Optimizing Capacity Allocation for Big Data Applications in Cloud Datacenters

Sebastiano Spicuglia*, Lydia Y. Chen†, Robert Birke†, Walter Binder*
*Università della Svizzera italiana (USI) – Faculty of Informatics
†IBM Research Zurich – Cloud & Computing Infrastructure

***Abstract*—To operate systems cost-effectively, cloud providers not only multiplex applications on the shared infrastructure but also dynamically allocate available resources, such as power and cores. Data intensive applications based on the MapReduce paradigm rapidly grow in popularity and importance in the Cloud. Such big data applications typically have high fan-out of components and workload dynamics. It is no mean feat to deploy and further optimize application performance within (stringent) resource budgets. In this paper, we develop a novel solution, OptiCA, that eases the deployment of big data applications on cloud and the control of application components so that desired performance metrics can be best achieved for any given resource budgets, in terms of core capacities. The control algorithm of OptiCA distributes the available core budget across co-executed applications and components, based on their "effective" demands obtained through non-intrusive profiling. Our proposed solution is able to achieve robust performance, i.e., with very minor degradation, in cases where resource budget decreases rapidly.**

I. Introduction

Today's cloud datacenter providers face ever stringent energy constraints, while delivering differentiated hosting services to applications. Various big data applications, based on the MapReduce computing paradigm, emerge as cores of business operation. One such example is Shark [1], an interactive analytic engine that can process SQL and advanced analytics at scale. The key feature of big data applications is high fan-out, composed of a large number of distributed components that exhibit disparate resource demands and complex execution dependencies. It is thus intricate to deploy such big data applications, especially in terms of resource allocation for each component.

In addition to the inherent complexity of system stacks, the difficulty of managing big data applications for datacenter providers is further exacerbated by the transient dynamics of cloud systems. First of all, the available system resources can fluctuate, due to energy optimization strategies which consider energy prices [2] and cooling conditions [3]. Also, the limited resources are shared among applications, each of which is associated with different business values and performance requirements. Second, as applications tend to take advantages of pay-as-you-go pricing models in the cloud, application workloads are highly transient and dynamic [4].

In this paper, we address a very challenging question: how can todays datacenter providers assist the deployment and resource allocation of big data applications under (stringent) resource constraints and meanwhile guarantee differentiated performance across co-executing allocations? To such an end, we develop a solution, termed OptiCA, that enables two aims: efficient configuration and allocating computational capacities, in the unit of divisible CPU cores, for big data applications. OptiCA systematically orchestrates the configuration of application components such that the deployment process can be easily automated and independent of the underlying hardware and thus portable. OptiCA assigns CPU cores budget to applications based on their business value and to each application component based on its effective demand, which is obtained non-intrusively by profiling of the components' utilization. In particular, we are able to control CPU cores being fractional utilize by application components such that allocated core budget of application is best met [1].

In terms of design, OptiCA is composed of four modules, namely manager, driver, controller, and power monitor. The former two focus on deploying and configuring the applications, while the latter two are responsible for the utilization monitoring and the budget enforcement. We evaluate OptiCA on a modern multicore cluster executing two state-of-the-art big data application benchmarks, BigDataBench [5] and Terasort [6], both built on two complex yet different software platforms. Our evaluation results show that OptiCA is able to effectively assign resources to the components such that performance degradation, in terms of completion time, is minimized in cases where available core budgets rapidly decrease.

Our contributions are twofold. We present an efficient and automated deployment process for big data applications. We develop a practical and robust solution which can accurately capture the resource demands of distributed components of big data applications, and further allocate the desired CPU resources. Our results are well validated by experiments conducted on prototype systems - representing the state-of-the-art execution environment and complex big data execution platforms in production systems.

The outline of this work is as follows. Section II provides an overview of the system and big data applications. We detail OptiCA and its allocation algorithm in Section III. In Section IV we present extensive evaluation results, followed by the related work in Section V. Section VI concludes with summary and future work.

II. System and Applications

In this section, we describe the general system setup and application model considered in this study. In particular, we consider a primary application and a background application,

[1] In this paper, we define computational capacities by the unit of divisible CPU cores, as CPU cores can be fractionally utilized. We interchangeably use computational capacity allocation and (divisible) (CPU) core allocation.

co-executing on a homogeneous cluster, which has a given resource budget in terms of fully utilized CPU cores.

