based on these numbers would always favor short tasks over longer once and fast machines over slower ones. This would not achieve the optimized matching of job requirements to server capabilities. For example, consider Nodes 1 and 3 in Table 1. Node 3 is almost 7.5 times faster than Node 1 in terms of CPU, but only 2.5 times faster in terms of disk. Hence, intuitively, disk intense jobs are better scheduled on Node 1, since the relativly higher CPU performance of Node 3 is better used for CPU intense jobs (if there are any). To account for this relativity of optimal resource matching, we normalize both jobs and machines to make their total requirements and capabilities sum to one for each resource x (here $x \in \{c,d\}$):

$$\text{norm}(\theta_x^i) = \frac{\mu_{\theta_x^i}}{\sum_k \mu_{\theta_k^i}} \qquad \text{norm}(\kappa_x^j) = \frac{\kappa_x^j}{\sum_{k=1}^5 \kappa_x^k}$$

5 Experimental Results

To evaluate the performance of *ThroughputScheduler* we conducted experiments on a five node Hadoop cluster at PARC (see Table 1).

5.1 Evaluation on Heterogeneous Jobs

We evaluate the performance of our scheduler on jobs with different resource requirements. Since the Hadoop benchmarks do not contain highly I/O intensive jobs (cf. Table 2), we constructed our own I/O intensive Map-Reduce job, J_{IO}. J_{IO} reads 1.5 GB from HDFS, and writes files totaling 15 GB back to HDFS. This resembles the resource requirements of many expand-translate-load (ETL) applications used in big data applications to pre-process data using Map-Reduce and writing into HBase, MongoDB, or another disk-backed database. We learn J_{IO}'s resource profile using the job learner described in the Explore section. The learned resource requirement of J_{IO} is listed in Table 2. To evaluate ThroughputScheduler on drastically heterogeneous job profiles, we run J_{IO} along with the Hadoop benchmark *Pi*, which is CPU intense. We compare the performance of ThroughputScheduler with FIFO- and FairScheduler—for a single user, CapacityScheduler is no different from FIFO.

5.1.1 Job Completion Time

We first compare the performance of the proposed scheduler in terms of overall job completion time. In case of multiple jobs, the overall job completion time is defined as the completion time of the job finishing last. In this experiment we study the effect of heterogeneity between job resource requirements, which we can quantify as the ratio of disk I/O to CPU requirement of a job: $h = \frac{\theta_d}{\theta_c}$. In order to vary this quantity we vary the I/O load of J_{IO} further by varying the replication factor of the cluster: the higher the replication factor, the higher the I/O load of a job. This impacts disk I/O intense jobs more than others.

These results show that ThroughputScheduler performs better than FIFO- and FairScheduler in all cases. The relative performance increase of our scheduler increases as the heterogeneity of the two jobs increase, as simulated by an increased replication factor: up to 40% compared to FIFO, and 20% compared to Fair. Note that both the Fair- and the ThroughputScheduler benefit from

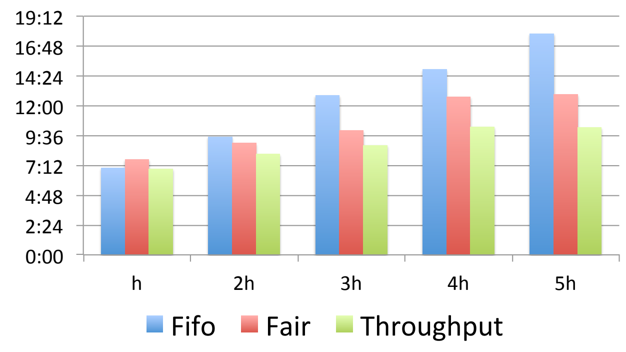

Figure 1: Overall job completion time in minutes (Y axis) on heterogeneous nodes at PARC for different relative values of $h = \frac{\theta_d}{\theta_c}$. Disk load θ_d is increased by increasing the replication number.

higher replication as they can better take advantage of data locality. The improvements of ThroughputScheduler beyond Fair- are purely due to our improved matching of jobs to computational resources.

Job	FIFO	Fair	Throughput
Pi	9 sec	9 sec	6 sec
J_{IO}	2 min 15 sec	2 min	2 min 10 sec

Table 3: Comparison of Average Mapping Time

To better understand the source of this speed-up, we considered the average mapping time for each job (*throughput*). Table 3 summarizes these results and provides the explanation for the speed-up: our scheduler improves the throughput of *Pi* by 33%, while maintaining the throughput of J_{IO} compared to the other schedulers. Since *Pi* has very many mapping tasks, these savings pay off for the overall time to completion.

