Virtual Hadoop: MapReduce over Docker Containers with an Auto-Scaling Mechanism for Heterogeneous Environments[*]

Yi-Wei Chen[1], Shih-Hao Hung[1], Chia-Heng Tu[1 2] and Chih Wei Yeh[1]
[1]National Taiwan University, Taipei 10617, Taiwan
[2]National Cheng Kung University, Tainan 70101, Taiwan
hungsh@csie.ntu.edu.tw

ABSTRACT

Hadoop is a widely used software framework for handling massive data. As heterogeneous computing gains its momentum, variants of Hadoop have been developed to offload the computation of the Hadoop applications onto the *heterogeneous processors*, such as GPUs, DSPs, and FPGA. Unfortunately, these variants do not support on-demand resource scaling for the deadline-aware applications in a sophisticated heterogeneous computing environment. In this work, we developed a framework called *Virtual Hadoop*, which scales out the required computing resources for the applications automatically to meet the given real-time requirements. We extended the methods of resource inference and allocation for the heterogeneous computing environments. On top of these methods, an *auto-scaling mechanism* was developed to dynamically allocate resources on-demand based on profile data and performance models for the application execution to meet the given time requirements. In addition, Virtual Hadoop can utilize Docker containers to facilitate the auto-scaling mechanism, where a container encapsulates a Hadoop node with the capability to leverage heterogeneous computing engines. Our experimental results reveal the efficiency of Virtual Hadoop, and hopefully the experiences and discussion presented in this paper will ease the adoption of heterogeneous computing for efficient big data processing.

CCS Concepts

•Software and its engineering → Middleware; Virtual machines; Massively parallel systems;

Keywords

MapReduce; Performance Model; Resource Allocation; Docker Container; General Purpose GPU (GPGPU); Heterogeneous System Architecture (HSA)

[*]This work was supported in part by the Ministry of Science and Technology, Taiwan under the grant No. 104-2622-8-002-002-. In addition, this work is also conducted under the support of QNAP Systems, Inc.

RACS '16, October 11-14, 2016, Odense, Denmark

Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4455-5/16/10…$15.00

DOI: http://dx.doi.org/10.1145/2987386.2987408

1. INTRODUCTION

The *Apache Hadoop* open source project is widely used for distributed computing and data storage [1]. From Microsoft to Amazon Web Services and Cloudera, the cloud service providers all have their customized Hadoop environments for data analysis services. When Hadoop is deployed as a service in the cloud, the Hadoop software stack often runs on top of a *virtual environment*, such as container or virtual machine, instead of a physical server, so as to improve service portability, server density, and on-demand execution. Among other virtualization techniques, *Docker container* has become a popular choice today for the service providers to host the Hadoop services, regarding the execution efficiency [2].

Recently, variants of Hadoop [3, 4, 5, 6] have been developed to take advantage of *heterogeneous computing* by offloading the computation kernels of a Hadoop application to the accelerators composed by multicore CPUs, Graphics Processing Units (GPUs), and/or even Field Programmable Gate Array (FPGA) boards. Among the Hadoop variants, the OpenCL-based approaches have gained some momentum (i.e., *HadoopCL* [4], and *Aparapi* [5]), since OpenCL is an open standard supported by most hardware vendors, which enables the use of different OpenCL-enabled accelerators in such Hadoop variants with minimal efforts. However, these variants do not pay attention to the support of the *on-demand resource scaling* for the deadline-aware application execution in the heterogeneous computing environments. Thus, the existing Hadoop variants would not be a proper solution if the target Hadoop framework is expected to dynamically expand the computing resources for the application execution so as to meet the given time constraints.