A. System

Here, we explain the resource budget available for co-executing applications in the system. A cloud cluster consists of N servers, each having K cores, and must fulfill certain power budgets that are often driven by energy pricing schemes [7]. We particularly consider the case where both servers and cores are homogeneous and servers are always on to guarantee access to the distributed data, e.g., HDFS [8]. We assume a core has a normalized capacity of one and can be utilized, in the range [0, 1], by multiple applications simultaneously. As the power consumption of a node is composed of a fixed term and dynamic term, which is shown linearly proportional to the aggregate core usage [9], we focus on *dynamic power budget*, which equals to the power budget subtracted by the fixed term of all nodes.

Further simplifying the idea of budget, we define the concept of global resource budget, representing the amount of available core capacities, $B \in [0, B^{max}]$, aggregated across all nodes. The maximum amount of core capacities, B^{max}, equals to the total number of fully-utilized cores in the cluster, i.e., $N \cdot K$. Note that as a core can be kept partially utilized, the core capacities and consumption are *continuous* values. Due to the linear relationship between the dynamic power and core usage, there is a one-to-one mapping between power budget and resource budget in the system considered. Consequently, we resort to control core utilization so to achieve any given power/resource budget in a non-intrusive manner. We need to ensure that the aggregated core usages of all nodes are less than the available budgets. Note that we assume that the values of available budgets are decided by a global optimization scheme which is out of the scope of the paper.

B. Big Data Applications

In terms of applications, we consider two types: primary *big data applications*, such as Shark [1] and YARN [10], and *background applications*, such as Pi (refer to Section IV-A for further details on the applications). These two application types differ in both their business values, performance requirements, and workload characteristics. Big data applications are highly distributed and composed of several components which can be hosted on the same physical nodes and cores where the background applications execute their work. Figure 1 depicts a cluster of four physical servers hosting components of Terasort and Pi, the primary and background application, respectively. Terasort is developed on the Apache Hadoop NextGen MapReduce (YARN).

For a better illustration of big data application components, we provide the example of a basic YARN installation to run MapReduce jobs. The underlying HDFS requires a minimum of one namenode plus three datanodes to satisfy a replication factor of three. Correspondingly, the following additional six components are needed: three nodemanagers running the map and reduce tasks, one resource-manager scheduling tasks, one job-history storing task statistics, and one Domain Name Server (DNS). A total of ten components are needed, each requiring a more or less lengthy cross configuration process

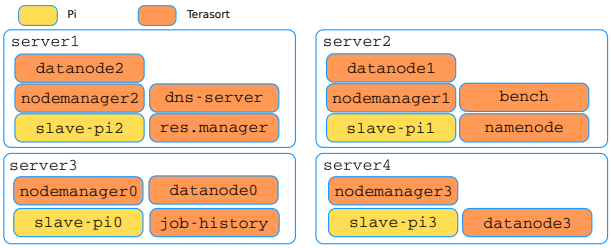


Fig. 1: Deployment of the Terasort and Pi on a cluster of 4 physical servers.

among components as well as of the underlying physical servers. Nevertheless, applications developed upon different types of data processing platforms, such as Shark, have additional application-specific components. More details about the applications can be found in Section IV. Note that the example described in this paragraph is slightly different from that one shown in Figure 1.

C. Application Core Budget

The challenge faced by providers is how to assist their clients to control the resource budget in a performance-fulfilling and non-intrusive way. We assume that providers need to allocate the resource budget across applications and their components, with respect to their relative business values. In particular, we consider a proportional fair budget allocation scheme for applications, whose weights are given by their business values. We assume that the estimation of such values is given and out of the scope of this paper. We denote the weights for big data application and background application executing on the cluster as α_1 and α_2, respectively. The resource budget allocated to application i is thus

$$B_i = \frac{\alpha_i}{\alpha_1 + \alpha_2} B, i = \{1, 2\} \quad (1)$$

The global cores consumption of all components belonging to an application should be less or equal to the budget. Essentially, our proposed solution aims at assigning and deploying the core budget across the distributed components, under a applications proportional fair scheme.