5.2 Performance on Benchmark Jobs

To estimate the performance of ThroughputScheduler on realistic workloads, we also experimented with the existing Hadoop example jobs. We ran the job combinations of concurrent jobs shown in Table 4.

$Comb_1$	Grep (15 GB) + Pi (1500 samples)
$Comb_2$	WordCount (15 GB) + Pi (1500 samples)
$Comb_3$	WordCount (15 GB) + Grep (15 GB)

Table 4: Job Combination

The performance comparison in terms of job completion time is presented in Figure 2. For these workloads ThroughputScheduler performs better than either of the other two in all cases. For $Comb_2$ the job completion time is reduced by 30% compared to FIFO. For $Comb_3$ all three schedulers perform similarly because both jobs are CPU intensive (cf. Table 2).

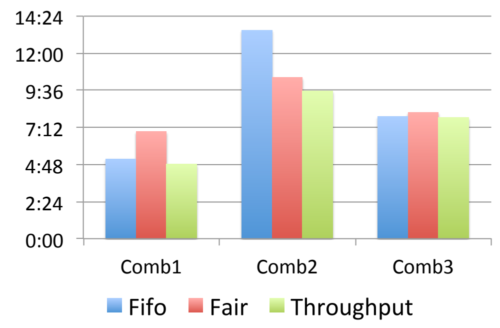

Figure 2: Job Completion time in minutes (Y axis) of combinations of Hadoop example jobs.

Job Combination	FIFO	Fair	Throughput
$Pi(1500sample), WC(15GB)$	440s	319s	310s
$Pi(1500sample), Grep(15GB)$	210s	224s	214s
$WC(15GB), Grep(15GB)$	225s	262s	214s

Table 5: Completion time of job combinations on a homogeneous cluster.

5.3 Performance on Homogeneous Cluster

We ran additional experiments on a set of homogeneous cluster nodes, to ensure such a setup would not cause ThroughputScheduler to produce inferior performance. These results are shown in Table 5.

6 Conclusion

ThroughputScheduler represents a unique method of scheduling jobs on heterogeneous Hadoop clusters using active learning. The framework learns both server capabilities and job task parameters autonomously. The resulting model can be used to optimize allocation of tasks to servers and thereby reduce overall execution time (and power consumption). Initial results confirm that ThroughputScheduler performs better than the default Hadoop schedulers for heterogenous clusters, and does not negatively impact performance even on homogeneous clusters.

While our demonstration uses the Hadoop system, the approach implemented by ThroughputScheduler is applicable to other framework of distributed computing as well.

References

[1] Apache hadoop nextgen mapreduce (yarn). `http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`.

[2] BALAKRISHNAN, S., RAJWAR, R., UPTON, M., AND LAI, K. The impact of performance asymmetry in emerging multicore architectures. In *In Proceedings of the 32nd Annual International Symposium on Computer Architecture* (2005), pp. 506–517.

[3] BORTNIKOV, E., FRANK, A., HILLEL, E., AND RAO, S. Predicting execution bottlenecks in map-reduce clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing* (Berkeley, CA, USA, 2012), HotCloud'12, USENIX Association, pp. 18–18.

[4] GHIASI, S., KELLER, T., AND RAWSON, F. Scheduling for heterogeneous processors in server systems. In *Proceedings of the 2nd conference on Computing frontiers* (New York, NY, USA, 2005), CF '05, ACM, pp. 199–210.

[5] KUMAR, K. A., KONISHETTY, V. K., VORUGANTI, K., AND RAO, G. V. P. Cash: context aware scheduler for hadoop. In *Proceedings of the International Conference on Advances in Computing, Communications and Informatics* (New York, NY, USA, 2012), ICACCI '12, ACM, pp. 52–61.

[6] KUMAR, R., TULLSEN, D. M., JOUPPI, N. P., AND RANGANATHAN, P. Heterogeneous chip multiprocessors. *Computer 38*, 11 (Nov. 2005), 32–38.

[7] LEE, G., CHUN, B.-G., AND KATZ, H. Heterogeneity-aware resource allocation and scheduling in the cloud. In *Proceedings of the 3rd USENIX conference on Hot topics in cloud computing* (Berkeley, CA, USA, 2011), HotCloud'11, USENIX Association, pp. 4–4.

[8] TIAN, C., ZHOU, H., HE, Y., AND ZHA, L. A dynamic mapreduce scheduler for heterogeneous workloads. In *Grid and Cooperative Computing, 2009. GCC '09. Eighth International Conference on* (2009), pp. 218–224.

[9] ZAHARIA, M., KONWINSKI, A., JOSEPH, A. D., KATZ, R., AND STOICA, I. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 29–42.