This paper describes *Virtual Hadoop*, the framework that we developed to tackle the resource management problem of the Hadoop cluster with heterogeneous accelerators. We enhanced the resource management modules (i.e., Resource Manager and Node Manager) in *Hadoop YARN* so that the extended Hadoop framework is able to be aware of the various types of computing resources within the system. Moreover, several software components have been added into the management modules to estimate the resources to be allocated for the Hadoop tasks and to schedule the tasks onto the corresponding computing engines, so that the time requirements of the applications could be satisfied as much as possible. The key features of Virtual Hadoop are highlighted as the following:

- Virtual Hadoop is integrated with the *Aparapi* framework [5] for computation acceleration of the Hadoop applications. In our case study, Virtual Hadoop is able to distribute the execution of the Java codes to the multicore CPUs, discrete GPUs (i.e., AMD Radeon 7970), and the integrated GPUs (i.e., the AMD Accelerated Processing Units, APUs). As the Aparapi based framework already supports various types of accelerators [6], Virtual Hadoop can be easily extended to support various accelerators, such as DSPs, and FPGA boards.

- We have extended the *resource management modules* of the stock Hadoop framework to estimate the execution times of the Hadoop tasks running on the heterogeneous computing engines, and to allocate the computing resources to these tasks for timely execution. In particular, for establishing performance models and estimating the execution time of heterogeneous computing engines, we extended the previous work [7] which models the performance of the Hadoop applications on general-purpose processors. With the extensions, our auto-scaling mechanism is able to scale the computation resources required by the Hadoop applications on demand. Our results show that, without the support of the on-demand resource scaling, the deviation between the time requirements and the actual execution times of the applications is up to 74 percent, and the deviation is down to 15 percent when the auto-scaling mechanism is applied.

- Virtual Hadoop can leverage the *Docker containers* to wrap up the execution environments of the Hadoop nodes for quick, efficient deployment, and is different from the design of the official Hadoop framework, where the Docker containers are used to wrap up the user codes, and served as an alternative to the *YARN containers* [8]. Overall, mirroring the software stacks of the Hadoop nodes into Docker images helps facilitate the deployment of the Hadoop nodes across the computers with heterogeneous accelerators.

The rest of this paper is organized as follows. Section 2 describes the architecture and the key components of Virtual Hadoop. Section 3 introduces our extended performance modeling method, and Section 4 describes how we use the performance modeling method for estimating the resources to be allocated for the target tasks so as to satisfy the time requirements for the tasks. Finally, Section 5 shows the results of the performance modeling method and the resource estimation scheme, and we conclude our work in Section 6.

2. VIRTUAL HADOOP

The architecture of Virtual Hadoop is illustrated in Figure 1. The framework consists of four components: *Application Profiler*, *Resource Estimator*, *QoS Scheduler*, and *Docker Communicator* in Hadoop's Resource Manager.

Application Profiler is responsible to collect the performance profiles of the interested applications as the inputs to *Resource Estimator*. Based on the expected execution times (i.e., the time constrains given by the user) of the applications, Resource Estimator calculates the required resources for these applications. With the estimated resources information, *QoS Scheduler* determines the priority and the execution order of the applications. Before the execution of

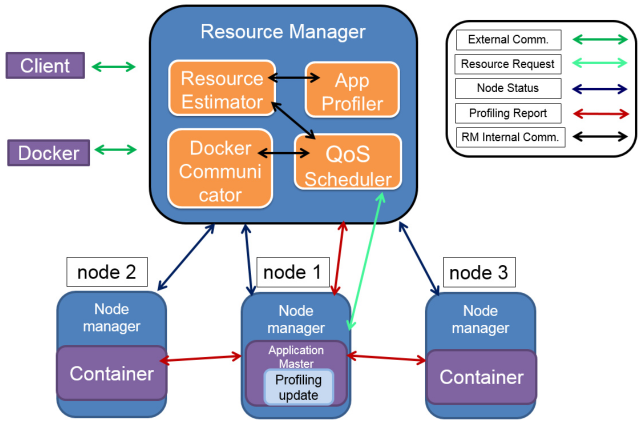

Figure 1: Overview of Virtual Hadoop.

the applications, Docker containers[1] are prepared and set up the execution context, where *Docker Communicator* is used to coordinate the use of Docker containers from the container pool, and to scale up the system resources for the Hadoop application.