III. OptiCA

In this section, we first detail the architecture of OptiCA that provides efficient deployment and budget allocation scheme of complex applications. In particular, our solution addresses the complexity of cross-configuration according to inter-component and component-node dependencies. Secondly, we describe a novel algorithm used in the controller module that allocates core budget to components, based on the concept of effective demand.

A. Architecture

OptiCA is composed of four modules, namely, *manager*, *driver*, *controller* and *powermon* (Figure 2). The former two are related to the application deployment and the latter two are used for computation allocation. Each application component is executed in a Docker container [11], which thus can be hosted on physical nodes with a great mobility. Note that the container is conceptually similar to the virtual machine but

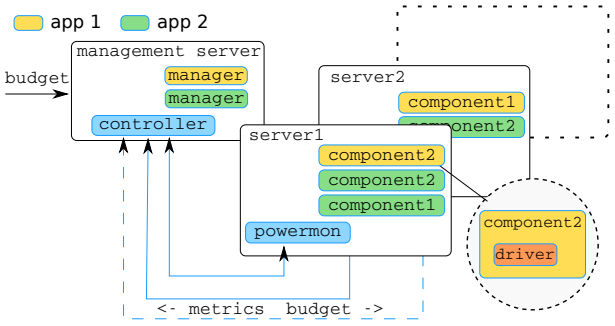


Fig. 2: Architecture of OptiCA.

more suitable for fast deployment due to its lower overhead. For instance, namenode and datanode for underlying Hadoop File System used by big data applications are hosted on two different containers. Essentially, the core budget received by a component can be seen as a container budget. An additional note on the applicability of proposed solution is that OptiCA is generally portable across Linux systems.

1) Deployment Related Modules: Prior to introducing the details of the modules related to the deployment, we use the basic example of Hadoop introduced in the previous section to illustrate the configuration process. First of all, a total of 10 components need to be installed, followed by the network configuration of all components, i.e., registering hostnames at DNS. Typically, the resource allocation of each component should be specified upon the deployment time by the application; otherwise, they are left unbounded. Then, Hadoop is configured, starting from formatting namenode, and setting the URL of component's configuration files. Afterwards, all components need to start in a specific sequence so to execute big data applications properly.

To ease the deployment of applications across platforms, our solution relies on one manager module per application and multiple driver modules per application component. Both modules are customized for applications, specifically extracting out configuration parameters and detailing out aforementioned configuration sequences. The manager initiates the deployment of all components and orchestrates the sequence of cross configurations among drivers, via a REST API over HTTP. Here, the components' resources are not defined; instead, they are modulated by our proposed "controller" module at run-time. The main responsibility of the manager is to cross-configure the network of the components. As big data applications are developed using the MapReduce paradigm, the manager modules of different big data applications are very similar and, thus, reproducible with minimum effort. Drivers are essentially daemons that automatically start upon the instantiation of components and apply the configuration given by the manager. In summary, the advantages of the proposed design are automation, reproducibility, and portability of lengthy configuration processes of big data applications on different platforms and hardware systems.

2) Core Allocation Related Modules: To control the budget allocation of each application and their components, we use a central controller module hosted on a dedicated management server and distributed powermon modules on each physical server. The controller splits the budget across applications and components. Each powermon has two key functionalities: monitoring the resource consumption of containers hosted on the same node and enforcing their budgets.

In particular, as described is Section II-C, the controller first splits the available core budget across big data application and background application based on their weights. Then, it decides the computation allocation of components for the big data application, based on the algorithm described in the following subsection. In addition to satisfy the application budget, the controller also needs to respect the constraints given by the physical server capacities: the sum of cores allocated to containers co-located on a server cannot exceed the total number of cores equipped on that server. The controller always gives high priority to the big data application, when there is insufficient core resource at server. In other words, the component of background application can only receive the residual core capacities, left by the big data application.

B. CPU Core Budgets for Component

Here, we describe how OptiCA decides the CPU computational budgets across the M_i components of an application, denoted as $C_{i,j}$, where $i = 1, 2$ and $j \in \{1 \dots M_i\}$. The sum of component budget must be less or equal to the application budget, B_i,

$$\sum_{j=1}^{M_i} C_{i,j} \leq B_i \tag{2}$$

The basic idea of the proposed algorithm is to allocate cores based on the components' *effective consumption*, $U_{i,j}$, collected during a profiling phase without any limitation. As shown in Figure 3, $U_{i,j}$ is defined as CPU consumption measured during non-idle periods of the profiling phase, i.e., utilization is strictly greater than 0. During the profiling phase, big data applications are given all the available core resources of the entire system, i.e., without any background application running and without any CPU capping enforced on any component. Big data applications are executed with a few iterations to obtain the average values of effective consumption. We note that the values of effective consumption is higher than average consumption, which considers both busy times and idle times. As such, effective consumption can better capture the real resource demand of each component.

We assign cores to components, based on their effective consumption as weights,

$$C_{i,j} = w_{i,j} \cdot B_i, \quad \text{where} \quad w_{i,j} = \frac{U_{i,j}}{\sum_{l=1}^{M_i} U_{i,l}} \tag{3}$$

Moreover, computation allocation of every component can not exceed the total available core capacity on a node, i.e., $C_{i,j} \leq K$. And, actual core utilization of each component needs to be less than the allocated capacity. In our solution, we do not specify any core-pinning for the components and we leave the decision up to the underlying OS scheduler. Note that the core budgets are continuous values, as we implement them via the concept of CPU time described as follows.

The computation allocation to components, $C_{i,j}$, is translated into the maximum CPU time given a time period, also

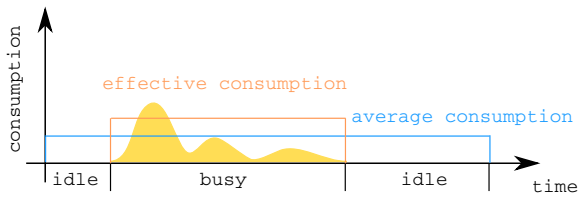


Fig. 3: Core consumption: average vs. effective

known as *CPU-bandwidth*. In particular, one can use the parameter pair $(quota, period)$ in the *cgroup*, where $period$ denotes the duration of using $C_{i,j}$ and $quota$ represents the resulting total core consumption, i.e., $quota = period \cdot C_{i,j}$. The range of period is between 1 ms and 1 s. On the one hand, a small $period$ provides a fine-grained control. On the other hand, the lower is the $period$, the higher is the overhead to enforce the $quota$. Based on an experimental evaluation, we choose a period of 100 ms. With such a value the budget can be best enforced without significant performance degradation.

Overall, the implementation of allocation schemes based on the effective consumption is both application and big-data platform independent, as we rely on standard Linux features. Moreover, the required profiling phase is non-intrusive to application, relying only on core utilization values associated with each component.

IV. Evaluation

In this section we evaluate our proposed allocation solution on a representative prototype cluster, hosting two primary big data applications, namely BigDataBench [5] and Terasort [6], as well as a background application computing π, termed Pi. BigDataBench and Terasort are built on two state-of-the-art big data platforms, Shark [1] and YARN [10], respectively. We present the performance degradation of big data applications, particularly the completion times, under different levels of core budgets. Moreover, we benchmark the proposed allocation schemes based on effective consumption against other approaches, which rely on the information of components' placement or average core consumption.

In the following, we first detail out the evaluation setup, including the system architecture of the cluster and the prototype of the software stacks of the applications. Then, we describe the parametrization of the proposed allocation schemes, followed by the extensive evaluation on representative big data systems.

A. Experiment Setup

Here, we first describe the available resources of the underlying system and application components. We then detail three experimental scenarios and three alternative approaches.

System: Our cluster consists of 4 physical servers, each of which equipped with 8 cores, at least 16 GB of RAM, and 2.7 TB RAID storage. There is an additional management node, hosting the managers of the applications and the controller. All servers are connected with 10 GB Ethernet links. Servers run Ubuntu 14.04. The maximum amount of available core capacity is $B^{max} = 32$ $(4 \cdot 8)$ to be distributed across co-located applications and their components.

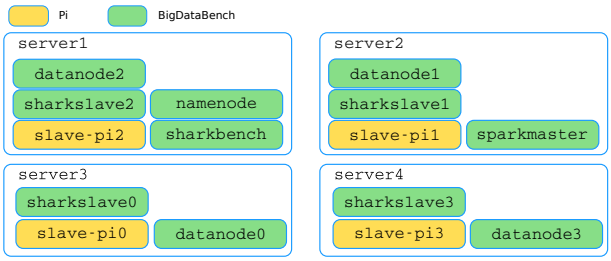


Fig. 4: Deployment of the BigDataBench and Pi on a cluster of 4 physical servers.