The auto-scaling mechanism supported by Virtual Hadoop is illustrated in Figure 2. Basically, a *deadline-first* scheduling algorithm is adopted, and the execution order is determined according to the time requirements (i.e., the expected times of the applications), and the order could be updated/changed whenever a new application is submitted to the Virtual Hadoop.

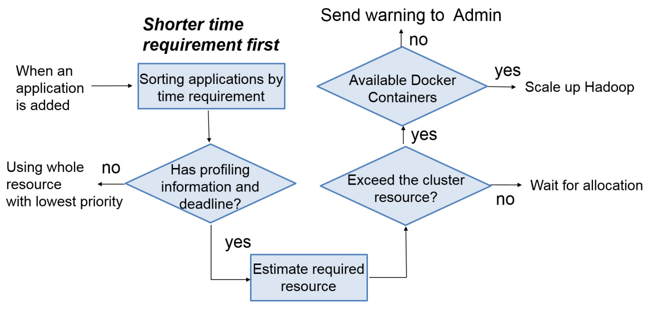
Shorter time requirement first
Send warning to Admin
no
When an application is added
Sorting applications by time requirement
Available Docker Containers
yes
Scale up Hadoop
yes
Using whole resource with lowest priority
no
Has profiling information and deadline?
Exceed the cluster resource?
Wait for allocation
no
yes
Estimate required resource

Figure 2: Auto-scaling in Virtual Hadoop.

The mechanism works as follows. When an application is submitted to Virtual Hadoop, QoS Scheduler sorts all the (execution order of) applications according to their time requirements. QoS Scheduler grants the resources to the application based on the calculation done by Resource Estimator and scales up/out the Virtual Hadoop if the required resources exceed the current resources of the Hadoop system. The scaling operations are performed by Docker Communicator to request for more Docker containers from the container pool, which has all the containers that can be accessed by the Hadoop administrator. If the pool runs out of container, Virtual Hadoop maintains the same size and alerts the Hadoop user about the insufficient resources. Note that the current implementation of Virtual Hadoop uses a basic

[1]Notice that a Docker container represents a Hadoop node. The container used in this paper is different from the container managed by the Hadoop YARN for task execution.

application scheduling algorithm to demonstrate the auto-scaling mechanism. To support of more rigorous timing constrains, real-time scheduling algorithms could be added to the QoS scheduler.

The key to a timing-aware execution of mixed workloads with Virtual Hadoop is automatic scaling of the heterogeneous resources based on performance modeling and resource estimation, which are discussed in the next two sections.

3. PERFORMANCE MODELING

Modeling the performance of MapReduce tasks on heterogeneous computing platforms is essential to estimate the required resources for the efficient task execution. We apply the *Makespan Theorem* [7] to establish performance bounds of MapReduce tasks for heterogeneous processing elements, and the summation of the established bounds of the Map and Reduce stages is the performance bounds of the MapReduce application. Before introducing the bounds of the Map and Reduce stages for the heterogeneous system, we summarize the assumptions of our design.

1. The resources to be managed (allocated and used) by Virtual Hadoop is the heterogeneous processing elements, including CPU, discrete GPU (denoted as *dG*), and integrated GPU (denoted as *iG*).

   Note that in this work, we adopt the OpenCL software to facilitate the heterogeneous computing for Hadoop tasks. By following the OpenCL standard, the unit of the processing elements, CPU, dG, and iG, is equal to the *compute unit* seen by the OpenCL software. For example, for a quad-core processor, it has four compute units from the OpenCL platform model. Furthermore, the integrated GPU refers to the GPU within the integrated graphics solutions, for example, the Graphics Core Next-based (GCN) GPU in AMD Accelerated Processing Unit (APU), which follows the Heterogeneous System Architecture (HSA). In this paper, we use *HSA* to refer to the integrated GPUs in the APU-like processor.

2. When allocating a GPU (either dG or iG) for computation, it implies that another CPU should be allocated as the CPU-GPU pair.

   For example, the resource vector <4, 1, 2> reveals the computing resources of <CPU, dG, iG> available on the system. If a dG is allocated to a new task on the system, the resource vector becomes <3, 0, 2>. It is designed to conform the OpenCL standard, where the context of the computation kernel is maintained by the CPU, and the computation acceleration is done by the GPU. Furthermore, the synchronous execution of the OpenCL kernel is adopted between the CPU and the paired GPU.

3. The execution of Map tasks does not overlap with that of Reduce tasks.

   The greedy assignment is adopted in this work to improve resource utilization, which means that when there is a runable task, it is assigned to the idle processing unit. More complex scheduling algorithms should be developed to support the modeling the overlapping of the Map and Reduce task execution, which is our future work.

3.1 Bounding Map Tasks Performance

By referencing the Makespan Theorem, the three variables, the number of tasks, the number of processing resource, and the execution time of the tasks, should be defined to model the application performance. Let N_{CPU}, N_{GPU}, and N_{HSA} be the number of tasks in CPUs, GPUs, and HSAs, respectively. $Alloc_{CPU}$, $Alloc_{GPU}$, and $Alloc_{HSA}$ are the quantity of CPUs, GPUs, and HSAs allocated to the MapReduce application. The average execution time of all Map tasks in CPUs, GPUs, and HSAs is denoted as Avg_{CPU}, Avg_{GPU}, and Avg_{HSA}, respectively. With the greedy assignment strategy in mind, the relation among the variables are defined as the equation below. Using the equation, it is simple to compute the number of tasks that should be assigned to the CPU, dG, and iG. Note that to obtain the outcome of the Equation 1, Application Profiler is used, which is introduced in Section 4.1, to generate the performance profile of the application before the calculation; for instance, Avg_{CPU}, Avg_{GPU}, and Avg_{HSA} should be known for the calculation.

$$N_{CPU} : N_{GPU} : N_{HSA} = \frac{Alloc_{CPU}}{Avg_{CPU}} : \frac{Alloc_{GPU}}{Avg_{GPU}} : \frac{Alloc_{HSA}}{Avg_{HSA}} \tag{1}$$

Let the maximum execution time of Map tasks in CPUs, GPUs, and HSAs be MAX_{CPU}, MAX_{GPU}, and MAX_{HSA} respectively. By the definitions of the lower bound and the upper bound of the application, which is specified in the previous work [7], we derive the the bounds of the Map tasks on CPU, dG, and iG, as listed in Table 1. With the bounds of the processing units, we are able to further derive the bounds of the Map tasks, where the lower bound is the minimum of the lower bounds in Table 1, whereas the upper bound is the maximum of the upper bounds in Table 1. The average bound is half of the snum of the lower and upper bounds. The definitions are available in Equation 2.

Table 1: Lower/Upper bounds for the processing units.

	Lower Bound	Upper Bound
CPU	$\frac{N_{CPU} \times Avg_{CPU}}{Alloc_{CPU}}$	$\frac{(N_{CPU} - 1) \times Avg_{CPU}}{Alloc_{CPU}} + Max_{CPU}$
GPU	$\frac{N_{GPU} \times Avg_{GPU}}{Alloc_{GPU}}$	$\frac{(N_{GPU} - 1) \times Avg_{GPU}}{Alloc_{GPU}} + Max_{GPU}$
HSA	$\frac{N_{HSA} \times Avg_{HSA}}{Alloc_{HSA}}$	$\frac{(N_{HSA} - 1) \times Avg_{HSA}}{Alloc_{HSA}} + Max_{HSA}$

$$\begin{cases} Lower_M = min\{Lower_{CPU},\ Lower_{GPU},\ Lower_{HSA}\} \\ Upper_M = max\{Upper_{CPU},\ Upper_{GPU},\ Upper_{HSA}\} \\ Avg_M = \frac{Lower_M + Upper_M}{2} \end{cases} \tag{2}$$

3.2 Bounding Reduce Tasks Performance

The Reduce stage is further decomposed into three phases: *shuffle*, *copy*, and *reduce*. With Assumption 3, the shuffle and copy phases are not allowed to be taken place until the termination of the Map stage. Let N_R be the number of Reduce tasks, and Avg_R and Max_R be average and max execution time of Reduce tasks respectively. Let $Alloc_R$ be the quantity of CPUs assigned to the Reduce stage. The performance bounds of the Reduce tasks are similar to those of the Map tasks listed in Equation 1 and 2, and Table 1.