Applications: Our target applications are BigDataBench and Terasort, as representative of primary big data workloads; and Pi, π estimation based on Monte Carlo method, is the co-executed background application. The performance metrics of interest are the average completion times for big data applications and throughput for Pi. In the following, we describe the specific configurations and the datasets of aforementioned benchmarks.

BigDataBench runs 5 different SQL queries over a varying-sized dataset stored in the HDFS filesystem. This benchmark is composed of 11 components: 4 sharkslaves, 4 datanodes, 1 namenode, 1 sharkbench and 1 sparkmaster. The queries are submitted to sharkbench, a central gateway that interprets them and defines a direct acyclic graph of tasks. The graph is sent to the sparkmaster, which orchestrates the sequence of tasks executions on the sharkslaves. The slaves, in turn, interact with namenode and datanodes to access the dataset stored in the HDFS filesystem. In particular, we select `query1a` and `query1b`.

Terasort is a standard Hadoop benchmark which sorts a given dataset stored on HDFS. Given the size of the target cluster, we choose a dataset of 10 millions rows, approximately 1GB. This benchmark is composed of 13 components: 1 namenode, 4 datanodes, 1 resourcemanger, 4 nodemanagers, 1 job-history, 1 DNS server, and 1 client named bench. Their functionalities are explained in Section II-B.

Pi is a distributed CPU-intensive micro-benchmark made of two kinds of components: master and slave. Slaves apply a simple Monte Carlo based algorithm and the master aggregates results from multiple slaves and computes an estimation of π. In our setup, we consider one master and four slaves.

In terms of experiments, we consider the following three co-execution scenarios: (1) executing `query1a` of BigDataBench co-located with Pi, shown in Figure 4, (2) executing query1b of BigDataBench co-located with Pi, shown in Figure 4, and (3) executing TeraSort co-located with Pi, shown in Figure 1. Moreover, we particularly consider the weights of the big data application and the background application as 0.8 and 0.2, respectively, so we split the available core budget between these two applications according to Equation 1.

Baseline algorithms: As a comparison basis for the algorithm proposed in Section III-B, to which we refer as `OptiCA`, we implement the following three simple algorithms to allocate core budget to application components.

- `fair`. The application core budget is split evenly across the components.

- `static`. The application core budget is first split evenly across the physical servers. Then, each server share is evenly split across the components. Note that `static` is equal to `fair` when the number of components per server is balanced.
- `util`. Each component receives the core capacity based on weights obtained from average core consumption values in the profiling phase. Note that the difference with the proposed allocation scheme is that we rely on the effective consumption values, collected only during non-idle times.

B. BigDataBench and Pi

Here, we present the results of executing `query1a` and `query1b` of BigDataBench with Pi, when the global budget ranges in $\{8, 12, 16, 24, 32\}$. The corresponding application budget for BigDataBench and Pi are $\{6.4, 9.6, 12.8, 19.2, 25.6\}$, and $\{1.6, 2.4, 3.2, 4.8, 6.4\}$, respectively. We summarize the average completion times of BigDataBench and throughput of Pi in Figure 5. In addition to the average bar, we also plot the minimum and the maximum completion times across the 5 repetitions of the same query. Overall, the completion times decrease and throughput increases with higher available core budgets, across all allocation schemes.

The trends of completion times of `query1a` and `query1b` are similar across different algorithms and under different core budgets. For a given budget, `OptiCA` always results into the lowest completion time, especially in the case of scarcity of resources. In particular, when application budget is 6.4, `OptiCA` can achieve query completion times 1.5 smaller than `util`, the second best algorithm. When the core resources are abundant, such as budget is 25.6 cores, the differences across algorithms are minor. Moreover, looking at performance degradation when the core budget reduces by three quarters, i.e.25.6 to 6.4, the average completion times under `OptiCA` increase by only 2X, whereas other algorithms results into an increment of 2.8X up to 4X.