The bounds of the Reduce tasks for the CPU are defined in Equation 3, which the bounds for dG and iG are similar to.

$$\begin{cases} Lower_R = \frac{N_R \times Avg_R}{Alloc_R} \\ Upper_R = \frac{(N_R - 1) \times Avg_R}{Alloc_R} + Max_R \\ Avg_R = \frac{Lower_R + Upper_R}{2} \end{cases} \quad (3)$$

3.3 Bounding Application Performance

The bounds of the MapReduce application are shown in Equation 4, which are derived from the bounds from the previous two subsections. Experiments have been performed to evaluate the deviation between the performance predicted by the bounds, and that specified by the user (i.e., the expected time). From our performance data, both the upper and average bounds deliver good results, except that the upper bounds sometimes may have higher deviation (against the expected time) than that delivered by the average bounds. The detailed results are shown in Section 5.

$$\begin{cases} Lower_{MR} = Lower_M + Lower_R \\ Upper_{MR} = Upper_M + Upper_R \\ Avg_{MR} = \frac{Lower_{MR} + Upper_{MR}}{2} \end{cases} \quad (4)$$

4. RESOURCE ESTIMATION

For on-demand resource scaling, the required resources should be computed based on the time requirement of the application. In particular, given the input dataset and the time requirement, we want to determine the proper number of CPUs, GPUs, or HSAs, such that the execution time of the application is close to the time requirement.

4.1 Application Profiler

The profiler is responsible for collecting the performance profile of the MapReduce applications. The profiler also helps translate the data into the format accepted by the Resource Estimator. In particular, the profiler leverages the Hadoop counters to obtain the performance metrics, maximum and average execution times of the Map and the Reduce tasks.

4.2 Resource Estimator

Resource Estimator leverages the performance model to estimate the performance bounds for the MapReduce application with the given resource allocation, the number of tasks in the application, and the application profiles. By altering the resources allocated for the application, the estimator is able to find the proper configuration (i.e., resource allocation scheme), where the execution time of the application meets the user's time requirement. A *bisection method* is adopted by the estimator to iteratively search for the proper resource allocation scheme.

The bisection method for resource allocation is illustrated in Figure 3, where the y-axis refers to the quantity of resources to be allocated for the application execution, and the x-axis is the corresponding execution time estimated by the performance model, which is defined in Section 3. The performance model can be viewed as the transformation function f to transfer the data points r in y-axis to those t in x-axis; $t_r = f(r)$, where with the resources to be allocated r, the transform function $f(r)$ estimates the corresponding execution time of the application t_r.

Resource Estimator checks if the system meets the user's time requirement with all the resources of the whole computer cluster. If not, a warning is sent to the user, and the application runs with all the resources on the system. If so, the iterative configuration searching process starts, which is elaborated with the example in Figure 3. First, the initial resource allocation scheme r_1, which uses half of the resources available on the physical system, is not sufficient for timely application execution (i.e., t_1 is longer than the *expected time*, the red dot), so more resources are added which results in the estimated time t_2. Next, as the estimated time t_2 is shorter than the expected time, the Resource Estimator try to use less resources r_3 for the computation, and the estimated time is t_3. The search is converged with the resource allocation r_3 since the corresponding estimated time t_3 is shorter than the expected time, and further searching (i.e., r_4) does not provide good results (i.e., the time, $t_4 = f(r_4)$, is longer than the expected time).

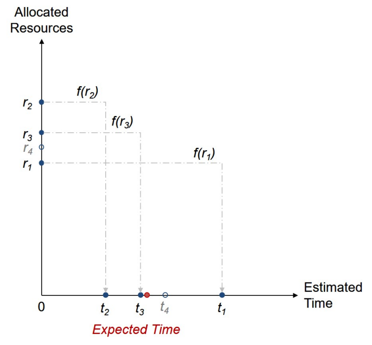

Figure 3: Example of using the bisection method for resource estimation.

5. EXPERIMENTAL RESULTS

The Hadoop system[2] built for the experiments is formed by the seven computers as listed in Table 2, where the discrete GPU is adopted in one machine and the integrated GPUs that are built within the Kaveri APUs are used in the other six machines. The software installed on these machines includes Ubuntu 14.04 64-bit, Oracle Java 1.8.0_45, and Hadoop 2.6.0. The Docker 1.6.2 is used to establish the container based Hadoop nodes, where each container can have access to two CPU cores and 4GB RAM, as well as one GPU core from the discrete (or integrated) GPU. Note that at most two containers run within one physical machine to avoid the resource contention problem, which is not considered in our current performance model, and at the same time, only one of the two containers has the privilege to access the GPU core (i.e., exclusive control over the GPU core). Three benchmarks, Gaussian Blur[3], K-Means[4], and Word Count[5], are used in our experiments to evaluate the

[2]To enforce Assumption 3, the parameter, mapreduce.job.reduce.slowstart.completedmaps, is set to one.
[3]The image processing algorithm reduces noise and blur images. The raw images are of 2 to 4GB in size, and the standard deviation for the pixels values (σ) is set to 4.0.
[4]The cluster number K is 1,000, and the five-dimensional data points are used with the size of 6 to 12GB.
[5]It computes the occurrence frequency of each word in the

efficiency of our approach. The first two benchmarks are able to run on both CPUs and GPUs[6], whereas the last one uses only CPUs.

In the following subsections, we first evaluate the efficiency of the performance bounds derived from our proposed performance model. Then, the accuracy of the estimated execution time is evaluated. Finally, the advantages of auto-scaling mechanism are discussed.

Table 2: Seven computers used in our experiments.

Nodes with Discrete GPU		
CPU	Intel Xeon i7-920(4C/8T, 2.67GHz)	1
GPU	AMD Radeon 7970(32GCN, 3GB GDDR5)	
RAM	6GB	
Storage	250GB, 7200rpm, HDD	
Nodes with Kaveri APUs		
A10-7850K	Quad-core CPU @1.7GHz + 8GPU)	2
A10-7800	Quad-core CPU @1.9GHz + 8GPU)	2
A8-7600	Quad-core CPU @1.4GHz + 6GPU)	2
RAM	16GB	
Storage	1TB, 7200rpm, HDD	

5.1 Performance Modeling

We evaluate the proposed performance model by comparing the estimated execution times with the time measured on the Hadoop cluster. We run the benchmarks on the computers to build the application profiles; Gaussian Blur runs with <12,1,3> and 2GB of input data, K-Means runs with <12,1,3> and 6GB of input data, and Word Count runs with <12,0,0> and 8GB of input data. With the application profiles, the performance bounds are used to predict the execution times of the MapReduce benchmarks running with different data sizes and different types/numbers of processing units. Using the K-Means as an example, we check the difference between the estimated and the measured times with the following configurations.

- Same resource usage <12,1,3>, larger dataset (12GB).
- Different resource usage <24,1,6>, same dataset (6GB).
- Different resource usage <24,1,6>, larger dataset (12GB).

The performance results are plotted in Figure 4, Figure 5, and Figure 6, where the data are normalized to the measured time. Overall, the estimated time derived by the average bound is close to the measured one. In particular, the deviation between the estimated (by the average bound) and the measured time is less than 15 percent in all cases. While the upper bound also deliver good results for K-Means or Word Count, the execution time is overestimated with the upper bound for Gaussian Blur (e.g., up to 37 percent estimation error). Hence, in order to offer accurate results, we use the average bound to compute the expected execution time for the benchmarks in the following experiments.