We present the throughput of Pi executing with `query1a` and `query1b`, in Figure 5b and 5d respectively. One can see most of algorithms achieve similar levels of throughput, except for the case of core budget of 6.4. This is due to all Pi slaves receiving similar amount of computation allocation. When Pi receives core budget of 6.4, `static` algorithm achieves the highest throughput, roughly 1.44 times better than `util` (the worst algorithm). This is attributed to the fact that `static` policy is able to even out the resource consumption between the two applications better. As such, when the total core budget is 32 (resources abundant), `static` policy can be considered as a good alternative to `OptiCA`, i.e., slightly lower completion time of `query1a` and `query1b` and a higher throughput of Pi.

Moreover, we also investigate how the allocated cores are actually consumed under different schemes. To formally quantify how effectively the budget is consumed, we propose a metric, termed slack, defined as the average difference between the budget and the global consumption samples. Table I summarizes the values of *slack* for all the algorithms, both the absolute and relative value (shown within brackets). The relative values can be interpreted as the percentage of global budget potentially wasted. For low budgets, `OptiCA` has a slack much smaller than all the other algorithms; while as the budget increases the difference among algorithms decreases. Combining this with the observation about completion times, we thus conclude that `OptiCA` can more effectively assign core capacities to components which have higher demands.

Budget	Algorithm			
	fair	static	util	OptiCA
8	4.8 (60%)	5.2 (65%)	4.8 (60%)	3.7 (46%)
12	7.4 (62%)	7.7 (64%)	7.3 (60%)	5.7 (47%)
16	10.0 (62%)	10.3 (64%)	9.6 (60%)	8.3 (52%)
24	15.5 (65%)	15.6 (65%)	14.5 (60%)	14.4 (60%)
32	23.1 (72%)	21.3 (67%)	22.5 (70%)	22.2 (70%)

TABLE I: BigDataBench and Pi: absolute and relative slack values across different algorithms.

C. Terasort and Pi

In this subsection, we present the results of executing Terasort and Pi, using the same set of budget values of the previous subsection. In Figure 6a and Figure 6b summarize the completion times of Terasort and throughput of Pi, respectively. The overall findings here are similar to the experiment of the previous subsection: `OptiCA` can achieve low completion times for Terasort and decent throughput for Pi when the core budget is low. In particular, the average completion time of `OptiCA` is lower than the second best algorithms by 30% to 40%, except in the case of budget being 9.6. Moreover, when the application budgets for Pi are 4.8 and 6.4, `static` policy can achieve higher throughput. Again, this can be explained by the free core capacity left by Terasort under different schemes. Both `util` and `OptiCA` tend to grab more core resources on physical server 1 and thus very little core capacity can be given to Pi computation. In addition to low average completion times, `OptiCA` systematically has the minimum completion-time under different core budgets.

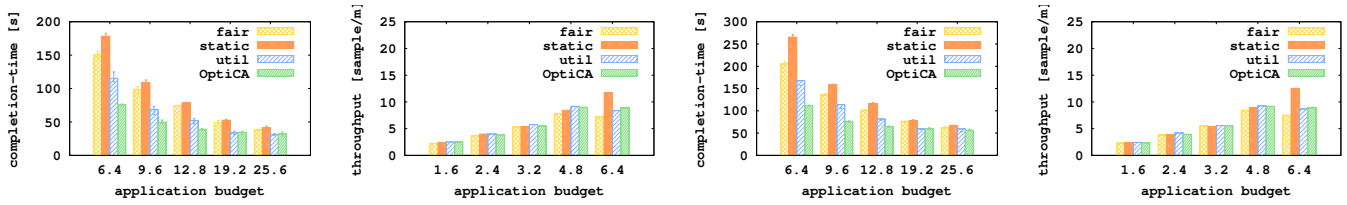
In summary, `OptiCA` is able to achieve robust performance for big data applications under different core budgets, by efficiently allocating component budgets.

V. Related work

Motivated by the increasing importance of big data applications in today's cloud data centers, there is a plethora of related studies addressing the performance issues of big data from the perspectives of management platforms, and resource allocation of MapReduce jobs and tasks.