5.2 Resource Estimation

The effectiveness of the resource estimation mechanism is evaluated by examining the estimated execution times of the applications. To evaluate the accuracy of the estimated performance, we run the three benchmarks with different

dataset, where Wikipedia history articles are used in the experiments with the size of 8 to 20GB.

[6]In current implementation, the Map tasks run on the CPUs and GPUs, whereas the Reduce tasks are on the CPUs.

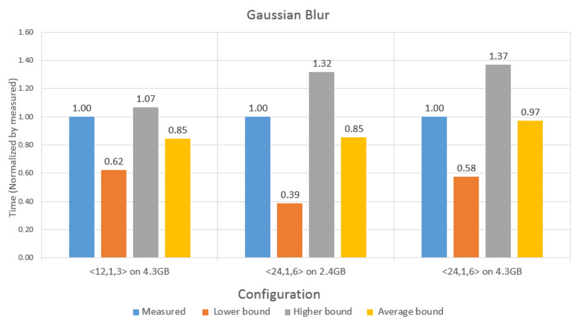

Figure 4: Estimated/Measured execution times for Gaussian Blur.

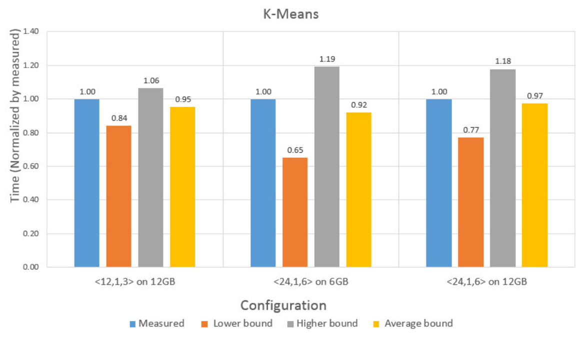

Figure 5: Estimated/Measured execution times for K-Means.

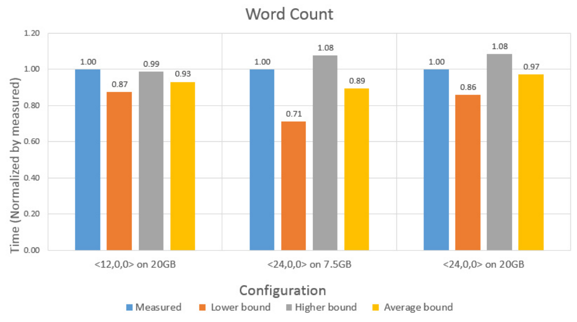

Figure 6: Estimated/Measured execution times for Word Count.

configurations, and plot the expected, estimated, and measured time in Figure 7, where the expected time (time requirement) for Gaussian Blur, K-Means, and Word Count, is 200, 550, and 400 seconds, respectively, and the data are normalized to the expected time. Interestingly, our results show that deviations between time requirement and measured time are less than ±7%, except for the last case of Gaussian Blur. Further experiments are required to investigate the high deviation for the Gaussian Blur.

5.3 Auto-Scaling Mechanism

We evaluate the mechanism by running the mixed workloads simultaneously on the same computer cluster. The required resources for the first benchmark are estimated before the benchmark runs on the system. Later, the auto-scaling mechanism is triggered if another benchmark is submitted

Table 3: **Effectiveness of auto-scaling mechanism. Smaller closeness value is better.**

Auto-scaling	Workloads	Time Requirement	Measured Time	Closeness
Off	K-Means/6GB	1.00	0.94	0.48
	Word Count/10GB	1.00	1.42	
On	K-Means/6GB	1.00	0.94	0.11
	Word Count/10GB	1.00	1.05	
Off	K-Means/8GB	1.00	1.02	0.74
	Word Count/10GB	1.00	1.72	
On	K-Means/8GB	1.00	0.89	0.19
	Word Count/10GB	1.00	1.08	

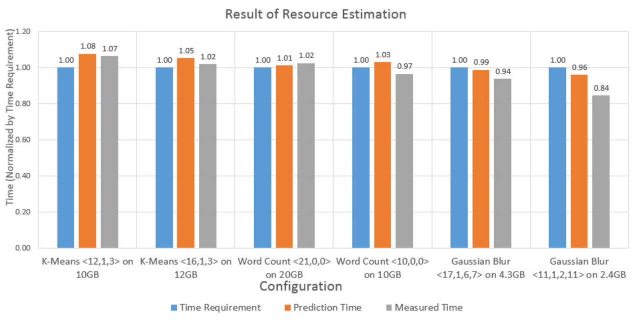

Figure 7: Accuracy of Estimated Performance.

to the system, and the combined resources of the running benchmark(s) and the new benchmark exceed that offered by the system[7]. In this section, we examine the staring overhead of the auto-scaling mechanism, and we show the effectiveness of the mechanism by turning on/off the auto-scaling function.

Table 4: Overhead of starting new containers.

Configuration	Time (s)	New Containers
K-Means/6GB + Word Count/10GB	3.769	1
K-Means/8GB + Word Count/10GB	3.572	2
K-Means/10GB + Word Count/10GB (1st Auto-scaling)	3.963	1
K-Means/10GB + Word Count/10GB (2nd Auto-scaling)	7.875	3
K-Means/6GB + Word Count/10GB + Gaussian Blur/4.3GB	6.412	3

The starting overhead of the auto-scaling mechanism is listed in Table 4. We observe that starting new 1 or 2 containers takes about 3.7 seconds on average. From our analysis, the overhead could be attributed to communication with the container pool, booting Node Manager, and the heartbeat transfer to the Resource Manager. Compared with the long execution times of the workloads, the starting overhead is relatively small. We think it is suitable for the Hadoop system with massive, long running programs.

We show the performance of the mixed workloads with and without the auto-scaling mechanism. For each running benchmark, the effectiveness is measured by the *closeness* function, $abs\left\{\frac{T_R - T}{T_R}\right\}$, where T_R is the time requirement, and T is the measured time of the benchmark. Table 3 lists the performance of the mixed workloads when the auto-scaling mechanism is turned on and off. It is shown that the resource starvation is encountered as the measured times of the workloads running without the auto-scaling support are significantly longer than the time requirements. By adding more containers (system resources) to the Hadoop system, the starvation problem is alleviated, i.e., smaller closeness values are obtained.

[7]The time requirement is set to 550 seconds for every running benchmark.

6. CONCLUSION

We developed Virtual Hadoop to help arrange the heterogeneous computing resources for the target workloads to satisfy their real-time requirements by designing an auto-scaling mechanism that works with the Docker containers. In this paper, we have shown our performance modeling technique for estimating the execution time of the workload with the given resource allocation scheme and the bisection method for our resource estimator to find the proper configuration of the resources. Our experimental results have demonstrated the efficiency of Virtual Hadoop, which encourages us to continue this development work and release the source code to the public in the near future.

7. REFERENCES

[1] Apache Hadoop. http://hadoop.apache.org/.

[2] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. *technology*, 28:32, 2014.

[3] CUDA On Hadoop. https://wiki.apache.org/hadoop/CUDA\%20On\%20Hadoop.

[4] Max Grossman, Mauricio Breternitz Jr, and Vivek Sarkar. HadoopCL: MapReduce on Distributed Heterogeneous Platforms through Seamless Integration of Hadoop and OpenCL. In *IPDPS Workshops*, pages 1918–1927, 2013.

[5] GitHub - aparapi/aparapi. https://github.com/aparapi/aparapi.

[6] Aparapi-Ucores - Aparapi for Unconventional Cores. https://gitlab.com/mora/aparapi-ucores.

[7] Abhishek Verma, Ludmila Cherkasova, and Roy H Campbell. Aria: automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 235–244. ACM, 2011.

[8] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.