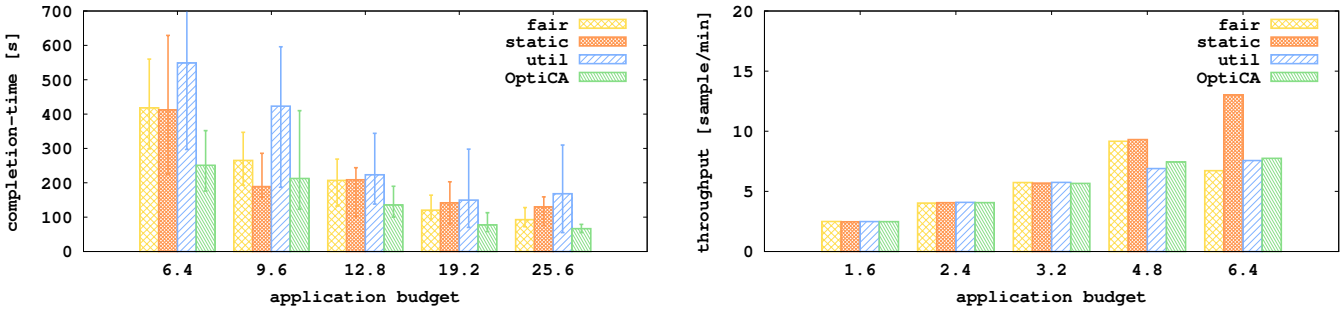
A. Big Data Platforms

Several platforms have emerged in recent years to address the challenges of resource management of big data applications at cluster level. Mesos [12] is a multi-platform manager enabling resource sharing across different big data platforms, such as Hadoop, Spark and MPI. YARN [10] is the next generation of the Hadoop, platform featuring fault-tolerance, high-scalability, while being extensible to different applications. Omega [13] addresses resource allocations across applications and resolves conflicts by optimistic concurrency control. In contrast to above studies, Ghit et al. [14] propose a resource management system to facilitate the deployment of MapReduce clusters in an on-demand fashion and with



(a) query1a (query1a + Pi) (b) Pi (query1a + Pi) (c) query1b (query1b + Pi) (d) Pi (query1b + Pi)

Fig. 5: BigDataBench and Pi: application performance under different resource allocation schemes.



(a) performance of Terasort co-located with simple-pi (b) performance of simple-pi co-located with Terasort

Fig. 6: TeraSort and Pi: application performance under different resource allocation schemes.

particular focus on component level. Overall, aforementioned studies tend to develop platform-specific resource management policies at application or cluster levels.

B. Resource Allocation and Scheduling of MapReduce

Studies on dynamical resources allocation to MapReduce tasks as well as jobs employ different methodologies, such as off-line profiling, on-line monitoring, machine learning, and economics cost model [15, 16]. Often, they focus on scenarios where resources are abundant. Verma et. al. [17, 18] developed a deadline-driven framework which can dynamically resize the map and reduce tasks and further allocate resources, in particular slots, to tasks as well as jobs. Their methodology is based on profiling and performance modeling. MROrchestrator [19] is a resource manager that monitors online the performance of each node, such that performance bottlenecks of MapReduce tasks can be detected to adjust resource allocations, such CPU and memory. AROMA [20] uses machine learning and optimization techniques to both configure and provision resources for MapReduce jobs. Nephele [21] focuses on scheduling jobs on different types of virtual machines based on their requirements of CPU, memory, network and disk. Zhang et al. [22] passively characterize the cost-performance trade-off when executing MapReduce jobs running on different kinds of Amazon EC2 virtual machines and conclude that a properly sized VM can results into a 70% cost saving without performance penalty. All in all, the prior art tends to investigate the resource allocation at level of MapReduce jobs or tasks, and, more importantly, in a dedicated cluster, except [23] which explicitly discusses the scheduling problem in a shared environment.

In contrast to the related work, OptiCA is a portable and platform independent solution that eases the deployment of applications and their components. OptiCA is designed to allocate core resources to components in a resource constrained situation, particularly suitable for highly distributed applications.

VI. Conclusion

In this paper, we present an integrated solution, OptiCA, to optimize the computation allocation of big data applications and their components, co-executed with background applications in a cloud datacenter. Two key functionalities of OptiCA are efficient deployment and computation allocation of big data components. Using the novel ideas of effective demands and fraction of core capacity, OptiCA is able to effectively allocate the required core resources. Our evaluation results show that OptiCA achieves low performance degradation of big data applications and a good throughput for background applications when the available core resources are low. For our future work, we plan to explore different schemes to allocate application core budgets and expand the evaluation on a larger number of applications and clusters.

VII. Acknowledgments

The research presented in this paper has been supported by the Swiss National Science Foundation (project 200021_141002). This work has been partly funded by the EU Commission under the FP7 GENiC project (Grant Agreement No 608826).

REFERENCES

[1] C. Engle, A. Lupher, R. Xin, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: Fast Data Analysis Using Coarse-grained Distributed Memory," in *Proceedings of SIGMOD'12*, pp. 689–692.

[2] Z. Liu, M. Lin, A. Wierman, S. H. Low, and L. L. H. Andrew, "Greening Geographical Load Balancing," in *Proceedings of SIGMETRICS'11*, pp. 233–244.

[3] N. El-Sayed, I. A. Stefanovici, G. Amvrosiadis, A. A. Hwang, and B. Schroeder, "Temperature Management in Data Centers: Why Some (Might) Like It Hot," in *Proceedings of SIGMETRICS'12*, pp. 163–174.

[4] R. Birke, L. Y. Chen, and E. Smirni, "Data centers in the cloud: A large scale performance study," in *Proceedings of CLOUD'12*, pp. 336–343.

[5] BigDataBench, "https://amplab.cs.berkeley.edu/benchmark/."

[6] Terasort, "http://sortbenchmark.org/yahoohadoop.pdf."

[7] H. Lim, A. Kansal, and J. Liu, "Power Budgeting for Virtualized Data Centers," in *Proceedings of ATC'11*, pp. 59–72.

[8] Hadoop, "http://hadoop.apache.org/."

[9] L. A. Barroso and U. Hölzle, "The Case for Energy-Proportional Computing," *IEEE Computer*, vol. 40, no. 12, pp. 33–37, 2007.

[10] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proceedings of SOCC'13*, pp. 63–78.

[11] Docker, "https://www.docker.com/."

[12] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center," in *Proceedings of NSDI'11*.

[13] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, Scalable Schedulers for Large Compute Clusters," in *Proceedings of EuroSys'13*, pp. 351–364.

[14] B. Ghit, N. Yigitbasi, and D. H. J. Epema, "Resource Management for Dynamic MapReduce Clusters in Multicluster Systems," in *Proceedings of SC'12*, pp. 1252–1259.

[15] F. Tian and K. Chen, "Towards Optimal Resource Provisioning for Running MapReduce Programs in Public Clouds," in *Proceedings of CLOUD'11*, pp. 155–162.

[16] I. Menache, O. Shamir, and N. Jain, "On-demand, Spot, or Both: Dynamic Resource Allocation for Executing Batch Jobs in the Cloud," in *Proceedings of ICAC'14*, pp. 177–187.

[17] A. Verma, L. Cherkasova, and R. H. Campbell, "ARIA: Automatic Resource Inference and Allocation for MapReduce Environments," in *Proceedings of ICAC'11*, pp. 235–244.

[18] A. Verma, L. Cherkasova, V. S. Kumar, and R. H. Campbell, "Deadline-based Workload Management for MapReduce Environments: Pieces of the Performance Puzzle," in *Proceedings of NOMS'12*, pp. 900–905.

[19] B. Sharma, R. Prabhakar, S. Lim, M. T. Kandemir, and C. R. Das, "MROrchestrator: A Fine-Grained Resource Orchestration Framework for MapReduce Clusters," in *Proceedings of CLOUD'12*, pp. 1–8.

[20] P. Lama and X. Zhou, "AROMA: Automated Resource Allocation and Configuration of MapReduce Environment in the Cloud," in *Proceedings of ICAC'12*, pp. 63–72.

[21] D. Warneke and O. Kao, "Exploiting Dynamic Resource Allocation for Efficient Parallel Data Processing in the Cloud," *IEEE Transactions Parallel Distributed Systems*, vol. 22, no. 6, pp. 985–997, 2011.

[22] Z. Zhang, L. Cherkasova, and B. T. Loo, "Optimizing Cost and Performance Trade-offs for Performance Job Processing in the Cloud," in *Proceedings of NOMS'14*, pp. 1–8.

[23] B. Sharma, T. Wood, and C. R. Das, "HybridMR: A Hierarchical MapReduce Scheduler for Hybrid Data Centers," in *Proceedings of ICDCS'13*, pp. 102–111.